Yaser S. Abu-Mostafa
Malik Magdon-Ismail
Hsuan-Tien Lin

Learning From Data

A SHORT COURSE

AMLbook.com

Learning From Data

A SHORT COURSE

Yaser S. Abu-Mostafa
California Institute of Technology

Malik Magdon-Ismail
Rensselaer Polytechnic Institute

Hsuan-Tien Lin
National Taiwan University

AMLbook**.com**

Yaser S. Abu Mostafa
Departments of Electrical Engineering
and Computer Science
California Institute of Technology
Pasadena, CA 91125, USA
yaser@caltech.edu

Malik Magdon Ismail
Department of Computer Science

Rensselaer Polytechnic Institute
Troy, NY 12180, USA
magdon@cs.rpi.edu

Hsuan Tien Lin
Department of Computer Science
and Information Engineering
National Taiwan University
Taipei, 106, Taiwan
htlin@csie.ntu.edu.tw

ISBN 10:1 60049 006 9
ISBN 13:978 1 60049 006 4

©2012 Yaser S. Abu Mostafa, Malik Magdon Ismail, Hsuan Tien Lin. 1.10

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the authors. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the authors, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act.

Limit of Liability/Disclaimer of Warranty: While the authors have used their best efforts in preparing this book, they make no representation or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. The authors shall not be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

The use in this publication of tradenames, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

This book was typeset by the authors and was printed and bound in the United States of America.

Contents

1. The Learning Problem 1
    1.1 Problem Setup 1
        1.1.1 Components of Learning 3
        1.1.2 A Simple Learning Model 5
        1.1.3 Learning versus Design 9
    1.2 Types of Learning 11
        1.2.1 Supervised Learning 11
        1.2.2 Reinforcement Learning 12
        1.2.3 Unsupervised Learning 13
        1.2.4 Other Views of Learning 14
    1.3 Is Learning Feasible? 15
        1.3.1 Outside the Data Set 16
        1.3.2 Probability to the Rescue 18
        1.3.3 Feasibility of Learning 24
    1.4 Error and Noise 27
        1.4.1 Error Measures 28
        1.4.2 Noisy Targets 30
    1.5 Problems 33
2. Training versus Testing 39
    2.1 Theory of Generalization 39
        2.1.1 Effective Number of Hypotheses 42
        2.1.2 Bounding the Growth Function 46
        2.1.3 The VC Dimension 50
        2.1.4 The VC Generalization Bound 53
    2.2 Interpreting the Generalization Bound 55
        2.2.1 Sample Complexity 57
        2.2.2 Penalty for Model Complexity 58
        2.2.3 The Test Set 59
        2.2.4 Other Target Types 61
    2.3 Approximation-Generalization Tradeoff 62
        2.3.1 Bias and Variance 62
        2.3.2 The Learning Curve 67
    2.4 Problems 69
3. The Linear Model 77
    3.1 Linear Classification 77
        3.1.1 Non-Separable Data 79
    3.2 Linear Regression 82
        3.2.1 The Algorithm 84

3.2.2 Generalization Issues 87
3.3 Logistic Regression 88
3.3.1 Predicting a Probability 89
3.3.2 Gradient Descent 93
3.4 Nonlinear Transformation 99
3.4.1 The Z Space 99
3.4.2 Computation and Generalization 104
3.5 Problems 109
4. Overfitting 119
4.1 When Does Overfitting Occur? 119
4.1.1 A Case Study: Overfitting with Polynomials 120
4.1.2 Catalysts for Overfitting 123
4.2 Regularization 126
4.2.1 A Soft Order Constraint 128
4.2.2 Weight Decay and Augmented Error 132
4.2.3 Choosing a Regularizer: Pill or Poison? 134
4.3 Validation 137
4.3.1 The Validation Set 138
4.3.2 Model Selection 141
4.3.3 Cross Validation 145
4.3.4 Theory Versus Practice 151
4.4 Problems 154
5. Three Learning Principles 167
5.1 Occam's Razor 167
5.2 Sampling Bias 171
5.3 Data Snooping 173
5.4 Problems 178
6. Similarity-Based Methods 181
6.1 Similarity 181
6.1.1 Similarity Measures 182
6.2 Nearest Neighbor 183
6.2.1 Nearest Neighbor is 2-Optimal 185
6.2.2 k-Nearest Neighbors (k-NN) 188
6.2.3 Improving the Efficiency of Nearest Neighbor 192
6.2.4 Nearest Neighbor is Nonparametric 197
6.2.5 Multiclass Data 199
6.2.6 Nearest Neighbor Regression 201
6.3 Radial Basis Functions 203
6.3.1 Radial Basis Function Networks 205
6.3.2 Fitting the Data 209

6.3.3 Unsupervised k-Means Clustering 211
6.3.4 Justifications and Extensions of RBFs.....214
6.4 Probability Density Estimation.....214
6.4.1 Gaussian Mixture Models (GMMs) 218
6.5 Problems 222
7. Neural Networks 233
7.1 The Multi-layer Perceptron (MLP).....233
7.2 Neural Networks 239
7.2.1 Notation 239
7.2.2 Forward Propagation.....241
7.2.3 Backpropagation Algorithm 243
7.2.4 Regression for Classification.....251
7.3 Approximation versus Generalization 251
7.4 Regularization and Validation.....255
7.4.1 Weight Based Complexity Penalties.....255
7.4.2 Early Stopping 256
7.4.3 Experiments With Digits Data.....259
7.5 Beefing Up Gradient Descent 261
7.5.1 Choosing the Learning Rate η 262
7.5.2 Conjugate Gradient Minimization.....266
7.6 Deep Learning: Networks with Many Layers.....268
7.6.1 A Greedy Deep Learning Algorithm.....271
7.7 Problems 275
8. Support Vector Machines 283
8.1 The Optimal Hyperplane 283
8.1.1 Finding the Fattest Separating Hyperplane 285
8.1.2 Is a Fat Separator Better?.....294
8.1.3 Non-Separable Data 300
8.2 Dual Formulation of the SVM.....303
8.2.1 Lagrange Dual for a QP-Problem 304
8.2.2 Dual of the Hard-Margin SVM.....307
8.2.3 Recovering the SVM from the Dual Solution.....311
8.3 Kernel Trick for SVM 315
8.3.1 Kernel Trick via Dual SVM 315
8.3.2 Choice of Kernels.....321
8.4 Soft-margin SVM 322
8.5 Problems 328
9. Learning Aides.....336
9.1 Input Preprocessing 336
9.2 Dimension Reduction and Feature Selection.....341

9.2.1 Principal Components Analysis (PCA) 342
9.2.2 Nonlinear Dimension Reduction 351
9.3 Hints And Invariances 354
9.3.1 Virtual Examples 358
9.3.2 Hints Versus Regularization 360
9.4 Data Cleaning 360
9.4.1 Using a Simpler Model to Identify Noisy Data 361
9.4.2 Computing a Validation Leverage Score 363
9.5 More on Validation 365
9.5.1 Rademacher Penalties 365
9.5.2 The Permutation and Bootstrap Penalties 366
9.5.3 Rademacher Generalization Bound 368
9.5.4 Out-of-Sample Error Estimates for Regression 369
9.6 Problems 373
Epilogue 387
Further Reading 389
Appendix A - Proof of the VC Bound 393
A.1 Relating Generalization Error to In-Sample Deviations 394
A.2 Bounding Worst Case Deviation Using the Growth Function 396
A.3 Bounding the Deviation between In-Sample Errors 397
e-Appendix B - Linear Algebra 399
B.1 Basic Properties of Vectors and Matrices 399
B.2 SVD and Pseudo-Inverse 403
B.3 Symmetric Matrices 404
B.3.1 Positive Semi-Definite Matrices 405
B.4 Trace and Determinant 405
B.4.1 Inverse and Determinant Identities 407
B.5 Inner Products, Metrix and Vector Norms 408
B.5.1 Linear Hyperplanes and Quadratic Forms 409
B.6 Vector and Matrix Calculus 410
B.6.1 Multidimensional Integration 410
B.6.2 Matrix Derivatives 411
e-Appendix C - The E-M Algorithm 413
C.1 Derivation of the E-M Algorithm 417
C.2 Problems 422

Chapter 1

The Learning Problem

If you show a picture to a three-year-old and ask if there is a tree in it, you will likely get the correct answer. If you ask a thirty-year-old what the definition of a tree is, you will likely get an inconclusive answer. We didn't learn what a tree is by studying the mathematical definition of trees. We learned it by looking at trees. In other words, we learned from 'data'.

Learning from data is used in situations where we don't have an analytic solution, but we do have data that we can use to construct an empirical solution. This premise covers a lot of territory, and indeed learning from data is one of the most widely used techniques in science, engineering, and economics, among other fields.

In this chapter, we present examples of learning from data and formalize the learning problem. We also discuss the main concepts associated with learning, and the different paradigms of learning that have been developed.

1.1 Problem Setup

What do financial forecasting, medical diagnosis, computer vision, and search engines have in common? They all have successfully utilized learning from data. The repertoire of such applications is quite impressive. Let us open the discussion with a real-life application to see how learning from data works.

Consider the problem of predicting how a movie viewer would rate the various movies out there. This is an important problem if you are a company that rents out movies, since you want to recommend to different viewers the movies they will like. Good recommender systems are so important to business that the movie rental company Netflix offered a prize of one million dollars to anyone who could improve their recommendations by a mere 10%.

The main difficulty in this problem is that the criteria that viewers use to rate movies are quite complex. Trying to model those explicitly is no easy task, so it may not be possible to come up with an analytic solution. However, we

Figure 1.1: A model for how a viewer rates a movie

know that the historical rating data reveal a lot about how people rate movies, so we may be able to construct a good empirical solution. There is a great deal of data available to movie rental companies, since they often ask their viewers to rate the movies that they have already seen.

Figure 1.1 illustrates a specific approach that was widely used in the million-dollar competition. Here is how it works. You describe a movie as a long array of different factors, e.g., how much comedy is in it, how complicated is the plot, how handsome is the lead actor, etc. Now, you describe each viewer with corresponding factors; how much do they like comedy, do they prefer simple or complicated plots, how important are the looks of the lead actor, and so on. How this viewer will rate that movie is now estimated based on the match/mismatch of these factors. For example, if the movie is pure comedy and the viewer hates comedies, the chances are he won't like it. If you take dozens of these factors describing many facets of a movie's content and a viewer's taste, the conclusion based on matching all the factors will be a good predictor of how the viewer will rate the movie.

The power of learning from data is that this entire process can be automated, without any need for analyzing movie content or viewer taste. To do so, the learning algorithm 'reverse-engineers' these factors based solely on pre-

vious ratings. It starts with random factors, then tunes these factors to make them more and more aligned with how viewers have rated movies before, until they are ultimately able to *predict* how viewers rate movies in general. The factors we end up with may not be as intuitive as 'comedy content', and in fact can be quite subtle or even incomprehensible. After all, the algorithm is only trying to find the best way to predict how a viewer would rate a movie, not necessarily explain to us how it is done. This algorithm was part of the winning solution in the million-dollar competition.

1.1.1 Components of Learning

The movie rating application captures the essence of learning from data, and so do many other applications from vastly different fields. In order to abstract the common core of the learning problem, we will pick one application and use it as a metaphor for the different components of the problem. Let us take credit approval as our metaphor.

Suppose that a bank receives thousands of credit card applications every day, and it wants to automate the process of evaluating them. Just as in the case of movie ratings, the bank knows of no magical formula that can pinpoint when credit should be approved, but it has a lot of data. This calls for learning from data, so the bank uses historical records of previous customers to figure out a good formula for credit approval.

Each customer record has personal information related to credit, such as annual salary, years in residence, outstanding loans, etc. The record also keeps track of whether approving credit for that customer was a good idea, i.e., did the bank make money on that customer. This data guides the construction of a successful formula for credit approval that can be used on future applicants.

Let us give names and symbols to the main components of this learning problem. There is the input $\mathbf{x}$ (customer information that is used to make a credit decision), the unknown target function $f\colon \mathcal{X} \to \mathcal{Y}$ (ideal formula for credit approval), where $\mathcal{X}$ is the input space (set of all possible inputs $\mathbf{x}$), and $\mathcal{Y}$ is the output space (set of all possible outputs, in this case just a yes/no decision). There is a data set $\mathcal{D}$ of input-output examples $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)$, where $y_n = f(\mathbf{x}_n)$ for $n = 1, ..., N$ (inputs corresponding to previous customers and the correct credit decision for them in hindsight). The examples are often referred to as data points. Finally, there is the learning algorithm that uses the data set $\mathcal{D}$ to pick a formula $g\colon \mathcal{X} \to \mathcal{Y}$ that approximates f. The algorithm chooses g from a set of candidate formulas under consideration, which we call the hypothesis set $\mathcal{H}$. For instance, $\mathcal{H}$ could be the set of all linear formulas from which the algorithm would choose the best linear fit to the data, as we will introduce later in this section.

When a new customer applies for credit, the bank will base its decision on g (the hypothesis that the learning algorithm produced), not on f (the ideal target function which remains unknown). The decision will be good only to the extent that g faithfully replicates f. To achieve that, the algorithm

Figure 1.2: Basic setup of the learning problem

chooses g that best matches f on the *training* examples of previous customers, with the hope that it will continue to match f on new customers. Whether or not this hope is justified remains to be seen. Figure 1.2 illustrates the components of the learning problem.

Exercise 1.1

Express each of the following tasks in the framework of learning from data by specifying the input space $\mathcal{X}$, output space $\mathcal{Y}$, target function $f : \mathcal{X} \to \mathcal{Y}$, and the specifics of the data set that we will learn from.

(a) Medical diagnosis: A patient walks in with a medical history and some symptoms, and you want to identify the problem.

(b) Handwritten digit recognition (for example postal zip code recognition for mail sorting).

(c) Determining if an email is spam or not.

(d) Predicting how an electric load varies with price, temperature, and day of the week.

(e) A problem of interest to you for which there is no analytic solution, but you have data from which to construct an empirical solution.

We will use the setup in Figure 1.2 as our definition of the learning problem. Later on, we will consider a number of refinements and variations to this basic setup as needed. However, the essence of the problem will remain the same. There is a target to be learned. It is unknown to us. We have a set of examples generated by the target. The learning algorithm uses these examples to look for a hypothesis that approximates the target.

1.1.2 A Simple Learning Model

Let us consider the different components of Figure 1.2. Given a specific learning problem, the target function and training examples are dictated by the problem. However, the learning algorithm and hypothesis set are not. These are solution tools that we get to choose. The hypothesis set and learning algorithm are referred to informally as the *learning model.*

Here is a simple model. Let $\mathcal{X} = \mathbb{R}^d$ be the input space, where $\mathbb{R}^d$ is the d-dimensional Euclidean space, and let $\mathcal{Y} = \{+1, -1\}$ be the output space, denoting a binary (yes/no) decision. In our credit example, different coordinates of the input vector $\mathbf{x} \in \mathbb{R}^d$ correspond to salary, years in residence, outstanding debt, and the other data fields in a credit application. The binary output y corresponds to approving or denying credit. We specify the hypothesis set $\mathcal{H}$ through a functional form that all the hypotheses $h \in \mathcal{H}$ share. The functional form $h(\mathbf{x})$ that we choose here gives different weights to the different coordinates of $\mathbf{x}$, reflecting their relative importance in the credit decision. The weighted coordinates are then combined to form a 'credit score' and the result is compared to a threshold value. If the applicant passes the threshold, credit is approved; if not, credit is denied:

$$\begin{aligned} \text{Approve credit if} \quad & \sum_{i=1}^{d} w_i x_i > \text{threshold}, \\ \text{Deny credit if} \quad & \sum_{i=1}^{d} w_i x_i < \text{threshold}. \end{aligned}$$

This formula can be written more compactly as

$$h(\mathbf{x}) = \text{sign}\left(\left(\sum_{i=1}^{d} w_i x_i\right) + b\right), \tag{1.1}$$

where $x_1, \cdots, x_d$ are the components of the vector $\mathbf{x}$; $h(\mathbf{x}) = +1$ means 'approve credit' and $h(\mathbf{x}) = -1$ means 'deny credit'; $\text{sign}(s) = +1$ if $s > 0$ and $\text{sign}(s) = -1$ if $s < 0$.[1] The weights are $w_1, \cdots, w_d$, and the threshold is determined by the bias term b since in Equation (1.1), credit is approved if $\sum_{i=1}^{d} w_i x_i > -b$.

This model of $\mathcal{H}$ is called the *perceptron*, a name that it got in the context of artificial intelligence. The learning algorithm will search $\mathcal{H}$ by looking for

[1] The value of $\text{sign}(s)$ when s 0 is a simple technicality that we ignore for the moment.

(a) Misclassified data (b) Perfectly classified data

Figure 1.3: Perceptron classification of linearly separable data in a two dimensional input space (a) Some training examples will be misclassified (blue points in red region and vice versa) for certain values of the weight parameters which define the separating line. (b) A final hypothesis that classifies all training examples correctly. ($\circ$ is $+1$ and $\times$ is -1.)

weights and bias that perform well on the data set. Some of the weights $w_1, \cdots, w_d$ may end up being negative, corresponding to an adverse effect on credit approval. For instance, the weight of the 'outstanding debt' field should come out negative since more debt is not good for credit. The bias value b may end up being large or small, reflecting how lenient or stringent the bank should be in extending credit. The optimal choices of weights and bias define the final hypothesis $g \in \mathcal{H}$ that the algorithm produces.

Exercise 1.2

Suppose that we use a perceptron to detect spam messages. Let's say that each email message is represented by the frequency of occurrence of keywords, and the output is $+1$ if the message is considered spam.

(a) Can you think of some keywords that will end up with a large positive weight in the perceptron?

(b) How about keywords that will get a negative weight?

(c) What parameter in the perceptron directly affects how many borderline messages end up being classified as spam?

Figure 1.3 illustrates what a perceptron does in a two-dimensional case ($d = 2$). The plane is split by a line into two regions, the $+1$ decision region and the -1 decision region. Different values for the parameters w_1, w_2, b correspond to different lines $w_1x_1 + w_2x_2 + b = 0$. If the data set is *linearly separable*, there will be a choice for these parameters that classifies all the training examples correctly.

To simplify the notation of the perceptron formula, we will treat the bias b as a weight $w_0 = b$ and merge it with the other weights into one vector $\mathbf{w} = [w_0, w_1, \cdots, w_d]^{\mathrm{T}}$, where ^{T} denotes the transpose of a vector, so $\mathbf{w}$ is a column vector. We also treat $\mathbf{x}$ as a column vector and modify it to become $\mathbf{x} = [x_0, x_1, \cdots, x_d]^{\mathrm{T}}$, where the added coordinate x_0 is fixed at $x_0 = 1$. Formally speaking, the input space is now

$$\mathcal{X} = \{1\} \times \mathbb{R}^d = \{[x_0, x_1, \cdots, x_d]^{\mathrm{T}} \mid x_0 = 1, x_1 \in \mathbb{R}, \cdots, x_d \in \mathbb{R}\}.$$

With this convention, $\mathbf{w}^{\mathrm{T}}\mathbf{x} = \sum_{i=0}^{d} w_i x_i$, and so Equation (1.1) can be rewritten in vector form as

$$h(\mathbf{x}) = \mathrm{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}). \tag{1.2}$$

We now introduce the *perceptron learning algorithm* (PLA). The algorithm will determine what $\mathbf{w}$ should be, based on the data. Let us assume that the data set is linearly separable, which means that there is a vector $\mathbf{w}$ that makes (1.2) achieve the correct decision $h(\mathbf{x}_n) = y_n$ on all the training examples, as shown in Figure 1.3.

Our learning algorithm will find this $\mathbf{w}$ using a simple iterative method. Here is how it works. At iteration t, where $t = 0, 1, 2, \ldots$, there is a current value of the weight vector, call it $\mathbf{w}(t)$. The algorithm picks an example from $(\mathbf{x}_1, y_1) \cdots (\mathbf{x}_N, y_N)$ that is currently misclassified, call it $(\mathbf{x}(t), y(t))$, and uses it to update $\mathbf{w}(t)$. Since the example is misclassified, we have $y(t) \neq \mathrm{sign}(\mathbf{w}^{\mathrm{T}}(t)\mathbf{x}(t))$. The update rule is

$$\mathbf{w}(t+1) = \mathbf{w}(t) + y(t)\mathbf{x}(t). \tag{1.3}$$

This rule moves the boundary in the direction of classifying $\mathbf{x}(t)$ correctly, as depicted in the figure above. The algorithm continues with further iterations until there are no longer misclassified examples in the data set.

Exercise 1.3

The weight update rule in (1.3) has the nice interpretation that it moves in the direction of classifying $\mathbf{x}(t)$ correctly.

(a) Show that $y(t)\mathbf{w}^{\mathrm{T}}(t)\mathbf{x}(t) < 0$. *[Hint: $\mathbf{x}(t)$ is misclassified by $\mathbf{w}(t)$.]*

(b) Show that $y(t)\mathbf{w}^{\mathrm{T}}(t+1)\mathbf{x}(t) > y(t)\mathbf{w}^{\mathrm{T}}(t)\mathbf{x}(t)$. *[Hint: Use (1.3).]*

(c) As far as classifying $\mathbf{x}(t)$ is concerned, argue that the move from $\mathbf{w}(t)$ to $\mathbf{w}(t+1)$ is a move 'in the right direction'.

Although the update rule in (1.3) considers only one training example at a time and may 'mess up' the classification of the other examples that are not involved in the current iteration, it turns out that the algorithm is guaranteed to arrive at the right solution in the end. The proof is the subject of Problem 1.3. The result holds regardless of which example we choose from among the misclassified examples in $(\mathbf{x}_1, y_1) \cdots (\mathbf{x}_N, y_N)$ at each iteration, and regardless of how we initialize the weight vector to start the algorithm. For simplicity, we can pick one of the misclassified examples at random (or cycle through the examples and always choose the first misclassified one), and we can initialize $\mathbf{w}(0)$ to the zero vector.

Within the infinite space of all weight vectors, the perceptron algorithm manages to find a weight vector that works, using a simple iterative process. This illustrates how a learning algorithm can effectively search an infinite hypothesis set using a finite number of simple steps. This feature is characteristic of many techniques that are used in learning, some of which are far more sophisticated than the perceptron learning algorithm.

Exercise 1.4

Let us create our own target function f and data set $\mathcal{D}$ and see how the perceptron learning algorithm works. Take $d = 2$ so you can visualize the problem, and choose a random line in the plane as your target function, where one side of the line maps to $+1$ and the other maps to -1. Choose the inputs $\mathbf{x}_n$ of the data set as random points in the plane, and evaluate the target function on each $\mathbf{x}_n$ to get the corresponding output y_n.

Now, generate a data set of size 20. Try the perceptron learning algorithm on your data set and see how long it takes to converge and how well the final hypothesis g matches your target f. You can find other ways to play with this experiment in Problem 1.4.

The perceptron learning algorithm succeeds in achieving its goal; finding a hypothesis that classifies all the points in the data set $\mathcal{D} = \{(\mathbf{x}_1, y_1) \cdots (\mathbf{x}_N, y_N)\}$ correctly. Does this mean that this hypothesis will also be successful in classifying new data points that are not in $\mathcal{D}$? This turns out to be the key question in the theory of learning, a question that will be thoroughly examined in this book.

(a) Coin data (b) Learned classifier

Figure 1.4: The learning approach to coin classification (a) Training data of pennies, nickels, dimes, and quarters (1, 5, 10, and 25 cents) are represented in a size mass space where they fall into clusters. (b) A classification rule is learned from the data set by separating the four clusters. A new coin will be classified according to the region in the size mass plane that it falls into.

1.1.3 Learning versus Design

So far, we have discussed what learning is. Now, we discuss what it is not. The goal is to distinguish between learning and a related approach that is used for similar problems. While learning is based on data, this other approach does not use data. It is a 'design' approach based on specifications, and is often discussed alongside the learning approach in pattern recognition literature.

Consider the problem of recognizing coins of different denominations, which is relevant to vending machines, for example. We want the machine to recognize quarters, dimes, nickels and pennies. We will contrast the 'learning from data' approach and the 'design from specifications' approach for this problem. We assume that each coin will be represented by its size and mass, a two-dimensional input.

In the learning approach, we are given a sample of coins from each of the four denominations and we use these coins as our data set. We treat the size and mass as the input vector, and the denomination as the output. Figure 1.4(a) shows what the data set may look like in the input space. There is some variation of size and mass within each class, but by and large coins of the same denomination cluster together. The learning algorithm searches for a hypothesis that classifies the data set well. If we want to classify a new coin, the machine measures its size and mass, and then classifies it according to the learned hypothesis in Figure 1.4(b).

In the design approach, we call the United States Mint and ask them about the specifications of different coins. We also ask them about the number

(a) Probabilistic model of data (b) Inferred classifier

Figure 1.5: The design approach to coin classification (a) A probabilistic model for the size, mass, and denomination of coins is derived from known specifications. The figure shows the high probability region for each denom ination (1, 5, 10, and 25 cents) according to the model. (b) A classification rule is derived analytically to minimize the probability of error in classifying a coin based on size and mass. The resulting regions for each denomination are shown.

of coins of each denomination in circulation, in order to get an estimate of the relative frequency of each coin. Finally, we make a physical model of the variations in size and mass due to exposure to the elements and due to errors in measurement. We put all of this information together and compute the full joint probability distribution of size, mass, and coin denomination (Figure 1.5(a)). Once we have that joint distribution, we can construct the optimal decision rule to classify coins based on size and mass (Figure 1.5(b)). The rule chooses the denomination that has the highest probability for a given size and mass, thus achieving the smallest possible probability of error.[2]

The main difference between the learning approach and the design approach is the role that data plays. In the design approach, the problem is well specified and one can analytically derive f without the need to see any data. In the learning approach, the problem is much less specified, and one needs data to pin down what f is.

Both approaches may be viable in some applications, but only the learning approach is possible in many applications where the target function is unknown. We are not trying to compare the utility or the performance of the two approaches. We are just making the point that the design approach is distinct from learning. This book is about learning.

[2]This is called Bayes optimal decision theory. Some learning models are based on the same theory by estimating the probability from data.

Exercise 1.5

Which of the following problems are more suited for the learning approach and which are more suited for the design approach?

(a) Determining the age at which a particular medical test should be performed

(b) Classifying numbers into primes and non-primes

(c) Detecting potential fraud in credit card charges

(d) Determining the time it would take a falling object to hit the ground

(e) Determining the optimal cycle for traffic lights in a busy intersection

1.2 Types of Learning

The basic premise of learning from data is the use of a set of observations to uncover an underlying process. It is a very broad premise, and difficult to fit into a single framework. As a result, different learning paradigms have arisen to deal with different situations and different assumptions. In this section, we introduce some of these paradigms.

The learning paradigm that we have discussed so far is called *supervised* learning. It is the most studied and most utilized type of learning, but it is not the only one. Some variations of supervised learning are simple enough to be accommodated within the same framework. Other variations are more profound and lead to new concepts and techniques that take on lives of their own. The most important variations have to do with the nature of the data set.

1.2.1 Supervised Learning

When the training data contains explicit examples of what the correct output should be for given inputs, then we are within the supervised learning setting that we have covered so far. Consider the hand-written digit recognition problem (task (b) of Exercise 1.1). A reasonable data set for this problem is a collection of images of hand-written digits, and for each image, what the digit actually is. We thus have a set of examples of the form (image , digit). The learning is supervised in the sense that some 'supervisor' has taken the trouble to look at each input, in this case an image, and determine the correct output, in this case one of the ten categories $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

While we are on the subject of variations, there is more than one way that a data set can be presented to the learning process. Data sets are typically created and presented to us in their entirety at the outset of the learning process. For instance, historical records of customers in the credit-card application, and previous movie ratings of customers in the movie rating application, are already there for us to use. This protocol of a 'ready' data set is the most

common in practice, and it is what we will focus on in this book. However, it is worth noting that two variations of this protocol have attracted a significant body of work.

One is *active learning*, where the data set is acquired through queries that we make. Thus, we get to choose a point $\mathbf{x}$ in the input space, and the supervisor reports to us the target value for $\mathbf{x}$. As you can see, this opens the possibility for strategic choice of the point $\mathbf{x}$ to maximize its information value, similar to asking a strategic question in a game of 20 questions.

Another variation is called *online learning*, where the data set is given to the algorithm one example at a time. This happens when we have streaming data that the algorithm has to process 'on the run'. For instance, when the movie recommendation system discussed in Section 1.1 is deployed, online learning can process new ratings from current users and movies. Online learning is also useful when we have limitations on computing and storage that preclude us from processing the whole data as a batch. We should note that online learning can be used in different paradigms of learning, not just in supervised learning.

1.2.2 Reinforcement Learning

When the training data does not explicitly contain the correct output for each input, we are no longer in a supervised learning setting. Consider a toddler learning not to touch a hot cup of tea. The experience of such a toddler would typically comprise a set of occasions when the toddler confronted a hot cup of tea and was faced with the decision of touching it or not touching it. Presumably, every time she touched it, the result was a high level of pain, and every time she didn't touch it, a much lower level of pain resulted (that of an unsatisfied curiosity). Eventually, the toddler learns that she is better off not touching the hot cup.

The training examples did not spell out what the toddler should have done, but they instead graded different actions that she has taken. Nevertheless, she uses the examples to reinforce the better actions, eventually learning what she should do in similar situations. This characterizes *reinforcement* learning, where the training example does not contain the target output, but instead contains some possible output together with a measure of how good that output is. In contrast to supervised learning where the training examples were of the form (input , correct output), the examples in reinforcement learning are of the form

$$(\text{ input , some output , grade for this output }).$$

Importantly, the example does not say how good other outputs would have been for this particular input.

Reinforcement learning is especially useful for learning how to play a game. Imagine a situation in backgammon where you have a choice between different actions and you want to identify the best action. It is not a trivial task to ascertain what the best action is at a given stage of the game, so we cannot

(a) Unlabeled Coin data (b) Unsupervised learning

Figure 1.6: Unsupervised learning of coin classification (a) The same data set of coins in Figure 1.4(a) is again represented in the size mass space, but without being labeled. They still fall into clusters. (b) An unsupervised classification rule treats the four clusters as different types. The rule may be somewhat ambiguous, as type 1 and type 2 could be viewed as one cluster

easily create supervised learning examples. If you use reinforcement learning instead, all you need to do is to take some action and report how well things went, and you have a training example. The reinforcement learning algorithm is left with the task of sorting out the information coming from different examples to find the best line of play.

1.2.3 Unsupervised Learning

In the unsupervised setting, the training data does not contain any output information at all. We are just given input examples $\mathbf{x}_1, \cdots, \mathbf{x}_N$. You may wonder how we could possibly learn anything from mere inputs. Consider the coin classification problem that we discussed earlier in Figure 1.4. Suppose that we didn't know the denomination of any of the coins in the data set. This *unlabeled data* is shown in Figure 1.6(a). We still get similar clusters, but they are now unlabeled so all points have the same 'color'. The decision regions in unsupervised learning may be identical to those in supervised learning, but without the labels (Figure 1.6(b)). However, the correct clustering is less obvious now, and even the number of clusters may be ambiguous.

Nonetheless, this example shows that we can learn *something* from the inputs by themselves. Unsupervised learning can be viewed as the task of spontaneously finding patterns and structure in input data. For instance, if our task is to categorize a set of books into topics, and we only use general properties of the various books, we can identify books that have similar properties and put them together in one category, without naming that category.

Unsupervised learning can also be viewed as a way to create a higher-level representation of the data. Imagine that you don't speak a word of Spanish, but your company will relocate you to Spain next month. They will arrange for Spanish lessons once you are there, but you would like to prepare yourself a bit before you go. All you have access to is a Spanish radio station. For a full month, you continuously bombard yourself with Spanish; this is an unsupervised learning experience since you don't know the meaning of the words. However, you gradually develop a better representation of the language in your brain by becoming more tuned to its common sounds and structures. When you arrive in Spain, you will be in a better position to start your Spanish lessons. Indeed, unsupervised learning can be a precursor to supervised learning. In other cases, it is a stand-alone technique.

Exercise 1.6

For each of the following tasks, identify which type of learning is involved (supervised, reinforcement, or unsupervised) and the training data to be used. If a task can fit more than one type, explain how and describe the training data for each type.

(a) Recommending a book to a user in an online bookstore

(b) Playing tic tac toe

(c) Categorizing movies into different types

(d) Learning to play music

(e) Credit limit: Deciding the maximum allowed debt for each bank customer

Our main focus in this book will be supervised learning, which is the most popular form of learning from data.

1.2.4 Other Views of Learning

The study of learning has evolved somewhat independently in a number of fields that started historically at different times and in different domains, and these fields have developed different emphases and even different jargons. As a result, learning from data is a diverse subject with many aliases in the scientific literature. The main field dedicated to the subject is called *machine learning*, a name that distinguishes it from human learning. We briefly mention two other important fields that approach learning from data in their own ways.

Statistics shares the basic premise of learning from data, namely the use of a set of observations to uncover an underlying process. In this case, the process is a probability distribution and the observations are samples from that distribution. Because statistics is a mathematical field, emphasis is given to situations where most of the questions can be answered with rigorous proofs. As a result, statistics focuses on somewhat idealized models and analyzes them in great detail. This is the main difference between the statistical approach

Figure 1.7: A visual learning problem. The first two rows show the training examples (each input $\mathbf{x}$ is a 9 bit vector represented visually as a 3×3 black and white array). The inputs in the first row have $f(\mathbf{x}) = -1$, and the inputs in the second row have $f(\mathbf{x}) = +1$. Your task is to learn from this data set what f is, then apply f to the test input at the bottom. Do you get -1 or $+1$?

to learning and how we approach the subject here. We make less restrictive assumptions and deal with more general models than in statistics. Therefore, we end up with weaker results that are nonetheless broadly applicable.

Data mining is a practical field that focuses on finding patterns, correlations, or anomalies in large relational databases. For example, we could be looking at medical records of patients and trying to detect a cause-effect relationship between a particular drug and long-term effects. We could also be looking at credit card spending patterns and trying to detect potential fraud. Technically, data mining is the same as learning from data, with more emphasis on data analysis than on prediction. Because databases are usually huge, computational issues are often critical in data mining. Recommender systems, which were illustrated in Section 1.1 with the movie rating example, are also considered part of data mining.

1.3 Is Learning Feasible?

The target function f is the object of learning. The most important assertion about the target function is that it is *unknown*. We really mean unknown.

This raises a natural question. How could a limited data set reveal enough information to pin down the entire target function? Figure 1.7 illustrates this

difficulty. A simple learning task with 6 training examples of a ± 1 target function is shown. Try to learn what the function is then apply it to the test input given. Do you get -1 or $+1$? Now, show the problem to your friends and see if they get the same answer.

The chances are the answers were not unanimous, and for good reason. There is simply more than one function that fits the 6 training examples, and some of these functions have a value of -1 on the test point and others have a value of $+1$. For instance, if the true f is $+1$ when the pattern is symmetric, the value for the test point would be $+1$. If the true f is $+1$ when the top left square of the pattern is white, the value for the test point would be -1. Both functions agree with all the examples in the data set, so there isn't enough information to tell us which would be the correct answer.

This does not bode well for the feasibility of learning. To make matters worse, we will now see that the difficulty we experienced in this simple problem is the rule, not the exception.

1.3.1 Outside the Data Set

When we get the training data $\mathcal{D}$, e.g., the first two rows of Figure 1.7, we know the value of f on all the points in $\mathcal{D}$. This doesn't mean that we have learned f, since it doesn't guarantee that we know anything about f outside of $\mathcal{D}$. We know what we have already seen, but that's not learning. That's memorizing.

Does the data set $\mathcal{D}$ tell us anything outside of $\mathcal{D}$ that we didn't know before? If the answer is yes, then we have learned *something*. If the answer is no, we can conclude that learning is not feasible.

Since we maintain that f is an unknown function, we can prove that f remains unknown outside of $\mathcal{D}$. Instead of going through a formal proof for the general case, we will illustrate the idea in a concrete case. Consider a Boolean target function over a three-dimensional input space $\mathcal{X} = \{0,1\}^3$. We are given a data set $\mathcal{D}$ of five examples represented in the table below. We denote the binary output by $\circ/\bullet$ for visual clarity,

$\mathbf{x}_n$	y_n
0 0 0	$\circ$
0 0 1	$\bullet$
0 1 0	$\bullet$
0 1 1	$\circ$
1 0 0	$\bullet$

where $y_n = f(\mathbf{x}_n)$ for $n = 1, 2, 3, 4, 5$. The advantage of this simple Boolean case is that we can enumerate the entire input space (since there are only $2^3 = 8$ distinct input vectors), and we can enumerate the set of all possible target functions (since f is a Boolean function on 3 Boolean inputs, and there are only $2^{2^3} = 256$ distinct Boolean functions on 3 Boolean inputs).

Let us look at the problem of learning f. Since f is unknown except inside $\mathcal{D}$, any function that agrees with $\mathcal{D}$ could conceivably be f. The table below shows all such functions $f_1, \cdots, f_8$. It also shows the data set $\mathcal{D}$ (in blue) and what the final hypothesis g may look like.

$\mathbf{x}$	y	g	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
0 0 0	○	○	○	○	○	○	○	○	○	○
0 0 1	●	●	●	●	●	●	●	●	●	●
0 1 0	●	●	●	●	●	●	●	●	●	●
0 1 1	○	○	○	○	○	○	○	○	○	○
1 0 0	●	●	●	●	●	●	●	●	●	●
1 0 1		?	○	○	○	○	●	●	●	●
1 1 0		?	○	○	●	●	○	○	●	●
1 1 1		?	○	●	○	●	○	●	○	●

The final hypothesis g is chosen based on the five examples in $\mathcal{D}$. The table shows the case where g is chosen to match f on these examples.

If we remain true to the notion of unknown target, we cannot exclude any of $f_1, \cdots, f_8$ from being the true f. Now, we have a dilemma. The whole purpose of learning f is to be able to predict the value of f on points that we haven't seen before. The quality of the learning will be determined by how close our prediction is to the true value. Regardless of what g predicts on the three points we haven't seen before (those outside of $\mathcal{D}$, denoted by red question marks), it can agree or disagree with the target, depending on which of $f_1, \cdots, f_8$ turns out to be the true target. It is easy to verify that any 3 bits that replace the red question marks are as good as any other 3 bits.

Exercise 1.7

For each of the following learning scenarios in the above problem, evaluate the performance of g on the three points in $\mathcal{X}$ outside $\mathcal{D}$. To measure the performance, compute how many of the 8 possible target functions agree with g on all three points, on two of them, on one of them, and on none of them.

(a) $\mathcal{H}$ has only two hypotheses, one that always returns '●' and one that always returns '○'. The learning algorithm picks the hypothesis that matches the data set the most.

(b) The same $\mathcal{H}$, but the learning algorithm now picks the hypothesis that matches the data set the *least*.

(c) $\mathcal{H} = \{\text{XOR}\}$ (only one hypothesis which is always picked), where XOR is defined by $\text{XOR}(\mathbf{x}) = \bullet$ if the number of 1's in $\mathbf{x}$ is odd and $\text{XOR}(\mathbf{x}) = \circ$ if the number is even.

(d) $\mathcal{H}$ contains all possible hypotheses (all Boolean functions on three variables), and the learning algorithm picks the hypothesis that agrees with all training examples, but otherwise disagrees the most with the XOR.

Figure 1.8: A random sample is picked from a bin of red and green marbles. The probability μ of red marbles in the bin is unknown. What does the fraction ν of red marbles in the sample tell us about μ?

It doesn't matter what the algorithm does or what hypothesis set $\mathcal{H}$ is used. Whether $\mathcal{H}$ has a hypothesis that perfectly agrees with $\mathcal{D}$ (as depicted in the table) or not, and whether the learning algorithm picks that hypothesis or picks another one that disagrees with $\mathcal{D}$ (different green bits), it makes no difference whatsoever as far as the performance outside of $\mathcal{D}$ is concerned. Yet the performance outside $\mathcal{D}$ is all that matters in learning!

This dilemma is not restricted to Boolean functions, but extends to the general learning problem. As long as f is an unknown function, knowing $\mathcal{D}$ cannot exclude any pattern of values for f outside of $\mathcal{D}$. Therefore, the predictions of g outside of $\mathcal{D}$ are meaningless.

Does this mean that learning from data is doomed? If so, this will be a very short book ☺. Fortunately, learning is alive and well, and we will see why. We won't have to change our basic assumption to do that. The target function will continue to be unknown, and we still mean *unknown*.

1.3.2 Probability to the Rescue

We will show that we can indeed infer something outside $\mathcal{D}$ using only $\mathcal{D}$, but in a probabilistic way. What we infer may not be much compared to learning a full target function, but it will establish the principle that we can reach outside $\mathcal{D}$. Once we establish that, we will take it to the general learning problem and pin down what we can and cannot learn.

Let's take the simplest case of picking a sample, and see when we can say something about the objects outside the sample. Consider a bin that contains red and green marbles, possibly infinitely many. The proportion of red and green marbles in the bin is such that if we pick a marble at random, the probability that it will be red is μ and the probability that it will be green is $1 - \mu$. We assume that the value of μ is unknown to us.

We pick a random sample of N independent marbles (with replacement) from this bin, and observe the fraction ν of red marbles within the sample (Figure 1.8). What does the value of ν tell us about the value of μ?

One answer is that regardless of the colors of the N marbles that we picked, we still don't know the color of any marble that we didn't pick. We can get mostly green marbles in the sample while the bin has mostly red marbles. Although this is certainly *possible*, it is by no means *probable*.

Exercise 1.8

If $\mu = 0.9$, what is the probability that a sample of 10 marbles will have $\nu \leq 0.1$? *[Hints: 1. Use binomial distribution. 2. The answer is a very small number.]*

The situation is similar to taking a poll. A random sample from a population tends to agree with the views of the population at large. The probability distribution of the random variable ν in terms of the parameter μ is well understood, and when the sample size is big, ν tends to be close to μ.

To quantify the relationship between ν and μ, we use a simple bound called the *Hoeffding Inequality*. It states that for any sample size N,

$$\mathbb{P}\left[|\nu - \mu| > \epsilon\right] \leq 2e^{-2\epsilon^2 N} \qquad \text{for any } \epsilon > 0. \tag{1.4}$$

Here, $\mathbb{P}[\cdot]$ denotes the probability of an event, in this case with respect to the random sample we pick, and ϵ is any positive value we choose. Putting Inequality (1.4) in words, it says that as the sample size N grows, it becomes exponentially unlikely that ν will deviate from μ by more than our 'tolerance' ϵ.

The only quantity that is random in (1.4) is ν which depends on the random sample. By contrast, μ is not random. It is just a constant, albeit unknown to us. There is a subtle point here. The utility of (1.4) is to infer the value of μ using the value of ν, although it is μ that affects ν, not vice versa. However, since the effect is that ν tends to be close to μ, we infer that μ 'tends' to be close to ν.

Although $\mathbb{P}\left[|\nu \quad \mu| > \epsilon\right]$ depends on μ, as μ appears in the argument and also affects the distribution of ν, we are able to bound the probability by $2e^{-2\epsilon^2 N}$ which does not depend on μ. Notice that only the size N of the sample affects the bound, not the size of the bin. The bin can be large or small, finite or infinite, and we still get the same bound when we use the same sample size.

Exercise 1.9

If $\mu = 0.9$, use the Hoeffding Inequality to bound the probability that a sample of 10 marbles will have $\nu \leq 0.1$ and compare the answer to the previous exercise.

If we choose ϵ to be very small in order to make ν a good approximation of μ, we need a larger sample size N to make the RHS of Inequality (1.4) small. We

can then assert that it is likely that ν will indeed be a good approximation of μ. Although this assertion does not give us the exact value of μ, and doesn't even guarantee that the approximate value holds, knowing that we are within $\pm\epsilon$ of μ most of the time is a significant improvement over not knowing anything at all.

The fact that the sample was randomly selected from the bin is the reason we are able to make any kind of statement about μ being close to ν. If the sample was not randomly selected but picked in a particular way, we would lose the benefit of the probabilistic analysis and we would again be in the dark outside of the sample.

How does the bin model relate to the learning problem? It seems that the unknown here was just the value of μ while the unknown in learning is an entire function $f\colon \mathcal{X} \to \mathcal{Y}$. The two situations can be connected. Take any single hypothesis $h \in \mathcal{H}$ and compare it to f on each point $\mathbf{x} \in \mathcal{X}$. If $h(\mathbf{x}) = f(\mathbf{x})$, color the point $\mathbf{x}$ green. If $h(\mathbf{x}) \neq f(\mathbf{x})$, color the point $\mathbf{x}$ red. The color that each point gets is not known to us, since f is unknown. However, if we pick $\mathbf{x}$ at random according to some probability distribution P over the input space $\mathcal{X}$, we know that $\mathbf{x}$ will be red with some probability, call it μ, and green with probability $1-\mu$. Regardless of the value of μ, the space $\mathcal{X}$ now behaves like the bin in Figure 1.8.

The training examples play the role of a sample from the bin. If the inputs $\mathbf{x}_1, \cdots, \mathbf{x}_N$ in $\mathcal{D}$ are picked independently according to P, we will get a random sample of red ($h(\mathbf{x}_n) \neq f(\mathbf{x}_n)$) and green ($h(\mathbf{x}_n) = f(\mathbf{x}_n)$) points. Each point will be red with probability μ and green with probability $1-\mu$. The color of each point will be known to us since both $h(\mathbf{x}_n)$ and $f(\mathbf{x}_n)$ are known for $n = 1, \cdots, N$ (the function h is our hypothesis so we can evaluate it on any point, and $f(\mathbf{x}_n) = y_n$ is given to us for all points in the data set $\mathcal{D}$). The learning problem is now reduced to a bin problem, under the assumption that the inputs in $\mathcal{D}$ are picked independently according to some distribution P on $\mathcal{X}$. Any P will translate to some μ in the equivalent bin. Since μ is allowed to be unknown, P can be unknown to us as well. Figure 1.9 adds this probabilistic component to the basic learning setup depicted in Figure 1.2.

With this equivalence, the Hoeffding Inequality can be applied to the learning problem, allowing us to make a prediction outside of $\mathcal{D}$. Using ν to predict μ tells us something about f, although it doesn't tell us what f is. What μ tells us is the error rate h makes in approximating f. If ν happens to be close to zero, we can predict that h will approximate f well over the entire input space. If not, we are out of luck.

Unfortunately, we have no control over ν in our current situation, since ν is based on a particular hypothesis h. In real learning, we explore an entire hypothesis set $\mathcal{H}$, looking for some $h \in \mathcal{H}$ that has a small error rate. If we have only one hypothesis to begin with, we are not really learning, but rather 'verifying' whether that particular hypothesis is good or bad. Let us see if we can extend the bin equivalence to the case where we have multiple hypotheses in order to capture real learning.

Figure 1.9: Probability added to the basic learning setup

To do that, we start by introducing more descriptive names for the different components that we will use. The error rate within the sample, which corresponds to ν in the bin model, will be called the *in-sample error*,

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

where $[\![\text{statement}]\!] = 1$ if the statement is true, and $= 0$ if the statement is false. We have made explicit the dependency of E_{in} on the particular h that we are considering. In the same way, we define the *out-of-sample error*

$$E_{\text{out}}(h) = \mathbb{P}\left[h(\mathbf{x}) \neq f(\mathbf{x}) \right] ,$$

which corresponds to μ in the bin model. The probability is based on the distribution P over $\mathcal{X}$ which is used to sample the data points $\mathbf{x}$.

Figure 1.10: Multiple bins depict the learning problem with M hypotheses

Substituting the new notation E_{in} for ν and E_{out} for μ, the Hoeffding Inequality (1.4) can be rewritten as

$$\mathbb{P}\left[|E_{\text{in}}(h) - E_{\text{out}}(h)| > \epsilon\right] \leq 2e^{-2\epsilon^2 N} \qquad \text{for any } \epsilon > 0, \tag{1.5}$$

where N is the number of training examples. The in-sample error E_{in}, just like ν, is a random variable that depends on the sample. The out-of-sample error E_{out}, just like μ, is unknown but not random.

Let us consider an entire hypothesis set $\mathcal{H}$ instead of just one hypothesis h, and assume for the moment that $\mathcal{H}$ has a finite number of hypotheses

$$\mathcal{H} = \{h_1, h_2, \cdots, h_M\}.$$

We can construct a bin equivalent in this case by having M bins as shown in Figure 1.10. Each bin still represents the input space $\mathcal{X}$, with the red marbles in the mth bin corresponding to the points $\mathbf{x} \in \mathcal{X}$ where $h_m(\mathbf{x}) \neq f(\mathbf{x})$. The probability of red marbles in the mth bin is $E_{\text{out}}(h_m)$ and the fraction of red marbles in the mth sample is $E_{\text{in}}(h_m)$, for $m = 1, \cdots, M$. Although the Hoeffding Inequality (1.5) still applies to each bin individually, the situation becomes more complicated when we consider all the bins simultaneously. Why is that? The inequality stated that

$$\mathbb{P}\left[|E_{\text{in}}(h) - E_{\text{out}}(h)| > \epsilon\right] \leq 2e^{-2\epsilon^2 N} \qquad \text{for any } \epsilon > 0,$$

where the hypothesis h is *fixed* before you generate the data set, and the probability is with respect to random data sets $\mathcal{D}$; we emphasize that the assumption "h is fixed *before* you generate the data set" is critical to the validity of this bound. If you are allowed to change h after you generate the data set, the assumptions that are needed to prove the Hoeffding Inequality no longer hold. With multiple hypotheses in $\mathcal{H}$, the learning algorithm picks

the final hypothesis g based on $\mathcal{D}$, i.e. *after* generating the data set. The statement we would like to make is not

$$\text{“}\mathbb{P}[|E_{\text{in}}(h_m) - E_{\text{out}}(h_m)| > \epsilon] \text{ is small”}$$

(for any particular, fixed $h_m \in \mathcal{H}$), but rather

$$\text{“}\mathbb{P}[|E_{\text{in}}(g) - E_{\text{out}}(g)| > \epsilon] \text{ is small” for the final hypothesis } g.$$

The hypothesis g is *not fixed* ahead of time before generating the data, because which hypothesis is selected to be g depends on the data. So, we cannot just plug in g for h in the Hoeffding inequality. The next exercise considers a simple coin experiment that further illustrates the difference between a fixed h and the final hypothesis g selected by the learning algorithm.

Exercise 1.10

Here is an experiment that illustrates the difference between a single bin and multiple bins. Run a computer simulation for flipping 1,000 fair coins. Flip each coin independently 10 times. Let's focus on 3 coins as follows: c_1 is the first coin flipped; c_{rand} is a coin you choose at random; $c_{\min}$ is the coin that had the minimum frequency of heads (pick the earlier one in case of a tie). Let ν_1, ν_{rand} and $\nu_{\min}$ be the fraction of heads you obtain for the respective three coins.

(a) What is μ for the three coins selected?

(b) Repeat this entire experiment a large number of times (e.g., 100,000 runs of the entire experiment) to get several instances of ν_1, ν_{rand} and $\nu_{\min}$ and plot the histograms of the distributions of ν_1, ν_{rand} and $\nu_{\min}$. Notice that which coins end up being c_{rand} and $c_{\min}$ may differ from one run to another.

(c) Using (b), plot estimates for $\mathbb{P}[|\nu - \mu| > \epsilon]$ as a function of ϵ, together with the Hoeffding bound $2e^{-2\epsilon^2 N}$ (on the same graph).

(d) Which coins obey the Hoeffding bound, and which ones do not? Explain why.

(e) Relate part (d) to the multiple bins in Figure 1.10.

The way to get around this is to try to bound $\mathbb{P}[|E_{\text{in}}(g) - E_{\text{out}}(g)| > \epsilon]$ in a way that does not depend on which g the learning algorithm picks. There is a simple but crude way of doing that. Since g has to be one of the h_m's regardless of the algorithm and the sample, it is always true that

$$\begin{aligned} \text{“}|E_{\text{in}}(g) - E_{\text{out}}(g)| > \epsilon\text{”} \quad \Longrightarrow \quad \text{“} \quad & |E_{\text{in}}(h_1) - E_{\text{out}}(h_1)| > \epsilon \\ \textbf{or } & |E_{\text{in}}(h_2) - E_{\text{out}}(h_2)| > \epsilon \\ & \cdots \\ \textbf{or } & |E_{\text{in}}(h_M) - E_{\text{out}}(h_M)| > \epsilon \text{”}. \end{aligned}$$

where $\mathcal{B}_1 \Longrightarrow \mathcal{B}_2$ means that event $\mathcal{B}_1$ implies event $\mathcal{B}_2$. Although the events on the RHS cover a lot more than the LHS, the RHS has the property we want; the hypotheses h_m are fixed. We now apply two basic rules in probability;

$$\text{if } \mathcal{B}_1 \Longrightarrow \mathcal{B}_2, \text{ then } \mathbb{P}\left[\mathcal{B}_1\right] \leq \mathbb{P}[\mathcal{B}_2],$$

and, if $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_M$ are any events, then

$$\mathbb{P}[\mathcal{B}_1 \textbf{ or } \mathcal{B}_2 \textbf{ or } \cdots \textbf{ or } \mathcal{B}_M] \leq \mathbb{P}[\mathcal{B}_1] + \mathbb{P}[\mathcal{B}_2] + \cdots + \mathbb{P}[\mathcal{B}_M].$$

The second rule is known as the *union bound.* Putting the two rules together, we get

$$\begin{aligned}
\mathbb{P}[\ |E_{\text{in}}(g) - E_{\text{out}}(g)| > \epsilon\ ] \quad \leq \quad & \mathbb{P}[\quad |E_{\text{in}}(h_1) - E_{\text{out}}(h_1)| > \epsilon \\
& \quad \textbf{or } |E_{\text{in}}(h_2) - E_{\text{out}}(h_2)| > \epsilon \\
& \quad \cdots \\
& \quad \textbf{or } |E_{\text{in}}(h_M) - E_{\text{out}}(h_M)| > \epsilon\ ] \\
\leq \quad & \sum_{m=1}^{M} \mathbb{P}\left[|E_{\text{in}}(h_m) \quad E_{\text{out}}(h_m)| > \epsilon\right].
\end{aligned}$$

Applying the Hoeffding Inequality (1.5) to the M terms one at a time, we can bound each term in the sum by $2e^{-2\epsilon^2 N}$. Substituting, we get

$$\mathbb{P}[|E_{\text{in}}(g) - E_{\text{out}}(g)| > \epsilon] \leq 2Me^{-2\epsilon^2 N}. \tag{1.6}$$

Mathematically, this is a 'uniform' version of (1.5). We are trying to simultaneously approximate all $E_{\text{out}}(h_m)$'s by the corresponding $E_{\text{in}}(h_m)$'s. This allows the learning algorithm to choose any hypothesis based on E_{in} and expect that the corresponding E_{out} will uniformly follow suit, regardless of which hypothesis is chosen.

The downside for uniform estimates is that the probability bound $2Me^{-2\epsilon^2 N}$ is a factor of M looser than the bound for a single hypothesis, and will only be meaningful if M is finite. We will improve on that in Chapter 2.

1.3.3 Feasibility of Learning

We have introduced two apparently conflicting arguments about the feasibility of learning. One argument says that we cannot learn anything outside of $\mathcal{D}$, and the other says that we can. We would like to reconcile these two arguments and pinpoint the sense in which learning is feasible:

1. Let us reconcile the two arguments. The question of whether $\mathcal{D}$ tells us anything outside of $\mathcal{D}$ that we didn't know before has two different answers. If we insist on a deterministic answer, which means that $\mathcal{D}$ tells us something certain about f outside of $\mathcal{D}$, then the answer is no. If we accept a probabilistic answer, which means that $\mathcal{D}$ tells us something likely about f outside of $\mathcal{D}$, then the answer is yes.

Exercise 1.11

We are given a data set $\mathcal{D}$ of 25 training examples from an unknown target function $f\colon \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \{-1, +1\}$. To learn f, we use a simple hypothesis set $\mathcal{H} = \{h_1, h_2\}$ where h_1 is the constant $+1$ function and h_2 is the constant -1.

We consider two learning algorithms, S (smart) and C (crazy). S chooses the hypothesis that agrees the most with $\mathcal{D}$ and C chooses the other hypothesis deliberately. Let us see how these algorithms perform out of sample from the deterministic and probabilistic points of view. Assume in the probabilistic view that there is a probability distribution on $\mathcal{X}$, and let $\mathbb{P}[f(\mathbf{x}) = +1] = p$.

(a) Can S produce a hypothesis that is *guaranteed* to perform better than random on any point outside $\mathcal{D}$?

(b) Assume for the rest of the exercise that all the examples in $\mathcal{D}$ have $y_n = +1$. Is it *possible* that the hypothesis that C produces turns out to be better than the hypothesis that S produces?

(c) If $p = 0.9$, what is the probability that S will produce a better hypothesis than C?

(d) Is there any value of p for which it is more likely than not that C will produce a better hypothesis than S?

By adopting the probabilistic view, we get a positive answer to the feasibility question without paying too much of a price. The only assumption we make in the probabilistic framework is that the examples in $\mathcal{D}$ are generated independently. We don't insist on using any particular probability distribution, or even on knowing what distribution is used. However, whatever distribution we use for generating the examples, we must also use when we evaluate how well g approximates f (Figure 1.9). That's what makes the Hoeffding Inequality applicable. Of course this ideal situation may not always happen in practice, and some variations of it have been explored in the literature.

2. Let us pin down what we mean by the feasibility of learning. Learning produces a hypothesis g to approximate the unknown target function f. If learning is successful, then g should approximate f well, which means $E_{\text{out}}(g) \approx 0$. However, this is not what we get from the probabilistic analysis. What we get instead is $E_{\text{out}}(g) \approx E_{\text{in}}(g)$. We still have to make $E_{\text{in}}(g) \approx 0$ in order to conclude that $E_{\text{out}}(g) \approx 0$.

We cannot guarantee that we will find a hypothesis that achieves $E_{\text{in}}(g) \approx 0$, but at least we will know if we find it. Remember that $E_{\text{out}}(g)$ is an unknown quantity, since f is unknown, but $E_{\text{in}}(g)$ is a quantity that we can evaluate. We have thus traded the condition $E_{\text{out}}(g) \approx 0$, one that we cannot ascertain, for the condition $E_{\text{in}}(g) \approx 0$, which we can ascertain. What enabled this is the Hoeffding Inequality (1.6):

$$\mathbb{P}[|E_{\text{in}}(g) \quad E_{\text{out}}(g)| > \epsilon] \leq 2Me^{\ 2\epsilon^2 N}$$

that assures us that $E_{\text{out}}(g) \approx E_{\text{in}}(g)$ so we can use E_{in} as a proxy for E_{out}.

Exercise 1.12

A friend comes to you with a learning problem. She says the target function f is *completely* unknown, but she has $4,000$ data points. She is willing to pay you to solve her problem and produce for her a g which approximates f. What is the best that you can promise her among the following:

(a) After learning you will provide her with a g that you will guarantee approximates f well out of sample.

(b) After learning you will provide her with a g, and with high probability the g which you produce will approximate f well out of sample.

(c) One of two things will happen.

(i) You will produce a hypothesis g;

(ii) You will declare that you failed.

If you do return a hypothesis g, then with high probability the g which you produce will approximate f well out of sample.

One should note that there are cases where we won't insist that $E_{\text{in}}(g) \approx 0$. Financial forecasting is an example where market unpredictability makes it impossible to get a forecast that has anywhere near zero error. All we hope for is a forecast that gets it right more often than not. If we get that, our bets will win in the long run. This means that a hypothesis that has $E_{\text{in}}(g)$ somewhat below 0.5 will work, provided of course that $E_{\text{out}}(g)$ is close enough to $E_{\text{in}}(g)$.

The feasibility of learning is thus split into two questions:

1. Can we make sure that $E_{\text{out}}(g)$ is close enough to $E_{\text{in}}(g)$?
2. Can we make $E_{\text{in}}(g)$ small enough?

The Hoeffding Inequality (1.6) addresses the first question only. The second question is answered after we run the learning algorithm on the actual data and see how small we can get E_{in} to be.

Breaking down the feasibility of learning into these two questions provides further insight into the role that different components of the learning problem play. One such insight has to do with the 'complexity' of these components.

The complexity of $\mathcal{H}$. If the number of hypotheses M goes up, we run more risk that $E_{\text{in}}(g)$ will be a poor estimator of $E_{\text{out}}(g)$ according to Inequality (1.6). M can be thought of as a measure of the 'complexity' of the

hypothesis set $\mathcal{H}$ that we use. If we want an affirmative answer to the first question, we need to keep the complexity of $\mathcal{H}$ in check. However, if we want an affirmative answer to the second question, we stand a better chance if $\mathcal{H}$ is more complex, since g has to come from $\mathcal{H}$. So, a more complex $\mathcal{H}$ gives us more flexibility in finding some g that fits the data well, leading to small $E_{\text{in}}(g)$. This tradeoff in the complexity of $\mathcal{H}$ is a major theme in learning theory that we will study in detail in Chapter 2.

The complexity of f. Intuitively, a complex target function f should be harder to learn than a simple f. Let us examine if this can be inferred from the two questions above. A close look at Inequality (1.6) reveals that the complexity of f does not affect how well $E_{\text{in}}(g)$ approximates $E_{\text{out}}(g)$. If we fix the hypothesis set and the number of training examples, the inequality provides the same bound whether we are trying to learn a simple f (for instance a constant function) or a complex f (for instance a highly nonlinear function). However, this doesn't mean that we can learn complex functions as easily as we learn simple functions. Remember that (1.6) affects the first question only. If the target function is complex, the second question comes into play since the data from a complex f are harder to fit than the data from a simple f. This means that we will get a worse value for $E_{\text{in}}(g)$ when f is complex. We might try to get around that by making our hypothesis set more complex so that we can fit the data better and get a lower $E_{\text{in}}(g)$, but then E_{out} won't be as close to E_{in} per (1.6). Either way we look at it, a complex f is harder to learn as we expected. In the extreme case, if f is too complex, we may not be able to learn it at all.

Fortunately, most target functions in real life are not too complex; we can learn them from a reasonable $\mathcal{D}$ using a reasonable $\mathcal{H}$. This is obviously a practical observation, not a mathematical statement. Even when we cannot learn a particular f, we will at least be able to tell that we can't. As long as we make sure that the complexity of $\mathcal{H}$ gives us a good Hoeffding bound, our success or failure in learning f can be determined by our success or failure in fitting the training data.

1.4 Error and Noise

We close this chapter by revisiting two notions in the learning problem in order to bring them closer to the real world. The first notion is what approximation means when we say that our hypothesis approximates the target function well. The second notion is about the nature of the target function. In many situations, there is noise that makes the output of f not uniquely determined by the input. What are the ramifications of having such a 'noisy' target on the learning problem?

1.4.1 Error Measures

Learning is not expected to replicate the target function perfectly. The final hypothesis g is only an approximation of f. To quantify how well g approximates f, we need to define an error measure[3] that quantifies how far we are from the target.

The choice of an error measure affects the outcome of the learning process. Different error measures may lead to different choices of the final hypothesis, even if the target and the data are the same, since the value of a particular error measure may be small while the value of another error measure in the same situation is large. Therefore, which error measure we use has consequences for what we learn. What are the criteria for choosing one error measure over another? We address this question here.

First, let's formalize this notion a bit. An error measure quantifies how well each hypothesis h in the model approximates the target function f,

$$\text{Error} \;=\; E(h, f).$$

While $E(h, f)$ is based on the entirety of h and f, it is almost universally defined based on the errors on individual input points $\mathbf{x}$. If we define a pointwise error measure $\mathsf{e}(h(\mathbf{x}), f(\mathbf{x}))$, the overall error will be the average value of this pointwise error. So far, we have been working with the classification error $\mathsf{e}(h(\mathbf{x}), f(\mathbf{x})) = [\![h(\mathbf{x}) \neq f(\mathbf{x})]\!]$.

In an ideal world, $E(h, f)$ should be user-specified. The same learning task in different contexts may warrant the use of different error measures. One may view $E(h, f)$ as the 'cost' of using h when you should use f. This cost depends on what h is used for, and cannot be dictated just by our learning techniques. Here is a case in point.

Example 1.1 (Fingerprint verification)**.** Consider the problem of verifying that a fingerprint belongs to a particular person. What is the appropriate error measure?

The target function takes as input a fingerprint, and returns $+1$ if it belongs to the right person, and -1 if it belongs to an intruder.

[3]This measure is also called an error *function* in the literature, and sometimes the error is referred to as *cost*, *objective*, or *risk*.

There are two types of error that our hypothesis h can make here. If the correct person is rejected ($h = -1$ but $f = +1$), it is called false reject, and if an incorrect person is accepted ($h = +1$ but $f = -1$), it is called false accept.

		f	
		$+1$	-1
h	$+1$	no error	false accept
	-1	false reject	no error

How should the error measure be defined in this problem? If the right person is accepted or an intruder is rejected, the error is clearly zero. We need to specify the error values for a false accept and for a false reject. The right values depend on the application.

Consider two potential clients of this fingerprint system. One is a supermarket who will use it at the checkout counter to verify that you are a member of a discount program. The other is the CIA who will use it at the entrance to a secure facility to verify that you are authorized to enter that facility.

For the supermarket, a false reject is costly because if a customer gets wrongly rejected, she may be discouraged from patronizing the supermarket in the future. All future revenue from this annoyed customer is lost. On the other hand, the cost of a false accept is minor. You just gave away a discount to someone who didn't deserve it, and that person left their fingerprint in your system they must be bold indeed.

For the CIA, a false accept is a disaster. An unauthorized person will gain access to a highly sensitive facility. This should be reflected in a much higher cost for the false accept. False rejects, on the other hand, can be tolerated since authorized persons are employees (rather than customers as with the supermarket). The inconvenience of retrying when rejected is just part of the job, and they must deal with it.

The costs of the different types of errors can be tabulated in a matrix. For our examples, the matrices might look like:

		f	
		$+1$	-1
h	$+1$	0	1
	-1	10	0

Supermarket

		f	
		$+1$	-1
h	$+1$	0	1000
	-1	1	0

CIA

These matrices should be used to weight the different types of errors when we compute the total error. When the learning algorithm minimizes a cost-weighted error measure, it automatically takes into consideration the utility of the hypothesis that it will produce. In the supermarket and CIA scenarios, this could lead to two completely different final hypotheses. □

The moral of this example is that the choice of the error measure depends on how the system is going to be used, rather than on any inherent criterion

Figure 1.11: The general (supervised) learning problem

that we can independently determine during the learning process. However, this ideal choice may not be possible in practice for two reasons. One is that the user may not provide an error specification, which is not uncommon. The other is that the weighted cost may be a difficult objective function for optimizers to work with. Therefore, we often look for other ways to define the error measure, sometimes with purely practical or analytic considerations in mind. We have already seen an example of this with the simple binary error used in this chapter, and we will see other error measures in later chapters.

1.4.2 Noisy Targets

In many practical applications, the data we learn from are not generated by a deterministic target function. Instead, they are generated in a noisy way such that the output is not uniquely determined by the input. For instance, in the credit-card example we presented in Section 1.1, two customers may have identical salaries, outstanding loans, etc., but end up with different credit behavior. Therefore, the credit 'function' is not really a deterministic function,

but a noisy one.

This situation can be readily modeled within the same framework that we have. Instead of $y = f(\mathbf{x})$, we can take the output y to be a random variable that is affected by, rather than determined by, the input $\mathbf{x}$. Formally, we have a target *distribution* $P(y \mid \mathbf{x})$ instead of a target function $y = f(\mathbf{x})$. A data point $(\mathbf{x}, y)$ is now generated by the joint distribution $P(\mathbf{x}, y) = P(\mathbf{x})P(y \mid \mathbf{x})$.

One can think of a noisy target as a deterministic target plus added noise. If y is real-valued for example, one can take the expected value of y given $\mathbf{x}$ to be the deterministic $f(\mathbf{x})$, and consider $y - f(\mathbf{x})$ as pure noise that is added to f.

This view suggests that a deterministic target function can be considered a special case of a noisy target, just with zero noise. Indeed, we can formally express any function f as a distribution $P(y \mid \mathbf{x})$ by choosing $P(y \mid \mathbf{x})$ to be zero for all y except $y = f(\mathbf{x})$. Therefore, there is no loss of generality if we consider the target to be a distribution rather than a function. Figure 1.11 modifies the previous Figures 1.2 and 1.9 to illustrate the general learning problem, covering both deterministic and noisy targets.

Exercise 1.13

Consider the bin model for a hypothesis h that makes an error with probability μ in approximating a deterministic target function f (both h and f are binary functions). If we use the same h to approximate a noisy version of f given by

$$P(y \mid \mathbf{x}) = \begin{cases} \lambda & y = f(\mathbf{x}), \\ 1 - \lambda & y \neq f(\mathbf{x}). \end{cases}$$

(a) What is the probability of error that h makes in approximating y?

(b) At what value of λ will the performance of h be independent of μ? *[Hint: The noisy target will look completely random.]*

There is a difference between the role of $P(y \mid \mathbf{x})$ and the role of $P(\mathbf{x})$ in the learning problem. While both distributions model probabilistic aspects of $\mathbf{x}$ and y, the target distribution $P(y \mid \mathbf{x})$ is what we are trying to learn, while the input distribution $P(\mathbf{x})$ only quantifies the relative importance of the point $\mathbf{x}$ in gauging how well we have learned.

Our entire analysis of the feasibility of learning applies to noisy target functions as well. Intuitively, this is because the Hoeffding Inequality (1.6) applies to an arbitrary, unknown target function. Assume we randomly picked all the y's according to the distribution $P(y \mid \mathbf{x})$ over the entire input space $\mathcal{X}$. This realization of $P(y \mid \mathbf{x})$ is effectively a target function. Therefore, the inequality will be valid no matter which particular random realization the 'target function' happens to be.

This does not mean that learning a noisy target is as easy as learning a deterministic one. Remember the two questions of learning? With the same learning model, E_{out} may be as close to E_{in} in the noisy case as it is in the

deterministic case, but E_{in} itself will likely be worse in the noisy case since it is hard to fit the noise.

In Chapter 2, where we prove a stronger version of (1.6), we will assume the target to be a probability distribution $P(y \mid \mathbf{x})$, thus covering the general case.

1.5 Problems

Problem 1.1 We have 2 opaque bags, each containing 2 balls. One bag has 2 black balls and the other has a black and a white ball. You pick a bag at random and then pick one of the balls in that bag at random. When you look at the ball it is black. You now pick the second ball from that same bag. What is the probability that this ball is also black? *[Hint: Use Bayes' Theorem:* $\mathbb{P}[A \text{ and } B] = \mathbb{P}[A \mid B]\,\mathbb{P}[B] = \mathbb{P}[B \mid A]\,\mathbb{P}[A]$.*]*

Problem 1.2 Consider the perceptron in two dimensions: $h(\mathbf{x}) = \text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x})$ where $\mathbf{w} = [w_0, w_1, w_2]^{\mathrm{T}}$ and $\mathbf{x} = [1, x_1, x_2]^{\mathrm{T}}$. Technically, $\mathbf{x}$ has three coordinates, but we call this perceptron two-dimensional because the first coordinate is fixed at 1.

(a) Show that the regions on the plane where $h(\mathbf{x}) = +1$ and $h(\mathbf{x}) = -1$ are separated by a line. If we express this line by the equation $x_2 = ax_1 + b$, what are the slope a and intercept b in terms of w_0, w_1, w_2?

(b) Draw a picture for the cases $\mathbf{w} = [1, 2, 3]^{\mathrm{T}}$ and $\mathbf{w} = -[1, 2, 3]^{\mathrm{T}}$.

In more than two dimensions, the $+1$ and -1 regions are separated by a *hy perplane*, the generalization of a line.

Problem 1.3 Prove that the PLA eventually converges to a linear separator for separable data. The following steps will guide you through the proof. Let $\mathbf{w}^*$ be an optimal set of weights (one which separates the data). The essential idea in this proof is to show that the PLA weights $\mathbf{w}(t)$ get "more aligned" with $\mathbf{w}^*$ with every iteration. For simplicity, assume that $\mathbf{w}(0) = \mathbf{0}$.

(a) Let $\rho = \min_{1 \le n \le N} y_n(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n)$. Show that $\rho > 0$.

(b) Show that $\mathbf{w}^{\mathrm{T}}(t)\mathbf{w}^* \ge \mathbf{w}^{\mathrm{T}}(t-1)\mathbf{w}^* + \rho$, and conclude that $\mathbf{w}^{\mathrm{T}}(t)\mathbf{w}^* \ge t\rho$. *[Hint: Use induction.]*

(c) Show that $\|\mathbf{w}(t)\|^2 \le \|\mathbf{w}(t-1)\|^2 + \|\mathbf{x}(t-1)\|^2$.

[Hint: $y(t-1) \cdot (\mathbf{w}^{\mathrm{T}}(t-1)\mathbf{x}(t-1)) \le 0$ *because* $\mathbf{x}(t-1)$ *was misclas sified by* $\mathbf{w}(t-1)$.*]*

(d) Show by induction that $\|\mathbf{w}(t)\|^2 \le tR^2$, where $R = \max_{1 \le n \le N} \|\mathbf{x}_n\|$.

(continued on next page)

(e) Using (b) and (d), show that

$$\frac{\mathbf{w}^{\mathrm{T}}(t)}{\|\mathbf{w}(t)\|}\mathbf{w}^* \geq \sqrt{t} \cdot \frac{\rho}{R},$$

and hence prove that

$$t \leq \frac{R^2\|\mathbf{w}^*\|^2}{\rho^2}.$$

[*Hint:* $\frac{\mathbf{w}^{\mathrm{T}}(t)\mathbf{w}^*}{\|\mathbf{w}(t)\|\,\|\mathbf{w}^*\|} \leq 1$. *Why?*]

In practice, PLA converges more quickly than the bound $\frac{R^2\|\mathbf{w}^*\|^2}{\rho^2}$ suggests. Nevertheless, because we do not know ρ in advance, we can't determine the number of iterations to convergence, which does pose a problem if the data is non-separable.

Problem 1.4 In Exercise 1.4, we use an artificial data set to study the perceptron learning algorithm. This problem leads you to explore the algorithm further with data sets of different sizes and dimensions.

(a) Generate a linearly separable data set of size 20 as indicated in Exercise 1.4. Plot the examples $\{(\mathbf{x}_n, y_n)\}$ as well as the target function f on a plane. Be sure to mark the examples from different classes differently, and add labels to the axes of the plot.

(b) Run the perceptron learning algorithm on the data set above. Report the number of updates that the algorithm takes before converging. Plot the examples $\{(\mathbf{x}_n, y_n)\}$, the target function f, and the final hypothesis g in the same figure. Comment on whether f is close to g.

(c) Repeat everything in (b) with another randomly generated data set of size 20. Compare your results with (b).

(d) Repeat everything in (b) with another randomly generated data set of size 100. Compare your results with (b).

(e) Repeat everything in (b) with another randomly generated data set of size $1,000$. Compare your results with (b).

(f) Modify the algorithm such that it takes $\mathbf{x}_n \in \mathbb{R}^{10}$ instead of $\mathbb{R}^2$. Randomly generate a linearly separable data set of size $1,000$ with $\mathbf{x}_n \in \mathbb{R}^{10}$ and feed the data set to the algorithm. How many updates does the algorithm take to converge?

(g) Repeat the algorithm on the same data set as (f) for 100 experiments. In the iterations of each experiment, pick $\mathbf{x}(t)$ randomly instead of deterministically. Plot a histogram for the number of updates that the algorithm takes to converge.

(h) Summarize your conclusions with respect to accuracy and running time as a function of N and d.

Problem 1.5 The perceptron learning algorithm works like this: In each it eration t, pick a random $(\mathbf{x}(t), y(t))$ and compute the 'signal' $s(t) = \mathbf{w}^{\mathrm{T}}(t)\mathbf{x}(t)$. If $y(t) \cdot s(t) \leq 0$, update $\mathbf{w}$ by

$$\mathbf{w}(t+1) \longleftarrow \mathbf{w}(t) + y(t) \cdot \mathbf{x}(t) \ ;$$

One may argue that this algorithm does not take the 'closeness' between $s(t)$ and $y(t)$ into consideration. Let's look at another perceptron learning algorithm: In each iteration, pick a random $(\mathbf{x}(t), y(t))$ and compute $s(t)$. If $y(t) \cdot s(t) \leq 1$, update $\mathbf{w}$ by

$$\mathbf{w}(t+1) \longleftarrow \mathbf{w}(t) + \eta \cdot (y(t) \quad s(t)) \cdot \mathbf{x}(t) \ ,$$

where η is a constant. That is, if $s(t)$ agrees with $y(t)$ well (their product is > 1), the algorithm does nothing. On the other hand, if $s(t)$ is further from $y(t)$, the algorithm changes $\mathbf{w}(t)$ more. In this problem, you are asked to implement this algorithm and study its performance.

(a) Generate a training data set of size 100 similar to that used in Exercise 1.4. Generate a test data set of size $10,000$ from the same process. To get g, run the algorithm above with $\eta = 100$ on the training data set, until a maximum of $1,000$ updates has been reached. Plot the training data set, the target function f, and the final hypothesis g on the same figure. Report the error on the test set.

(b) Use the data set in (a) and redo everything with $\eta = 1$.

(c) Use the data set in (a) and redo everything with $\eta = 0.01$.

(d) Use the data set in (a) and redo everything with $\eta = 0.0001$.

(e) Compare the results that you get from (a) to (d).

The algorithm above is a variant of the so called Adaline (*Adaptive Linear Neuron*) algorithm for perceptron learning.

Problem 1.6 Consider a sample of 10 marbles drawn independently from a bin that holds red and green marbles. The probability of a red marble is μ. For $\mu = 0.05$, $\mu = 0.5$, and $\mu = 0.8$, compute the probability of getting no red marbles ($\nu = 0$) in the following cases.

(a) We draw only one such sample. Compute the probability that $\nu = 0$.

(b) We draw $1,000$ independent samples. Compute the probability that (at least) one of the samples has $\nu = 0$.

(c) Repeat (b) for $1,000,000$ independent samples.

Problem 1.7 A sample of heads and tails is created by tossing a coin a number of times independently. Assume we have a number of coins that generate different samples independently. For a given coin, let the probability of heads (probability of error) be μ. The probability of obtaining k heads in N tosses of this coin is given by the binomial distribution:

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

Remember that the training error ν is $\frac{k}{N}$.

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

(b) For the case $N = 6$ and 2 coins with $\mu = 0.5$ for both coins, plot the probability

$$P[\max_i |\nu_i - \mu_i| > \epsilon]$$

for ϵ in the range $[0, 1]$ (the $\max$ is over coins). On the same plot show the bound that would be obtained using the Hoeffding Inequality . Remember that for a single coin, the Hoeffding bound is

$$P[|\nu - \mu| > \epsilon] \leq 2e^{-2N\epsilon^2}.$$

[Hint: Use $P[A \text{ or } B] = P[A] + P[B]$ $P[A \text{ and } B] = P[A] + P[B] - P[A]P[B]$, where the last equality follows by independence, to evaluate $P[\max \ldots]$]

Problem 1.8 The Hoeffding Inequality is one form of the *law of large numbers*. One of the simplest forms of that law is the *Chebyshev Inequality*, which you will prove here.

(a) If t is a non negative random variable, prove that for any $\alpha > 0$, $\mathbb{P}[t \geq \alpha] \leq \mathbb{E}(t)/\alpha$.

(b) If u is any random variable with mean μ and variance σ^2, prove that for any $\alpha > 0$, $\mathbb{P}[(u \quad \mu)^2 \geq \alpha] \leq \frac{\sigma^2}{\alpha}$. *[Hint: Use (a)]*

(c) If $u_1, \cdots, u_N$ are iid random variables, each with mean μ and variance σ^2, and $u = \frac{1}{N}\sum_{n=1}^{N} u_n$, prove that for any $\alpha > 0$,

$$\mathbb{P}[(u \quad \mu)^2 \geq \alpha] \leq \frac{\sigma^2}{N\alpha}.$$

Notice that the RHS of this Chebyshev Inequality goes down linearly in N, while the counterpart in Hoeffding's Inequality goes down exponentially. In Problem 1.9, we develop an exponential bound using a similar approach.

Problem 1.9 In this problem, we derive a form of the law of large numbers that has an exponential bound, called the *Chernoff bound*. We focus on the simple case of flipping a fair coin, and use an approach similar to Problem 1.8.

(a) Let t be a (finite) random variable, α be a positive constant, and s be a positive parameter. If $T(s) = \mathbb{E}_t(e^{st})$, prove that

$$\mathbb{P}[t \geq \alpha] \;\leq\; e^{-s\alpha}T(s) \;.$$

[Hint: e^{st} is monotonically increasing in t.]

(b) Let $u_1, \cdots, u_N$ be iid random variables, and let $u = \frac{1}{N}\sum_{n=1}^{N} u_n$. If $U(s) = \mathbb{E}_{u_n}(e^{su_n})$ (for any n), prove that

$$\mathbb{P}[u \geq \alpha] \;\leq\; \left(e^{-s\alpha}U(s)\right)^N \;.$$

(c) Suppose $\mathbb{P}[u_n = 0] = \mathbb{P}[u_n = 1] = \frac{1}{2}$ (fair coin). Evaluate $U(s)$ as a function of s, and minimize $e^{-s\alpha}U(s)$ with respect to s for fixed α, $0 < \alpha < 1$.

(d) Conclude in (c) that, for $0 < \epsilon < \frac{1}{2}$,

$$\mathbb{P}[u \geq \mathbb{E}(u) + \epsilon] \;\leq\; 2^{-\beta N} \;,$$

where $\beta = 1 + (\frac{1}{2} + \epsilon)\log_2(\frac{1}{2} + \epsilon) + (\frac{1}{2} - \epsilon)\log_2(\frac{1}{2} - \epsilon)$ and $\mathbb{E}(u) = \frac{1}{2}$. Show that $\beta > 0$, hence the bound is exponentially decreasing in N.

Problem 1.10 Assume that $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N, \mathbf{x}_{N+1}, \ldots, \mathbf{x}_{N+M}\}$ and $\mathcal{Y} = \{-1, +1\}$ with an unknown target function $f\colon \mathcal{X} \to \mathcal{Y}$. The training data set $\mathcal{D}$ is $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)$. Define the *off-training-set error* of a hypothesis h with respect to f by

$$E_{\text{off}}(h, f) = \frac{1}{M}\sum_{m=1}^{M} [\![h(\mathbf{x}_{N+m}) \neq f(\mathbf{x}_{N+m})]\!] \,.$$

(a) Say $f(\mathbf{x}) = +1$ for all $\mathbf{x}$ and

$$h(\mathbf{x}) = \begin{cases} +1, & \text{for } \mathbf{x} = \mathbf{x}_k \text{ and } k \text{ is odd and } 1 \leq k \leq M+N \\ -1, & \text{otherwise} \end{cases} \;.$$

What is $E_{\text{off}}(h, f)$?

(b) We say that a target function f can 'generate' $\mathcal{D}$ in a noiseless setting if $y_n = f(\mathbf{x}_n)$ for all $(\mathbf{x}_n, y_n) \in \mathcal{D}$. For a fixed $\mathcal{D}$ of size N, how many possible $f\colon \mathcal{X} \to \mathcal{Y}$ can generate $\mathcal{D}$ in a noiseless setting?

(c) For a given hypothesis h and an integer k between 0 and M, how many of those f in (b) satisfy $E_{\text{off}}(h, f) = \frac{k}{M}$?

(d) For a given hypothesis h, if all those f that generate $\mathcal{D}$ in a noiseless setting are equally likely in probability, what is the expected off training-set error $\mathbb{E}_f[E_{\text{off}}(h, f)]$?

(continued on next page)

(e) A deterministic algorithm A is defined as a procedure that takes $\mathcal{D}$ as an input, and outputs a hypothesis $h = A(\mathcal{D})$. Argue that for any two deterministic algorithms A_1 and A_2,

$$\mathbb{E}_f\left[E_{\text{off}}(A_1(\mathcal{D}), f)\right] = \mathbb{E}_f\left[E_{\text{off}}(A_2(\mathcal{D}), f)\right].$$

You have now proved that in a noiseless setting, for a fixed $\mathcal{D}$, if all possible f are equally likely, any two deterministic algorithms are equivalent in terms of the expected off training set error. Similar results can be proved for more general settings.

Problem 1.11 The matrix which tabulates the cost of various errors for the CIA and Supermarket applications in Example 1.1 is called a *risk* or *loss matrix*.

For the two risk matrices in Example 1.1, explicitly write down the in sample error E_{in} that one should minimize to obtain g. This in-sample error should weight the different types of errors based on the risk matrix. *[Hint: Consider $y_n = +1$ and $y_n = -1$ separately.]*

Problem 1.12 This problem investigates how changing the error measure can change the result of the learning process. You have N data points $y_1 \leq \cdots \leq y_N$ and wish to estimate a 'representative' value.

(a) If your algorithm is to find the hypothesis h that minimizes the in sample sum of squared deviations,

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

then show that your estimate will be the in sample mean,

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

(b) If your algorithm is to find the hypothesis h that minimizes the in sample sum of absolute deviations,

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

then show that your estimate will be the in sample median h_{med}, which is any value for which half the data points are at most h_{med} and half the data points are at least h_{med}.

(c) Suppose y_N is perturbed to $y_N + \epsilon$, where $\epsilon \to \infty$. So, the single data point y_N becomes an outlier. What happens to your two estimators h_{mean} and h_{med}?

Chapter 2

Training versus Testing

Before the final exam, a professor may hand out some practice problems and solutions to the class. Although these problems are not the exact ones that will appear on the exam, studying them will help you do better. They are the 'training set' in your learning.

If the professor's goal is to help you do better in the exam, why not give out the exam problems themselves? Well, nice try ☺. Doing well in the exam is not the goal in and of itself. The goal is for you to learn the course material. The exam is merely a way to gauge how well you have learned the material. If the exam problems are known ahead of time, your performance on them will no longer accurately gauge how well you have learned.

The same distinction between training and testing happens in learning from data. In this chapter, we will develop a mathematical theory that characterizes this distinction. We will also discuss the conceptual and practical implications of the contrast between training and testing.

2.1 Theory of Generalization

The out-of-sample error E_{out} measures how well our training on $\mathcal{D}$ has *generalized* to data that we have not seen before. E_{out} is based on the performance over the entire input space $\mathcal{X}$. Intuitively, if we want to estimate the value of E_{out} using a sample of data points, these points must be 'fresh' test points that have not been used for training, similar to the questions on the final exam that have not been used for practice.

The in sample error E_{in}, by contrast, is based on data points that have been used for training. It expressly measures training performance, similar to your performance on the practice problems that you got before the final exam. Such performance has the benefit of looking at the solutions and adjusting accordingly, and may not reflect the ultimate performance in a real test. We began the analysis of in-sample error in Chapter 1, and we will extend this

analysis to the general case in this chapter. We will also make the contrast between a training set and a test set more precise.

A word of warning: this chapter is the heaviest in this book in terms of mathematical abstraction. To make it easier on the not-so-mathematically inclined, we will tell you which part you can safely skip without 'losing the plot'. The mathematical results provide fundamental insights into learning from data, and we will interpret these results in practical terms.

Generalization error. We have already discussed how the value of E_{in} does not always generalize to a similar value of E_{out}. Generalization is a key issue in learning. One can define the *generalization error* as the discrepancy between E_{in} and E_{out}.[1] The Hoeffding Inequality (1.6) provides a way to characterize the generalization error with a probabilistic bound,

$$\mathbb{P}\left[|E_{\text{in}}(g) - E_{\text{out}}(g)| > \epsilon\right] \leq 2Me^{-2\epsilon^2 N},$$

for any $\epsilon > 0$. This can be rephrased as follows. Pick a tolerance level δ, for example $\delta = 0.05$, and assert with probability at least $1 \quad \delta$ that

$$E_{\text{out}}(g) \leq E_{\text{in}}(g) + \sqrt{\frac{1}{2N} \ln \frac{2M}{\delta}}. \tag{2.1}$$

We refer to the type of inequality in (2.1) as a *generalization bound* because it bounds E_{out} in terms of E_{in}. To see that the Hoeffding Inequality implies this generalization bound, we rewrite (1.6) as follows: with probability at least $1 \quad 2Me^{\ -2N\epsilon^2}$, $|E_{\text{out}} \quad E_{\text{in}}| \leq \epsilon$, which implies $E_{\text{out}} \leq E_{\text{in}} + \epsilon$. We may now identify $\delta = 2Me^{\ -2N\epsilon^2}$, from which $\epsilon = \sqrt{\frac{1}{2N} \ln \frac{2M}{\delta}}$, and (2.1) follows.

Notice that the other side of $|E_{\text{out}} \quad E_{\text{in}}| \leq \epsilon$ also holds, that is, $E_{\text{out}} \geq E_{\text{in}} - \epsilon$ for all $h \in \mathcal{H}$. This is important for learning, but in a more subtle way. Not only do we want to know that the hypothesis g that we choose (say the one with the best training error) will continue to do well out of sample (i.e., $E_{\text{out}} \leq E_{\text{in}} + \epsilon$), but we also want to be sure that we did the best we could with our $\mathcal{H}$ (no other hypothesis $h \in \mathcal{H}$ has $E_{\text{out}}(h)$ significantly better than $E_{\text{out}}(g)$). The $E_{\text{out}}(h) \geq E_{\text{in}}(h) - \epsilon$ direction of the bound assures us that we couldn't do much better because every hypothesis with a higher E_{in} than the g we have chosen will have a comparably higher E_{out}.

The error bound $\sqrt{\frac{1}{2N} \ln \frac{2M}{\delta}}$ in (2.1), or 'error bar' if you will, depends on M, the size of the hypothesis set $\mathcal{H}$. If $\mathcal{H}$ is an infinite set, the bound goes to infinity and becomes meaningless. Unfortunately, almost all interesting learning models have infinite $\mathcal{H}$, including the simple perceptron which we discussed in Chapter 1.

In order to study generalization in such models, we need to derive a counterpart to (2.1) that deals with infinite $\mathcal{H}$. We would like to replace M with

[1] Sometimes 'generalization error' is used as another name for E_{out}, but not in this book.

something finite, so that the bound is meaningful. To do this, we notice that the way we got the M factor in the first place was by taking the disjunction of events:

$$\begin{aligned} &\text{“}|E_{\text{in}}(h_1) \quad E_{\text{out}}(h_1)| > \epsilon\text{”} \ \textbf{ or} \\ &\text{“}|E_{\text{in}}(h_2) \quad E_{\text{out}}(h_2)| > \epsilon\text{”} \ \textbf{ or} \\ &\qquad\qquad \vdots \\ &\text{“}|E_{\text{in}}(h_M) - E_{\text{out}}(h_M)| > \epsilon\text{”} \ , \end{aligned} \tag{2.2}$$

which is guaranteed to include the event “$|E_{\text{in}}(g) \quad E_{\text{out}}(g)| > \epsilon$” since g is always one of the hypotheses in $\mathcal{H}$. We then over-estimated the probability using the union bound. Let $\mathcal{B}_m$ be the ($\mathcal{B}$ad) event that “$|E_{\text{in}}(h_m) \quad E_{\text{out}}(h_m)| > \epsilon$”. Then,

$$\mathbb{P}[\mathcal{B}_1 \textbf{ or } \mathcal{B}_2 \textbf{ or } \cdots \textbf{ or } \mathcal{B}_M] \leq \mathbb{P}[\mathcal{B}_1] + \mathbb{P}[\mathcal{B}_2] + \cdots + \mathbb{P}[\mathcal{B}_M].$$

If the events $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_M$ are strongly overlapping, the union bound becomes particularly loose as illustrated in the figure to the right for an example with 3 hypotheses; the areas of different events correspond to their probabilities. The union bound says that the total area covered by $\mathcal{B}_1$, $\mathcal{B}_2$, or $\mathcal{B}_3$ is smaller than the sum of the individual areas, which is true but is a gross overestimate when the areas overlap heavily as in this example. The events “$|E_{\text{in}}(h_m) \quad E_{\text{out}}(h_m)| > \epsilon$”; $m = 1, \cdots, M$, are often strongly overlapping. If h_1 is very similar to h_2 for instance, the two events “$|E_{\text{in}}(h_1) \quad E_{\text{out}}(h_1)| > \epsilon$” and “$|E_{\text{in}}(h_2) \quad E_{\text{out}}(h_2)| > \epsilon$” are likely to coincide for most data sets. In a typical learning model, many hypotheses are indeed very similar. If you take the perceptron model for instance, as you slowly vary the weight vector $\mathbf{w}$, you get infinitely many hypotheses that differ from each other only infinitesimally.

The mathematical theory of generalization hinges on this observation. Once we properly account for the overlaps of the different hypotheses, we will be able to replace the number of hypotheses M in (2.1) by an effective number which is finite even when M is infinite, and establish a more useful condition under which E_{out} is close to E_{in}.

2.1.1 Effective Number of Hypotheses

We now introduce the *growth function*, the quantity that will formalize the effective number of hypotheses. The growth function is what will replace M

in the generalization bound (2.1). It is a combinatorial quantity that captures how different the hypotheses in $\mathcal{H}$ are, and hence how much overlap the different events in (2.2) have.

We will start by defining the growth function and studying its basic properties. Next, we will show how we can bound the value of the growth function. Finally, we will show that we can replace M in the generalization bound with the growth function. These three steps will yield the generalization bound that we need, which applies to infinite $\mathcal{H}$. We will focus on binary target functions for the purpose of this analysis, so each $h \in \mathcal{H}$ maps $\mathcal{X}$ to $\{-1, +1\}$.

The definition of the growth function is based on the number of different hypotheses that $\mathcal{H}$ can implement, but only over a finite sample of points rather than over the entire input space $\mathcal{X}$. If $h \in \mathcal{H}$ is applied to a finite sample $\mathbf{x}_1, \cdots, \mathbf{x}_N \in \mathcal{X}$, we get an N-tuple $h(\mathbf{x}_1), \cdots, h(\mathbf{x}_N)$ of ± 1's. Such an N-tuple is called a *dichotomy* since it splits $\mathbf{x}_1, \cdots, \mathbf{x}_N$ into two groups: those points for which h is -1 and those for which h is $+1$. Each $h \in \mathcal{H}$ generates a dichotomy on $\mathbf{x}_1, \cdots, \mathbf{x}_N$, but two different h's may generate the same dichotomy if they happen to give the same pattern of ± 1's on this particular sample.

Definition 2.1. *Let $\mathbf{x}_1, \cdots, \mathbf{x}_N \in \mathcal{X}$. The dichotomies generated by $\mathcal{H}$ on these points are defined by*

$$\mathcal{H}(\mathbf{x}_1, \cdots, \mathbf{x}_N) \;=\; \{\, (h(\mathbf{x}_1), \cdots, h(\mathbf{x}_N)) \mid h \in \mathcal{H} \}\,. \tag{2.3}$$

One can think of the dichotomies $\mathcal{H}(\mathbf{x}_1, \cdots, \mathbf{x}_N)$ as a set of hypotheses just like $\mathcal{H}$ is, except that the hypotheses are seen through the eyes of N points only. A larger $\mathcal{H}(\mathbf{x}_1, \cdots, \mathbf{x}_N)$ means $\mathcal{H}$ is more 'diverse' generating more dichotomies on $\mathbf{x}_1, \cdots, \mathbf{x}_N$. The growth function is based on the number of dichotomies.

Definition 2.2. *The growth function is defined for a hypothesis set $\mathcal{H}$ by*

$$m_{\mathcal{H}}(N) = \max_{\mathbf{x}_1, \cdots, \mathbf{x}_N \in \mathcal{X}} |\mathcal{H}(\mathbf{x}_1, \cdots, \mathbf{x}_N)|,$$

where $|\cdot|$ denotes the cardinality (number of elements) of a set.

In words, $m_{\mathcal{H}}(N)$ is the maximum number of dichotomies that can be generated by $\mathcal{H}$ on any N points. To compute $m_{\mathcal{H}}(N)$, we consider all possible choices of N points $\mathbf{x}_1, \cdots, \mathbf{x}_N$ from $\mathcal{X}$ and pick the one that gives us the most dichotomies. Like M, $m_{\mathcal{H}}(N)$ is a measure of the number of hypotheses in $\mathcal{H}$, except that a hypothesis is now considered on N points instead of the entire $\mathcal{X}$. For any $\mathcal{H}$, since $\mathcal{H}(\mathbf{x}_1, \cdots, \mathbf{x}_N) \subseteq \{-1, +1\}^N$ (the set of all possible dichotomies on any N points), the value of $m_{\mathcal{H}}(N)$ is at most $\left|\{-1, +1\}^N\right|$, hence

$$m_{\mathcal{H}}(N) \leq 2^N.$$

If $\mathcal{H}$ is capable of generating all possible dichotomies on $\mathbf{x}_1, \cdots, \mathbf{x}_N$, then $\mathcal{H}(\mathbf{x}_1, \cdots, \mathbf{x}_N) = \{-1, +1\}^N$ and we say that $\mathcal{H}$ can *shatter* $\mathbf{x}_1, \cdots, \mathbf{x}_N$. This signifies that $\mathcal{H}$ is as diverse as can be on this particular sample.

Figure 2.1: Illustration of the growth function for a two dimensional per ceptron. The dichotomy of red versus blue on the 3 colinear points in part (a) cannot be generated by a perceptron, but all 8 dichotomies on the 3 points in part (b) can. By contrast, the dichotomy of red versus blue on the 4 points in part (c) cannot be generated by a perceptron. At most 14 out of the possible 16 dichotomies on any 4 points can be generated.

Example 2.1. If $\mathcal{X}$ is a Euclidean plane and $\mathcal{H}$ is a two-dimensional perceptron, what are $m_{\mathcal{H}}(3)$ and $m_{\mathcal{H}}(4)$? Figure 2.1(a) shows a dichotomy on 3 points that the perceptron cannot generate, while Figure 2.1(b) shows another 3 points that the perceptron can shatter, generating all $2^3 = 8$ dichotomies. Because the definition of $m_{\mathcal{H}}(N)$ is based on the maximum number of dichotomies, $m_{\mathcal{H}}(3) = 8$ in spite of the case in Figure 2.1(a).

In the case of 4 points, Figure 2.1(c) shows a dichotomy that the perceptron cannot generate. One can verify that there are no 4 points that the perceptron can shatter. The most a perceptron can do on any 4 points is 14 dichotomies out of the possible 16, where the 2 missing dichotomies are as depicted in Figure 2.1(c) with blue and red corresponding to -1, $+1$ or to $+1$, -1. Hence, $m_{\mathcal{H}}(4) = 14$. □

Let us now illustrate how to compute $m_{\mathcal{H}}(N)$ for some simple hypothesis sets. These examples will confirm the intuition that $m_{\mathcal{H}}(N)$ grows faster when the hypothesis set $\mathcal{H}$ becomes more complex. This is what we expect of a quantity that is meant to replace M in the generalization bound (2.1).

Example 2.2. Let us find a formula for $m_{\mathcal{H}}(N)$ in each of the following cases.

1. *Positive rays*: $\mathcal{H}$ consists of all hypotheses $h\colon \mathbb{R} \to \{-1, +1\}$ of the form $h(x) = \text{sign}(x - a)$, i.e., the hypotheses are defined in a one-dimensional input space, and they return -1 to the left of some value a and $+1$ to the right of a.

To compute $m_{\mathcal{H}}(N)$, we notice that given N points, the line is split by the points into $N+1$ regions. The dichotomy we get on the N points is decided by which region contains the value a. As we vary a, we will get $N+1$ different dichotomies. Since this is the most we can get for any N points, the growth function is

$$m_{\mathcal{H}}(N) = N + 1.$$

Notice that if we picked N points where some of the points coincided (which is allowed), we will get less than $N+1$ dichotomies. This does not affect the value of $m_{\mathcal{H}}(N)$ since it is defined based on the maximum number of dichotomies.

2. *Positive intervals*: $\mathcal{H}$ consists of all hypotheses in one dimension that return $+1$ within some interval and -1 otherwise. Each hypothesis is specified by the two end values of that interval.

To compute $m_{\mathcal{H}}(N)$, we notice that given N points, the line is again split by the points into $N+1$ regions. The dichotomy we get is decided by which two regions contain the end values of the interval, resulting in $\binom{N+1}{2}$ different dichotomies. If both end values fall in the same region, the resulting hypothesis is the constant -1 regardless of which region it is. Adding up these possibilities, we get

$$m_{\mathcal{H}}(N) = \binom{N+1}{2} + 1 = \frac{1}{2}N^2 + \frac{1}{2}N + 1.$$

Notice that $m_{\mathcal{H}}(N)$ grows as the square of N, faster than the linear $m_{\mathcal{H}}(N)$ of the 'simpler' positive ray case.

3. *Convex sets*: $\mathcal{H}$ consists of all hypotheses in two dimensions $h\colon \mathbb{R}^2 \to \{-1, +1\}$ that are positive inside some convex set and negative elsewhere (a set is convex if the line segment connecting any two points in the set lies entirely within the set).

To compute $m_{\mathcal{H}}(N)$ in this case, we need to choose the N points carefully. Per the next figure, choose N points on the perimeter of a circle. Now consider any dichotomy on these points, assigning an arbitrary pattern of ± 1's to the N points. If you connect the $+1$ points with a polygon, the hypothesis made up of the closed interior of the polygon (which has to be convex since its vertices are on the perimeter of a circle) agrees with the dichotomy on all N points. For the dichotomies that have less than three $+1$ points, the convex set will be a line segment, a point, or an empty set.

This means that any dichotomy on these N points can be realized using a convex hypothesis, so $\mathcal{H}$ manages to shatter these points and the growth function has the maximum possible value

$$m_{\mathcal{H}}(N) = 2^N.$$

Notice that if the N points were chosen at random in the plane rather than on the perimeter of a circle, many of the points would be 'internal' and we wouldn't be able to shatter all the points with convex hypotheses as we did for the perimeter points. However, this doesn't matter as far as $m_{\mathcal{H}}(N)$ is concerned, since it is defined based on the maximum (2^N in this case). □

It is not practical to try to compute $m_{\mathcal{H}}(N)$ for every hypothesis set we use. Fortunately, we don't have to. Since $m_{\mathcal{H}}(N)$ is meant to replace M in (2.1), we can use an upper bound on $m_{\mathcal{H}}(N)$ instead of the exact value, and the inequality in (2.1) will still hold. Getting a good bound on $m_{\mathcal{H}}(N)$ will prove much easier than computing $m_{\mathcal{H}}(N)$ itself, thanks to the notion of a *break point*.

Definition 2.3. *If no data set of size k can be shattered by $\mathcal{H}$, then k is said to be a break point for $\mathcal{H}$.*

If k is a break point, then $m_{\mathcal{H}}(k) < 2^k$. Example 2.1 shows that $k = 4$ is a break point for two-dimensional perceptrons. In general, it is easier to find a break point for $\mathcal{H}$ than to compute the full growth function for that $\mathcal{H}$.

Exercise 2.1

By inspection, find a break point k for each hypothesis set in Example 2.2 (if there is one). Verify that $m_{\mathcal{H}}(k) < 2^k$ using the formulas derived in that Example.

We now use the break point k to derive a bound on the growth function $m_{\mathcal{H}}(N)$ for all values of N. For example, the fact that no 4 points can be shattered by

the two-dimensional perceptron puts a significant constraint on the number of dichotomies that can be realized by the perceptron on 5 or more points. We will exploit this idea to get a significant bound on $m_{\mathcal{H}}(N)$ in general.

2.1.2 Bounding the Growth Function

The most important fact about growth functions is that if the condition $m_{\mathcal{H}}(N) = 2^N$ breaks at any point, we can bound $m_{\mathcal{H}}(N)$ for all values of N by a simple polynomial based on this break point. The fact that the bound is polynomial is crucial. Absent a break point (as is the case in the convex hypothesis example), $m_{\mathcal{H}}(N) = 2^N$ for all N. If $m_{\mathcal{H}}(N)$ replaced M in Equation (2.1), the bound $\sqrt{\frac{1}{2N}\ln\frac{2M}{\delta}}$ on the generalization error would not go to zero regardless of how many training examples N we have. However, if $m_{\mathcal{H}}(N)$ can be bounded by a polynomial any polynomial , the generalization error will go to zero as $N \to \infty$. This means that we will generalize well given a sufficient number of examples.

> **Begin safe skip:** If you trust our math, you can skip the following part without compromising the logical sequence. A similar green box will tell you when to rejoin.

To prove the polynomial bound, we will introduce a combinatorial quantity that counts the maximum number of dichotomies given that there is a break point, without having to assume any particular form of $\mathcal{H}$. This bound will therefore apply to any $\mathcal{H}$.

Definition 2.4. *$B(N,k)$ is the maximum number of dichotomies on N points such that no subset of size k of the N points can be shattered by these dichotomies.*

The definition of $B(N,k)$ assumes a break point k, then tries to find the most dichotomies on N points without imposing any further restrictions. Since $B(N,k)$ is defined as a maximum, it will serve as an upper bound for any $m_{\mathcal{H}}(N)$ that has a break point k;

$$m_{\mathcal{H}}(N) \leq B(N,k) \quad \text{if } k \text{ is a break point for } \mathcal{H}.$$

The notation B comes from 'Binomial' and the reason will become clear shortly. To evaluate $B(N,k)$, we start with the two boundary conditions $k = 1$ and $N = 1$.

$$\begin{aligned} B(N,1) &= 1 \\ B(1,k) &= 2 \text{ for } k > 1 \,. \end{aligned}$$

$B(N,1) = 1$ for all N since if no subset of size 1 can be shattered, then only one dichotomy can be allowed. A second different dichotomy must differ on at least one point and then that subset of size 1 would be shattered. $B(1,k) = 2$ for $k > 1$ since in this case there do not even exist subsets of size k; the constraint is vacuously true and we have 2 possible dichotomies ($+1$ and -1) on the one point.

We now assume $N \geq 2$ and $k \geq 2$ and try to develop a recursion. Consider the $B(N,k)$ dichotomies in definition 2.4, where no k points can be shattered. We list these dichotomies in the following table,

	# of rows	$\mathbf{x}_1$	$\mathbf{x}_2$	$\dots$	$\mathbf{x}_{N-1}$	$\mathbf{x}_N$
		$+1$	$+1$	$\dots$	$+1$	$+1$
		-1	$+1$	$\dots$	$+1$	-1
S_1	α	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
		$+1$	-1	$\dots$	-1	-1
		-1	$+1$	$\dots$	-1	$+1$
		$+1$	-1	$\dots$	$+1$	$+1$
		-1	-1	$\dots$	$+1$	$+1$
S_2^+	β	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
		$+1$	-1	$\dots$	$+1$	$+1$
		-1	-1	$\dots$	-1	$+1$
S_2		$+1$	-1	$\dots$	$+1$	-1
		-1	-1	$\dots$	$+1$	-1
S_2^-	β	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
		$+1$	-1	$\dots$	$+1$	-1
		-1	-1	$\dots$	-1	-1

where $\mathbf{x}_1, \cdots, \mathbf{x}_N$ in the table are labels for the N points of the dichotomy. We have chosen a convenient order in which to list the dichotomies, as follows. Consider the dichotomies on $\mathbf{x}_1, \cdots, \mathbf{x}_{N-1}$. Some dichotomies on these $N-1$ points appear only once (with either $+1$ or -1 in the $\mathbf{x}_N$ column, but not both). We collect these dichotomies in the set S_1. The remaining dichotomies on the first $N-1$ points appear twice, once with $+1$ and once with -1 in the $\mathbf{x}_N$ column. We collect these dichotomies in the set S_2 which can be divided into two equal parts, S_2^+ and S_2^- (with $+1$ and -1 in the $\mathbf{x}_N$ column, respectively). Let S_1 have α rows, and let S_2^+ and S_2^- have β rows each. Since the total number of rows in the table is $B(N,k)$ by construction, we have

$$B(N,k) = \alpha + 2\beta. \tag{2.4}$$

The total number of different dichotomies on the first $N-1$ points is given by $\alpha + \beta$; since S_2^+ and S_2^- are identical on these $N-1$ points, their dichotomies are redundant. Since no subset of k of these first $N-1$ points can

be shattered (since no k-subset of all N points can be shattered), we deduce that

$$\alpha + \beta \leq B(N \quad 1, k) \tag{2.5}$$

by definition of B. Further, no subset of size $k-1$ of the first $N-1$ points can be shattered by the dichotomies in S_2^+. If there existed such a subset, then taking the corresponding set of dichotomies in S_2^- and adding $\mathbf{x}_N$ to the data points yields a subset of size k that is shattered, which we know cannot exist in this table by definition of $B(N, k)$. Therefore,

$$\beta \leq B(N-1, k \quad 1). \tag{2.6}$$

Substituting the two Inequalities (2.5) and (2.6) into (2.4), we get

$$B(N, k) \leq B(N-1, k) + B(N \quad 1, k \quad 1). \tag{2.7}$$

We can use (2.7) to recursively compute a bound on $B(N, k)$, as shown in the following table.

		k						
		1	2	3	4	5	6	...
	1	1	2	2	2	2	2	...
	2	1	3	4	4	4	4	...
	3	1	4	7	8	8	8	...
N			↘	↓				
	4	1	5	11	...	...	...	...
	5	1	6	⋮	⋱			
	6	1	7	⋮		⋱		
	⋮	⋮	⋮	⋮			⋱	

where the first row ($N = 1$) and the first column ($k = 1$) are the boundary conditions that we already calculated. We can also use the recursion to bound $B(N, k)$ analytically.

Lemma 2.3 (Sauer's Lemma)**.**

$$B(N, k) \leq \sum_{i=0}^{k-1} \binom{N}{i}$$

Proof. The statement is true whenever $k = 1$ or $N = 1$, by inspection. The proof is by induction on N. Assume the statement is true for all $N \leq N_o$ and all k. We need to prove the statement for $N = N_o + 1$ and all k. Since the statement is already true when $k = 1$ (for all values of N) by the initial condition, we only need to worry about $k \geq 2$. By (2.7),

$$B(N_o + 1, k) \leq B(N_o, k) + B(N_o, k-1).$$

Applying the induction hypothesis to each term on the RHS, we get

$$\begin{aligned} B(N_o+1,k) &\leq \sum_{i=0}^{k-1}\binom{N_o}{i}+\sum_{i=0}^{k-2}\binom{N_o}{i} \\ &= 1+\sum_{i=1}^{k-1}\binom{N_o}{i}+\sum_{i=1}^{k-1}\binom{N_o}{i-1} \\ &= 1+\sum_{i=1}^{k-1}\left[\binom{N_o}{i}+\binom{N_o}{i-1}\right] \\ &= 1+\sum_{i=1}^{k-1}\binom{N_o+1}{i}=\sum_{i=0}^{k-1}\binom{N_o+1}{i}, \end{aligned}$$

where the combinatorial identity $\binom{N_o+1}{i} \quad \binom{N_o}{i}+\binom{N_o}{i-1}$ has been used. This identity can be proved by noticing that to calculate the number of ways to pick i objects from N_o+1 distinct objects, either the first object is included, in $\binom{N_o}{i-1}$ ways, or the first object is not included, in $\binom{N_o}{i}$ ways. We have thus proved the induction step, so the statement is true for all N and k. ■

It turns out that $B(N,k)$ in fact equals $\sum_{i=0}^{k-1}\binom{N}{i}$ (see Problem 2.4), but we only need the inequality of Lemma 2.3 to bound the growth function. For a given break point k, the bound $\sum_{i=0}^{k-1}\binom{N}{i}$ is polynomial in N, as each term in the sum is polynomial (of degree $i \leq k-1$). Since $B(N,k)$ is an upper bound on any $m_{\mathcal{H}}(N)$ that has a break point k, we have proved

> **End safe skip:** Those who skipped are now rejoining us. The next theorem states that any growth function $m_{\mathcal{H}}(N)$ with a break point is bounded by a polynomial.

Theorem 2.4. If $m_{\mathcal{H}}(k) < 2^k$ for some value k, then

$$m_{\mathcal{H}}(N) \leq \sum_{i=0}^{k-1}\binom{N}{i} \tag{2.8}$$

for all N. The RHS is polynomial in N of degree $k-1$.

The implication of Theorem 2.4 is that if $\mathcal{H}$ has a break point, we have what we want to ensure good generalization; a polynomial bound on $m_{\mathcal{H}}(N)$.

Exercise 2.2

(a) Verify the bound of Theorem 2.4 in the three cases of Example 2.2:

(i) Positive rays: $\mathcal{H}$ consists of all hypotheses in one dimension of the form $h(x) = \text{sign}(x - a)$.

(ii) Positive intervals: $\mathcal{H}$ consists of all hypotheses in one dimension that are positive within some interval and negative elsewhere.

(iii) Convex sets: $\mathcal{H}$ consists of all hypotheses in two dimensions that are positive inside some convex set and negative elsewhere.

(Note: you can use the break points you found in Exercise 2.1.)

(b) Does there exist a hypothesis set for which $m_{\mathcal{H}}(N) = N + 2^{\lfloor N/2 \rfloor}$ (where $\lfloor N/2 \rfloor$ is the largest integer $\leq N/2$)?

2.1.3 The VC Dimension

Theorem 2.4 bounds the entire growth function in terms of any break point. The smaller the break point, the better the bound. This leads us to the following definition of a single parameter that characterizes the growth function.

Definition 2.5. *The Vapnik-Chervonenkis dimension of a hypothesis set $\mathcal{H}$, denoted by $d_{\text{vc}}(\mathcal{H})$ or simply d_{vc}, is the largest value of N for which $m_{\mathcal{H}}(N) = 2^N$. If $m_{\mathcal{H}}(N) = 2^N$ for all N, then $d_{\text{vc}}(\mathcal{H}) = \infty$.*

If d_{vc} is the VC dimension of $\mathcal{H}$, then $k = d_{\text{vc}} + 1$ is a break point for $m_{\mathcal{H}}$ since $m_{\mathcal{H}}(N)$ cannot equal 2^N for any $N > d_{\text{vc}}$ by definition. It is easy to see that no smaller break point exists since $\mathcal{H}$ can shatter d_{vc} points, hence it can also shatter any subset of these points.

Exercise 2.3

Compute the VC dimension of $\mathcal{H}$ for the hypothesis sets in parts (i), (ii), (iii) of Exercise 2.2(a).

Since $k = d_{\text{vc}} + 1$ is a break point for $m_{\mathcal{H}}$, Theorem 2.4 can be rewritten in terms of the VC dimension:

$$m_{\mathcal{H}}(N) \leq \sum_{i=0}^{d_{\text{vc}}} \binom{N}{i}. \tag{2.9}$$

Therefore, the VC dimension is the order of the polynomial bound on $m_{\mathcal{H}}(N)$. It is also the best we can do using this line of reasoning, because no smaller break point than $k = d_{\text{vc}} + 1$ exists. The form of the polynomial bound can be further simplified to make the dependency on d_{vc} more salient. We state a useful form here, which can be proved by induction (Problem 2.5).

$$m_{\mathcal{H}}(N) \leq N^{d_{\text{vc}}} + 1. \tag{2.10}$$

Now that the growth function has been bounded in terms of the VC dimension, we have only one more step left in our analysis, which is to replace the number of hypotheses M in the generalization bound (2.1) with the growth function $m_{\mathcal{H}}(N)$. If we manage to do that, the VC dimension will play a pivotal role in the generalization question. If we were to directly replace M by $m_{\mathcal{H}}(N)$ in (2.1), we would get a bound of the form

$$E_{\text{out}} \overset{?}{\leq} E_{\text{in}} + \sqrt{\frac{1}{2N} \ln \frac{2m_{\mathcal{H}}(N)}{\delta}} \ .$$

Unless $d_{\text{vc}}(\mathcal{H}) = \infty$, we know that $m_{\mathcal{H}}(N)$ is bounded by a polynomial in N; thus, $\ln m_{\mathcal{H}}(N)$ grows logarithmically in N regardless of the order of the polynomial, and so it will be crushed by the $\frac{1}{N}$ factor. Therefore, for any fixed tolerance δ, the bound on E_{out} will be arbitrarily close to E_{in} for sufficiently large N.

Only if $d_{\text{vc}}(\mathcal{H}) = \infty$ will this argument fail, as the growth function in this case is exponential in N. For any finite value of d_{vc}, the error bar will converge to zero at a speed determined by d_{vc}, since d_{vc} is the order of the polynomial. The smaller d_{vc} is, the faster the convergence to zero.

It turns out that we cannot just replace M with $m_{\mathcal{H}}(N)$ in the generalization bound (2.1), but rather we need to make other adjustments as we will see shortly. However, the general idea above is correct, and d_{vc} will still play the role that we discussed here. One implication of this discussion is that there is a division of models into two classes. The 'good models' have finite d_{vc}, and for sufficiently large N, E_{in} will be close to E_{out}; for good models, the in-sample performance generalizes to out of sample. The 'bad models' have infinite d_{vc}. With a bad model, no matter how large the data set is, we cannot make generalization conclusions from E_{in} to E_{out} based on the VC analysis.[2]

Because of its significant role, it is worthwhile to try to gain some insight about the VC dimension before we proceed to the formalities of deriving the new generalization bound. One way to gain insight about d_{vc} is to try to compute it for learning models that we are familiar with. Perceptrons are one case where we can compute d_{vc} exactly. This is done in two steps. First, we show that d_{vc} is at least a certain value, then we show that it is at most the same value. There is a logical difference in arguing that d_{vc} is at least a certain value, as opposed to at most a certain value. This is because

$$d_{\text{vc}} \geq N \iff \text{there } \textbf{exists } \mathcal{D} \text{ of size } N \text{ such that } \mathcal{H} \text{ shatters } \mathcal{D},$$

hence we have different conclusions in the following cases.

1. There is a set of N points that can be shattered by $\mathcal{H}$. In this case, we can conclude that $d_{\text{vc}} \geq N$.

[2] In some cases with infinite d_{vc}, such as the convex sets that we discussed, alternative analysis based on an 'average' growth function can establish good generalization behavior.

2. Any set of N points can be shattered by $\mathcal{H}$. In this case, we have more than enough information to conclude that $d_{\text{vc}} \geq N$.

3. There is a set of N points that cannot be shattered by $\mathcal{H}$. Based only on this information, we cannot conclude anything about the value of d_{vc}.

4. No set of N points can be shattered by $\mathcal{H}$. In this case, we can conclude that $d_{\text{vc}} < N$.

Exercise 2.4

Consider the input space $\mathcal{X} = \{1\} \times \mathbb{R}^d$ (including the constant coordinate $x_0 = 1$). Show that the VC dimension of the perceptron (with $d+1$ parameters, counting w_0) is exactly $d+1$ by showing that it is at least $d+1$ and at most $d+1$, as follows.

(a) To show that $d_{\text{vc}} \geq d+1$, find $d+1$ points in $\mathcal{X}$ that the perceptron can shatter. *[Hint: Construct a nonsingular $(d+1) \times (d+1)$ matrix whose rows represent the $d+1$ points, then use the nonsingularity to argue that the perceptron can shatter these points.]*

(b) To show that $d_{\text{vc}} \leq d+1$, show that no set of $d+2$ points in $\mathcal{X}$ can be shattered by the perceptron. *[Hint: Represent each point in $\mathcal{X}$ as a vector of length $d+1$, then use the fact that any $d+2$ vectors of length $d+1$ have to be linearly dependent. This means that some vector is a linear combination of all the other vectors. Now, if you choose the class of these other vectors carefully, then the classification of the dependent vector will be dictated. Conclude that there is some dichotomy that cannot be implemented, and therefore that for $N \geq d+2$, $m_{\mathcal{H}}(N) < 2^N$.]*

The VC dimension of a d-dimensional perceptron[3] is indeed $d+1$. This is consistent with Figure 2.1 for the case $d = 2$, which shows a VC dimension of 3. The perceptron case provides a nice intuition about the VC dimension, since $d+1$ is also the number of parameters in this model. One can view the VC dimension as measuring the 'effective' number of parameters. The more parameters a model has, the more diverse its hypothesis set is, which is reflected in a larger value of the growth function $m_{\mathcal{H}}(N)$. In the case of perceptrons, the effective parameters correspond to explicit parameters in the model, namely $w_0, w_1, \cdots, w_d$. In other models, the effective parameters may be less obvious or implicit. The VC dimension measures these effective parameters or 'degrees of freedom' that enable the model to express a diverse set of hypotheses.

Diversity is not necessarily a good thing in the context of generalization. For example, the set of all possible hypotheses is as diverse as can be, so $m_{\mathcal{H}}(N) = 2^N$ for all N and $d_{\text{vc}}(\mathcal{H}) = \infty$. In this case, no generalization at all is to be expected, as the final version of the generalization bound will show.

[3] $\mathcal{X}$ $\{1\} \times \mathbb{R}^d$ is considered d dimensional since the first coordinate x_0 1 is fixed.

2.1.4 The VC Generalization Bound

If we treated the growth function as an effective number of hypotheses, and replaced M in the generalization bound (2.1) with $m_{\mathcal{H}}(N)$, the resulting bound would be

$$E_{\text{out}}(g) \overset{?}{\leq} E_{\text{in}}(g) + \sqrt{\frac{1}{2N} \ln \frac{2m_{\mathcal{H}}(N)}{\delta}}. \tag{2.11}$$

It turns out that this is not exactly the form that will hold. The quantities in red need to be technically modified to make (2.11) true. The correct bound, which is called the VC generalization bound, is given in the following theorem; it holds for any binary target function f, any hypothesis set $\mathcal{H}$, any learning algorithm $\mathcal{A}$, and any input probability distribution P.

Theorem 2.5 (VC generalization bound)**.** For any tolerance $\delta > 0$,

$$E_{\text{out}}(g) \leq E_{\text{in}}(g) + \sqrt{\frac{8}{N} \ln \frac{4m_{\mathcal{H}}(2N)}{\delta}} \tag{2.12}$$

with probability $\geq 1 - \delta$.

If you compare the blue items in (2.12) to their red counterparts in (2.11), you notice that all the blue items move the bound in the weaker direction. However, as long as the VC dimension is finite, the error bar still converges to zero (albeit at a slower rate), since $m_{\mathcal{H}}(2N)$ is also polynomial of order d_{vc} in N, just like $m_{\mathcal{H}}(N)$. This means that, with enough data, *each and every hypothesis* in an infinite $\mathcal{H}$ with a finite VC dimension will generalize well from E_{in} to E_{out}. The key is that the effective number of hypotheses, represented by the finite growth function, has replaced the actual number of hypotheses in the bound.

The VC generalization bound is the most important mathematical result in the theory of learning. Since the formal proof is somewhat lengthy and technical, we illustrate the main ideas in a sketch of the proof, and include the formal proof as an appendix. There are two parts to the proof; the justification that the growth function can replace the number of hypotheses in the first place, and the reason why we had to change the red items in (2.11) into the blue items in (2.12).

Sketch of the proof. The data set $\mathcal{D}$ is the source of randomization in the original Hoeffding Inequality. Consider the space of all possible data sets. Let us think of this space as a 'canvas' (Figure 2.2(a)). Each $\mathcal{D}$ is a point on that canvas. The probability of a point is determined by which $\mathbf{x}_n$'s in $\mathcal{X}$ happen to be in that particular $\mathcal{D}$, and is calculated based on the distribution P over $\mathcal{X}$. Let's think of probabilities of different events as areas on that canvas, so the total area of the canvas is 1.

(a) Hoeffding Inequality (b) Union Bound (c) VC Bound

Figure 2.2: Illustration of the proof of the VC bound, where the 'canvas' represents the space of all data sets, with areas corresponding to probabili ties. (a) For a given hypothesis, the colored points correspond to data sets where E_{in} does not generalize well to E_{out}. The Hoeffding Inequality guar antees a small colored area. (b) For several hypotheses, the union bound assumes no overlaps, so the total colored area is large. (c) The VC bound keeps track of overlaps, so it estimates the total area of bad generalization to be relatively small.

For a given hypothesis $h \in \mathcal{H}$, the event "$|E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon$" consists of all points $\mathcal{D}$ for which the statement is true. For a particular h, let us paint all these 'bad' points using one color. What the basic Hoeffding Inequality tells us is that the colored area on the canvas will be small (Figure 2.2(a)).

Now, if we take another $h \in \mathcal{H}$, the event "$|E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon$" may contain different points, since the event depends on h. Let us paint these points with a different color. The area covered by all the points we colored will be at most the sum of the two individual areas, which is the case only if the two areas have no points in common. This is the worst case that the union bound considers. If we keep throwing in a new colored area for each $h \in \mathcal{H}$, and never overlap with previous colors, the canvas will soon be mostly covered in color (Figure 2.2(b)). Even if each h contributed very little, the sheer number of hypotheses will eventually make the colored area cover the whole canvas. This was the problem with using the union bound in the Hoeffding Inequality (1.6), and not taking the overlaps of the colored areas into consideration.

The bulk of the VC proof deals with how to account for the overlaps. Here is the idea. If you were told that the hypotheses in $\mathcal{H}$ are such that each point on the canvas that is colored will be colored 100 times (because of 100 different h's), then the total colored area is now 1/100 of what it would have been if the colored points had not overlapped at all. This is the essence of the VC bound as illustrated in (Figure 2.2(c)). The argument goes as follows.

Many hypotheses share the same dichotomy on a given $\mathcal{D}$, since there are finitely many dichotomies even with an infinite number of hypotheses. Any statement based on $\mathcal{D}$ alone will be simultaneously true or simultaneously false for all the hypotheses that look the same on that particular $\mathcal{D}$. What the growth function enables us to do is to account for this kind of hypothesis redundancy in a precise way, so we can get a factor similar to the '100' in the above example.

When $\mathcal{H}$ is infinite, the redundancy factor will also be infinite since the hypotheses will be divided among a finite number of dichotomies. Therefore, the reduction in the total colored area when we take the redundancy into consideration will be dramatic. If it happens that the number of dichotomies is only a polynomial, the reduction will be so dramatic as to bring the total probability down to a very small value. This is the essence of the proof of Theorem 2.5.

The reason $m_{\mathcal{H}}(2N)$ appears in the VC bound instead of $m_{\mathcal{H}}(N)$ is that the proof uses a sample of $2N$ points instead of N points. Why do we need $2N$ points? The event "$|E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon$" depends not only on $\mathcal{D}$, but also on the entire $\mathcal{X}$ because $E_{\text{out}}(h)$ is based on $\mathcal{X}$. This breaks the main premise of grouping h's based on their behavior on $\mathcal{D}$, since aspects of each h outside of $\mathcal{D}$ affect the truth of "$|E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon$." To remedy that, we consider the artificial event "$|E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \epsilon$" instead, where E_{in} and E'_{in} are based on two samples $\mathcal{D}$ and $\mathcal{D}'$ each of size N. This is where the $2N$ comes from. It accounts for the total size of the two samples $\mathcal{D}$ and $\mathcal{D}'$. Now, the truth of the statement "$|E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \epsilon$" depends exclusively on the total sample of size $2N$, and the above redundancy argument will hold.

Of course we have to justify why the two-sample condition "$|E_{\text{in}}(h)$ $E'_{\text{in}}(h)| > \epsilon$" can replace the original condition "$|E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon$." In doing so, we end up having to shrink the ϵ's by a factor of 4, and also end up with a factor of 2 in the estimate of the overall probability. This accounts for the $\frac{8}{N}$ instead of $\frac{1}{2N}$ in the VC bound and for having 4 instead of 2 as the multiplicative factor of the growth function. When you put all this together, you get the formula in (2.12). □

2.2 Interpreting the Generalization Bound

The VC generalization bound (2.12) is a universal result in the sense that it applies to all hypothesis sets, learning algorithms, input spaces, probability distributions, and binary target functions. It can be extended to other types of target functions as well. Given the generality of the result, one would suspect that the bound it provides may not be particularly tight in any given case, since the same bound has to cover a lot of different cases. Indeed, the bound is quite loose.

Exercise 2.5

Suppose we have a simple learning model whose growth function is $m_{\mathcal{H}}(N) = N + 1$, hence $d_{\text{vc}} = 1$. Use the VC bound (2.12) to estimate the probability that E_{out} will be within 0.1 of E_{in} given 100 training examples. *[Hint: The estimate will be ridiculous.]*

Why is the VC bound so loose? The slack in the bound can be attributed to a number of technical factors. Among them,

1. The basic Hoeffding Inequality used in the proof already has a slack. The inequality gives the same bound whether E_{out} is close to 0.5 or close to zero. However, the variance of E_{in} is quite different in these two cases. Therefore, having one bound capture both cases will result in some slack.

2. Using $m_{\mathcal{H}}(N)$ to quantify the number of dichotomies on N points, regardless of which N points are in the data set, gives us a worst-case estimate. This does allow the bound to be independent of the probability distribution P over $\mathcal{X}$. However, we would get a more tuned bound if we considered specific $\mathbf{x}_1, \cdots, \mathbf{x}_N$ and used $|\mathcal{H}(\mathbf{x}_1, \cdots, \mathbf{x}_N)|$ or its expected value instead of the upper bound $m_{\mathcal{H}}(N)$. For instance, in the case of convex sets in two dimensions, which we examined in Example 2.2, if you pick N points at random in the plane, they will likely have far fewer dichotomies than 2^N, while $m_{\mathcal{H}}(N) = 2^N$.

3. Bounding $m_{\mathcal{H}}(N)$ by a simple polynomial of order d_{vc}, as given in (2.10), will contribute further slack to the VC bound.

Some effort could be put into tightening the VC bound, but many highly technical attempts in the literature have resulted in only diminishing returns. The reality is that the VC line of analysis leads to a very loose bound. Why did we bother to go through the analysis then? Two reasons. First, the VC analysis is what establishes the feasibility of learning for infinite hypothesis sets, the only kind we use in practice. Second, although the bound is loose, it tends to be equally loose for different learning models, and hence is useful for comparing the generalization performance of these models. This is an observation from practical experience, not a mathematical statement. In real applications, learning models with lower d_{vc} tend to generalize better than those with higher d_{vc}. Because of this observation, the VC analysis proves useful in practice, and some rules of thumb have emerged in terms of the VC dimension. For instance, requiring that N be at least $10 \times d_{\text{vc}}$ to get decent generalization is a popular rule of thumb.

Thus, the VC bound can be used as a guideline for generalization, relatively if not absolutely. With this understanding, let us look at the different ways the bound is used in practice.

2.2.1 Sample Complexity

The sample complexity denotes how many training examples N are needed to achieve a certain generalization performance. The performance is specified by two parameters, ϵ and δ. The error tolerance ϵ determines the allowed generalization error, and the confidence parameter δ determines how often the error tolerance ϵ is violated. How fast N grows as ϵ and δ become smaller[4] indicates how much data is needed to get good generalization.

We can use the VC bound to estimate the sample complexity for a given learning model. Fix $\delta > 0$, and suppose we want the generalization error to be at most ϵ. From Equation (2.12), the generalization error is bounded by $\sqrt{\frac{8}{N}\ln\frac{4m_{\mathcal{H}}(2N)}{\delta}}$, and so it suffices to make $\sqrt{\frac{8}{N}\ln\frac{4m_{\mathcal{H}}(2N)}{\delta}} \leq \epsilon$. It follows that

$$N \geq \frac{8}{\epsilon^2}\ln\left(\frac{4m_{\mathcal{H}}(2N)}{\delta}\right)$$

suffices to obtain generalization error at most ϵ (with probability at least $1-\delta$). This gives an implicit bound for the sample complexity N, since N appears on both sides of the inequality. If we replace $m_{\mathcal{H}}(2N)$ in (2.12) by its polynomial upper bound in (2.10) which is based on the the VC dimension, we get a similar bound

$$N \geq \frac{8}{\epsilon^2}\ln\left(\frac{4((2N)^{d_{\text{vc}}}+1)}{\delta}\right), \tag{2.13}$$

which is again implicit in N. We can obtain a numerical value for N using simple iterative methods.

Example 2.6. Suppose that we have a learning model with $d_{\text{vc}} = 3$ and would like the generalization error to be at most 0.1 with confidence 90% (so $\epsilon = 0.1$ and $\delta = 0.1$). How big a data set do we need? Using (2.13), we need

$$N \geq \frac{8}{0.1^2}\ln\left(\frac{4(2N)^3+4}{0.1}\right).$$

Trying an initial guess of $N = 1,000$ in the RHS, we get

$$N \geq \frac{8}{0.1^2}\ln\left(\frac{4(2\times 1000)^3+4}{0.1}\right) \approx 21,193.$$

We then try the new value $N = 21,193$ in the RHS and continue this iterative process, rapidly converging to an estimate of $N \approx 30,000$. If d_{vc} were 4, a similar calculation will find that $N \approx 40,000$. For $d_{\text{vc}} = 5$, we get $N \approx 50,000$. You can see that the inequality suggests that the number of examples needed is approximately proportional to the VC dimension, as has been observed in practice. The constant of proportionality it suggests is 10,000, which is a *gross* overestimate; a more practical constant of proportionality is closer to 10. □

[4]The term 'complexity' comes from a similar metaphor in computational complexity.

2.2.2 Penalty for Model Complexity

Sample complexity fixes the performance parameters ϵ (generalization error) and δ (confidence parameter) and estimates how many examples N are needed. In most practical situations, however, we are given a fixed data set $\mathcal{D}$, so N is also fixed. In this case, the relevant question is what performance can we expect given this particular N. The bound in (2.12) answers this question: with probability at least $1-\delta$,

$$E_{\text{out}}(g) \leq E_{\text{in}}(g) + \sqrt{\frac{8}{N}\ln\left(\frac{4m_{\mathcal{H}}(2N)}{\delta}\right)}\,.$$

If we use the polynomial bound based on d_{vc} instead of $m_{\mathcal{H}}(2N)$, we get another valid bound on the out-of-sample error,

$$E_{\text{out}}(g) \leq E_{\text{in}}(g) + \sqrt{\frac{8}{N}\ln\left(\frac{4((2N)^{d_{\text{vc}}}+1)}{\delta}\right)}\,. \tag{2.14}$$

Example 2.7. Suppose that $N = 100$ and we have a 90% confidence requirement ($\delta = 0.1$). We could ask what error bar can we offer with this confidence, if $\mathcal{H}$ has $d_{\text{vc}} = 1$. Using (2.14), we have

$$E_{\text{out}}(g) \leq E_{\text{in}}(g) + \sqrt{\frac{8}{100}\ln\left(\frac{4(201)}{0.1}\right)} \approx E_{\text{in}}(g) + 0.848 \tag{2.15}$$

with confidence $\geq 90\%$. This is a pretty poor bound on E_{out}. Even if $E_{\text{in}} = 0$, E_{out} may still be close to 1. If $N = 1,000$, then we get $E_{\text{out}}(g) \leq E_{\text{in}}(g) + 0.301$, a somewhat more respectable bound. □

Let us look more closely at the two parts that make up the bound on E_{out} in (2.12). The first part is E_{in}, and the second part is a term that increases as the VC dimension of $\mathcal{H}$ increases.

$$E_{\text{out}}(g) \leq E_{\text{in}}(g) + \Omega(N, \mathcal{H}, \delta), \tag{2.16}$$

where

$$\begin{aligned}\Omega(N,\mathcal{H},\delta) &= \sqrt{\frac{8}{N}\ln\left(\frac{4m_{\mathcal{H}}(2N)}{\delta}\right)} \\ &\leq \sqrt{\frac{8}{N}\ln\left(\frac{4((2N)^{d_{\text{vc}}}+1)}{\delta}\right)}\,.\end{aligned}$$

One way to think of $\Omega(N, \mathcal{H}, \delta)$ is that it is a penalty for model complexity. It penalizes us by worsening the bound on E_{out} when we use a more complex $\mathcal{H}$ (larger d_{vc}). If someone manages to fit a simpler model with the same training

Figure 2.3: When we use a more complex learning model, one that has higher VC dimension d_{vc}, we are likely to fit the training data better re sulting in a lower in sample error, but we pay a higher penalty for model complexity. A combination of the two, which estimates the out of sample error, thus attains a minimum at some intermediate d^*_{vc}.

error, they will get a more favorable estimate for E_{out}. The penalty $\Omega(N, \mathcal{H}, \delta)$ gets worse if we insist on higher confidence (lower δ), and it gets better when we have more training examples, as we would expect.

Although $\Omega(N, \mathcal{H}, \delta)$ goes up when $\mathcal{H}$ has a higher VC dimension, E_{in} is likely to go down with a higher VC dimension as we have more choices within $\mathcal{H}$ to fit the data. Therefore, we have a tradeoff: more complex models help E_{in} and hurt $\Omega(N, \mathcal{H}, \delta)$. The optimal model is a compromise that minimizes a combination of the two terms, as illustrated informally in Figure 2.3.

2.2.3 The Test Set

As we have seen, the generalization bound gives us a loose estimate of the out-of-sample error E_{out} based on E_{in}. While the estimate can be useful as a guideline for the training process, it is next to useless if the goal is to get an accurate forecast of E_{out}. If you are developing a system for a customer, you need a more accurate estimate so that your customer knows how well the system is expected to perform.

An alternative approach that we alluded to in the beginning of this chapter is to estimate E_{out} by using a *test set*, a data set that was not involved in the training process. The final hypothesis g is evaluated on the test set, and the result is taken as an estimate of E_{out}. We would like to now take a closer look at this approach.

Let us call the error we get on the test set E_{test}. When we report E_{test} as our estimate of E_{out}, we are in fact asserting that E_{test} generalizes very well to E_{out}. After all, E_{test} is just a sample estimate like E_{in}. How do we know

that E_{test} generalizes well? We can answer this question with authority now that we have developed the theory of generalization in concrete mathematical terms.

The effective number of hypotheses that matters in the generalization behavior of E_{test} is 1. There is only one hypothesis as far as the test set is concerned, and that's the final hypothesis g that the training phase produced. This hypothesis would not change if we used a different test set as it would if we used a different training set. Therefore, the simple Hoeffding Inequality is valid in the case of a test set. Had the choice of g been affected by the test set in any shape or form, it wouldn't be considered a *test* set any more and the simple Hoeffding Inequality would not apply.

Therefore, the generalization bound that applies to E_{test} is the simple Hoeffding Inequality with one hypothesis. This is a much tighter bound than the VC bound. For example, if you have $1,000$ data points in the test set, E_{test} will be within $\pm 5\%$ of E_{out} with probability $\geq 98\%$. The bigger the test set you use, the more accurate E_{test} will be as an estimate of E_{out}.

Exercise 2.6

A data set has 600 examples. To properly test the performance of the final hypothesis, you set aside a randomly selected subset of 200 examples which are never used in the training phase; these form a test set. You use a learning model with $1,000$ hypotheses and select the final hypothesis g based on the 400 training examples. We wish to estimate $E_{\text{out}}(g)$. We have access to two estimates: $E_{\text{in}}(g)$, the in sample error on the 400 training examples; and, $E_{\text{test}}(g)$, the test error on the 200 test examples that were set aside.

(a) Using a 5% error tolerance ($\delta = 0.05$), which estimate has the higher 'error bar'?

(b) Is there any reason why you shouldn't reserve even more examples for testing?

Another aspect that distinguishes the test set from the training set is that the test set is not biased. Both sets are finite samples that are bound to have some variance due to sample size, but the test set doesn't have an optimistic or pessimistic bias in its estimate of E_{out}. The training set has an optimistic bias, since it was used to choose a hypothesis that looked good *on it*. The VC generalization bound implicitly takes that bias into consideration, and that's why it gives a huge error bar. The test set just has straight finite-sample variance, but no bias. When you report the value of E_{test} to your customer and they try your system on new data, they are as likely to be pleasantly surprised as unpleasantly surprised, though quite likely not to be surprised at all.

There is a price to be paid for having a test set. The test set does not affect the outcome of our learning process, which only uses the training set. The test set just tells us how well we did. Therefore, if we set aside some

of the data points provided by the customer as a test set, we end up using fewer examples for training. Since the training set is used to select one of the hypotheses in $\mathcal{H}$, training examples are essential to finding a good hypothesis. If we take a big chunk of the data for testing and end up with too few examples for training, we may not get a good hypothesis from the training part even if we can reliably evaluate it in the testing part. We may end up reporting to the customer, with high confidence mind you, that the g we are delivering is terrible ☺. There is thus a tradeoff to setting aside test examples. We will address that tradeoff in more detail and learn some clever tricks to get around it in Chapter 4.

In some of the learning literature, E_{test} is used as synonymous with E_{out}. When we report experimental results in this book, we will often treat E_{test} based on a large test set as if it was E_{out} because of the closeness of the two quantities.

2.2.4 Other Target Types

Although the VC analysis was based on binary target functions, it can be extended to real-valued functions, as well as to other types of functions. The proofs in those cases are quite technical, and they do not add to the insight that the VC analysis of binary functions provides. Therefore, we will introduce an alternative approach that covers real-valued functions and provides new insights into generalization. The approach is based on bias-variance analysis, and will be discussed in the next section.

In order to deal with real-valued functions, we need to adapt the definitions of E_{in} and E_{out} that have so far been based on binary functions. We defined E_{in} and E_{out} in terms of binary error; either $h(\mathbf{x}) = f(\mathbf{x})$ or else $h(\mathbf{x}) \neq f(\mathbf{x})$. If f and h are real-valued, a more appropriate error measure would gauge how far $f(\mathbf{x})$ and $h(\mathbf{x})$ are from each other, rather than just whether their values are exactly the same.

An error measure that is commonly used in this case is the squared error $\mathsf{e}(h(\mathbf{x}), f(\mathbf{x})) = (h(\mathbf{x}) - f(\mathbf{x}))^2$. We can define in-sample and out-of-sample versions of this error measure. The out-of-sample error is based on the expected value of the error measure over the entire input space $\mathcal{X}$,

$$E_{\text{out}}(h) = \mathbb{E}\left[(h(\mathbf{x}) - f(\mathbf{x}))^2\right],$$

while the in-sample error is based on averaging the error measure over the data set,

$$E_{\text{in}}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n) - f(\mathbf{x}_n))^2.$$

These definitions make E_{in} a sample estimate of E_{out} just as it was in the case of binary functions. In fact, the error measure used for binary functions can also be expressed as a squared error.

Exercise 2.7

For binary target functions, show that $\mathbb{P}[h(\mathbf{x}) \neq f(\mathbf{x})]$ can be written as an expected value of a mean squared error measure in the following cases.

(a) The convention used for the binary function is 0 or 1.

(b) The convention used for the binary function is ± 1.

[Hint: The difference between (a) and (b) is just a scale.]

Just as the sample frequency of error converges to the overall probability of error per Hoeffding's Inequality, the sample average of squared error converges to the expected value of that error (assuming finite variance). This is a manifestation of what is referred to as the 'law of large numbers' and Hoeffding's Inequality is just one form of that law. The same issues of the data set size and the hypothesis set complexity come into play just as they did in the VC analysis.

2.3 Approximation-Generalization Tradeoff

The VC analysis showed us that the choice of $\mathcal{H}$ needs to strike a balance between approximating f on the training data and generalizing on new data. The ideal $\mathcal{H}$ is a singleton hypothesis set containing only the target function. Unfortunately, we are better off buying a lottery ticket than hoping to have this $\mathcal{H}$. Since we do not know the target function, we resort to a larger model hoping that it will contain a good hypothesis, and hoping that the data will pin down that hypothesis. When you select your hypothesis set, you should balance these two conflicting goals; to have some hypothesis in $\mathcal{H}$ that can approximate f, and to enable the data to zoom in on the right hypothesis.

The VC generalization bound is one way to look at this tradeoff. If $\mathcal{H}$ is too simple, we may fail to approximate f well and end up with a large in-sample error term. If $\mathcal{H}$ is too complex, we may fail to generalize well because of the large model complexity term. There is another way to look at the approximation-generalization tradeoff which we will present in this section. It is particularly suited for squared error measures, rather than the binary error used in the VC analysis. The new way provides a different angle; instead of bounding E_{out} by E_{in} plus a penalty term Ω, we will decompose E_{out} into two different error terms.

2.3.1 Bias and Variance

The bias-variance decomposition of out-of-sample error is based on squared error measures. The out-of-sample error is

$$E_{\text{out}}(g^{(\mathcal{D})}) = \mathbb{E}_{\mathbf{x}}\left[(g^{(\mathcal{D})}(\mathbf{x}) - f(\mathbf{x}))^2\right], \tag{2.17}$$

where $\mathbb{E}_{\mathbf{x}}$ denotes the expected value with respect to $\mathbf{x}$ (based on the probability distribution on the input space $\mathcal{X}$). We have made explicit the dependence of the final hypothesis g on the data $\mathcal{D}$, as this will play a key role in the current analysis. We can rid Equation (2.17) of the dependence on a particular data set by taking the expectation with respect to all data sets. We then get the expected out-of-sample error for our learning model, independent of any particular realization of the data set,

$$\begin{aligned}
\mathbb{E}_{\mathcal{D}}\left[E_{\text{out}}(g^{(\mathcal{D})})\right] &= \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{x}}[(g^{(\mathcal{D})}(\mathbf{x}) - f(\mathbf{x}))^2]\right] \\
&= \mathbb{E}_{\mathbf{x}}\left[\mathbb{E}_{\mathcal{D}}[(g^{(\mathcal{D})}(\mathbf{x}) - f(\mathbf{x}))^2]\right] \\
&= \mathbb{E}_{\mathbf{x}}\left[\mathbb{E}_{\mathcal{D}}[g^{(\mathcal{D})}(\mathbf{x})^2] - 2\,\mathbb{E}_{\mathcal{D}}\left[g^{(\mathcal{D})}(\mathbf{x})\right]f(\mathbf{x}) + f(\mathbf{x})^2\right].
\end{aligned}$$

The term $\mathbb{E}_{\mathcal{D}}[g^{(\mathcal{D})}(\mathbf{x})]$ gives an 'average function', which we denote by $\bar{g}(\mathbf{x})$. One can interpret $\bar{g}(\mathbf{x})$ in the following operational way. Generate many data sets $\mathcal{D}_1, \ldots, \mathcal{D}_K$ and apply the learning algorithm to each data set to produce final hypotheses $g_1, \ldots, g_K$. We can then estimate the average function for any $\mathbf{x}$ by $\bar{g}(\mathbf{x}) \approx \frac{1}{K}\sum_{k=1}^{K} g_k(\mathbf{x})$. Essentially, we are viewing $g(\mathbf{x})$ as a random variable, with the randomness coming from the randomness in the data set; $\bar{g}(\mathbf{x})$ is the expected value of this random variable (for a particular $\mathbf{x}$), and $\bar{g}$ is a *function*, the average function, composed of these expected values. The function $\bar{g}$ is a little counterintuitive; for one thing, $\bar{g}$ need not be in the model's hypothesis set, even though it is the average of functions that are.

Exercise 2.8

(a) Show that if $\mathcal{H}$ is closed under linear combination (any linear combination of hypotheses in $\mathcal{H}$ is also a hypothesis in $\mathcal{H}$), then $\bar{g} \in \mathcal{H}$.

(b) Give a model for which the average function $\bar{g}$ is not in the model's hypothesis set. *[Hint: Use a very simple model.]*

(c) For binary classification, do you expect $\bar{g}$ to be a binary function?

We can now rewrite the expected out-of-sample error in terms of $\bar{g}$:

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}[E_{\text{out}}(g^{(\mathcal{D})})] \\
=\ &\mathbb{E}_{\mathbf{x}}\left[\mathbb{E}_{\mathcal{D}}[g^{(\mathcal{D})}(\mathbf{x})^2] - 2\bar{g}(\mathbf{x})f(\mathbf{x}) + f(\mathbf{x})^2\right], \\
=\ &\mathbb{E}_{\mathbf{x}}\Big[\underbrace{\mathbb{E}_{\mathcal{D}}[g^{(\mathcal{D})}(\mathbf{x})^2] - \bar{g}(\mathbf{x})^2}_{\mathbb{E}_{\mathcal{D}}\left[(g^{(\mathcal{D})}(\mathbf{x}) - \bar{g}(\mathbf{x}))^2\right]} + \underbrace{\bar{g}(\mathbf{x})^2 - 2\bar{g}(\mathbf{x})f(\mathbf{x}) + f(\mathbf{x})^2}_{(\bar{g}(\mathbf{x}) - f(\mathbf{x}))^2}\Big],
\end{aligned}$$

where the last reduction follows since $\bar{g}(\mathbf{x})$ is constant with respect to $\mathcal{D}$. The term $(\bar{g}(\mathbf{x}) - f(\mathbf{x}))^2$ measures how much the average function that we would learn using different data sets $\mathcal{D}$ deviates from the target function that generated these data sets. This term is appropriately called the bias:

$$\text{bias}(\mathbf{x}) = (\bar{g}(\mathbf{x}) - f(\mathbf{x}))^2,$$

as it measures how much our learning model is biased away from the target function.[5] This is because $\bar{g}$ has the benefit of learning from an unlimited number of data sets, so it is only limited in its ability to approximate f by the limitation in the learning model itself. The term $\mathbb{E}_{\mathcal{D}}\left[(g^{(\mathcal{D})}(\mathbf{x}) \quad \bar{g}(\mathbf{x}))^2\right]$ is the variance of the random variable $g^{(\mathcal{D})}(\mathbf{x})$,

$$\mathsf{var}(\mathbf{x}) = \mathbb{E}_{\mathcal{D}}[(g^{(\mathcal{D})}(\mathbf{x}) - \bar{g}(\mathbf{x}))^2],$$

which measures the variation in the final hypothesis, depending on the data set. We thus arrive at the bias-variance decomposition of out-of-sample error,

$$\begin{aligned}\mathbb{E}_{\mathcal{D}}[E_{\text{out}}(g^{(\mathcal{D})})] &= \mathbb{E}_{\mathbf{x}}[\mathsf{bias}(\mathbf{x}) + \mathsf{var}(\mathbf{x})] \\ &= \mathsf{bias} + \mathsf{var},\end{aligned}$$

where $\mathsf{bias} = \mathbb{E}_{\mathbf{x}}[\mathsf{bias}(\mathbf{x})]$ and $\mathsf{var} = \mathbb{E}_{\mathbf{x}}[\mathsf{var}(\mathbf{x})]$. Our derivation assumed that the data was noiseless. A similar derivation with noise in the data would lead to an additional noise term in the out-of-sample error (Problem 2.22). The noise term is unavoidable no matter what we do, so the terms we are interested in are really the bias and var.

The approximation-generalization tradeoff is captured in the bias-variance decomposition. To illustrate, let's consider two extreme cases: a very small model (with one hypothesis) and a very large one with all hypotheses.

Very small model. Since there is only one hypothesis, both the average function $\bar{g}$ and the final hypothesis $g^{(\mathcal{D})}$ will be the same, for any data set. Thus, $\mathsf{var} = 0$. The bias will depend solely on how well this single hypothesis approximates the target f, and unless we are extremely lucky, we expect a large bias.

Very large model. The target function is in $\mathcal{H}$. Different data sets will lead to different hypotheses that agree with f on the data set, and are spread around f in the red region. Thus, $\mathsf{bias} \approx 0$ because $\bar{g}$ is likely to be close to f. The var is large (heuristically represented by the size of the red region in the figure).

One can also view the variance as a measure of 'instability' in the learning model. Instability manifests in wild reactions to small variations or idiosyncrasies in the data, resulting in vastly different hypotheses.

[5] What we call bias is sometimes called bias^2 in the literature.

Example 2.8. Consider a target function $f(x) = \sin(\pi x)$ and a data set of size $N = 2$. We sample x uniformly in $[-1, 1]$ to generate a data set $(x_1, y_1), (x_2, y_2)$; and fit the data using one of two models:

$\mathcal{H}_0$: Set of all lines of the form $h(x) = b$;

$\mathcal{H}_1$: Set of all lines of the form $h(x) = ax + b$.

For $\mathcal{H}_0$, we choose the constant hypothesis that best fits the data (the horizontal line at the midpoint, $b = \frac{y_1+y_2}{2}$). For $\mathcal{H}_1$, we choose the line that passes through the two data points (x_1, y_1) and (x_2, y_2). Repeating this process with many data sets, we can estimate the bias and the variance. The figures which follow show the resulting fits on the same (random) data sets for both models.

$\mathcal{H}_0$ $\mathcal{H}_1$

With $\mathcal{H}_1$, the learned hypothesis is wilder and varies extensively depending on the data set. The bias-var analysis is summarized in the next figures.

$\mathcal{H}_0$
bias $= 0.50$;
var $= 0.25$.

$\mathcal{H}_1$
bias $= 0.21$;
var $= 1.69$.

Average hypothesis $\bar{g}$ (red) with $\mathsf{var}(x)$ indicated by the gray shaded region that is $\bar{g}(x) \pm \sqrt{\mathsf{var}(x)}$.

For $\mathcal{H}_1$, the average hypothesis $\bar{g}$ (red line) is a reasonable fit with a fairly small bias of 0.21. However, the large variability leads to a high var of 1.69 resulting in a large expected out-of-sample error of 1.90. With the simpler

model $\mathcal{H}_0$, the fits are much less volatile and we have a significantly lower var of 0.25, as indicated by the shaded region. However, the average fit is now the zero function, resulting in a higher bias of 0.50. The total out-of-sample error has a much smaller expected value of 0.75. The simpler model wins by significantly decreasing the var at the expense of a smaller increase in bias.

Notice that we are not comparing how well the red curves (the average hypotheses) fit the sine. These curves are only conceptual, since in real learning we do not have access to the multitude of data sets needed to generate them. We have one data set, and the simpler model results in a better out-of-sample error on average as we fit our model to just this one data. However, the var term decreases as N increases, so if we get a bigger and bigger data set, the bias term will be the dominant part of E_{out}, and $\mathcal{H}_1$ will win. □

The learning algorithm plays a role in the bias-variance analysis that it did not play in the VC analysis. Two points are worth noting.

1. By design, the VC analysis is based purely on the hypothesis set $\mathcal{H}$, independently of the learning algorithm $\mathcal{A}$. In the bias-variance analysis, both $\mathcal{H}$ and the algorithm $\mathcal{A}$ matter. With the same $\mathcal{H}$, using a different learning algorithm can produce a different $g^{(\mathcal{D})}$. Since $g^{(\mathcal{D})}$ is the building block of the bias-variance analysis, this may result in different bias and var terms.

2. Although the bias-variance analysis is based on squared-error measure, the learning algorithm itself does not have to be based on minimizing the squared error. It can use any criterion to produce $g^{(\mathcal{D})}$ based on $\mathcal{D}$. However, once the algorithm produces $g^{(\mathcal{D})}$, we measure its bias and variance using squared error.

Unfortunately, the bias and variance cannot be computed in practice, since they depend on the target function and the input probability distribution (both unknown). Thus, the bias-variance decomposition is a conceptual tool which is helpful when it comes to developing a model. There are two typical goals when we consider bias and variance. The first is to try to lower the variance without significantly increasing the bias, and the second is to lower the bias without significantly increasing the variance. These goals are achieved by different techniques, some principled and some heuristic. Regularization is one of these techniques that we will discuss in Chapter 4. Reducing the bias without increasing the variance requires some prior information regarding the target function to steer the selection of $\mathcal{H}$ in the direction of f, and this task is largely application-specific. On the other hand, reducing the variance without compromising the bias can be done through general techniques.

2.3.2 The Learning Curve

We close this chapter with an important plot that illustrates the tradeoffs that we have seen so far. The *learning curves* summarize the behavior of the

in-sample and out-of-sample errors as we vary the size of the training set.

After learning with a particular data set $\mathcal{D}$ of size N, the final hypothesis $g^{(\mathcal{D})}$ has in-sample error $E_{\text{in}}(g^{(\mathcal{D})})$ and out-of-sample error $E_{\text{out}}(g^{(\mathcal{D})})$, both of which depend on $\mathcal{D}$. As we saw in the bias-variance analysis, the expectation with respect to all data sets of size N gives the expected errors: $\mathbb{E}_{\mathcal{D}}[E_{\text{in}}(g^{(\mathcal{D})})]$ and $\mathbb{E}_{\mathcal{D}}[E_{\text{out}}(g^{(\mathcal{D})})]$. These expected errors are functions of N, and are called the learning curves of the model. We illustrate the learning curves for a simple learning model and a complex one, based on actual experiments.

Simple Model **Complex Model**

Notice that for the simple model, the learning curves converge more quickly but to worse ultimate performance than for the complex model. This behavior is typical in practice. For both simple and complex models, the out-of-sample learning curve is decreasing in N, while the in-sample learning curve is increasing in N. Let us take a closer look at these curves and interpret them in terms of the different approaches to generalization that we have discussed.

In the VC analysis, E_{out} was expressed as the sum of E_{in} and a generalization error that was bounded by Ω, the penalty for model complexity. In the bias-variance analysis, E_{out} was expressed as the sum of a bias and a variance. The following learning curves illustrate these two approaches side by side.

VC Analysis **Bias-Variance Analysis**

The VC analysis bounds the generalization error which is illustrated on the left.[6] The bias-variance analysis is illustrated on the right. The bias-variance illustration is somewhat idealized, since it assumes that, for every N, the average learned hypothesis $\bar{g}$ has the same performance as the best approximation to f in the learning model.

When the number of data points increases, we move to the right on the learning curves and both the generalization error and the variance term shrink, as expected. The learning curve also illustrates an important point about E_{in}. As N increases, E_{in} edges toward the smallest error that the learning model can achieve in approximating f. For small N, the value of E_{in} is actually smaller than that 'smallest possible' error. This is because the learning model has an easier task for smaller N; it only needs to approximate f on the N points regardless of what happens outside those points. Therefore, it can achieve a superior fit on those points, albeit at the expense of an inferior fit on the rest of the points as shown by the corresponding value of E_{out}.

[6] For the learning curve, we take the expected values of all quantities with respect to $\mathcal{D}$ of size N.

2.4 Problems

Problem 2.1 In Equation (2.1), set $\delta = 0.03$ and let

$$\epsilon(M, N, \delta) = \sqrt{\frac{1}{2N} \ln \frac{2M}{\delta}}.$$

(a) For $M = 1$, how many examples do we need to make $\epsilon \leq 0.05$?

(b) For $M = 100$, how many examples do we need to make $\epsilon \leq 0.05$?

(c) For $M = 10,000$, how many examples do we need to make $\epsilon \leq 0.05$?

Problem 2.2 Show that for the learning model of positive rectangles (aligned horizontally or vertically), $m_{\mathcal{H}}(4) = 2^4$ and $m_{\mathcal{H}}(5) < 2^5$. Hence, give a bound for $m_{\mathcal{H}}(N)$.

Problem 2.3 Compute the maximum number of dichotomies, $m_{\mathcal{H}}(N)$, for these learning models, and consequently compute d_{vc}, the VC dimension.

(a) Positive or negative ray: $\mathcal{H}$ contains the functions which are $+1$ on $[a, \infty)$ (for some a) together with those that are $+1$ on $(-\infty, a]$ (for some a).

(b) Positive or negative interval: $\mathcal{H}$ contains the functions which are $+1$ on an interval $[a, b]$ and -1 elsewhere or -1 on an interval $[a, b]$ and $+1$ elsewhere.

(c) Two concentric spheres in $\mathbb{R}^d$: $\mathcal{H}$ contains the functions which are $+1$ for $a \leq \sqrt{x_1^2 + \ldots + x_d^2} \leq b$.

Problem 2.4 Show that $B(N, k) = \sum_{i=0}^{k-1} \binom{N}{i}$ by showing the other direction to Lemma 2.3, namely that

$$B(N, k) \geq \sum_{i=0}^{k-1} \binom{N}{i}.$$

To do so, construct a specific set of $\sum_{i=0}^{k-1} \binom{N}{i}$ dichotomies that does not shatter any subset of k variables. *[Hint: Try limiting the number of -1's in each dichotomy.]*

Problem 2.5 Prove by induction that $\sum_{i=0}^{D} \binom{N}{i} \leq N^D + 1$, hence

$$m_{\mathcal{H}}(N) \leq N^{d_{\text{vc}}} + 1.$$

Problem 2.6 Prove that for $N \geq d$,

$$\sum_{i=0}^{d} \binom{N}{i} \leq \left(\frac{eN}{d}\right)^d.$$

We suggest you first show the following intermediate steps.

(a) $\sum_{i=0}^{d} \binom{N}{i} \leq \sum_{i=0}^{d} \binom{N}{i} \left(\frac{N}{d}\right)^{d\ \ i} \leq \left(\frac{N}{d}\right)^d \sum_{i=0}^{N} \binom{N}{i} \left(\frac{d}{N}\right)^i$.

(b) $\sum_{i=0}^{N} \binom{N}{i} \left(\frac{d}{N}\right)^i \leq e^d$. *[Hints: Binomial theorem;* $\left(1+\frac{1}{x}\right)^x \leq e$ *for* $x > 0$.*]*

Hence, argue that $m_{\mathcal{H}}(N) \leq \left(\frac{eN}{d_{\text{VC}}}\right)^{d_{\text{VC}}}$.

Problem 2.7 Plot the bounds for $m_{\mathcal{H}}(N)$ given in Problems 2.5 and 2.6 for $d_{\text{VC}} = 2$ and $d_{\text{VC}} = 5$. When do you prefer one bound over the other?

Problem 2.8 Which of the following are possible growth functions $m_{\mathcal{H}}(N)$ for some hypothesis set:

$$1+N;\ 1+N+\frac{N(N-1)}{2};\ 2^N;\ 2^{\lfloor\sqrt{N}\rfloor};\ 2^{\lfloor N/2\rfloor};\ 1+N+\frac{N(N-1)(N-2)}{6}.$$

Problem 2.9 **[hard]** For the perceptron in d dimensions, show that

$$m_{\mathcal{H}}(N) = 2\sum_{i=0}^{d} \binom{N-1}{i}.$$

[Hint: Cover(1965) in Further Reading.]

Use this formula to verify that $d_{\text{VC}} = d+1$ by evaluating $m_{\mathcal{H}}(d+1)$ and $m_{\mathcal{H}}(d+2)$. Plot $m_{\mathcal{H}}(N)/2^N$ for $d = 10$ and $N \in [1, 40]$. If you generate a random dichotomy on N points in 10 dimensions, give an upper bound on the probability that the dichotomy will be separable for $N = 10, 20, 40$.

Problem 2.10 Show that $m_{\mathcal{H}}(2N) \leq m_{\mathcal{H}}(N)^2$, and hence obtain a generalization bound which only involves $m_{\mathcal{H}}(N)$.

Problem 2.11 Suppose $m_{\mathcal{H}}(N) = N+1$, so $d_{\text{VC}} = 1$. You have 100 training examples. Use the generalization bound to give a bound for E_{out} with confidence 90%. Repeat for $N = 10,000$.

Problem 2.12 For an $\mathcal{H}$ with $d_{\text{vc}} = 10$, what sample size do you need (as prescribed by the generalization bound) to have a 95% confidence that your generalization error is at most 0.05?

Problem 2.13

(a) Let $\mathcal{H} = \{h_1, h_2, \ldots, h_M\}$ with some finite M. Prove that $d_{\text{vc}}(\mathcal{H}) \leq \log_2 M$.

(b) For hypothesis sets $\mathcal{H}_1$, $\mathcal{H}_2$, $\cdots$, $\mathcal{H}_K$ with finite VC dimensions $d_{\text{vc}}(\mathcal{H}_k)$, derive and prove the tightest upper and lower bound that you can get on $d_{\text{vc}}\left(\cap_{k=1}^{K} \mathcal{H}_k\right)$.

(c) For hypothesis sets $\mathcal{H}_1$, $\mathcal{H}_2$, $\cdots$, $\mathcal{H}_K$ with finite VC dimensions $d_{\text{vc}}(\mathcal{H}_k)$, derive and prove the tightest upper and lower bounds that you can get on $d_{\text{vc}}\left(\cup_{k=1}^{K} \mathcal{H}_k\right)$.

Problem 2.14 Let $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_K$ be K hypothesis sets with finite VC dimension d_{vc}. Let $\mathcal{H} = \mathcal{H}_1 \cup \mathcal{H}_2 \cup \cdots \cup \mathcal{H}_K$ be the union of these models.

(a) Show that $d_{\text{vc}}(\mathcal{H}) < K(d_{\text{vc}} + 1)$.

(b) Suppose that ℓ satisfies $2^\ell > 2K\ell^{d_{\text{vc}}}$. Show that $d_{\text{vc}}(\mathcal{H}) \leq \ell$.

(c) Hence, show that

$$d_{\text{vc}}(\mathcal{H}) \quad \leq \quad \min\big(K(d_{\text{vc}} + 1), 7(d_{\text{vc}} + K)\log_2(d_{\text{vc}}K)\big).$$

That is, $d_{\text{vc}}(\mathcal{H}) = O\big(\max(d_{\text{vc}}, K)\log_2 \max(d_{\text{vc}}, K)\big)$ is not too bad.

Problem 2.15 The monotonically increasing hypothesis set is

$$\mathcal{H} = \{h \mid \mathbf{x}_1 \geq \mathbf{x}_2 \implies h(\mathbf{x}_1) \geq h(\mathbf{x}_2)\},$$

where $\mathbf{x}_1 \geq \mathbf{x}_2$ if and only if the inequality is satisfied for every component.

(a) Give an example of a monotonic classifier in two dimensions, clearly showing the $+1$ and -1 regions.

(b) Compute $m_{\mathcal{H}}(N)$ and hence the VC dimension. *[Hint: Consider a set of N points generated by first choosing one point, and then generating the next point by increasing the first component and decreasing the second component until N points are obtained.]*

Problem 2.16 In this problem, we will consider $\mathcal{X} = \mathbb{R}$. That is, $\mathbf{x}$ x is a one dimensional variable. For a hypothesis set

$$\mathcal{H} = \left\{ h_{\mathbf{c}} \;\middle|\; h_{\mathbf{c}}(x) = \text{sign}\left(\sum_{i=0}^{D} c_i x^i \right) \right\},$$

prove that the VC dimension of $\mathcal{H}$ is exactly $(D+1)$ by showing that

(a) There are $(D+1)$ points which are shattered by $\mathcal{H}$.

(b) There are no $(D+2)$ points which are shattered by $\mathcal{H}$.

Problem 2.17 The VC dimension depends on the input space as well as $\mathcal{H}$. For a fixed $\mathcal{H}$, consider two input spaces $\mathcal{X}_1 \subseteq \mathcal{X}_2$. Show that the VC dimension of $\mathcal{H}$ with respect to input space $\mathcal{X}_1$ is at most the VC dimension of $\mathcal{H}$ with respect to input space $\mathcal{X}_2$.

How can the result of this problem be used to answer part (b) in Problem 2.16? *[Hint: How is Problem 2.16 related to a perceptron in D dimensions?]*

Problem 2.18 The VC dimension of the perceptron hypothesis set corresponds to the number of parameters $(w_0, w_1, \cdots, w_d)$ of the set, and this observation is 'usually' true for other hypothesis sets. However, we will present a counter example here. Prove that the following hypothesis set for $x \in \mathbb{R}$ has an infinite VC dimension:

$$\mathcal{H} = \left\{ h_\alpha \;\middle|\; h_\alpha(x) = (-1)^{\lfloor \alpha x \rfloor}, \text{ where } \alpha \in \mathbb{R} \right\},$$

where $\lfloor A \rfloor$ is the biggest integer $\leq A$ (the floor function). This hypothesis has only one parameter α but 'enjoys' an infinite VC dimension. *[Hint: Consider $x_1, \ldots, x_N$, where x_n 10^n, and show how to implement an arbitrary dichotomy $y_1, \ldots, y_N$.]*

Problem 2.19 This problem derives a bound for the VC dimension of a complex hypothesis set that is built from simpler hypothesis sets via composi tion. Let $\mathcal{H}_1, \ldots, \mathcal{H}_K$ be hypothesis sets with VC dimension $d_1, \ldots, d_K$. Fix $h_1, \ldots, h_K$, where $h_i \in \mathcal{H}_i$. Define a vector $\mathbf{z}$ obtained from $\mathbf{x}$ to have components $h_i(\mathbf{x})$. Note that $\mathbf{x} \in \mathbb{R}^d$, but $\mathbf{z} \in \{-1, +1\}^K$. Let $\tilde{\mathcal{H}}$ be a hypothesis set of functions that take inputs in $\mathbb{R}^K$. So

$$\tilde{h} \in \tilde{\mathcal{H}}\colon \mathbf{z} \in \mathbb{R}^K \mapsto \{+1, -1\},$$

and suppose that $\tilde{\mathcal{H}}$ has VC dimension $\tilde{d}$.

We can apply a hypothesis in $\tilde{\mathcal{H}}$ to the $\mathbf{z}$ constructed from $(h_1, \ldots, h_K)$. This is the composition of the hypothesis set $\tilde{\mathcal{H}}$ with $(\mathcal{H}_1, \ldots, \mathcal{H}_K)$. More formally, the composed hypothesis set $\mathcal{H} = \tilde{\mathcal{H}} \circ (\mathcal{H}_1, \ldots, \mathcal{H}_K)$ is defined by $h \in \mathcal{H}$ if

$$h(\mathbf{x}) = \tilde{h}(h_1(\mathbf{x}), \ldots, h_K(\mathbf{x})), \qquad \tilde{h} \in \tilde{\mathcal{H}};\ h_i \in \mathcal{H}_i.$$

(a) Show that

$$m_{\mathcal{H}}(N) \leq m_{\tilde{\mathcal{H}}}(N) \prod_{i=1}^{K} m_{\mathcal{H}_i}(N). \tag{2.18}$$

[Hint: Fix N points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and fix $h_1, \ldots, h_K$. This generates N transformed points $\mathbf{z}_1, \ldots, \mathbf{z}_N$. These $\mathbf{z}_1, \ldots, \mathbf{z}_N$ can be dichotomized in at most $m_{\tilde{\mathcal{H}}}(N)$ ways, hence for fixed $(h_1, \ldots, h_K)$, $(\mathbf{x}_1, \ldots, \mathbf{x}_N)$ can be dichotomized in at most $m_{\tilde{\mathcal{H}}}(N)$ ways. Through the eyes of $\mathbf{x}_1, \ldots, \mathbf{x}_N$, at most how many hypotheses are there (effectively) in $\mathcal{H}_i$? Use this bound to bound the effective number of K-tuples $(h_1, \ldots, h_K)$ that need to be considered. Finally, argue that you can bound the number of dichotomies that can be implemented by the product of the number of possible K-tuples $(h_1, \ldots, h_K)$ and the number of dichotomies per K-tuple.]

(b) Use the bound $m(N) \leq \left(\frac{eN}{d_{\text{VC}}}\right)^{d_{\text{VC}}}$ to get a bound for $m_{\mathcal{H}}(N)$ in terms of $\tilde{d}, d_1, \ldots, d_K$.

(c) Let $D = \tilde{d} + \sum_{i=1}^{K} d_i$, and assume that $D > 2e \log_2 D$. Show that

$$d_{\text{VC}}(\mathcal{H}) \leq 2D \log_2 D.$$

(d) If $\mathcal{H}_i$ and $\tilde{\mathcal{H}}$ are all perceptron hypothesis sets, show that

$$d_{\text{VC}}(\mathcal{H}) = O(dK \log(dK)).$$

In the next chapter, we will further develop the simple linear model. This linear model is the building block of many other models, such as neural networks. The results of this problem show how to bound the VC dimension of the more complex models built in this manner.

Problem 2.20 There are a number of bounds on the generalization error ϵ, all holding with probability at least $1 - \delta$.

(a) Original VC-bound:

$$\epsilon \leq \sqrt{\frac{8}{N} \ln \frac{4m_{\mathcal{H}}(2N)}{\delta}}.$$

(b) Rademacher Penalty Bound:

$$\epsilon \leq \sqrt{\frac{2\ln(2Nm_{\mathcal{H}}(N))}{N}} + \sqrt{\frac{2}{N} \ln \frac{1}{\delta}} + \frac{1}{N}.$$

(continued on next page)

(c) Parrondo and Van den Broek:

$$\epsilon \leq \sqrt{\frac{1}{N}\left(2\epsilon + \ln\frac{6m_{\mathcal{H}}(2N)}{\delta}\right)}.$$

(d) Devroye:

$$\epsilon \leq \sqrt{\frac{1}{2N}\left(4\epsilon(1+\epsilon) + \ln\frac{4m_{\mathcal{H}}(N^2)}{\delta}\right)}.$$

Note that (c) and (d) are implicit bounds in ϵ. Fix $d_{\text{VC}} = 50$ and $\delta = 0.05$ and plot these bounds as a function of N. Which is best?

Problem 2.21 Assume the following theorem to hold

Theorem

$$\mathbb{P}\left[\frac{E_{\text{out}}(\text{g}) - E_{\text{in}}(\text{g})}{\sqrt{E_{\text{out}}(\text{g})}} > \epsilon\right] \leq c \cdot m_{\mathcal{H}}(2N)\exp\left(-\frac{\epsilon^2 N}{4}\right),$$

where c is a constant that is a little bigger than 6.

This bound is useful because sometimes what we care about is not the absolute generalization error but instead a relative generalization error (one can imagine that a generalization error of 0.01 is more significant when $E_{\text{out}} = 0.01$ than when $E_{\text{out}} = 0.5$). Convert this to a generalization bound by showing that with probability at least $1 - \delta$,

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

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

Problem 2.22 When there is noise in the data, $E_{\text{out}}(g^{(\mathcal{D})}) = \mathbb{E}_{\mathbf{x},y}[(g^{(\mathcal{D})}(\mathbf{x}) - y(\mathbf{x}))^2]$, where $y(\mathbf{x}) = f(\mathbf{x}) + \epsilon$. If ϵ is a zero mean noise random variable with variance σ^2, show that the bias variance decomposition becomes

$$\mathbb{E}_{\mathcal{D}}[E_{\text{out}}(g^{(\mathcal{D})})] = \sigma^2 + \text{bias} + \text{var}.$$

Problem 2.23 Consider the learning problem in Example 2.8, where the input space is $\mathcal{X} = [-1, +1]$, the target function is $f(x) = \sin(\pi x)$, and the input probability distribution is uniform on $\mathcal{X}$. Assume that the training set $\mathcal{D}$ has only two data points (picked independently), and that the learning algorithm picks the hypothesis that minimizes the in sample mean squared error. In this problem, we will dig deeper into this case.

For each of the following learning models, find (analytically or numerically) (i) the best hypothesis that approximates f in the mean squared error sense (assume that f is known for this part), (ii) the expected value (with respect to $\mathcal{D}$) of the hypothesis that the learning algorithm produces, and (iii) the expected out of sample error and its bias and var components.

(a) The learning model consists of all hypotheses of the form $h(x) = ax + b$ (if you need to deal with the infinitesimal probability case of two identical data points, choose the hypothesis tangential to f).

(b) The learning model consists of all hypotheses of the form $h(x) = ax$. This case was not covered in Example 2.8.

(c) The learning model consists of all hypotheses of the form $h(x) = b$.

Problem 2.24 Consider a simplified learning scenario. Assume that the input dimension is one. Assume that the input variable x is uniformly distributed in the interval $[-1, 1]$. The data set consists of 2 points $\{x_1, x_2\}$ and assume that 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 learning algorithm returns the line fitting these two points as g ($\mathcal{H}$ consists of functions of the form $h(x) = ax + b$). We are interested in the test performance (E_{out}) of our learning system with respect to the squared error measure, the bias and the var.

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

(b) Describe an experiment that you could run to determine (numerically) $\bar{g}(x)$, E_{out}, bias, and var.

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

(d) Compute analytically what E_{out}, bias and var should be.

Chapter 3

The Linear Model

We often wonder how to draw a line between two categories; right versus wrong, personal versus professional life, useful email versus spam, to name a few. A line is intuitively our first choice for a decision boundary. In learning, as in life, a line is also a good first choice.

In Chapter 1, we (and the machine ☺) learned a procedure to 'draw a line' between two categories based on data (the perceptron learning algorithm). We started by taking the hypothesis set $\mathcal{H}$ that included all possible lines (actually hyperplanes). The algorithm then searched for a good line in $\mathcal{H}$ by iteratively correcting the errors made by the current candidate line, in an attempt to improve E_{in}. As we saw in Chapter 2, the linear model set of lines has a small VC dimension and so is able to generalize well from E_{in} to E_{out}.

The aim of this chapter is to further develop the basic linear model into a powerful tool for learning from data. We branch into three important problems: the classification problem that we have seen and two other important problems called *regression* and *probability estimation*. The three problems come with different but related algorithms, and cover a lot of territory in learning from data. As a rule of thumb, when faced with learning problems, it is generally a winning strategy to try a linear model first.

3.1 Linear Classification

The linear model for classifying data into two classes uses a hypothesis set of linear classifiers, where each h has the form

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}),$$

for some column vector $\mathbf{w} \in \mathbb{R}^{d+1}$, where d is the dimensionality of the input space, and the added coordinate $x_0 = 1$ corresponds to the bias 'weight' w_0 (recall that the input space $\mathcal{X} = \{1\} \times \mathbb{R}^d$ is considered d-dimensional since the added coordinate $x_0 = 1$ is fixed). We will use h and $\mathbf{w}$ interchangeably

to refer to the hypothesis when the context is clear. When we left Chapter 1, we had two basic criteria for learning:

1. Can we make sure that $E_{\text{out}}(g)$ is close to $E_{\text{in}}(g)$? This ensures that what we have learned in sample will generalize out of sample.

2. Can we make $E_{\text{in}}(g)$ small? This ensures that what we have learned in sample is a good hypothesis.

The first criterion was studied in Chapter 2. Specifically, the VC dimension of the linear model is only $d+1$ (Exercise 2.4). Using the VC generalization bound (2.12), and the bound (2.10) on the growth function in terms of the VC dimension, we conclude that with high probability,

$$E_{\text{out}}(g) = E_{\text{in}}(g) + O\left(\sqrt{\frac{d}{N}\ln N}\right). \tag{3.1}$$

Thus, when N is sufficiently large, E_{in} and E_{out} will be close to each other (see the definition of $O(\cdot)$ in the Notation table), and the first criterion for learning is fulfilled.

The second criterion, making sure that E_{in} is small, requires first and foremost that there is *some* linear hypothesis that has small E_{in}. If there isn't such a linear hypothesis, then learning certainly can't find one. So, let's suppose for the moment that there is a linear hypothesis with small E_{in}. In fact, let's suppose that the data is linearly separable, which means there is some hypothesis $\mathbf{w}^*$ with $E_{\text{in}}(\mathbf{w}^*) = 0$. We will deal with the case when this is not true shortly.

In Chapter 1, we introduced the perceptron learning algorithm (PLA). Start with an arbitrary weight vector $\mathbf{w}(0)$. Then, at every time step $t \geq 0$, select *any* misclassified data point $(\mathbf{x}(t), y(t))$, and update $\mathbf{w}(t)$ as follows:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + y(t)\mathbf{x}(t).$$

The intuition is that the update is attempting to correct the error in classifying $\mathbf{x}(t)$. The remarkable thing is that this incremental approach of learning based on one data point at a time works. As discussed in Problem 1.3, it can be proved that the PLA will eventually stop updating, ending at a solution $\mathbf{w}_{\text{PLA}}$ with $E_{\text{in}}(\mathbf{w}_{\text{PLA}}) = 0$. Although this result applies to a restricted setting (linearly separable data), it is a significant step. The PLA is clever – it doesn't naïvely test every linear hypothesis to see if it (the hypothesis) separates the data; that would take infinitely long. Using an iterative approach, the PLA manages to search an *infinite* hypothesis set and output a linear separator in (provably) *finite* time.

As far as PLA is concerned, linear separability is a property of the *data*, not the *target*. A linearly separable $\mathcal{D}$ could have been generated either from a linearly separable target, or (by chance) from a target that is not linearly separable. The convergence proof of PLA guarantees that the algorithm will

(a) Few noisy data. (b) Nonlinearly separable.

Figure 3.1: Data sets that are not linearly separable but are (a) linearly separable after discarding a few examples, or (b) separable by a more so phisticated curve.

work in both these cases, and produce a hypothesis with $E_{\text{in}} = 0$. Further, in both cases, you can be confident that this performance will generalize well out of sample, according to the VC bound.

Exercise 3.1

Will PLA ever stop updating if the data is not linearly separable?

3.1.1 Non-Separable Data

We now address the case where the data is not linearly separable. Figure 3.1 shows two data sets that are not linearly separable. In Figure 3.1(a), the data becomes linearly separable after the removal of just two examples, which could be considered noisy examples or outliers. In Figure 3.1(b), the data can be separated by a circle rather than a line. In both cases, there will always be a misclassified training example if we insist on using a linear hypothesis, and hence PLA will never terminate. In fact, its behavior becomes quite unstable, and can jump from a good perceptron to a very bad one within one update; the quality of the resulting E_{in} cannot be guaranteed. In Figure 3.1(a), it seems appropriate to stick with a line, but to somehow tolerate noise and output a hypothesis with a small E_{in}, not necessarily $E_{\text{in}} = 0$. In Figure 3.1(b), the linear model does not seem to be the correct model in the first place, and we will discuss a technique called nonlinear transformation for this situation in Section 3.4.

The situation in Figure 3.1(a) is actually encountered very often: even though a linear classifier seems appropriate, the data may not be linearly separable because of outliers or noise. To find a hypothesis with the minimum E_{in}, we need to solve the combinatorial optimization problem:

$$\min_{\mathbf{w}\in\mathbb{R}^{d+1}} \underbrace{\frac{1}{N}\sum_{n=1}^{N} [\![\text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) \neq y_n]\!]}_{E_{\text{in}}(\mathbf{w})}. \tag{3.2}$$

The difficulty in solving this problem arises from the discrete nature of both $\text{sign}(\cdot)$ and $[\![\cdot]\!]$. In fact, minimizing $E_{\text{in}}(\mathbf{w})$ in (3.2) in the general case is known to be NP-hard, which means there is no known efficient algorithm for it, and if you discovered one, you would become really, really famous ☺. Thus, one has to resort to approximately minimizing E_{in}.

One approach for getting an approximate solution is to extend PLA through a simple modification into what is called the *pocket algorithm.* Essentially, the pocket algorithm keeps 'in its pocket' the best weight vector encountered up to iteration t in PLA. At the end, the best weight vector will be reported as the final hypothesis. This simple algorithm is shown below.

The pocket algorithm:

1: Set the pocket weight vector $\hat{\mathbf{w}}$ to $\mathbf{w}(0)$ of PLA.
2: **for** $t = 0, \ldots, T \quad 1$ **do**
3: Run PLA for one update to obtain $\mathbf{w}(t+1)$.
4: Evaluate $E_{\text{in}}(\mathbf{w}(t+1))$.
5: If $\mathbf{w}(t+1)$ is better than $\hat{\mathbf{w}}$ in terms of E_{in}, set $\hat{\mathbf{w}}$ to $\mathbf{w}(t+1)$.
6: Return $\hat{\mathbf{w}}$.

The original PLA only checks *some* of the examples using $\mathbf{w}(t)$ to identify $(\mathbf{x}(t), y(t))$ in each iteration, while the pocket algorithm needs an additional step that evaluates *all* examples using $\mathbf{w}(t+1)$ to get $E_{\text{in}}(\mathbf{w}(t+1))$. The additional step makes the pocket algorithm much slower than PLA. In addition, there is no guarantee for how fast the pocket algorithm can converge to a good E_{in}. Nevertheless, it is a useful algorithm to have on hand because of its simplicity. Other, more efficient approaches for obtaining good approximate solutions have been developed based on different optimization techniques, as shown later in this chapter.

Exercise 3.2

Take $d = 2$ and create a data set $\mathcal{D}$ of size $N = 100$ that is not linearly separable. You can do so by first choosing a random line in the plane as your target function and the inputs $\mathbf{x}_n$ of the data set as random points in the plane. Then, evaluate the target function on each $\mathbf{x}_n$ to get the corresponding output y_n. Finally, flip the labels of $\frac{N}{10}$ randomly selected y_n's and the data set will likely become non separable.

Now, try the pocket algorithm on your data set using $T = 1,000$ iterations. Repeat the experiment 20 times. Then, plot the average $E_{\text{in}}(\mathbf{w}(t))$ and the average $E_{\text{in}}(\hat{\mathbf{w}})$ (which is also a function of t) on the same figure and see how they behave when t increases. Similarly, use a test set of size $1,000$ and plot a figure to show how $E_{\text{out}}(\mathbf{w}(t))$ and $E_{\text{out}}(\hat{\mathbf{w}})$ behave.

Example 3.1 (Handwritten digit recognition)**.** We sample some digits from the US Postal Service Zip Code Database. These 16×16 pixel images are preprocessed from the scanned handwritten zip codes. The goal is to recognize the digit in each image. We alluded to this task in part (b) of Exercise 1.1. A quick look at the images reveals that this is a non-trivial task (even for a human), and typical human E_{out} is about 2.5%. Common confusion occurs between the digits $\{4, 9\}$ and $\{2, 7\}$. A machine-learned hypothesis which can achieve such an error rate would be highly desirable.

Let's first decompose the big task of separating ten digits into smaller tasks of separating two of the digits. Such a decomposition approach from *multiclass* to *binary* classification is commonly used in many learning algorithms. We will focus on digits $\{1, 5\}$ for now. A human approach to determining the digit corresponding to an image is to look at the shape (or other properties) of the black pixels. Thus, rather than carrying all the information in the 256 pixels, it makes sense to summarize the information contained in the image into a few *features*. Let's look at two important features here: intensity and symmetry. Digit 5 usually occupies more black pixels than digit 1, and hence the average pixel intensity of digit 5 is higher. On the other hand, digit 1 is symmetric while digit 5 is not. Therefore, if we define asymmetry as the average absolute difference between an image and its flipped versions, and symmetry as the negation of asymmetry, digit 1 would result in a higher symmetry value. A scatter plot for these intensity and symmetry features for some of the digits is shown next.

While the digits can be roughly separated by a line in the plane representing these two features, there are poorly written digits (such as the '5' depicted in the top-left corner) that prevent a perfect linear separation.

We now run PLA and pocket on the data set and see what happens. Since the data set is not linearly separable, PLA will not stop updating. In fact, as can be seen in Figure 3.2(a), its behavior can be quite unstable. When it is forcibly terminated at iteration $1,000$, PLA gives a line that has a poor $E_{\text{in}} = 2.24\%$ and $E_{\text{out}} = 6.37\%$. On the other hand, if the pocket algorithm is applied to the same data set, as shown in Figure 3.2(b), we can obtain a line that has a better $E_{\text{in}} = 0.45\%$ and a better $E_{\text{out}} = 1.89\%$. □

3.2 Linear Regression

Linear regression is another useful linear model that applies to real-valued target functions.[1] It has a long history in statistics, where it has been studied in great detail, and has various applications in social and behavioral sciences. Here, we discuss linear regression from a learning perspective, where we derive the main results with minimal assumptions.

Let us revisit our application in credit approval, this time considering a regression problem rather than a classification problem. Recall that the bank has customer records that contain information fields related to personal credit, such as annual salary, years in residence, outstanding loans, etc. Such variables can be used to learn a linear classifier to decide on credit approval. Instead of just making a binary decision (approve or not), the bank also wants to set a proper credit limit for each approved customer. Credit limits are traditionally determined by human experts. The bank wants to automate this task, as it did with credit approval.

[1] Regression, a term inherited from earlier work in statistics, means y is real valued.

(a) PLA (b) Pocket

Figure 3.2: Comparison of two linear classification algorithms for sep arating digits 1 and 5. E_{in} and E_{out} are plotted versus iteration number and below that is the learned hypothesis g. (a) A version of the PLA which selects a random training example and updates $\mathbf{w}$ if that example is misclas sified (hence the flat regions when no update is made). This version avoids searching all the data at every iteration. (b) The pocket algorithm.

This is a regression learning problem. The bank uses historical records to construct a data set $\mathcal{D}$ of examples $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)$, where $\mathbf{x}_n$ is customer information and y_n is the credit limit set by one of the human experts in the bank. Note that y_n is now a real number (positive in this case) instead of just a binary value ± 1. The bank wants to use learning to find a hypothesis g that replicates how human experts determine credit limits.

Since there is more than one human expert, and since each expert may not be perfectly consistent, our target will not be a deterministic function $y = f(\mathbf{x})$. Instead, it will be a noisy target formalized as a distribution of the random variable y that comes from the different views of different experts as well as the variation within the views of each expert. That is, the label y_n comes from some distribution $P(y \mid \mathbf{x})$ instead of a deterministic function $f(\mathbf{x})$. Nonetheless, as we discussed in previous chapters, the nature of the problem is not changed. We have an *unknown* distribution $P(\mathbf{x}, y)$ that generates

each $(\mathbf{x}_n, y_n)$, and we want to find a hypothesis g that minimizes the error between $g(\mathbf{x})$ and y with respect to that distribution.

The choice of a linear model for this problem presumes that there is a linear combination of the customer information fields that would properly approximate the credit limit as determined by human experts. If this assumption does not hold, we cannot achieve a small error with a linear model. We will deal with this situation when we discuss nonlinear transformation later in the chapter.

3.2.1 The Algorithm

The linear regression algorithm is based on minimizing the squared error between $h(\mathbf{x})$ and y.[2]

$$E_{\text{out}}(h) = \mathbb{E}\left[(h(\mathbf{x}) \quad y)^2\right],$$

where the expected value is taken with respect to the joint probability distribution $P(\mathbf{x}, y)$. The goal is to find a hypothesis that achieves a small $E_{\text{out}}(h)$. Since the distribution $P(\mathbf{x}, y)$ is unknown, $E_{\text{out}}(h)$ cannot be computed. Similar to what we did in classification, we resort to the in-sample version instead,

$$E_{\text{in}}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n) \quad y_n)^2.$$

In linear regression, h takes the form of a linear combination of the components of $\mathbf{x}$. That is,

$$h(\mathbf{x}) = \sum_{i=0}^{d} w_i x_i = \mathbf{w}^{\mathrm{T}}\mathbf{x},$$

where $x_0 = 1$ and $\mathbf{x} \in \{1\} \times \mathbb{R}^d$ as usual, and $\mathbf{w} \in \mathbb{R}^{d+1}$. For the special case of linear h, it is very useful to have a matrix representation of $E_{\text{in}}(h)$. First, define the data matrix $\mathrm{X} \in \mathbb{R}^{N\times(d+1)}$ to be the $N \times (d+1)$ matrix whose rows are the inputs $\mathbf{x}_n$ as row vectors, and define the target vector $\mathbf{y} \in \mathbb{R}^N$ to be the column vector whose components are the target values y_n. The in-sample error is a function of $\mathbf{w}$ and the data $\mathrm{X}, \mathbf{y}$:

$$\begin{aligned} E_{\text{in}}(\mathbf{w}) &= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n \quad y_n)^2 \\ &= \frac{1}{N}\|\mathrm{X}\mathbf{w} - \mathbf{y}\|^2 \qquad (3.3) \\ &= \frac{1}{N}(\mathbf{w}^{\mathrm{T}}\mathrm{X}^{\mathrm{T}}\mathrm{X}\mathbf{w} - 2\mathbf{w}^{\mathrm{T}}\mathrm{X}^{\mathrm{T}}\mathbf{y} + \mathbf{y}^{\mathrm{T}}\mathbf{y}), \qquad (3.4) \end{aligned}$$

where $\|\cdot\|$ is the Euclidean norm of a vector, and (3.3) follows because the nth component of the vector $\mathrm{X}\mathbf{w} - \mathbf{y}$ is exactly $\mathbf{w}^{\mathrm{T}}\mathbf{x}_n \quad y_n$. The linear regression

[2] The term 'linear regression' has been historically confined to squared error measures.

(a) one dimension (line) (b) two dimensions (hyperplane)

Figure 3.3: The solution hypothesis (in blue) of the linear regression algo rithm in one and two dimensions. The sum of squared errors is minimized.

algorithm is derived by minimizing $E_{\text{in}}(\mathbf{w})$ over all possible $\mathbf{w} \in \mathbb{R}^{d+1}$, as formalized by the following optimization problem:

$$\mathbf{w}_{\text{lin}} = \underset{\mathbf{w}\in\mathbb{R}^{d+1}}{\operatorname{argmin}} E_{\text{in}}(\mathbf{w}). \tag{3.5}$$

Figure 3.3 illustrates the solution in one and two dimensions. Since Equation (3.4) implies that $E_{\text{in}}(\mathbf{w})$ is differentiable, we can use standard matrix calculus to find the $\mathbf{w}$ that minimizes $E_{\text{in}}(\mathbf{w})$ by requiring that the gradient of E_{in} with respect to $\mathbf{w}$ is the zero vector, i.e., $\nabla E_{\text{in}}(\mathbf{w}) = \mathbf{0}$. The gradient is a (column) vector whose ith component is $[\nabla E_{\text{in}}(\mathbf{w})]_i = \frac{\partial}{\partial w_i} E_{\text{in}}(\mathbf{w})$. By explicitly computing $\frac{\partial}{\partial w_i}$, the reader can verify the following gradient identities,

$$\nabla_{\mathbf{w}}(\mathbf{w}^{\mathrm{T}}\mathrm{A}\mathbf{w}) = (\mathrm{A} + \mathrm{A}^{\mathrm{T}})\mathbf{w}, \qquad \nabla_{\mathbf{w}}(\mathbf{w}^{\mathrm{T}}\mathbf{b}) = \mathbf{b}.$$

These identities are the matrix analog of ordinary differentiation of quadratic and linear functions. To obtain the gradient of E_{in}, we take the gradient of each term in (3.4) to obtain

$$\nabla E_{\text{in}}(\mathbf{w}) = \frac{2}{N}(\mathrm{X}^{\mathrm{T}}\mathrm{X}\mathbf{w} - \mathrm{X}^{\mathrm{T}}\mathbf{y}).$$

Note that both $\mathbf{w}$ and $\nabla E_{\text{in}}(\mathbf{w})$ are column vectors. Finally, to get $\nabla E_{\text{in}}(\mathbf{w})$ to be $\mathbf{0}$, one should solve for $\mathbf{w}$ that satisfies

$$\mathrm{X}^{\mathrm{T}}\mathrm{X}\mathbf{w} = \mathrm{X}^{\mathrm{T}}\mathbf{y}.$$

If $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is invertible, $\mathbf{w} = \mathrm{X}^{\dagger}\mathbf{y}$ where $\mathrm{X}^{\dagger} = (\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}$ is the *pseudo-inverse* of X. The resulting $\mathbf{w}$ is the unique optimal solution to (3.5). If $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is not

invertible, a pseudo-inverse can still be defined, but the solution will not be unique (see Problem 3.15). In practice, $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is invertible in most of the cases since N is often much bigger than $d+1$, so there will likely be $d+1$ linearly independent vectors $\mathbf{x}_n$. We have thus derived the following *linear regression algorithm.*

Linear regression algorithm:

1: Construct the matrix X and the vector $\mathbf{y}$ from the data set $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)$, where each $\mathbf{x}$ includes the $x_0 = 1$ bias coordinate, as follows

$$\underbrace{\mathrm{X} = \begin{bmatrix} -\mathbf{x}_1^{\mathrm{T}}- \\ -\mathbf{x}_2^{\mathrm{T}}- \\ \vdots \\ -\mathbf{x}_N^{\mathrm{T}}- \end{bmatrix}}_{\text{input data matrix}}, \qquad \underbrace{\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\text{target vector}}.$$

2: Compute the pseudo-inverse $\mathrm{X}^\dagger$ of the matrix X. If $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is invertible,

$$\mathrm{X}^\dagger = (\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}.$$

3: Return $\mathbf{w}_{\text{lin}} = \mathrm{X}^\dagger \mathbf{y}$.

This algorithm is sometimes referred to as *ordinary least squares* (OLS). It may seem that, compared with the perceptron learning algorithm, linear regression doesn't really look like 'learning', in the sense that the hypothesis $\mathbf{w}_{\text{lin}}$ comes from an analytic solution (matrix inversion and multiplications) rather than from iterative learning steps. Well, as long as the hypothesis $\mathbf{w}_{\text{lin}}$ has a decent out-of-sample error, then learning *has* occurred. Linear regression is a rare case where we have an analytic formula for learning that is easy to evaluate. This is one of the reasons why the technique is so widely used. It should be noted that there are methods for computing the pseudo-inverse directly without inverting a matrix, and that these methods are numerically more stable than matrix inversion.

Linear regression has been analyzed in great detail in statistics. We would like to mention one of the analysis tools here since it relates to in-sample and out-of-sample errors, and that is the *hat matrix* H. Here is how H is defined. The linear regression weight vector $\mathbf{w}_{\text{lin}}$ is an attempt to map the inputs X to the outputs $\mathbf{y}$. However, $\mathbf{w}_{\text{lin}}$ does not produce $\mathbf{y}$ exactly, but produces an estimate

$$\hat{\mathbf{y}} = \mathrm{X}\mathbf{w}_{\text{lin}}$$

which differs from $\mathbf{y}$ due to in-sample error. Substituting the expression for $\mathbf{w}_{\text{lin}}$ (assuming $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is invertible), we get

$$\hat{\mathbf{y}} = \mathrm{X}(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}\mathbf{y}.$$

Therefore the estimate $\hat{\mathbf{y}}$ is a linear transformation of the actual $\mathbf{y}$ through matrix multiplication with H, where

$$\mathrm{H} = \mathrm{X}(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}. \tag{3.6}$$

Since $\hat{\mathbf{y}} = \mathrm{H}\mathbf{y}$, the matrix H 'puts a hat' on $\mathbf{y}$, hence the name. The hat matrix is a very special matrix. For one thing, $\mathrm{H}^2 = \mathrm{H}$, which can be verified using the above expression for H. This and other properties of H will facilitate the analysis of in-sample and out-of-sample errors of linear regression.

Exercise 3.3

Consider the hat matrix $\mathrm{H} = \mathrm{X}(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}$, where X is an N by $d+1$ matrix, and $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is invertible.

(a) Show that H is symmetric.

(b) Show that $\mathrm{H}^K = \mathrm{H}$ for any positive integer K.

(c) If I is the identity matrix of size N, show that $(\mathrm{I}-\mathrm{H})^K = \mathrm{I}-\mathrm{H}$ for any positive integer K.

(d) Show that $\mathrm{trace}(\mathrm{H}) = d+1$, where the trace is the sum of diagonal elements. *[Hint:* $\mathrm{trace}(\mathrm{AB}) = \mathrm{trace}(\mathrm{BA})$.*]*

3.2.2 Generalization Issues

Linear regression looks for the optimal weight vector in terms of the in-sample error E_{in}, which leads to the usual generalization question: Does this guarantee decent out-of-sample error E_{out}? The short answer is yes. There is a regression version of the VC generalization bound (3.1) that similarly bounds E_{out}. In the case of linear regression in particular, there are also exact formulas for the expected E_{out} and E_{in} that can be derived under simplifying assumptions. The general form of the result is

$$E_{\text{out}}(g) = E_{\text{in}}(g) + O\left(\frac{d}{N}\right),$$

where $E_{\text{out}}(g)$ and $E_{\text{in}}(g)$ are the expected values. This is comparable to the classification bound in (3.1).

Exercise 3.4

Consider a noisy target $y = \mathbf{w}^{*\mathrm{T}}\mathbf{x} + \epsilon$ for generating the data, where ϵ is a noise term with zero mean and σ^2 variance, independently generated for every example $(\mathbf{x}, y)$. The expected error of the best possible linear fit to this target is thus σ^2.

For the data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, denote the noise in y_n as ϵ_n and let $\boldsymbol{\epsilon} = [\epsilon_1, \epsilon_2, \ldots, \epsilon_N]^{\mathrm{T}}$; assume that $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is invertible. By following

(continued on next page)

the steps below, show that the expected in sample error of linear regression with respect to $\mathcal{D}$ is given by

$$\mathbb{E}_{\mathcal{D}}[E_{\text{in}}(\mathbf{w}_{\text{lin}})] = \sigma^2\left(1 - \frac{d+1}{N}\right).$$

(a) Show that the in sample estimate of $\mathbf{y}$ is given by $\hat{\mathbf{y}} = \mathrm{X}\mathbf{w}^* + \mathrm{H}\boldsymbol{\epsilon}$.

(b) Show that the in sample error vector $\hat{\mathbf{y}} - \mathbf{y}$ can be expressed by a matrix times $\boldsymbol{\epsilon}$. What is the matrix?

(c) Express $E_{\text{in}}(\mathbf{w}_{\text{lin}})$ in terms of $\boldsymbol{\epsilon}$ using (b), and simplify the expression using Exercise 3.3(c).

(d) Prove that $\mathbb{E}_{\mathcal{D}}[E_{\text{in}}(\mathbf{w}_{\text{lin}})] = \sigma^2\left(1 - \frac{d+1}{N}\right)$ using (c) and the independence of $\epsilon_1, \cdots, \epsilon_N$. *[Hint: The sum of the diagonal elements of a matrix (the trace) will play a role. See Exercise 3.3(d).]*

For the expected out of sample error, we take a special case which is easy to analyze. Consider a test data set $\mathcal{D}_{\text{test}} = \{(\mathbf{x}_1, y_1'), \ldots, (\mathbf{x}_N, y_N')\}$, which shares the same input vectors $\mathbf{x}_n$ with $\mathcal{D}$ but with a different realization of the noise terms. Denote the noise in y_n' as ϵ_n' and let $\boldsymbol{\epsilon}' = [\epsilon_1', \epsilon_2', \ldots, \epsilon_N']^{\mathrm{T}}$. Define $E_{\text{test}}(\mathbf{w}_{\text{lin}})$ to be the average squared error on $\mathcal{D}_{\text{test}}$.

(e) Prove that $\mathbb{E}_{\mathcal{D},\boldsymbol{\epsilon}'}[E_{\text{test}}(\mathbf{w}_{\text{lin}})] = \sigma^2\left(1 + \frac{d+1}{N}\right)$.

The special test error E_{test} is a very restricted case of the general out-of sample error. Some detailed analysis shows that similar results can be obtained for the general case, as shown in Problem 3.11.

Figure 3.4 illustrates the learning curve of linear regression under the assumptions of Exercise 3.4. The best possible linear fit has expected error σ^2. The expected in-sample error is smaller, equal to $\sigma^2(1 - \frac{d+1}{N})$ for $N \geq d+1$. The learned linear fit has eaten into the in-sample noise as much as it could with the $d+1$ degrees of freedom that it has at its disposal. This occurs because the fitting cannot distinguish the noise from the 'signal.' On the other hand, the expected out-of-sample error is $\sigma^2(1 + \frac{d+1}{N})$, which is more than the unavoidable error of σ^2. The additional error reflects the drift in $\mathbf{w}_{\text{lin}}$ due to fitting the in-sample noise.

3.3 Logistic Regression

The core of the linear model is the 'signal' $s = \mathbf{w}^{\mathrm{T}}\mathbf{x}$ that combines the input variables linearly. We have seen two models based on this signal, and we are now going to introduce a third. In linear regression, the signal itself is taken as the output, which is appropriate if you are trying to predict a real response that could be unbounded. In linear classification, the signal is thresholded at zero to produce a ± 1 output, appropriate for binary decisions. A third possibility, which has wide application in practice, is to output a *probability*,

Figure 3.4: The learning curve for linear regression.

a value between 0 and 1. Our new model is called *logistic* regression. It has similarities to both previous models, as the output is real (like regression) but bounded (like classification).

Example 3.2 (Prediction of heart attacks)**.** Suppose we want to predict the occurrence of heart attacks based on a person's cholesterol level, blood pressure, age, weight, and other factors. Obviously, we cannot predict a heart attack with any certainty, but we may be able to predict how likely it is to occur given these factors. Therefore, an output that varies continuously between 0 and 1 would be a more suitable model than a binary decision. The closer y is to 1, the more likely that the person will have a heart attack. □

3.3.1 Predicting a Probability

Linear classification uses a hard threshold on the signal $s = \mathbf{w}^{\mathrm{T}}\mathbf{x}$,

$$h(\mathbf{x}) = \operatorname{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}),$$

while linear regression uses no threshold at all,

$$h(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}.$$

In our new model, we need something in between these two cases that smoothly restricts the output to the probability range $[0, 1]$. One choice that accomplishes this goal is the logistic regression model,

$$h(\mathbf{x}) = \theta(\mathbf{w}^{\mathrm{T}}\mathbf{x}),$$

where θ is the so-called *logistic* function $\theta(s) = \frac{e^s}{1+e^s}$ whose output is between 0 and 1.

The output can be interpreted as a probability for a binary event (heart attack or no heart attack, digit '1' versus digit '5', etc.). Linear classification also deals with a binary event, but the difference is that the 'classification' in logistic regression is allowed to be uncertain, with intermediate values between 0 and 1 reflecting this uncertainty. The logistic function θ is referred to as a *soft threshold*, in contrast to the hard threshold in classification. It is also called a *sigmoid* because its shape looks like a flattened out 's'.

Exercise 3.5

Another popular soft threshold is the hyperbolic tangent

$$\tanh(s) = \frac{e^s - e^{\ s}}{e^s + e^{\ s}}.$$

(a) How is tanh related to the logistic function θ? *[Hint: shift and scale]*

(b) Show that $\tanh(s)$ converges to a hard threshold for large $|s|$, and converges to no threshold for small $|s|$ *[Hint: Formalize the figure below.]*

The specific formula of $\theta(s)$ will allow us to define an error measure for learning that has analytical and computational advantages, as we will see shortly. Let us first look at the target that logistic regression is trying to learn. The target is a probability, say of a patient being at risk for heart attack, that depends on the input $\mathbf{x}$ (the characteristics of the patient). Formally, we are trying to learn the target function

$$f(\mathbf{x}) = \mathbb{P}[y = +1 \mid \mathbf{x}].$$

The data does not give us the value of f explicitly. Rather, it gives us samples generated by this probability, e.g., patients who had heart attacks and patients who didn't. Therefore, the data is in fact generated by a noisy target $P(y \mid \mathbf{x})$,

$$P(y \mid \mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{for } y = +1; \\ 1 - f(\mathbf{x}) & \text{for } y = -1. \end{cases} \tag{3.7}$$

To learn from such data, we need to define a proper error measure that gauges how close a given hypothesis h is to f in terms of these noisy ± 1 examples.

Error measure. The standard error measure $\mathsf{e}(h(\mathbf{x}), y)$ used in logistic regression is based on the notion of *likelihood*; how 'likely' is it that we would get this output y from the input $\mathbf{x}$ if the target distribution $P(y \mid \mathbf{x})$ was indeed captured by our hypothesis $h(\mathbf{x})$? Based on (3.7), that likelihood would be

$$P(y \mid \mathbf{x}) - \begin{cases} h(\mathbf{x}) & \text{for } y = +1; \\ 1 \quad h(\mathbf{x}) & \text{for } y = -1. \end{cases}$$

We substitute for $h(\mathbf{x})$ by its value $\theta(\mathbf{w}^{\mathrm{T}}\mathbf{x})$, and use the fact that $1 \quad \theta(s) = \theta(\quad s)$ (easy to verify) to get

$$P(y \mid \mathbf{x}) = \theta(y\ \mathbf{w}^{\mathrm{T}}\mathbf{x}). \tag{3.8}$$

One of our reasons for choosing the mathematical form $\theta(s) = e^s/(1+e^s)$ is that it leads to this simple expression for $P(y \mid \mathbf{x})$.

Since the data points $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ are independently generated, the probability of getting all the y_n's in the data set from the corresponding $\mathbf{x}_n$'s would be the product

$$\prod_{n=1}^{N} P(y_n \mid \mathbf{x}_n).$$

The method of *maximum likelihood* selects the hypothesis h which maximizes this probability.[3] We can equivalently minimize a more convenient quantity,

$$-\frac{1}{N} \ln\left(\prod_{n=1}^{N} P(y_n \mid \mathbf{x}_n)\right) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(\frac{1}{P(y_n \mid \mathbf{x}_n)}\right),$$

since '$-\frac{1}{N}\ln(\cdot)$' is a monotonically decreasing function. Substituting with Equation (3.8), we would be minimizing

$$\frac{1}{N}\sum_{n=1}^{N} \ln\left(\frac{1}{\theta(y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n)}\right)$$

with respect to the weight vector $\mathbf{w}$. The fact that we are *minimizing* this quantity allows us to treat it as an 'error measure.' Substituting the functional form for $\theta(y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n)$ produces the in-sample error measure for logistic regression,

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(1 + e^{-y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n}\right). \tag{3.9}$$

The implied pointwise error measure is $\mathsf{e}(h(\mathbf{x}_n), y_n) = \ln(1+e^{-y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n})$. Notice that this error measure is small when $y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n$ is large and *positive*, which would imply that $\text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) = y_n$. Therefore, as our intuition would expect, the error measure encourages $\mathbf{w}$ to 'classify' each $\mathbf{x}_n$ correctly.

[3] Although the method of maximum likelihood is intuitively plausible, its rigorous justification as an inference tool continues to be discussed in the statistics community.

Exercise 3.6 [Cross-entropy error measure]

(a) More generally, if we are learning from ± 1 data to predict a noisy target $P(y \mid \mathbf{x})$ with candidate hypothesis h, show that the maximum likelihood method reduces to the task of finding h that minimizes

$$E_{\text{in}}(\mathbf{w}) = \sum_{n=1}^{N} [\![y_n = +1]\!] \ln \frac{1}{h(\mathbf{x}_n)} + [\![y_n = -1]\!] \ln \frac{1}{1 - h(\mathbf{x}_n)}.$$

(b) For the case $h(\mathbf{x}) = \theta(\mathbf{w}^{\mathrm{T}}\mathbf{x})$, argue that minimizing the in sample error in part (a) is equivalent to minimizing the one in (3.9).

For two probability distributions $\{p, 1-p\}$ and $\{q, 1 \quad q\}$ with binary outcomes, the cross entropy (from information theory) is

$$p \log \frac{1}{q} + (1-p) \log \frac{1}{1-q}.$$

The in sample error in part (a) corresponds to a cross entropy error measure on the data point $(\mathbf{x}_n, y_n)$, with $p = [\![y_n = +1]\!]$ and $q = h(\mathbf{x}_n)$.

For linear classification, we saw that minimizing E_{in} for the perceptron is a combinatorial optimization problem; to solve it, we introduced a number of algorithms such as the perceptron learning algorithm and the pocket algorithm. For linear regression, we saw that training can be done using the analytic pseudo-inverse algorithm for minimizing E_{in} by setting $\nabla E_{\text{in}}(\mathbf{w}) = \mathbf{0}$. These algorithms were developed based on the specific form of linear classification or linear regression, so none of them would apply to logistic regression.

To train logistic regression, we will take an approach similar to linear regression in that we will try to set $\nabla E_{\text{in}}(\mathbf{w}) = \mathbf{0}$. Unfortunately, unlike the case of linear regression, the mathematical form of the gradient of E_{in} for logistic regression is not easy to manipulate, so an analytic solution is not feasible.

Exercise 3.7

For logistic regression, show that

$$\begin{aligned} \nabla E_{\text{in}}(\mathbf{w}) &= -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n}} \\ &= \frac{1}{N} \sum_{n=1}^{N} -y_n \mathbf{x}_n \theta(-y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n). \end{aligned}$$

Argue that a 'misclassified' example contributes more to the gradient than a correctly classified one.

Instead of analytically setting the gradient to zero, we will *iteratively* set it to zero. To do so, we will introduce a new algorithm, *gradient descent*. Gradient

descent is a very general algorithm that can be used to train many other learning models with smooth error measures. For logistic regression, gradient descent has particularly nice properties.

3.3.2 Gradient Descent

Gradient descent is a general technique for minimizing a twice-differentiable function, such as $E_{\text{in}}(\mathbf{w})$ in logistic regression. A useful physical analogy of gradient descent is a ball rolling down a hilly surface. If the ball is placed on a hill, it will roll down, coming to rest at the bottom of a valley. The same basic idea underlies gradient descent. $E_{\text{in}}(\mathbf{w})$ is a 'surface' in a high-dimensional space. At step 0, we start somewhere on this surface, at $\mathbf{w}(0)$, and try to roll down this surface, thereby decreasing E_{in}. One thing which you immediately notice from the physical analogy is that the ball will not necessarily come to rest in the lowest valley of the entire surface. Depending on where you start the ball rolling, you will end up at the bottom of one of the valleys a *local minimum*. In general, the same applies to gradient descent. Depending on your starting weights, the path of descent will take you to a local minimum in the error surface.

A particular advantage for logistic regression with the cross-entropy error is that the picture looks much nicer. There is only one valley! So, it does not matter where you start your ball rolling, it will always roll down to the same (unique) *global minimum*. This is a consequence of the fact that $E_{\text{in}}(\mathbf{w})$ is a *convex* function of $\mathbf{w}$, a mathematical property that implies a single 'valley' as shown to the right. This means that gradient descent will not be trapped in local minima when minimizing such convex error measures.[4]

Let's now determine how to 'roll' down the E_{in}-surface. We would like to take a step in the direction of steepest descent, to gain the biggest bang for our buck. Suppose that we take a small step of size η in the direction of a unit vector $\hat{\mathbf{v}}$. The new weights are $\mathbf{w}(0) + \eta\hat{\mathbf{v}}$. Since η is small, using the Taylor expansion to first order, we compute the change in E_{in} as

$$\begin{aligned}\Delta E_{\text{in}} &= E_{\text{in}}(\mathbf{w}(0) + \eta\hat{\mathbf{v}}) \quad E_{\text{in}}(\mathbf{w}(0)) \\ &= \eta\nabla E_{\text{in}}(\mathbf{w}(0))^{\mathrm{T}}\hat{\mathbf{v}} + O(\eta^2) \\ &\geq \quad \eta\|\nabla E_{\text{in}}(\mathbf{w}(0))\|,\end{aligned}$$

[4] In fact, the squared in-sample error in linear regression is also convex, which is why the analytic solution found by the pseudo-inverse is guaranteed to have optimal in-sample error.

where we have ignored the small term $O(\eta^2)$. Since $\hat{\mathbf{v}}$ is a unit vector, equality holds if and only if

$$\hat{\mathbf{v}} = -\frac{\nabla E_{\text{in}}(\mathbf{w}(0))}{\|\nabla E_{\text{in}}(\mathbf{w}(0))\|}. \tag{3.10}$$

This direction, specified by $\hat{\mathbf{v}}$, leads to the largest decrease in E_{in} for a given step size η.

Exercise 3.8

The claim that $\hat{\mathbf{v}}$ is the direction which gives largest decrease in E_{in} only holds for small η. Why?

There is nothing to prevent us from continuing to take steps of size η, re-evaluating the direction $\hat{\mathbf{v}}_t$ at each iteration $t = 0, 1, 2, \ldots$. How large a step should one take at each iteration? This is a good question, and to gain some insight, let's look at the following examples.

η too small η too large variable η just right

A fixed step size (if it is too small) is inefficient when you are far from the local minimum. On the other hand, too large a step size when you are close to the minimum leads to bouncing around, possibly even increasing E_{in}. Ideally, we would like to take large steps when far from the minimum to get in the right ballpark quickly, and then small (more careful) steps when close to the minimum. A simple heuristic can accomplish this: far from the minimum, the norm of the gradient is typically large, and close to the minimum, it is small. Thus, we could set $\eta_t = \eta\|\nabla E_{\text{in}}\|$ to obtain the desired behavior for the variable step size; choosing the step size proportional to the norm of the gradient will also conveniently cancel the term normalizing the unit vector $\hat{\mathbf{v}}$ in Equation (3.10), leading to the *fixed learning rate gradient descent* algorithm for minimizing E_{in} (with redefined η):

Fixed learning rate gradient descent:

1: Initialize the weights at time step $t = 0$ to $\mathbf{w}(0)$.
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Compute the gradient $\mathbf{g}_t = \nabla E_{\text{in}}(\mathbf{w}(t))$.
4: Set the direction to move, $\mathbf{v}_t = -\mathbf{g}_t$.
5: Update the weights: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \mathbf{v}_t$.
6: Iterate to the next step until it is time to stop.
7: Return the final weights.

In the algorithm, $\mathbf{v}_t$ is a direction that is no longer restricted to unit length. The parameter η (the *learning rate*) has to be specified. A typically good choice for η is around 0.1 (a purely practical observation). To use gradient descent, one must compute the gradient. This can be done explicitly for logistic regression (see Exercise 3.7).

Example 3.3. Gradient descent is a general algorithm for minimizing twice-differentiable functions. We can apply it to the logistic regression in-sample error to return weights that approximately minimize

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \ln\left(1 + e^{\ y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n}\right).$$

Logistic regression algorithm:

1: Initialize the weights at time step $t = 0$ to $\mathbf{w}(0)$.
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Compute the gradient

$$\mathbf{g}_t = -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\mathrm{T}}(t) \mathbf{x}_n}}.$$

4: Set the direction to move, $\mathbf{v}_t = -\mathbf{g}_t$.
5: Update the weights: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \mathbf{v}_t$.
6: Iterate to the next step until it is time to stop.
7: Return the final weights $\mathbf{w}$.

□

Initialization and termination. We have two more loose ends to tie: the first is how to choose $\mathbf{w}(0)$, the initial weights, and the second is how to set the criterion for "...until it is time to stop" in step 6 of the gradient descent algorithm. In some cases, such as logistic regression, initializing the weights $\mathbf{w}(0)$ as zeros works well. However, in general, it is safer to initialize the weights randomly, so as to avoid getting stuck on a perfectly symmetric hilltop. Choosing each weight independently from a Normal distribution with zero mean and small variance usually works well in practice.

That takes care of initialization, so we now move on to termination. How do we decide when to stop? Termination is a non-trivial topic in optimization. One simple approach, as we encountered in the pocket algorithm, is to set an upper bound on the number of iterations, where the upper bound is typically in the thousands, depending on the amount of training time we have. The problem with this approach is that there is no guarantee on the quality of the final weights.

Another plausible approach is based on the gradient being zero at any minimum. A natural termination criterion would be to stop once $\|\mathbf{g}_t\|$ drops below a certain threshold. Eventually this must happen, but we do not know when it will happen. For logistic regression, a combination of the two conditions (setting a large upper bound for the number of iterations, and a small lower bound for the size of the gradient) usually works well in practice.

There is a problem with relying solely on the size of the gradient to stop, which is that you might stop prematurely as illustrated on the right. When the iteration reaches a relatively flat region (which is more common than you might suspect), the algorithm will prematurely stop when we may want to continue. So one solution is to require that termination occurs only if the error change is small and the error itself is small. Ultimately a combination of termination criteria (a maximum number of iterations, marginal error improvement, coupled with small value for the error itself) works reasonably well.

Example 3.4. By way of summarizing linear models, we revisit our old friend the credit example. If the goal is to decide whether to approve or deny, then we are in the realm of classification; if you want to assign an amount of credit line, then linear regression is appropriate; if you want to predict the probability that someone will default, use logistic regression.

The three linear models have their respective goals, error measures, and algorithms. Nonetheless, they not only share similar sets of linear hypotheses, but are in fact related in other ways. We would like to point out one important relationship: Both logistic regression and linear regression can be used in linear classification. Here is how.

Logistic regression produces a final hypothesis $g(\mathbf{x})$ which is our estimate of $\mathbb{P}[y = +1 \mid \mathbf{x}]$. Such an estimate can easily be used for classification by

setting a threshold on $g(\mathbf{x})$; a natural threshold is $\frac{1}{2}$, which corresponds to classifying $+1$ if $+1$ is more likely. This choice for threshold corresponds to using the logistic regression weights as weights in the perceptron for classification. Not only can logistic regression weights be used for classification in this way, but they can also be used as a way to train the perceptron model. The perceptron learning problem (3.2) is a very hard combinatorial optimization problem. The convexity of E_{in} in logistic regression makes the optimization problem much easier to solve. Since the logistic function is a soft version of a hard threshold, the logistic regression weights should be good weights for classification using the perceptron.

A similar relationship exists between classification and linear regression. Linear regression can be used with any real-valued target function, which includes real values that are ± 1. If $\mathbf{w}_{\text{lin}}^{\mathrm{T}}\mathbf{x}$ is fit to ± 1 values, $\text{sign}(\mathbf{w}_{\text{lin}}^{\mathrm{T}}\mathbf{x})$ will likely agree with these values and make good classification predictions. In other words, the linear regression weights $\mathbf{w}_{\text{lin}}$, which are easily computed using the pseudo-inverse, are also an approximate solution for the perceptron model. The weights can be directly used for classification, or used as an initial condition for the pocket algorithm to give it a head start. □

Exercise 3.9

Consider pointwise error measures $\mathrm{e}_{\text{class}}(s, y) = [\![y \neq \text{sign}(s)]\!]$, $\mathrm{e}_{\text{sq}}(s, y) = (y - s)^2$, and $\mathrm{e}_{\log}(s, y) = \ln(1 + \exp(-ys))$, where the signal $s = \mathbf{w}^{\mathrm{T}}\mathbf{x}$.

(a) For $y = +1$, plot $\mathrm{e}_{\text{class}}$, e_{sq} and $\frac{1}{\ln 2}\mathrm{e}_{\log}$ versus s, on the same plot.

(b) Show that $\mathrm{e}_{\text{class}}(s, y) \leq \mathrm{e}_{\text{sq}}(s, y)$, and hence that the classification error is upper bounded by the squared error.

(c) Show that $\mathrm{e}_{\text{class}}(s, y) \leq \frac{1}{\ln 2}\mathrm{e}_{\log}(s, y)$, and, as in part (b), get an upper bound (up to a constant factor) using the logistic regression error.

These bounds indicate that minimizing the squared or logistic regression error should also decrease the classification error, which justifies using the weights returned by linear or logistic regression as approximations for classification.

Stochastic gradient descent. The version of gradient descent we have described so far is known as *batch* gradient descent the gradient is computed for the error on the whole data set before a weight update is done. A sequential version of gradient descent known as *stochastic gradient descent* (SGD) turns out to be very efficient in practice. Instead of considering the full batch gradient on all N training data points, we consider a stochastic version of the gradient. First, pick a training data point $(\mathbf{x}_n, y_n)$ uniformly at random (hence the name 'stochastic'), and consider only the error on that data point

(in the case of logistic regression),

$$\mathsf{e}_n(\mathbf{w}) = \ln\left(1 + e^{-y_n \mathbf{w}^{\mathsf{T}} \mathbf{x}_n}\right).$$

The gradient of this single data point's error is used for the weight update in exactly the same way that the gradient was used in batch gradient descent. The gradient needed for the weight update of SGD is (see Exercise 3.7)

$$\nabla \mathsf{e}_n(\mathbf{w}) = \frac{-y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\mathsf{T}} \mathbf{x}_n}},$$

and the weight update is $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \mathsf{e}_n(\mathbf{w})$. Insight into why SGD works can be gained by looking at the expected value of the change in the weight (the expectation is with respect to the random point that is selected). Since n is picked uniformly at random from $\{1, \ldots, N\}$, the expected weight change is

$$\eta \cdot \frac{1}{N} \sum_{n=1}^{N} \nabla \mathsf{e}_n(\mathbf{w}).$$

This is exactly the same as the deterministic weight change from the batch gradient descent weight update. That is, 'on average' the minimization proceeds in the right direction, but is a bit wiggly. In the long run, these random fluctuations cancel out. The computational cost is cheaper by a factor of N, though, since we compute the gradient for only one point per iteration, rather than for all N points as we do in batch gradient descent.

Notice that SGD is similar to PLA in that it decreases the error with respect to one data point at a time. Minimizing the error on one data point may interfere with the error on the rest of the data points that are not considered at that iteration. However, also similar to PLA, the interference cancels out on average as we have just argued.

Exercise 3.10

(a) Define an error for a single data point $(\mathbf{x}_n, y_n)$ to be

$$\mathsf{e}_n(\mathbf{w}) = \max(0, -y_n \mathbf{w}^{\mathsf{T}} \mathbf{x}_n).$$

Argue that PLA can be viewed as SGD on e_n with learning rate $\eta = 1$.

(b) For logistic regression with a very large $\mathbf{w}$, argue that minimizing E_{in} using SGD is similar to PLA. This is another indication that the logistic regression weights can be used as a good approximation for classification.

SGD is successful in practice, often beating the batch version and other more sophisticated algorithms. In fact, SGD was an important part of the algorithm that won the million-dollar Netflix competition, discussed in Section 1.1. It scales well to large data sets, and is naturally suited to online learning, where

a stream of data present themselves to the learning algorithm sequentially. The randomness introduced by processing one data point at a time can be a plus, helping the algorithm to avoid flat regions and local minima in the case of a complicated error surface. However, it is challenging to choose a suitable termination criterion for SGD. A good stopping criterion should consider the total error on all the data, which can be computationally demanding to evaluate at each iteration.

3.4 Nonlinear Transformation

All formulas for the linear model have used the sum

$$\mathbf{w}^{\mathrm{T}}\mathbf{x} = \sum_{i=0}^{d} w_i x_i \tag{3.11}$$

as the main quantity in computing the hypothesis output. This quantity is linear, not only in the x_i's but also in the w_i's. A closer inspection of the corresponding learning algorithms shows that *the linearity in w_i's* is the key property for deriving these algorithms; the x_i's are just constants as far as the algorithm is concerned. This observation opens the possibility for allowing nonlinear versions of x_i's while still remaining in the analytic realm of linear models, because the form of Equation (3.11) remains linear in the w_i parameters.

Consider the credit limit problem for instance. It makes sense that the 'years in residence' field would affect a person's credit since it is correlated with stability. However, it is less plausible that the credit limit would grow *linearly* with the number of years in residence. More plausibly, there is a threshold (say 1 year) below which the credit limit is affected negatively and another threshold (say 5 years) above which the credit limit is affected positively. If x_i is the input variable that measures years in residence, then two nonlinear 'features' derived from it, namely $[\![x_i < 1]\!]$ and $[\![x_i > 5]\!]$, would allow a linear formula to reflect the credit limit better.

We have already seen the use of features in the classification of handwritten digits, where intensity and symmetry features were derived from input pixels. Nonlinear transforms can be further applied to those features, as we will see shortly, creating more elaborate features and improving the performance. The scope of linear methods expands significantly when we represent the input by a set of appropriate features.

3.4.1 The $\mathcal{Z}$ Space

Consider the situation in Figure 3.1(b) where a linear classifier can't fit the data. By transforming the inputs x_1, x_2 in a nonlinear fashion, we will be able to separate the data with more complicated boundaries while still using the

simple PLA as a building block. Let's start by looking at the circle in Figure 3.5(a), which is a replica of the non-separable case in Figure 3.1(b). The circle represents the following equation:

$$x_1^2 + x_2^2 = 0.6.$$

That is, the nonlinear hypothesis $h(\mathbf{x}) = \text{sign}(-0.6 + x_1^2 + x_2^2)$ separates the data set perfectly. We can view the hypothesis as a *linear* one after applying a nonlinear transformation on $\mathbf{x}$. In particular, consider $z_0 = 1$, $z_1 = x_1^2$ and $z_2 = x_2^2$,

$$\begin{aligned} h(\mathbf{x}) &= \text{sign}\left(\underbrace{(-0.6)}_{\tilde{w}_0} \cdot \underbrace{1}_{z_0} + \underbrace{1}_{\tilde{w}_1} \cdot \underbrace{x_1^2}_{z_1} + \underbrace{1}_{\tilde{w}_2} \cdot \underbrace{x_2^2}_{z_2}\right) \\ &= \text{sign}\left(\underbrace{[\tilde{w}_0 \ \tilde{w}_1 \ \tilde{w}_2]}_{\tilde{\mathbf{w}}^{\mathsf{T}}} \underbrace{\begin{bmatrix} 1 \\ z_1 \\ z_2 \end{bmatrix}}_{\mathbf{z}}\right). \end{aligned}$$

where the vector $\mathbf{z}$ is obtained from $\mathbf{x}$ through a nonlinear transform Φ,

$$\mathbf{z} = \Phi(\mathbf{x}).$$

We can plot the data in terms of $\mathbf{z}$ instead of $\mathbf{x}$, as depicted in Figure 3.5(b). For instance, the point $\mathbf{x}_1$ in Figure 3.5(a) is transformed to the point $\mathbf{z}_1$ in Figure 3.5(b) and the point $\mathbf{x}_2$ is transformed to the point $\mathbf{z}_2$. The space $\mathcal{Z}$, which contains the $\mathbf{z}$ vectors, is referred to as the *feature space* since its coordinates are higher-level features derived from the raw input $\mathbf{x}$. We designate different quantities in $\mathcal{Z}$ with a tilde version of their counterparts in $\mathcal{X}$, e.g., the dimensionality of $\mathcal{Z}$ is $\tilde{d}$ and the weight vector is $\tilde{\mathbf{w}}$.[5] The transform Φ that takes us from $\mathcal{X}$ to $\mathcal{Z}$ is called a *feature transform*, which in this case is

$$\Phi(\mathbf{x}) = (1, x_1^2, x_2^2). \tag{3.12}$$

In general, some points in the $\mathcal{Z}$ space may not be valid transforms of any $\mathbf{x} \in \mathcal{X}$, and multiple points in $\mathcal{X}$ may be transformed to the same $\mathbf{z} \in \mathcal{Z}$, depending on the nonlinear transform Φ.

The usefulness of the transform above is that the nonlinear hypothesis h (circle) in the $\mathcal{X}$ space can be represented by a linear hypothesis (line) in the $\mathcal{Z}$ space. Indeed, any linear hypothesis $\tilde{h}$ in $\mathbf{z}$ corresponds to a (possibly nonlinear) hypothesis of $\mathbf{x}$ given by

$$h(\mathbf{x}) = \tilde{h}(\Phi(\mathbf{x})).$$

[5] $\mathcal{Z}$ $\{1\} \times \mathbb{R}^{\tilde{d}}$, where $\tilde{d}$ 2 in this case. We treat $\mathcal{Z}$ as $\tilde{d}$ dimensional since the added coordinate z_0 1 is fixed.

(a) Data in $\mathcal{X}$-space (b) Transformed data in $\mathcal{Z}$ space

$$\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \qquad \mathbf{z} = \Phi(\mathbf{x}) = \begin{bmatrix} 1 \\ x_1^2 \\ x_2^2 \end{bmatrix}$$

Figure 3.5: (a) The original data set that is not linearly separable, but separable by a circle. (b) The transformed data set that is linearly separable in the $\mathcal{Z}$ space. In the figure, $\mathbf{x}_1$ maps to $\mathbf{z}_1$ and $\mathbf{x}_2$ maps to $\mathbf{z}_2$; the circular separator in the $\mathcal{X}$ space maps to the linear separator in the $\mathcal{Z}$ space.

The set of these hypotheses h is denoted by $\mathcal{H}_\Phi$. For instance, when using the feature transform in (3.12), each $h \in \mathcal{H}_\Phi$ is a quadratic curve in $\mathcal{X}$ that corresponds to some line $\tilde{h}$ in $\mathcal{Z}$.

Exercise 3.11

Consider the feature transform Φ in (3.12). What kind of boundary in $\mathcal{X}$ does a hyperplane $\tilde{\mathbf{w}}$ in $\mathcal{Z}$ correspond to in the following cases? Draw a picture that illustrates an example of each case.

(a) $\tilde{w}_1 > 0, \tilde{w}_2 < 0$

(b) $\tilde{w}_1 > 0, \tilde{w}_2 = 0$

(c) $\tilde{w}_1 > 0, \tilde{w}_2 > 0, \tilde{w}_0 < 0$

(d) $\tilde{w}_1 > 0, \tilde{w}_2 > 0, \tilde{w}_0 > 0$

Because the transformed data set $(\mathbf{z}_1, y_1), \cdots, (\mathbf{z}_N, y_N)$ in Figure 3.5(b) is linearly separable in the feature space $\mathcal{Z}$, we can apply PLA on the transformed data set to obtain $\tilde{\mathbf{w}}_{\text{PLA}}$, the PLA solution, which gives us a final hypothesis $g(\mathbf{x}) = \text{sign}(\tilde{\mathbf{w}}_{\text{PLA}}^{\text{T}}\mathbf{z})$ in the $\mathcal{X}$ space, where $\mathbf{z} = \Phi(\mathbf{x})$. The whole process of applying the feature transform before running PLA for linear classification is depicted in Figure 3.6.

The in-sample error in the input space $\mathcal{X}$ is the same as in the feature space $\mathcal{Z}$, so $E_{\text{in}}(g) = 0$. Hyperplanes that achieve $E_{\text{in}}(\tilde{\mathbf{w}}_{\text{PLA}}) = 0$ in $\mathcal{Z}$ correspond to separating curves in the original input space $\mathcal{X}$. For instance,

Figure 3.6: The nonlinear transform for separating non separable data.

as shown in Figure 3.6, the PLA may select the line $\tilde{\mathbf{w}}_{\text{PLA}} = (-0.6, 0.6, 1)$ that separates the transformed data $(\mathbf{z}_1, y_1), \cdots, (\mathbf{z}_N, y_N)$. The corresponding hypothesis $g(\mathbf{x}) = \text{sign}(-0.6 + 0.6 \cdot x_1^2 + x_2^2)$ will separate the original data $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)$. In this case, the decision boundary is an ellipse in $\mathcal{X}$.

How does the feature transform affect the VC bound (3.1)? If we honestly decide on the transform Φ before seeing the data, then with probability at least $1 - \delta$, the bound (3.1) remains true by using $d_{\text{VC}}(\mathcal{H}_\Phi)$ as the VC dimension. For instance, consider the feature transform Φ in (3.12). We know that $\mathcal{Z} = \{1\} \times \mathbb{R}^2$. Since $\mathcal{H}_\Phi$ is the perceptron in $\mathcal{Z}$, $d_{\text{VC}}(\mathcal{H}_\Phi) \leq 3$ (the $\leq$ is because some points $\mathbf{z} \in \mathcal{Z}$ may not be valid transforms of any $\mathbf{x}$, so some dichotomies may not be realizable). We can then substitute N, $d_{\text{VC}}(\mathcal{H}_\Phi)$, and δ into the VC bound. After running PLA on the transformed data set, if we succeed in

getting some g with $E_{\text{in}}(g) = 0$, we can claim that g will perform well out of sample.

It is very important to understand that the claim above is valid only if you decide on Φ *before* seeing the data or trying any algorithms. What if we first try using lines to separate the data, fail, and then use the circles? Then we are effectively using a model that contains both lines and circles, and d_{vc} is no longer 3.

Exercise 3.12

We know that in the Euclidean plane, the perceptron model $\mathcal{H}$ cannot implement all 16 dichotomies on 4 points. That is, $m_{\mathcal{H}}(4) < 16$. Take the feature transform Φ in (3.12).

(a) Show that $m_{\mathcal{H}_\Phi}(3) = 8$.

(b) Show that $m_{\mathcal{H}_\Phi}(4) < 16$.

(c) Show that $m_{\mathcal{H} \cup \mathcal{H}_\Phi}(4) = 16$.

That is, if you used lines, $d_{\text{vc}} = 3$; if you used elipses, $d_{\text{vc}} = 3$; if you used lines and elipses, $d_{\text{vc}} > 3$.

Worse yet, if you actually *look* at the data (e.g., look at the points in Figure 3.1(a)) before deciding on a suitable Φ, you forfeit most of what you learned in Chapter 2 ☹. You have inadvertently explored a huge hypothesis space in your mind to come up with a specific Φ that would work *for this data set*. If you invoke a generalization bound now, you will be charged for the VC dimension of the full space that you explored in your mind, not just the space that Φ creates.

This does not mean that Φ should be chosen blindly. In the credit limit problem for instance, we suggested nonlinear features based on the 'years in residence' field that may be more suitable for linear regression than the raw input. This was based on our understanding of the problem, not on 'snooping' into the training data. Therefore, we pay no price in terms of generalization, and we may well gain a dividend in performance because of a good choice of features.

The feature transform Φ can be general, as long as it is chosen before seeing the data set (as if we cannot emphasize this enough). For instance, you may have noticed that the feature transform in (3.12) only allows us to get very limited types of quadratic curves. Ellipses that do not center at the origin in $\mathcal{X}$ cannot correspond to a hyperplane in $\mathcal{Z}$. To get all possible quadratic curves in $\mathcal{X}$, we could consider the more general feature transform $\mathbf{z} = \Phi_2(\mathbf{x})$,

$$\Phi_2(\mathbf{x}) = (1, x_1, x_2, x_1^2, x_1x_2, x_2^2), \tag{3.13}$$

which gives us the flexibility to represent any quadratic curve in $\mathcal{X}$ by a hyperplane in $\mathcal{Z}$ (the subscript 2 of Φ is for polynomials of degree 2 - quadratic curves). The price we pay is that $\mathcal{Z}$ is now five-dimensional instead of two-dimensional, and hence d_{vc} is doubled from 3 to 6.

Exercise 3.13

Consider the feature transform $\mathbf{z} = \Phi_2(\mathbf{x})$ in (3.13). How can we use a hyperplane $\tilde{\mathbf{w}}$ in $\mathcal{Z}$ to represent the following boundaries in $\mathcal{X}$?

(a) The parabola $(x_1 - 3)^2 + x_2 = 1$.

(b) The circle $(x_1 - 3)^2 + (x_2 - 4)^2 = 1$.

(c) The ellipse $2(x_1 - 3)^2 + (x_2 - 4)^2 = 1$.

(d) The hyperbola $(x_1 - 3)^2 - (x_2 \quad 4)^2 \quad 1$.

(e) The ellipse $2(x_1 + x_2 - 3)^2 + (x_1 - x_2 - 4)^2 = 1$.

(f) The line $2x_1 + x_2 = 1$.

One may further extend Φ_2 to a feature transform Φ_3 for cubic curves in $\mathcal{X}$, or more generally define the feature transform Φ_Q for degree-Q curves in $\mathcal{X}$. The feature transform Φ_Q is called the *Qth order polynomial transform.*

The power of the feature transform should be used with care. It may not be worth it to insist on linear separability and employ a highly complex surface to achieve that. Consider the case of Figure 3.1(a). If we insist on a feature transform that linearly separates the data, it may lead to a significant increase of the VC dimension. As we see in Figure 3.7, no line can separate the training examples perfectly, and neither can any quadratic nor any third-order polynomial curves. Thus, we need to use a fourth-order polynomial transform:

$$\Phi_4(\mathbf{x}) = (1, x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3, x_1^4, x_1^3x_2, x_1^2x_2^2, x_1x_2^3, x_2^4).$$

If you look at the fourth-order decision boundary in Figure 3.7(b), you don't need the VC analysis to tell you that this is an overkill that is unlikely to generalize well to new data. A better option would have been to ignore the two misclassified examples in Figure 3.7(a), separate the other examples perfectly with the line, and accept the small but nonzero E_{in}. Indeed, sometimes our best bet is to go with a simpler hypothesis set while tolerating a small E_{in}.

While our discussion of feature transforms has focused on classification problems, these transforms can be applied equally to regression problems. Both linear regression and logistic regression can be implemented in the feature space $\mathcal{Z}$ instead of the input space $\mathcal{X}$. For instance, linear regression is often coupled with a feature transform to perform nonlinear regression. The N by $d+1$ input matrix X in the algorithm is replaced with the N by $\tilde{d}+1$ matrix Z, while the output vector $\mathbf{y}$ remains the same.

3.4.2 Computation and Generalization

Although using a larger Q gives us more flexibility in terms of the shape of decision boundaries in $\mathcal{X}$, there is a price to be paid. Computation is one issue, and generalization is the other.

Computation is an issue because the feature transform Φ_Q maps a two-dimensional vector $\mathbf{x}$ to $\tilde{d} = \frac{Q(Q+3)}{2}$ dimensions, which increases the memory

(a) Linear fit (b) 4th order polynomial fit

Figure 3.7: Illustration of the nonlinear transform using a data set that is not linearly separable; (a) a line separates the data after omitting a few points, (b) a fourth order polynomial separates all the points.

and computational costs. Things could get worse if $\mathbf{x}$ is in a higher dimension to begin with.

Exercise 3.14

Consider the Qth order polynomial transform Φ_{Q} for $\mathcal{X} = \mathbb{R}^d$. What is the dimensionality $\tilde{d}$ of the feature space $\mathcal{Z}$ (excluding the fixed coordinate $z_0 = 1$). Evaluate your result on $d \in \{2, 3, 5, 10\}$ and $Q \in \{2, 3, 5, 10\}$.

The other important issue is generalization. If Φ_{Q} is the feature transform of a two-dimensional input space, there will be $\tilde{d} = \frac{Q(Q+3)}{2}$ dimensions in $\mathcal{Z}$, and $d_{\mathrm{vc}}(\mathcal{H}_\Phi)$ can be as high as $\frac{Q(Q+3)}{2} + 1$. This means that the second term in the VC bound (3.1) can grow significantly. In other words, we would have a weaker guarantee that E_{out} will be small. For instance, if we use $\Phi = \Phi_{50}$, the VC dimension of $\mathcal{H}_\Phi$ could be as high as $\frac{(50)(53)}{2} + 1 = 1326$ instead of the original $d_{\mathrm{vc}} = 3$. Applying the rule of thumb that the amount of data needed is proportional to the VC dimension, we would need hundreds of times more data than we would if we didn't use a feature transform, in order to achieve the same level of generalization error.

Exercise 3.15

High-dimensional feature transforms are by no means the only transforms that we can use. We can take the tradeoff in the other direction, and use low dimensional feature transforms as well (to achieve an even lower generalization error bar).

(continued on next page)

Consider the following feature transform, which maps a d-dimensional $\mathbf{x}$ to a one-dimensional $\mathbf{z}$, keeping only the kth coordinate of $\mathbf{x}$.

$$\Phi_{(k)}(\mathbf{x}) = (1, x_k). \tag{3.14}$$

Let $\mathcal{H}_k$ be the set of perceptrons in the feature space.

(a) Prove that $d_{\text{vc}}(\mathcal{H}_k) = 2$.

(b) Prove that $d_{\text{vc}}(\cup_{k=1}^{d} \mathcal{H}_k) \leq 2(\log_2 d + 1)$.

$\mathcal{H}_k$ is called the *decision stump* model on dimension k.

The problem of generalization when we go to high-dimensional space is sometimes balanced by the advantage we get in approximating the target better. As we have seen in the case of using quadratic curves instead of lines, the transformed data became linearly separable, reducing E_{in} to 0. In general, when choosing the appropriate dimension for the feature transform, we cannot avoid the approximation-generalization tradeoff,

higher $\tilde{d}$	better chance of being linearly separable ($E_{\text{in}} \downarrow$)	$d_{\text{vc}} \uparrow$
lower $\tilde{d}$	possibly not linearly separable ($E_{\text{in}} \uparrow$)	$d_{\text{vc}} \downarrow$

Therefore, choosing a feature transform before seeing the data is a non-trivial task. When we apply learning to a particular problem, some understanding of the problem can help in choosing features that work well. More generally, there are some guidelines for choosing a suitable transform, or a suitable model, which we will discuss in Chapter 4.

Exercise 3.16

Write down the steps of the algorithm that combines Φ_3 with linear regression. How about using Φ_{10} instead? Where is the main computational bottleneck of the resulting algorithm?

Example 3.5. Let's revisit the handwritten digit recognition example. We can try a different way of decomposing the big task of separating ten digits to smaller tasks. One decomposition is to separate digit 1 from all the other digits. Using intensity and symmetry as our input variables like we did before, the scatter plot of the training data is shown next. A line can roughly separate digit 1 from the rest, but a more complicated curve might do better.

We use linear regression (for classification), first without any feature transform. The results are shown below (LHS). We get $E_{\text{in}} = 2.13\%$ and $E_{\text{out}} = 2.38\%$.

Linear model
$E_{\text{in}} = 2.13\%$
$E_{\text{out}} = 2.38\%$

3rd order polynomial model
$E_{\text{in}} = 1.75\%$
$E_{\text{out}} = 1.87\%$

Classification of the digits data ('1' versus 'not 1') using linear and third order polynomial models.

When we run linear regression with Φ_3, the third-order polynomial transform, we obtain a better fit to the data, with a lower $E_{\text{in}} = 1.75\%$. The result is depicted in the RHS of the figure. In this case, the better in-sample fit also resulted in a better out-of-sample performance, with $E_{\text{out}} = 1.87\%$. □

Linear models, a final pitch. The linear model (for classification or regression) is an often overlooked resource in the arena of learning from data. Since efficient learning algorithms exist for linear models, they are low overhead. They are also very robust and have good generalization properties. A sound

policy to follow when learning from data is to *first* try a linear model. Because of the good generalization properties of linear models, not much can go wrong. If you get a good fit to the data (low E_{in}), then you are done. If you do not get a good enough fit to the data and decide to go for a more complex model, you will pay a price in terms of the VC dimension as we have seen in Exercise 3.12, but the price is modest.

3.5 Problems

Problem 3.1 Consider the double semi-circle "toy" learning task below.

There are two semi circles of width thk with inner radius rad, separated by sep as shown (red is -1 and blue is $+1$). The center of the top semi circle is aligned with the middle of the edge of the bottom semi circle. This task is linearly separable when $sep \geq 0$, and not so for $sep < 0$. Set $rad = 10$, $thk = 5$ and $sep = 5$. Then, generate $2,000$ examples uniformly, which means you will have approximately $1,000$ examples for each class.

(a) Run the PLA starting from $\mathbf{w} = \mathbf{0}$ until it converges. Plot the data and the final hypothesis.

(b) Repeat part (a) using the linear regression (for classification) to obtain $\mathbf{w}$. Explain your observations.

Problem 3.2 For the double semi circle task in Problem 3.1, vary sep in the range $\{0.2, 0.4, \ldots, 5\}$. Generate $2,000$ examples and run the PLA starting with $\mathbf{w} = \mathbf{0}$. Record the number of iterations PLA takes to converge.

Plot sep versus the number of iterations taken for PLA to converge. Explain your observations. *[Hint: Problem 1.3.]*

Problem 3.3 For the double semi circle task in Problem 3.1, set $sep = -5$ and generate $2,000$ examples.

(a) What will happen if you run PLA on those examples?

(b) Run the pocket algorithm for $100,000$ iterations and plot E_{in} versus the iteration number t.

(c) Plot the data and the final hypothesis in part (b).

(continued on next page)

(d) Use the linear regression algorithm to obtain the weights $\mathbf{w}$, and compare this result with the pocket algorithm in terms of computation time and quality of the solution.

(e) Repeat (b) – (d) with a 3rd order polynomial feature transform.

Problem 3.4 In Problem 1.5, we introduced the Adaptive Linear Neuron (Adaline) algorithm for classification. Here, we derive Adaline from an optimization perspective.

(a) Consider $E_n(\mathbf{w}) = (\max(0, 1 - y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n))^2$. Show that $E_n(\mathbf{w})$ is continuous and differentiable. Write down the gradient $\nabla E_n(\mathbf{w})$.

(b) Show that $E_n(\mathbf{w})$ is an upper bound for $[\![\mathrm{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) \neq y_n]\!]$. Hence, $\frac{1}{N}\sum_{n=1}^{N} E_n(\mathbf{w})$ is an upper bound for the in sample classification error $E_{\mathrm{in}}(\mathbf{w})$.

(c) Argue that the Adaline algorithm in Problem 1.5 performs stochastic gradient descent on $\frac{1}{N}\sum_{n=1}^{N} E_n(\mathbf{w})$.

Problem 3.5

(a) Consider

$$E_n(\mathbf{w}) = \max(0, 1 - y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n).$$

Show that $E_n(\mathbf{w})$ is continuous and differentiable except when $y_n = \mathbf{w}^{\mathrm{T}}\mathbf{x}_n$.

(b) Show that $E_n(\mathbf{w})$ is an upper bound for $[\![\mathrm{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) \neq y_n]\!]$. Hence, $\frac{1}{N}\sum_{n=1}^{N} E_n(\mathbf{w})$ is an upper bound for the in sample classification error $E_{\mathrm{in}}(\mathbf{w})$.

(c) Apply stochastic gradient descent on $\frac{1}{N}\sum_{n=1}^{N} E_n(\mathbf{w})$ (ignoring the singular case of $\mathbf{w}^{\mathrm{T}}\mathbf{x}_n = y_n$) and derive a new perceptron learning algorithm.

Problem 3.6 Derive a linear programming algorithm to fit a linear model for classification using the following steps. A linear program is an optimization problem of the following form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & \mathbf{c}^{\mathrm{T}}\mathbf{z} \\ \text{subject to} \quad & \mathrm{A}\mathbf{z} \leq \mathbf{b}. \end{aligned}$$

A, $\mathbf{b}$ and $\mathbf{c}$ are parameters of the linear program and $\mathbf{z}$ is the optimization variable. This is such a well studied optimization problem that most mathematics software have canned optimization functions which solve linear programs.

(a) For linearly separable data, show that for some $\mathbf{w}$, $y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) \geq 1$ for $n = 1, \ldots, N$.

(b) Formulate the task of finding a separating $\mathbf{w}$ for separable data as a linear program. You need to specify what the parameters $\mathrm{A}, \mathbf{b}, \mathbf{c}$ are and what the optimization variable $\mathbf{z}$ is.

(c) If the data is not separable, the condition in (a) cannot hold for every n. Thus introduce the violation $\xi_n \geq 0$ to capture the amount of violation for example $\mathbf{x}_n$. So, for $n = 1, \ldots, N$,

$$\begin{aligned} y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) &\geq 1 - \xi_n, \\ \xi_n &\geq 0. \end{aligned}$$

Naturally, we would like to minimize the amount of violation. One intu itive approach is to minimize $\sum_{n=1}^{N} \xi_n$, i.e., we want $\mathbf{w}$ that solves

$$\begin{aligned} \min_{\mathbf{w},\xi_n} \quad & \sum_{n=1}^{N} \xi_n \\ \text{subject to} \quad & y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) \geq 1 - \xi_n, \\ & \xi_n \geq 0, \end{aligned}$$

where the inequalities must hold for $n = 1, \ldots, N$. Formulate this prob lem as a linear program.

(d) Argue that the linear program you derived in (c) and the optimization problem in Problem 3.5 are equivalent.

Problem 3.7 Use the linear programming algorithm from Problem 3.6 on the learning task in Problem 3.1 for the separable ($sep = 5$) and the non-separable ($sep = -5$) cases.

Compare your results to the linear regression approach with and without the 3rd order polynomial feature transform.

Problem 3.8 For linear regression, the out of sample error is

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

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

$$h^*(\mathbf{x}) = \mathbb{E}[y \mid \mathbf{x}].$$

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

Problem 3.9 Assuming that $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is invertible, show by direct comparison with Equation (3.4) that $E_{\text{in}}(\mathbf{w})$ can be written as

$$\begin{aligned}&E_{\text{in}}(\mathbf{w})\\&= (\mathbf{w} - (\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}\mathbf{y})^{\mathrm{T}}(\mathrm{X}^{\mathrm{T}}\mathrm{X})(\mathbf{w} - (\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}\mathbf{y}) + \mathbf{y}^{\mathrm{T}}(\mathrm{I} - \mathrm{X}(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}})\mathbf{y}.\end{aligned}$$

Use this expression for E_{in} to obtain $\mathbf{w}_{\text{lin}}$. What is the in sample error? *[Hint: The matrix $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is positive definite.]*

Problem 3.10 Exercise 3.3 studied some properties of the hat matrix $\mathrm{H} = \mathrm{X}(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}$, where X is a N by $d+1$ matrix, and $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is invertible. Show the following additional properties.

(a) Every eigenvalue of H is either 0 or 1. *[Hint: Exercise 3.3(b).]*

(b) Show that the trace of a symmetric matrix equals the sum of its eigenvalues. *[Hint: Use the spectral theorem and the cyclic property of the trace. Note that the same result holds for non-symmetric matrices, but is a little harder to prove.]*

(c) How many eigenvalues of H are 1? What is the rank of H? *[Hint: Exercise 3.3(d).]*

Problem 3.11 Consider the linear regression problem setup in Exercise 3.4, where the data comes from a genuine linear relationship with added noise. The noise for the different data points is assumed to be iid with zero mean and variance σ^2. Assume that the 2nd moment matrix $\Sigma = \mathbb{E}_{\mathbf{x}}[\mathbf{x}\mathbf{x}^{\mathrm{T}}]$ is non-singular. Follow the steps below to show that, with high probability, the out-of-sample error on average is

$$E_{\text{out}}(\mathbf{w}_{\text{lin}}) = \sigma^2\left(1 + \frac{d+1}{N} + o(\tfrac{1}{N})\right).$$

(a) For a test point $\mathbf{x}$, show that the error $y - g(\mathbf{x})$ is

$$\epsilon - \mathbf{x}^{\mathrm{T}}(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}\boldsymbol{\epsilon},$$

where ϵ is the noise realization for the test point and $\boldsymbol{\epsilon}$ is the vector of noise realizations on the data.

(b) Take the expectation with respect to the test point, i.e., $\mathbf{x}$ and ϵ, to obtain an expression for E_{out}. Show that

$$E_{\text{out}} = \sigma^2 + \operatorname{trace}\left(\Sigma(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\mathrm{X}^{\mathrm{T}}(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\right).$$

[Hints: $a = \operatorname{trace}(a)$ for any scalar a; $\operatorname{trace}(\mathrm{AB}) = \operatorname{trace}(\mathrm{BA})$; expectation and trace commute.]

(c) What is $\mathbb{E}_{\boldsymbol{\epsilon}}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}]$?

(d) Take the expectation with respect to ϵ to show that, on average,

$$E_{\text{out}} = \sigma^2 + \frac{\sigma^2}{N}\text{trace}\left(\Sigma(\tfrac{1}{N}\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\right).$$

Note that $\frac{1}{N}\mathrm{X}^{\mathrm{T}}\mathrm{X} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}$ is an N sample estimate of Σ. So $\frac{1}{N}\mathrm{X}^{\mathrm{T}}\mathrm{X} \approx \Sigma$. If $\frac{1}{N}\mathrm{X}^{\mathrm{T}}\mathrm{X} = \Sigma$, then what is E_{out} on average?

(e) Show that (after taking the expectation over the data noise) with high probability,

$$E_{\text{out}} = \sigma^2\left(1 + \frac{d+1}{N} + o(\tfrac{1}{N})\right).$$

[Hint: By the law of large numbers $\frac{1}{N}\mathrm{X}^{\mathrm{T}}\mathrm{X}$ converges in probability to Σ, and so by continuity of the inverse at Σ, $\left(\frac{1}{N}\mathrm{X}^{\mathrm{T}}\mathrm{X}\right)^{-1}$ converges in probability to Σ^{-1}.]

Problem 3.12 In linear regression, the in sample predictions are given by $\hat{\mathbf{y}} = \mathrm{H}\mathbf{y}$, where $\mathrm{H} = \mathrm{X}(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}$. Show that H is a projection matrix, i.e. $\mathrm{H}^2 = \mathrm{H}$. So $\hat{\mathbf{y}}$ is the projection of $\mathbf{y}$ onto some space. What is this space?

Problem 3.13 This problem creates a linear regression algorithm from a good algorithm for linear classification. As illustrated, the idea is to take the original data and shift it in one direction to get the $+1$ data points; then, shift it in the opposite direction to get the -1 data points.

Original data for the one dimensional regression prob lem

Shifted data viewed as a two dimensional classifica tion problem

More generally, The data $(\mathbf{x}_n, y_n)$ can be viewed as data points in $\mathbb{R}^{d+1}$ by treating the y value as the $(d+1)$th coordinate.

(continued on next page)

Now, construct positive and negative points

$$\begin{aligned}\mathcal{D}_+ &= (\mathbf{x}_1, y_1) + \mathbf{a}, \ldots, (\mathbf{x}_N, y_N) + \mathbf{a} \\ \mathcal{D}_- &= (\mathbf{x}_1, y_1) - \mathbf{a}, \ldots, (\mathbf{x}_N, y_N) - \mathbf{a},\end{aligned}$$

where $\mathbf{a}$ is a perturbation parameter. You can now use the linear programming algorithm in Problem 3.6 to separate $\mathcal{D}_+$ from $\mathcal{D}_-$. The resulting separating hyperplane can be used as the regression 'fit' to the original data.

(a) How many weights are learned in the classification problem? How many weights are needed for the linear fit in the regression problem?

(b) The linear fit requires weights $\mathbf{w}$, where $h(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}$. Suppose the weights returned by solving the classification problem are $\mathbf{w}_{\text{class}}$. Derive an expression for $\mathbf{w}$ as a function of $\mathbf{w}_{\text{class}}$.

(c) Generate a data set $y_n = x_n^2 + \sigma\epsilon_n$ with $N = 50$, where x_n is uniform on $[0, 1]$ and ϵ_n is zero mean Gaussian noise; set $\sigma = 0.1$. Plot $\mathcal{D}_+$ and $\mathcal{D}_-$ for $\mathbf{a} = \begin{bmatrix} 0 \\ 0.1 \end{bmatrix}$.

(d) Give comparisons of the resulting fits from running the classification approach and the analytic pseudo-inverse algorithm for linear regression.

Problem 3.14 In a regression setting, assume the target function is linear, so $f(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathbf{w}_f$, and $\mathbf{y} = \mathrm{X}\mathbf{w}_f + \boldsymbol{\epsilon}$, where the entries in $\boldsymbol{\epsilon}$ are zero mean, iid with variance σ^2. In this problem derive the bias and variance as follows.

(a) Show that the average function is $\bar{g}(\mathbf{x}) = f(\mathbf{x})$, no matter what the size of the data set, as long as $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is invertible. What is the bias?

(b) What is the variance? *[Hint: Problem 3.11]*

Problem 3.15 In the text we derived that the linear regression solution weights must satisfy $\mathrm{X}^{\mathrm{T}}\mathrm{X}\mathbf{w} = \mathrm{X}^{\mathrm{T}}\mathbf{y}$. If $\mathrm{X}^{\mathrm{T}}\mathrm{X}$ is not invertible, the solution $\mathbf{w}_{\text{lin}} = (\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}\mathbf{y}$ won't work. In this event, there will be many solutions for $\mathbf{w}$ that minimize E_{in}. Here, you will derive one such solution. Let ρ be the rank of X. Assume that the singular value decomposition (SVD) of X is $\mathrm{X} = \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}$, where $\mathrm{U} \in \mathbb{R}^{N\times\rho}$ satisfies $\mathrm{U}^{\mathrm{T}}\mathrm{U} = \mathrm{I}_\rho$, $\mathrm{V} \in \mathbb{R}^{(d+1)\times\rho}$ satisfies $\mathrm{V}^{\mathrm{T}}\mathrm{V} = \mathrm{I}_\rho$, and $\Gamma \in \mathbb{R}^{\rho\times\rho}$ is a positive diagonal matrix.

(a) Show that $\rho < d + 1$.

(b) Show that $\mathbf{w}_{\text{lin}} = \mathrm{V}\Gamma^{-1}\mathrm{U}^{\mathrm{T}}\mathbf{y}$ satisfies $\mathrm{X}^{\mathrm{T}}\mathrm{X}\mathbf{w}_{\text{lin}} = \mathrm{X}^{\mathrm{T}}\mathbf{y}$, and hence is a solution.

(c) Show that for any other solution that satisfies $\mathrm{X}^{\mathrm{T}}\mathrm{X}\mathbf{w} = \mathrm{X}^{\mathrm{T}}\mathbf{y}$, $\|\mathbf{w}_{\text{lin}}\| < \|\mathbf{w}\|$. That is, the solution we have constructed is the minimum norm set of weights that minimizes E_{in}.

Problem 3.16 In Example 3.4, it is mentioned that the output of the final hypothesis $g(\mathbf{x})$ learned using logistic regression can be thresholded to get a 'hard' (± 1) classification. This problem shows how to use the risk matrix introduced in Example 1.1 to obtain such a threshold.

Consider fingerprint verification, as in Example 1.1. After learning from the data using logistic regression, you produce the final hypothesis

$$g(\mathbf{x}) = \mathbb{P}[y = +1 \mid \mathbf{x}],$$

which is your estimate of the probability that $y = +1$. Suppose that the cost matrix is given by

		True classification	
		$+1$ (correct person)	-1 (intruder)
you say	$+1$	0	c_a
	-1	c_r	0

For a new person with fingerprint $\mathbf{x}$, you compute $g(\mathbf{x})$ and you now need to de cide whether to accept or reject the person (i.e., you need a hard classification). So, you will accept if $g(\mathbf{x}) \geq \kappa$, where κ is the threshold.

(a) Define the cost(accept) as your expected cost if you accept the person. Similarly define cost(reject). Show that

$$\begin{aligned} \text{cost(accept)} &= (1 - g(\mathbf{x}))c_a, \\ \text{cost(reject)} &= g(\mathbf{x})c_r. \end{aligned}$$

(b) Use part (a) to derive a condition on $g(\mathbf{x})$ for accepting the person and hence show that

$$\kappa = \frac{c_a}{c_a + c_r}.$$

(c) Use the cost matrices for the Supermarket and CIA applications in Example 1.1 to compute the threshold κ for each of these two cases. Give some intuition for the thresholds you get.

Problem 3.17 Consider a function

$$E(u, v) = e^u + e^{2v} + e^{uv} + u^2 - 3uv + 4v^2 - 3u - 5v,$$

(a) Approximate $E(u + \Delta u, v + \Delta v)$ by $\hat{E}_1(\Delta u, \Delta v)$, where $\hat{E}_1$ is the first-order Taylor's expansion of E around $(u, v) = (0, 0)$. Suppose $\hat{E}_1(\Delta u, \Delta v) = a_u \Delta u + a_v \Delta v + a$. What are the values of a_u, a_v, and a?

(continued on next page)

(b) Minimize $\hat{E}_1$ over all possible $(\Delta u, \Delta v)$ such that $\|(\Delta u, \Delta v)\| = 0.5$. In this chapter, we proved that the optimal column vector $\begin{bmatrix}\Delta u\\ \Delta v\end{bmatrix}$ is parallel to the column vector $-\nabla E(u,v)$, which is called the *negative gradient direction*. Compute the optimal $(\Delta u, \Delta v)$ and the resulting $E(u+\Delta u, v+\Delta v)$.

(c) Approximate $E(u+\Delta u, v+\Delta v)$ by $\hat{E}_2(\Delta u, \Delta v)$, where $\hat{E}_2$ is the second order Taylor's expansion of E around $(u,v) = (0,0)$. Suppose

$$\hat{E}_2(\Delta u, \Delta v) = b_{uu}(\Delta u)^2 + b_{vv}(\Delta v)^2 + b_{uv}(\Delta u)(\Delta v) + b_u \Delta u + b_v \Delta v + b.$$

What are the values of b_{uu}, b_{vv}, b_{uv}, b_u, b_v, and b?

(d) Minimize $\hat{E}_2$ over all possible $(\Delta u, \Delta v)$ (regardless of length). Use the fact that $\nabla^2 E(u,v)\big|_{(0,0)}$ (the Hessian matrix at $(0,0)$) is positive definite to prove that the optimal column vector

$$\begin{bmatrix}\Delta u^*\\ \Delta v^*\end{bmatrix} = -\left(\nabla^2 E(u,v)\right)^{-1} \nabla E(u,v),$$

which is called the *Newton direction*.

(e) Numerically compute the following values:

(i) the vector $(\Delta u, \Delta v)$ of length 0.5 along the Newton direction, and the resulting $E(u+\Delta u, v+\Delta v)$.

(ii) the vector $(\Delta u, \Delta v)$ of length 0.5 that minimizes $E(u+\Delta u, v+\Delta v)$, and the resulting $E(u+\Delta u, v+\Delta v)$. (*Hint: Let* $\Delta u = 0.5 \sin\theta$.)

Compare the values of $E(u+\Delta u, v+\Delta v)$ in (b), (e i), and (e ii). Briefly state your findings.

The negative gradient direction and the Newton direction are quite fundamental for designing optimization algorithms. It is important to understand these directions and put them in your toolbox for designing learning algorithms.

Problem 3.18 Take the feature transform Φ_2 in Equation (3.13) as Φ.

(a) Show that $d_{\text{vc}}(\mathcal{H}_\Phi) \le 6$.

(b) Show that $d_{\text{vc}}(\mathcal{H}_\Phi) > 4$. *[Hint: Exercise 3.12]*

(c) Give an upper bound on $d_{\text{vc}}(\mathcal{H}_{\Phi_k})$ for $\mathcal{X} = \mathbb{R}^d$.

(d) Define

$$\tilde{\Phi}_2 \colon \mathbf{x} \to (1, x_1, x_2, x_1 + x_2, x_1 - x_2, x_1^2, x_1 x_2, x_2 x_1, x_2^2) \text{ for } \mathbf{x} \in \mathbb{R}^2.$$

Argue that $d_{\text{vc}}(\mathcal{H}_{\Phi_2}) = d_{\text{vc}}(\mathcal{H}_{\tilde{\Phi}_2})$. In other words, while $\tilde{\Phi}_2(\mathcal{X}) \in \mathbb{R}^9$, $d_{\text{vc}}(\mathcal{H}_{\tilde{\Phi}_2}) \le 6 < 9$. Thus, the dimension of $\Phi(\mathcal{X})$ only gives an upper bound of $d_{\text{vc}}(\mathcal{H}_\Phi)$, and the exact value of $d_{\text{vc}}(\mathcal{H}_\Phi)$ can depend on the components of the transform.

Problem 3.19 A Transformer thinks the following procedures would work well in learning from two-dimensional data sets of any size. Please point out if there are any potential problems in the procedures:

(a) Use the feature transform

$$\Phi(\mathbf{x}) = \begin{cases} (\underbrace{0, \cdots, 0}_{n-1}, 1, 0, \cdots) & \text{if } \mathbf{x} = \mathbf{x}_n \\ (0, 0, \cdots, 0) & \text{otherwise .} \end{cases}$$

before running PLA.

(b) Use the feature transform Φ with

$$\phi_n(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_n\|^2}{2\gamma^2}\right)$$

using some very small γ.

(c) Use the feature transform Φ that consists of all

$$\phi_{i,j}(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - (i,j)\|^2}{2\gamma^2}\right),$$

before running PLA, with $i \in \{0, \frac{1}{100}, \ldots, 1\}$ and $j \in \{0, \frac{1}{100}, \ldots, 1\}$.

Chapter 4

Overfitting

Paraskavedekatriaphobia[1] (fear of Friday the 13th), and superstitions in general, are perhaps the most illustrious cases of the human ability to overfit. Unfortunate events are memorable, and given a few such memorable events, it is natural to *try* and find an explanation. In the future, will there be more unfortunate events on Friday the 13th's than on any other day?

Overfitting is the phenomenon where fitting the observed facts (data) well no longer indicates that we will get a decent out-of-sample error, and may actually lead to the opposite effect. You have probably seen cases of overfitting when the learning model is more complex than is necessary to represent the target function. The model uses its additional degrees of freedom to fit idiosyncrasies in the data (for example, noise), yielding a final hypothesis that is inferior. Overfitting can occur even when the hypothesis set contains only functions which are *far simpler* than the target function, and so the plot thickens ☺.

The ability to deal with overfitting is what separates professionals from amateurs in the field of learning from data. We will cover three themes: When does overfitting occur? What are the tools to combat overfitting? How can one estimate the degree of overfitting and 'certify' that a model is good, or better than another? Our emphasis will be on techniques that work well in practice.

4.1 When Does Overfitting Occur?

Overfitting literally means "Fitting the data more than is warranted." The main case of overfitting is when you pick the hypothesis with lower E_{in}, and it results in higher E_{out}. This means that E_{in} alone is no longer a good guide for learning. Let us start by identifying the cause of overfitting.

[1] from the Greek *paraskevi* (Friday), *dekatreis* (thirteen), *phobia* (fear)

Consider a simple one-dimensional regression problem with five data points. We do not know the target function, so let's select a general model, maximizing our chance to capture the target function. Since 5 data points can be fit by a 4th order polynomial, we select 4th order polynomials.

The result is shown on the right. The target function is a 2nd order polynomial (blue curve), with a *little* added noise in the data points. Though the target is simple, the learning algorithm used the full power of the 4th order polynomial to fit the data exactly, but the result does not look anything like the target function. The data has been 'overfit.' The little noise in the data has misled the learning, for if there were no noise, the fitted red curve would exactly match the target. This is a typical overfitting scenario, in which a complex model uses its additional degrees of freedom to 'learn' the noise.

The fit has zero in-sample error but huge out-of-sample error, so this is a case of bad generalization (as discussed in Chapter 2) – a likely outcome when overfitting is occurring. However, our definition of overfitting goes beyond bad generalization for any given hypothesis. Instead, overfitting applies to a *process*: in this case, the process of picking a hypothesis with lower and lower E_{in} resulting in higher and higher E_{out}.

4.1.1 A Case Study: Overfitting with Polynomials

Let's dig deeper to gain a better understanding of when overfitting occurs. We will illustrate the main concepts using data in one-dimension and polynomial regression, a special case of a linear model that uses the feature transform $x \mapsto (1, x, x^2, \cdots)$. Consider the two regression problems below:

(a) 10th order target function (b) 50th order target function

In both problems, the target function is a polynomial and the data set $\mathcal{D}$ contains 15 data points. In (a), the target function is a 10th order polynomial

(a) Noisy low order target (b) Noiseless high order target

Figure 4.1: Fits using 2nd and 10th order polynomials to 15 data points. In (a), the data are noisy and the target is a 10th order polynomial. In (b) the data are noiseless and the the target is a 50th order polynomial.

and the sampled data are noisy (the data do not lie on the target function curve). In (b), the target function is a 50th order polynomial and the data are noiseless.

The best 2nd and 10th order fits are shown in Figure 4.1, and the in-sample and out-of-sample errors are given in the following table.

	10th order noisy target	
	2nd Order	10th Order
E_{in}	0.050	0.034
E_{out}	0.127	**9.00**

	50th order noiseless target	
	2nd Order	10th Order
E_{in}	0.029	10^{-5}
E_{out}	0.120	**7680**

What the learning algorithm sees is the data, not the target function. In both cases, the 10th order polynomial heavily overfits the data, and results in a nonsensical final hypothesis which does not resemble the target function. The 2nd order fits do not capture the full nature of the target function either, but they do at least capture its general trend, resulting in significantly lower out-of-sample error. The 10th order fits have lower in-sample error and higher out-of-sample error, so this is indeed a case of overfitting that results in pathologically bad generalization.

Exercise 4.1

Let $\mathcal{H}_2$ and $\mathcal{H}_{10}$ be the 2nd and 10th order hypothesis sets respectively. Specify these sets as parameterized sets of functions. Show that $\mathcal{H}_2 \subset \mathcal{H}_{10}$.

These two examples reveal some surprising phenomena. Let's consider first the 10th order target function, Figure 4.1(a). Here is the scenario. Two learners, O (for overfitted) and R (for restricted), know that the target function is a 10th order polynomial, and that they will receive 15 noisy data points. Learner O

Figure 4.2: Overfitting is occurring for N in the shaded gray region because by choosing $\mathcal{H}_{10}$ which has *better* E_{in}, you get *worse* E_{out}.

uses model $\mathcal{H}_{10}$, which is known to contain the target function, and finds the best fitting hypothesis to the data. Learner R uses model $\mathcal{H}_2$, and similarly finds the best fitting hypothesis to the data.

The surprising thing is that learner R wins (lower out-of-sample error) by using the smaller model, even though she has *knowingly* given up the ability to implement the true target function. Learner R trades off a worse in-sample error for a huge gain in the generalization error, ultimately resulting in lower out-of-sample error. What is funny here? A folklore belief about learning is that best results are obtained by incorporating as much information about the target function as is available. But as we see here, even if we *know* the order of the target and naively incorporate this knowledge by choosing the model accordingly ($\mathcal{H}_{10}$), the performance is inferior to that demonstrated by the more 'stable' 2nd order model.

The models $\mathcal{H}_2$ and $\mathcal{H}_{10}$ were in fact the ones used to generate the learning curves in Chapter 2, and we use those same learning curves to illustrate overfitting in Figure 4.2. If you mentally superimpose the two plots, you can see that there is a range of N for which $\mathcal{H}_{10}$ has lower E_{in} but higher E_{out} than $\mathcal{H}_2$ does, a case in point of overfitting.

Is learner R always going to prevail? Certainly not. For example, if the data was noiseless, then indeed learner O would recover the target function exactly from 15 data points, while learner R would have no hope. This brings us to the second example, Figure 4.1(b). Here, the data *is* noiseless, but the target function is very complex (50th order polynomial). Again learner R wins, and again because learner O heavily overfits the data.

4.1.2 Catalysts for Overfitting

A skeptical reader should ask whether the examples in Figure 4.1 are just pathological constructions created by the authors, or is overfitting a real phenomenon which has to be considered carefully when learning from data? The next exercise guides you through an experimental design for studying overfitting within our current setup. We will use the results from this experiment to serve two purposes: to convince you that overfitting is not the result of some rare pathological construction, and to unravel some of the conditions conducive to overfitting.

Exercise 4.2 [Experimental design for studying overfitting]

This is a reading exercise that sets up an experimental framework to study various aspects of overfitting. The reader interested in implementing the experiment can find the details fleshed out in Problem 4.4. The input space is $\mathcal{X} = [-1,1]$, with uniform input probability density, $P(x) = \frac{1}{2}$. We consider the two models $\mathcal{H}_2$ and $\mathcal{H}_{10}$.

The target is a degree-Q_f polynomial, which we write $f(x) = \sum_{q=0}^{Q_f} a_q L_q(x)$, where $L_i(x)$ are polynomials of increasing complexity (the Legendre polynomials). The data set is $\mathcal{D} = (x_1, y_1), \ldots, (x_N, y_N)$, where $y_n = f(x_n) + \sigma\epsilon_n$ and ϵ_n are *iid* (independent and identically distributed) standard Normal random variates.

For a single experiment, with specified values for Q_f, N, σ, generate a random degree-Q_f target function by selecting coefficients a_i independently from a standard Normal, rescaling them so that $\mathbb{E}_{\mathbf{a},x}\left[f^2\right] = 1$. Generate a data set, selecting $x_1, \ldots, x_N$ independently according to $P(x)$ and $y_n = f(x_n) + \sigma\epsilon_n$. Let g_2 and g_{10} be the best fit hypotheses to the data from $\mathcal{H}_2$ and $\mathcal{H}_{10}$ respectively, with out-of-sample errors $E_{\text{out}}(g_2)$ and $E_{\text{out}}(g_{10})$.

Vary Q_f, N, σ, and for each combination of parameters, run a large number of experiments, each time computing $E_{\text{out}}(g_2)$ and $E_{\text{out}}(g_{10})$. Averaging these out-of-sample errors gives estimates of the expected out-of-sample error for the given learning scenario (Q_f, N, σ) using $\mathcal{H}_2$ and $\mathcal{H}_{10}$.

Exercise 4.2 set up an experiment to study how the noise level σ^2, the target complexity Q_f, and the number of data points N relate to overfitting. We compare the final hypothesis $g_{10} \in \mathcal{H}_{10}$ (larger model) to the final hypothesis $g_2 \in \mathcal{H}_2$ (smaller model). Clearly, $E_{\text{in}}(g_{10}) \leq E_{\text{in}}(g_2)$ since g_{10} has more degrees of freedom to fit the data. What is surprising is how often g_{10} overfits the data, resulting in $E_{\text{out}}(g_{10}) > E_{\text{out}}(g_2)$. Let us define the overfit measure as $E_{\text{out}}(g_{10}) \quad E_{\text{out}}(g_2)$. The more positive this measure is, the more severe overfitting would be.

Figure 4.3 shows how the extent of overfitting depends on certain parameters of the learning problem (the results are from our implementation of Exercise 4.2). In the figure, the colors map to the level of overfitting, with redder

(a) Stochastic noise (b) Deterministic noise

Figure 4.3: How overfitting depends on the noise σ^2, the target function complexity Q_f, and the number of data points N. The colors map to the overfit measure $E_{\text{out}}(\mathcal{H}_{10}) - E_{\text{out}}(\mathcal{H}_2)$. In (a) we see how overfitting depends on σ^2 and N, with $Q_f = 20$. As σ^2 increases we are adding stochastic noise to the data. In (b) we see how overfitting depends on Q_f and N, with $\sigma^2 = 0.1$. As Q_f increases we are adding deterministic noise to the data.

regions showing worse overfitting. These red regions are large overfitting is real, and here to stay.

Figure 4.3(a) reveals that there is less overfitting when the noise level σ^2 drops or when the number of data points N increases (the linear pattern in Figure 4.3(a) is typical). Since the 'signal' f is normalized to $\mathbb{E}[f^2] = 1$, the noise level σ^2 is automatically calibrated to the signal level. Noise leads the learning astray, and the larger, more complex model is more susceptible to noise than the simpler one because it has more ways to go astray. Figure 4.3(b) reveals that target function complexity Q_f affects overfitting in a similar way to noise, albeit nonlinearly. To summarize,

Number of data points	$\uparrow$	Overfitting	$\downarrow$
Noise	$\uparrow$	Overfitting	$\uparrow$
Target complexity	$\uparrow$	Overfitting	$\uparrow$

Deterministic noise. Why does a higher target complexity lead to more overfitting when comparing the same two models? The intuition is that for a given learning model, there is a best approximation to the target function. The part of the target function 'outside' this best fit acts like noise in the data. We can call this *deterministic noise* to differentiate it from the random *stochastic noise*. Just as stochastic noise cannot be modeled, the deterministic noise is that part of the target function which cannot be modeled. The learning algorithm should not attempt to fit the noise; however, it cannot distinguish noise from signal. On a finite data set, the algorithm inadvertently uses some

Figure 4.4: Deterministic noise. h^* is the best fit to f in $\mathcal{H}_2$. The shading illustrates deterministic noise for this learning problem.

of the degrees of freedom to fit the noise, which can result in overfitting and a spurious final hypothesis.

Figure 4.4 illustrates deterministic noise for a quadratic model fitting a more complex target function. While stochastic and deterministic noise have similar effects on overfitting, there are two basic differences between the two types of noise. First, if we generated the same data ($\mathbf{x}$ values) again, the deterministic noise would not change but the stochastic noise would. Second, different models capture different 'parts' of the target function, hence the same data set will have different deterministic noise depending on which model we use. In reality, we work with one model at a time and have only one data set on hand. Hence, we have one realization of the noise to work with and the algorithm cannot differentiate between the two types of noise.

Exercise 4.3

Deterministic noise depends on $\mathcal{H}$, as some models approximate f better than others.

(a) Assume $\mathcal{H}$ is fixed and we increase the complexity of f. Will deterministic noise in general go up or down? Is there a higher or lower tendency to overfit?

(b) Assume f is fixed and we decrease the complexity of $\mathcal{H}$. Will deterministic noise in general go up or down? Is there a higher or lower tendency to overfit? *[Hint: There is a race between two factors that affect overfitting in opposite ways, but one wins.]*

The bias-variance decomposition, which we discussed in Section 2.3.1 (see also Problem 2.22) is a useful tool for understanding how noise affects performance:

$$\mathbb{E}_{\mathcal{D}}[E_{\text{out}}] = \sigma^2 + \text{bias} + \text{var}.$$

The first two terms reflect the direct impact of the stochastic and deterministic noise. The variance of the stochastic noise is σ^2 and the bias is directly

related to the deterministic noise in that it captures the model's inability to approximate f. The var term is indirectly impacted by both types of noise, capturing a model's susceptibility to being led astray by the noise.

4.2 Regularization

Regularization is our first weapon to combat overfitting. It constrains the learning algorithm to improve out-of-sample error, especially when noise is present. To whet your appetite, look at what a little regularization can do for our first overfitting example in Section 4.1. Though we only used a very small 'amount' of regularization, the fit improves dramatically.

without regularization with regularization

Now that we have your attention, we would like to come clean. Regularization is as much an art as it is a science. Most of the methods used successfully in practice are heuristic methods. However, these methods are grounded in a mathematical framework that is developed for special cases. We will discuss both the mathematical and the heuristic, trying to maintain a balance that reflects the reality of the field.

Speaking of heuristics, one view of regularization is through the lens of the VC bound, which bounds E_{out} using a model complexity penalty $\Omega(\mathcal{H})$:

$$E_{\text{out}}(h) \leq E_{\text{in}}(h) + \Omega(\mathcal{H}) \qquad \text{for all } h \in \mathcal{H}. \tag{4.1}$$

So, we are better off if we fit the data using a simple $\mathcal{H}$. Extrapolating one step further, we should be better off by fitting the data using a 'simple' h from $\mathcal{H}$. The essence of regularization is to concoct a measure $\Omega(h)$ for the complexity of an individual hypothesis. Instead of minimizing $E_{\text{in}}(h)$ alone, one minimizes a combination of $E_{\text{in}}(h)$ and $\Omega(h)$. This avoids overfitting by constraining the learning algorithm to fit the data well using a simple hypothesis.

Example 4.1. One popular regularization technique is *weight decay*, which measures the complexity of a hypothesis h by the size of the coefficients used to represent h (e.g. in a linear model). This heuristic prefers mild lines with

small offset and slope, to wild lines with bigger offset and slope. We will get to the mechanics of weight decay shortly, but for now let's focus on the outcome.

We apply weight decay to fitting the target $f(x) = \sin(\pi x)$ using $N = 2$ data points (as in Example 2.8). We sample x uniformly in $[\ \ 1, 1]$, generate a data set and fit a line to the data (our model is $\mathcal{H}_1$). The figures below show the resulting fits on the same (random) data sets with and without regularization.

without regularization with regularization

Without regularization, the learned function varies extensively depending on the data set. As we have seen in Example 2.8, a constant model scored $E_{\text{out}} = 0.75$, handily beating the performance of the (unregularized) linear model that scored $E_{\text{out}} = 1.90$. With a little weight decay regularization, the fits to *the same data sets* are considerably less volatile. This results in a significantly lower $E_{\text{out}} = 0.56$ that beats both the constant model and the unregularized linear model.

The bias-variance decomposition helps us to understand how the regularized version beat both the unregularized version as well as the constant model.

without regularization
bias $= 0.21$;
var $= 1.69$.

with regularization
bias $= 0.23$;
var $= 0.33$.

Average hypothesis $\bar{g}$ (red) with $\mathsf{var}(x)$ indicated by the gray shaded region that is $\bar{g}(x) \pm \sqrt{\mathsf{var}(x)}$.

As expected, regularization reduced the var term rather dramatically from 1.69 down to 0.33. The price paid in terms of the bias (quality of the average fit) was

modest, only slightly increasing from 0.21 to 0.23. The result was a significant decrease in the expected out-of-sample error because bias+var decreased. This is the crux of regularization. By constraining the learning algorithm to select 'simpler' hypotheses from $\mathcal{H}$, we sacrifice a little bias for a significant gain in the var. □

This example also illustrates why regularization is needed. The linear model is *too sophisticated for the amount of data we have,* since a line can perfectly fit any 2 points. This need would persist even if we changed the target function, as long as we have either stochastic or deterministic noise. The need for regularization depends on the *quantity and quality of the data.* Given our meager data set, our choices were either to take a simpler model, such as the model with constant functions, or to constrain the linear model. It turns out that

In practice, this is the rule not the exception.

Enough heuristics. Let's develop the mathematics of regularization.

4.2.1 A Soft Order Constraint

In this section, we derive a regularization method that applies to a wide variety of learning problems. To simplify the math, we will use the concrete setting of regression using Legendre polynomials, the polynomials of increasing complexity used in Exercise 4.2. So, let's first formally introduce you to the Legendre polynomials.

Consider a learning model where $\mathcal{H}$ is the set of polynomials in one variable $x \in [\ \ 1, 1]$. Instead of expressing the polynomials in terms of consecutive powers of x, we will express them as a combination of Legendre polynomials in x. Legendre polynomials are a standard set of polynomials with nice analytic properties that result in simpler derivations. The zeroth-order Legendre polynomial is the constant $L_0(x) = 1$, and the first few Legendre polynomials are illustrated below.

As you can see, when the order of the Legendre polynomial increases, the curve gets more complex. Legendre polynomials are orthogonal to each other within $x \in [\ \ 1, 1]$, and

just like it can be written as a linear combination of powers of x.

Polynomial models are a special case of linear models in a space $\mathcal{Z}$, under a nonlinear transformation $\Phi\colon \mathcal{X} \to \mathcal{Z}$. Here, for the Qth order polynomial model, Φ transforms x into a vector $\mathbf{z}$ of Legendre polynomials,

$$\mathbf{z} \quad \begin{bmatrix} 1 \\ L_1(x) \\ \vdots \\ L_Q(x) \end{bmatrix}.$$

Our hypothesis set $\mathcal{H}_Q$ is a linear combination of these polynomials,

$$\mathcal{H}_Q = \left\{ h \;\middle|\; h(x) = \mathbf{w}^{\mathrm{T}}\mathbf{z} = \sum_{q=0}^{Q} w_q L_q(x) \right\}_{\mathbf{w}\in\mathbb{R}^{Q+1}},$$

where $L_0(x) \quad 1$. As usual, we will sometimes refer to the hypothesis h by its weight vector $\mathbf{w}$.[2] Since each h is linear in $\mathbf{w}$, we can use the machinery of linear regression from Chapter 3 to minimize the squared error

$$E_{\text{in}}(\mathbf{w}) \quad \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}^{\mathrm{T}}\mathbf{z}_n \quad y_n)^2. \tag{4.2}$$

The case of polynomial regression with squared-error measure illustrates the main ideas of regularization well, and facilitates a solid mathematical derivation. Nonetheless, our discussion will generalize in practice to non-linear, multi-dimensional settings with more general error measures. The baseline algorithm (without regularization) is to minimize E_{in} over the hypotheses in $\mathcal{H}_Q$ to produce the final hypothesis $g(x) \quad \mathbf{w}_{\text{lin}}^{\mathrm{T}}\mathbf{z}$, where $\mathbf{w}_{\text{lin}} \quad \underset{\mathbf{w}}{\operatorname{argmin}}\, E_{\text{in}}(\mathbf{w})$.

Exercise 4.4

Let $\mathrm{Z} \quad [\mathbf{z}_1 \quad \dots \quad \mathbf{z}_N]^{\mathrm{T}}$ be the data matrix (assume Z has full column rank); let $\mathbf{w}_{\text{lin}} \quad (\mathrm{Z}^{\mathrm{T}}\mathrm{Z})^{-1}\mathrm{Z}^{\mathrm{T}}\mathbf{y}$; and let $\mathrm{H} \quad \mathrm{Z}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z})^{-1}\mathrm{Z}^{\mathrm{T}}$ (the hat matrix of Exercise 3.3). Show that

$$E_{\text{in}}(\mathbf{w}) = \frac{(\mathbf{w} \quad \mathbf{w}_{\text{lin}})^{\mathrm{T}}\mathrm{Z}^{\mathrm{T}}\mathrm{Z}(\mathbf{w} \quad \mathbf{w}_{\text{lin}}) + \mathbf{y}^{\mathrm{T}}(\mathrm{I} \quad \mathrm{H})\mathbf{y}}{N}, \tag{4.3}$$

where I is the identity matrix.

(a) What value of $\mathbf{w}$ minimizes E_{in}?

(b) What is the minimum in sample error?

The task of regularization, which results in a final hypothesis $\mathbf{w}_{\text{reg}}$ instead of the simple $\mathbf{w}_{\text{lin}}$, is to constrain the learning so as to prevent overfitting the

[2] We used $\tilde{\mathbf{w}}$ and $\tilde{d}$ for the weight vector and dimension in $\mathcal{Z}$. Since we are explicitly dealing with polynomials and $\mathcal{Z}$ is the only space around, we use $\mathbf{w}$ and Q for simplicity.

data. We have already seen an example of constraining the learning; the set $\mathcal{H}_2$ can be thought of as a constrained version of $\mathcal{H}_{10}$ in the sense that some of the $\mathcal{H}_{10}$ weights are required to be zero. That is, $\mathcal{H}_2$ is a subset of $\mathcal{H}_{10}$ defined by $\mathcal{H}_2 \quad \{\mathbf{w} \mid \mathbf{w} \in \mathcal{H}_{10}; w_q \quad 0 \text{ for } q \geq 3\}$. Requiring some weights to be 0 is a *hard* constraint. We have seen that such a hard constraint on the order can help, for example $\mathcal{H}_2$ is better than $\mathcal{H}_{10}$ when there is a lot of noise and N is small. Instead of requiring some weights to be zero, we can force the weights to be small but not necessarily zero through a softer constraint such as

$$\sum_{q=0}^{Q} w_q^2 \leq C.$$

This is a 'soft order' constraint because it only encourages each weight to be small, without changing the order of the polynomial by explicitly setting some weights to zero. The in-sample optimization problem becomes:

$$\min_{\mathbf{w}} E_{\text{in}}(\mathbf{w}) \quad \text{subject to} \quad \mathbf{w}^{\mathrm{T}}\mathbf{w} \leq C. \tag{4.4}$$

The data determines the optimal weight sizes, given the total budget C which determines the amount of regularization; the larger C is, the weaker the constraint and the smaller the amount of regularization. We can define the soft-order-constrained hypothesis set $\mathcal{H}(C)$ by

$$\mathcal{H}(C) = \{h \mid h(x) = \mathbf{w}^{\mathrm{T}}\mathbf{z} \ , \ \mathbf{w}^{\mathrm{T}}\mathbf{w} \leq C\}.$$

Equation (4.4) is equivalent to minimizing E_{in} over $\mathcal{H}(C)$. If $C_1 < C_2$, then $\mathcal{H}(C_1) \subset \mathcal{H}(C_2)$ and so $d_{\text{vc}}(\mathcal{H}(C_1)) \leq d_{\text{vc}}(\mathcal{H}(C_2))$, and we expect better generalization with $\mathcal{H}(C_1)$. Let the regularized weights $\mathbf{w}_{\text{reg}}$ be the solution to (4.4).

Solving for $\mathbf{w}_{\text{reg}}$. If $\mathbf{w}_{\text{lin}}^{\mathrm{T}}\mathbf{w}_{\text{lin}} \leq C$ then $\mathbf{w}_{\text{reg}}$ $\mathbf{w}_{\text{lin}}$ because $\mathbf{w}_{\text{lin}} \in \mathcal{H}(C)$. If $\mathbf{w}_{\text{lin}} \notin \mathcal{H}(C)$, then not only is $\mathbf{w}_{\text{reg}}^{\mathrm{T}}\mathbf{w}_{\text{reg}} \leq C$, but in fact $\mathbf{w}_{\text{reg}}^{\mathrm{T}}\mathbf{w}_{\text{reg}} \quad C$ ($\mathbf{w}_{\text{reg}}$ uses the entire budget C; see Problem 4.10).

We thus need to minimize E_{in} subject to the *equality* constraint $\mathbf{w}^{\mathrm{T}}\mathbf{w} \quad C$. The situation is illustrated to the right. The weights $\mathbf{w}$ must lie on the surface of the sphere $\mathbf{w}^{\mathrm{T}}\mathbf{w} = C$; the normal vector to this surface at $\mathbf{w}$ is the vector $\mathbf{w}$ itself (also in red). A surface of constant E_{in} is shown in blue; this surface is a quadratic surface (see Exercise 4.4) and the normal to this surface is $\nabla E_{\text{in}}(\mathbf{w})$.

while still

remaining on the surface. If $\mathbf{w}_{\text{reg}}$ is to be optimal, then for some positive parameter λ_C

$$\nabla E_{\text{in}}(\mathbf{w}_{\text{reg}}) = -2\lambda_C \mathbf{w}_{\text{reg}},$$

i.e., ∇E_{in} must be parallel to $\mathbf{w}_{\text{reg}}$, the normal vector to the constraint surface (the scaling by 2 is for mathematical convenience and the negative sign is because ∇E_{in} and $\mathbf{w}$ are in opposite directions). Equivalently, $\mathbf{w}_{\text{reg}}$ satisfies

$$\nabla \left(E_{\text{in}}(\mathbf{w}) + \lambda_C \mathbf{w}^{\text{T}}\mathbf{w}\right)\Big|_{\mathbf{w}=\mathbf{w}_{\text{reg}}} = \mathbf{0},$$

because $\nabla(\mathbf{w}^{\text{T}}\mathbf{w}) \quad 2\mathbf{w}$. So, for some $\lambda_C > 0$, $\mathbf{w}_{\text{reg}}$ locally minimizes

$$E_{\text{in}}(\mathbf{w}) + \lambda_C \mathbf{w}^{\text{T}}\mathbf{w}. \tag{4.5}$$

The parameter λ_C and the vector $\mathbf{w}_{\text{reg}}$ (both of which depend on C and the data) must be chosen so as to simultaneously satisfy the gradient equality and the weight norm constraint $\mathbf{w}_{\text{reg}}^{\text{T}}\mathbf{w}_{\text{reg}} \quad C$.[3] That $\lambda_C > 0$ is intuitive since we are enforcing smaller weights, and minimizing $E_{\text{in}}(\mathbf{w}) + \lambda_C \mathbf{w}^{\text{T}}\mathbf{w}$ would not lead to smaller weights if λ_C were negative. Note that if $\mathbf{w}_{\text{lin}}^{\text{T}}\mathbf{w}_{\text{lin}} \leq C$, $\mathbf{w}_{\text{reg}} \quad \mathbf{w}_{\text{lin}}$ and minimizing (4.5) still holds with $\lambda_C \quad 0$. Therefore, we have an equivalence between solving the constrained problem (4.4) and the unconstrained minimization of (4.5).

Other variations of the constraint in (4.4) can be used to emphasize some weights over the others. Consider the constraint $\sum_{q=0}^{Q} \gamma_q w_q^2 \leq C$. The importance γ_q given to weight w_q determines the type of regularization. For example, $\gamma_q \quad q$ or $\gamma_q \quad e^q$ encourages a low-order fit, and $\gamma_q \quad (1+q)^{-1}$ or $\gamma_q = e^{-q}$ encourages a high-order fit. In extreme cases, one recovers hard-order constraints by choosing some $\gamma_q \quad 0$ and some $\gamma_q \to \infty$.

Exercise 4.5 [Tikhonov regularizer]

A more general soft constraint is the *Tikhonov* regularization constraint

$$\mathbf{w}^{\text{T}}\Gamma^{\text{T}}\Gamma\mathbf{w} \leq C$$

which can capture relationships among the w_i (the matrix Γ is the Tikhonov regularizer).

(a) What should Γ be to obtain the constraint $\sum_{q=0}^{Q} w_q^2 \leq C$?

(b) What should Γ be to obtain the constraint $(\sum_{q=0}^{Q} w_q)^2 \leq C$?

[3] λ_C is known as a Lagrange multiplier and an alternate derivation of these same results can be obtained via the theory of Lagrange multipliers for constrained optimization.

4.2.2 Weight Decay and Augmented Error

The soft-order constraint for a given value of C is a constrained minimization of E_{in}. Equation (4.5) suggests that we may equivalently solve an unconstrained minimization of a different function. Let's define the *augmented error*,

$$E_{\text{aug}}(\mathbf{w}) = E_{\text{in}}(\mathbf{w}) + \lambda \mathbf{w}^{\mathrm{T}}\mathbf{w}, \tag{4.6}$$

where $\lambda \geq 0$ is now a free parameter at our disposal. The augmented error has two terms. The first is the in-sample error which we are used to minimizing, and the second is a *penalty term*. Notice that this fits the heuristic view of regularization that we discussed earlier, where the penalty for complexity is defined for each individual h instead of $\mathcal{H}$ as a whole. When $\lambda = 0$, we have the usual in-sample error. For $\lambda > 0$, minimizing the augmented error corresponds to minimizing a penalized in-sample error. The value of λ controls the amount of regularization. As discussed in Problem 4.8, if we minimize the augmented error using an iterative method like gradient descent, we will have a reduction of the in-sample error together with a gradual shrinking of the weights, hence the name weight 'decay.'

There is an equivalence between the soft order constraint and augmented error minimization. In the soft-order constraint, the amount of regularization is controlled by the parameter C. From (4.5), there is a particular λ_C (depending on C and the data $\mathcal{D}$), for which minimizing the augmented error $E_{\text{aug}}(\mathbf{w})$ leads to the same final hypothesis $\mathbf{w}_{\text{reg}}$. A larger C allows larger weights and is a weaker soft-order constraint; this corresponds to smaller λ, i.e., less emphasis on the penalty term $\mathbf{w}^{\mathrm{T}}\mathbf{w}$ in the augmented error.

Have we gained from the augmented error view? Yes, because augmented error minimization is unconstrained, which is generally easier than constrained minimization. For example, we can obtain a closed form solution for linear models or use a method like stochastic gradient descent to carry out the minimization. However, augmented error minimization is not so easy to interpret. There are no values for the weights which are explicitly forbidden, as there are in the soft-order constraint. For a given C, the soft-order constraint corresponds to selecting a hypothesis from the smaller set $\mathcal{H}(C)$, and so from our VC analysis we should expect better generalization when C decreases (λ increases). It is through the relationship between λ and C that one has a theoretical justification of weight decay as a method for regularization.

We focused on the soft-order constraint $\mathbf{w}^{\mathrm{T}}\mathbf{w} \leq C$ with corresponding augmented error $E_{\text{aug}}(\mathbf{w}) = E_{\text{in}}(\mathbf{w}) + \lambda \mathbf{w}^{\mathrm{T}}\mathbf{w}$. However, our discussion applies more generally. There is a duality between the minimization of the in-sample

error over a constrained hypothesis set and the unconstrained minimization of an augmented error. We may choose to live in either world, but more often than not, the unconstrained minimization of the augmented error is more convenient.

In our definition of $E_{\text{aug}}(\mathbf{w})$ in Equation (4.6), we only highlighted the dependence on $\mathbf{w}$. There are two other quantities under our control, namely the amount of regularization, λ, and the nature of the regularizer which we chose to be $\mathbf{w}^{\mathrm{T}}\mathbf{w}$. In general, the augmented error for a hypothesis $h \in \mathcal{H}$ is

$$E_{\text{aug}}(h, \lambda, \Omega) = E_{\text{in}}(h) + \frac{\lambda}{N}\Omega(h). \tag{4.7}$$

For weight decay, $\Omega(h) \quad \mathbf{w}^{\mathrm{T}}\mathbf{w}$, which penalizes large weights. The penalty term has two components: the regularizer $\Omega(h)$ (the type of regularization) which penalizes a particular property of h; and the *regularization parameter* λ (the amount of regularization). The need for regularization goes down as the number of data points goes up, so we factored out $\frac{1}{N}$; this allows the optimal choice for λ to be less sensitive to N. This is just a redefinition of the λ that we have been using, in order to make it a more stable parameter that is easier to interpret. Notice how Equation (4.7) resembles the VC bound (4.1) as we anticipated in the heuristic view of regularization. This is why we use the same notation Ω for both the penalty on individual hypotheses $\Omega(h)$ and the penalty on the whole set $\Omega(\mathcal{H})$. The correspondence between the complexity of $\mathcal{H}$ and the complexity of an individual h will be discussed further in Section 5.1.

The regularizer Ω is typically fixed ahead of time, before seeing the data; sometimes the problem itself can dictate an appropriate regularizer.

Exercise 4.6

We have seen both the hard-order constraint and the soft-order constraint. Which do you expect to be more useful for binary classification using the perceptron model? *[Hint: sign*$(\mathbf{w}^{\mathrm{T}}\mathbf{x}) \quad$ *sign*$(\alpha\mathbf{w}^{\mathrm{T}}\mathbf{x})$ *for any* $\alpha > 0$*.]*

The optimal regularization parameter, however, typically depends on the data. The choice of the optimal λ is one of the applications of *validation*, which we will discuss shortly.

Example 4.2. Linear models with weight decay. Linear models are important enough that it is worthwhile to spell out the details of augmented error minimization in this case. From Exercise 4.4, the augmented error is

$$E_{\text{aug}}(\mathbf{w}) = \frac{(\mathbf{w} - \mathbf{w}_{\text{lin}})^{\mathrm{T}}\mathrm{Z}^{\mathrm{T}}\mathrm{Z}(\mathbf{w} - \mathbf{w}_{\text{lin}}) + \lambda\mathbf{w}^{\mathrm{T}}\mathbf{w} + \mathbf{y}^{\mathrm{T}}(\mathrm{I} - \mathrm{H})\mathbf{y}}{N},$$

where Z is the transformed data matrix and $\mathbf{w}_{\text{lin}} \quad (\mathrm{Z}^{\mathrm{T}}\mathrm{Z})^{-1}\mathrm{Z}^{\mathrm{T}}\mathbf{y}$. The reader may verify, after taking the derivatives of E_{aug} and setting $\nabla_{\mathbf{w}}E_{\text{aug}} \quad \mathbf{0}$, that

$$\mathbf{w}_{\text{reg}} = (\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathrm{Z}^{\mathrm{T}}\mathbf{y}.$$

As expected, $\mathbf{w}_{\text{reg}}$ will go to zero as $\lambda \to \infty$, due to the $\lambda\mathrm{I}$ term. The predictions on the in-sample data are given by $\hat{\mathbf{y}} = \mathrm{Z}\mathbf{w}_{\text{reg}} = \mathrm{H}(\lambda)\mathbf{y}$, where

$$\mathrm{H}(\lambda) = \mathrm{Z}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathrm{Z}^{\mathrm{T}}.$$

The matrix $\mathrm{H}(\lambda)$ plays an important role in defining the effective complexity of a model. When $\lambda = 0$, H is the hat matrix of Exercises 3.3 and 4.4, which satisfies $\mathrm{H}^2 = \mathrm{H}$ and $\mathrm{trace}(\mathrm{H}) = d + 1$. The vector of in-sample errors, which are also called residuals, is $\mathbf{y} - \hat{\mathbf{y}} = (\mathrm{I} - \mathrm{H}(\lambda))\mathbf{y}$, and the in-sample error E_{in} is $E_{\text{in}}(\mathbf{w}_{\text{reg}}) = \frac{1}{N}\mathbf{y}^{\mathrm{T}}(\mathrm{I} - \mathrm{H}(\lambda))^2\mathbf{y}$. □

We can now apply weight decay regularization to the first overfitting example that opened this chapter. The results for different λ's are shown in Figure 4.5.

Figure 4.5: Weight decay applied to Example 4.2 with different values for the regularization parameter λ. The red fit gets flatter as we increase λ.

As you can see, even very little regularization goes a long way, but too much regularization results in an overly flat curve at the expense of in-sample fit. Another case we saw earlier is Example 4.1, where we fit a linear model to a sinusoid. The regularization used there was also weight decay, with $\lambda = 0.1$.

4.2.3 Choosing a Regularizer: Pill or Poison?

We have presented a number of ways to constrain a model: hard-order constraints where we simply use a lower-order model, soft-order constraints where we constrain the parameters of the model, and augmented error where we add a penalty term to an otherwise unconstrained minimization of error. Augmented error is the most popular form of regularization, for which we need to choose the regularizer $\Omega(h)$ and the regularization parameter λ.

In practice, the choice of Ω is largely heuristic. Finding a perfect Ω is as difficult as finding a perfect $\mathcal{H}$. It depends on information that, by the very nature of learning, we don't have. However, there are regularizers we can work with that have stood the test of time, such as weight decay. Some forms of regularization work and some do not, depending on the specific application and the data. Figure 4.5 illustrated that even the amount of regularization

(a) Uniform regularizer (b) Low order regularizer

Figure 4.6: Out of sample performance for the uniform and low order reg ularizers using model $\mathcal{H}_{15}$, with $\sigma^2 = 0.5$, $Q_f = 15$ and $N = 30$. Overfitting occurs in the shaded region because lower E_{in} (lower λ) leads to higher E_{out}. Underfitting occurs when λ is too large, because the learning algorithm has too little flexibility to fit the data.

has to be chosen carefully. Too much regularization (too harsh a constraint) leaves the learning too little flexibility to fit the data and leads to *underfitting*, which can be just as bad as overfitting.

If so many choices can go wrong, why do we bother with regularization in the first place? Regularization is a necessary evil, with the operative word being *necessary*. If our model is too sophisticated for the amount of data we have, we are doomed. By applying regularization, we have a chance. By applying the proper regularization, we are in good shape. Let us experiment with two choices of a regularizer for the model $\mathcal{H}_{15}$ of 15th order polynomials, using the experimental design in Exercise 4.2:

1. A uniform regularizer: $\Omega_{\text{unif}}(\mathbf{w}) \quad \sum_{q=0}^{15} w_q^2$.

2. A low-order regularizer: $\Omega_{\text{low}}(\mathbf{w}) = \sum_{q=0}^{15} q w_q^2$.

The first encourages all weights to be small, uniformly; the second pays more attention to the higher order weights, encouraging a lower order fit. Figure 4.6 shows the performance for different values of the regularization parameter λ. As you decrease λ, the optimization pays less attention to the penalty term and more to E_{in}, and so E_{in} will decrease (Problem 4.7). In the shaded region, E_{out} increases as you decrease E_{in} (decrease λ) – the regularization parameter is too small and there is not enough of a constraint on the learning, leading to decreased performance because of overfitting. In the unshaded region, the regularization parameter is too large, over-constraining the learning and not giving it enough flexibility to fit the data, leading to decreased performance because of underfitting. As can be observed from the figure, the price paid for overfitting is generally more severe than underfitting. It usually pays to be conservative.

(a) Stochastic noise (b) Deterministic noise

Figure 4.7: Performance of the uniform regularizer at different levels of noise. The optimal λ is highlighted for each curve.

The optimal regularization parameter for the two cases is quite different and the performance can be quite sensitive to the choice of regularization parameter. However, the promising message from the figure is that though the behaviors are quite different, the performances of the two regularizers are comparable (around 0.76), *if we choose the right λ for each.*

We can also use this experiment to study how performance with regularization depends on the noise. In Figure 4.7(a), when $\sigma^2 = 0$, no amount of regularization helps (i.e., the optimal regularization parameter is $\lambda \quad 0$), which is not a surprise because there is no stochastic or deterministic noise in the data (both target and model are 15th order polynomials). As we add more stochastic noise, the overall performance degrades as expected. Note that the optimal value for the regularization parameter increases with noise, which is also expected based on the earlier discussion that the potential to overfit increases as the noise increases; hence, constraining the learning more should help. Figure 4.7(b) shows what happens when we add deterministic noise, keeping the stochastic noise at zero. This is accomplished by increasing Q_f (the target complexity), thereby adding deterministic noise, but keeping everything else the same. Comparing parts (a) and (b) of Figures 4.7 provides another demonstration of how the effects of deterministic and stochastic noise are similar. When either is present, it *is* helpful to regularize, and the more noise there is, the larger the amount of regularization you need.

What happens if you pick the wrong regularizer? To illustrate, we picked a regularizer which encourages large weights (weight growth) versus weight decay which encourages small weights. As you can see, in this case, weight growth does not help the cause of overfitting. If we happened to choose weight growth as our regularizer, we would still be OK as long as we have

a good way to pick the regularization parameter the optimal regularization parameter in this case is $\lambda = 0$, and we are no worse off than not regularizing. No regularizer will be ideal for all settings, or even for a specific setting since we never have perfect information, but they all tend to work with varying success, *if the amount of regularization λ is set to the correct level.* Thus, the entire burden rests on picking the right λ, a task that can be addressed by a technique called validation, which is the topic of the next section.

The lesson learned is that some form of regularization is necessary, as learning is quite sensitive to stochastic and deterministic noise. The best way to constrain the learning is in the 'direction' of the target function, and more of a constraint is needed when there is more noise. Even though we don't know either the target function or the noise, regularization helps by reducing the impact of the noise. Most common models have hypothesis sets which are naturally parameterized so that smaller parameters lead to smoother hypotheses. Thus, a weight decay type of regularizer constrains the learning towards smoother hypotheses. This helps, because stochastic noise is 'high frequency' (non-smooth). Similarly, deterministic noise (the part of the target function which cannot be modeled) also tends to be non-smooth. Thus, constraining the learning towards smoother hypotheses 'hurts' our ability to overfit the noise more than it hurts our ability to fit the useful information. These are empirical observations, not theoretically justifiable statements.

Regularization and the VC dimension. Regularization (for example soft-order selection by minimizing the augmented error) poses a problem for the VC line of reasoning. As λ goes up, the learning algorithm changes but the hypothesis set does not, so d_{VC} will not change. We argued that $\lambda \uparrow$ in the augmented error corresponds to $C \downarrow$ in the soft-order constrained model. So, more regularization corresponds to an effectively smaller model, and we expect better generalization for a small increase in E_{in} even though the VC dimension of the model we are actually using with augmented error does not change. This suggests a heuristic that works well in practice, which is to use an 'effective VC dimension' instead of the VC dimension. For linear perceptrons, the VC dimension equals the number of free parameters $d+1$, and so an *effective number of parameters* is a good surrogate for the VC dimension in the VC bound. The effective number of parameters will go down as λ increases, and so the effective VC dimension will reflect better generalization with increased regularization. Problems 4.13, 4.14, and 4.15 explore the notion of an effective number of parameters.

4.3 Validation

So far, we have identified overfitting as a problem, noise (stochastic and deterministic) as a cause, and regularization as a cure. In this section, we introduce another cure, called *validation.* One can think of both regularization and val-

idation as attempts at minimizing E_{out} rather than just E_{in}. Of course the true E_{out} is not available to us, so we need an estimate of E_{out} based on information available to us in sample. In some sense, this is the Holy Grail of machine learning: to find an in-sample estimate of the out-of-sample error. Regularization attempts to minimize E_{out} by working through the equation

$$E_{\text{out}}(h) \quad E_{\text{in}}(h) + \underbrace{\text{overfit penalty}}_{\text{regularization estimates this quantity}},$$

and concocting a heuristic term that emulates the penalty term. Validation, on the other hand, cuts to the chase and estimates the out-of-sample error directly.

$$\underbrace{E_{\text{out}}(h)}_{\text{validation estimates this quantity}} \quad E_{\text{in}}(h) + \text{overfit penalty}.$$

Estimating the out-of-sample error directly is nothing new to us. In Section 2.2.3, we introduced the idea of a *test set*, a subset of $\mathcal{D}$ that is not involved in the learning process and is used to evaluate the final hypothesis. The test error E_{test}, unlike the in-sample error E_{in}, is an unbiased estimate of E_{out}.

4.3.1 The Validation Set

The idea of a *validation set* is almost identical to that of a test set. We remove a subset from the data; this subset is not used in training. We then use this held-out subset to estimate the out-of-sample error. The held-out set is effectively out-of-sample, because it has not been used during the learning.

However, there is a difference between a validation set and a test set. Although the validation set will not be directly used for training, it will be used in making certain choices in the learning process. The minute a set affects the learning process in any way, it is no longer a *test* set. However, as we will see, the way the validation set is used in the learning process is so benign that its estimate of E_{out} remains almost intact.

Let us first look at how the validation set is created. The first step is to partition the data set $\mathcal{D}$ into a *training set* $\mathcal{D}_{\text{train}}$ of size $(N-K)$ and a *validation set* $\mathcal{D}_{\text{val}}$ of size K. Any partitioning method which does not depend on the values of the data points will do; for example, we can select $N-K$ points at random for training and the remaining for validation.

Now, we run the learning algorithm using the training set $\mathcal{D}_{\text{train}}$ to obtain a final hypothesis $g^- \in \mathcal{H}$, where the 'minus' superscript indicates that some data points were taken out of the training. We then compute the validation error for g^- using the validation set $\mathcal{D}_{\text{val}}$:

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

where $\mathsf{e}(g(\mathbf{x}), y)$ is the pointwise error measure which we introduced in Section 1.4.1. For classification, $\mathsf{e}(g(\mathbf{x}), y) = [\![g(\mathbf{x}) \neq y]\!]$ and for regression using squared error, $\mathsf{e}(g(\mathbf{x}), y) = (g(\mathbf{x}) - y)^2$.

The validation error is an *unbiased* estimate of E_{out} because the final hypothesis g^- was created independently of the data points in the validation set. Indeed, taking the expectation of E_{val} with respect to the data points in $\mathcal{D}_{\text{val}}$,

$$\begin{aligned}
\mathbb{E}_{\mathcal{D}_{\text{val}}}\left[E_{\text{val}}(g^-)\right] &= \frac{1}{K}\sum_{\mathbf{x}_n \in \mathcal{D}_{\text{val}}} \mathbb{E}_{\mathcal{D}_{\text{val}}}\left[\mathsf{e}\left(g^-(\mathbf{x}_n), y_n\right)\right], \\
&= \frac{1}{K}\sum_{\mathbf{x}_n \in \mathcal{D}_{\text{val}}} E_{\text{out}}(g^-), \\
&= E_{\text{out}}(g^-). \qquad (4.8)
\end{aligned}$$

The first step uses the linearity of expectation, and the second step follows because $\mathsf{e}(g(\mathbf{x}_n), y_n)$ depends only on $\mathbf{x}_n$ and so

$$\mathbb{E}_{\mathcal{D}_{\text{val}}}\left[\mathsf{e}\left(g(\mathbf{x}_n), y_n\right)\right] = \mathbb{E}_{\mathbf{x}_n}\left[\mathsf{e}\left(g(\mathbf{x}_n), y_n\right)\right] = E_{\text{out}}(g).$$

How reliable is E_{val} at estimating E_{out}? In the case of classification, one can use the VC bound to predict how good the validation error is as an estimate for the out-of-sample error. We can view $\mathcal{D}_{\text{val}}$ as an 'in-sample' data set on which we computed the error of the single hypothesis g. We can thus apply the VC bound for a finite model with one hypothesis in it (the Hoeffding bound). With high probability,

$$E_{\text{out}}(g) \leq E_{\text{val}}(g) + O\left(\frac{1}{\sqrt{K}}\right). \qquad (4.9)$$

While Inequality (4.9) applies to binary target functions, we may use the variance of E_{val} as a more generally applicable measure of the reliability. The next exercise studies how the variance of E_{val} depends on K (the size of the validation set), and implies that a similar bound holds for regression. The conclusion is that the error between $E_{\text{val}}(g)$ and $E_{\text{out}}(g)$ drops as $\sigma(g)/\sqrt{K}$, where $\sigma(g)$ is bounded by a constant in the case of classification.

Exercise 4.7

Fix g (learned from $\mathcal{D}_{\text{train}}$) and define $\sigma^2_{\text{val}} \stackrel{\text{def}}{=} \text{Var}_{\mathcal{D}_{\text{val}}}[E_{\text{val}}(g^-)]$. We consider how σ^2_{val} depends on K. Let

$$\sigma^2(g^-) = \text{Var}_{\mathbf{x}}[\mathsf{e}(g^-(\mathbf{x}), y)]$$

be the pointwise variance in the out-of-sample error of g.

(a) Show that $\sigma^2_{\text{val}} = \frac{1}{K}\sigma^2(g)$.

(b) In a classification problem, where $\mathsf{e}(g(\mathbf{x}), y) = [\![g(\mathbf{x}) \neq y]\!]$, express σ^2_{val} in terms of $\mathbb{P}[g^-(\mathbf{x}) \neq y]$.

(c) Show that for any g^- in a classification problem, $\sigma^2_{\text{val}} \leq \frac{1}{4K}$.

(continued on next page)

(d) Is there a uniform upper bound for $\mathrm{Var}[E_{\mathrm{val}}(g\ )]$ similar to (c) in the case of regression with squared error $\mathrm{e}(g\ (\mathbf{x}), y) \quad (g\ (\mathbf{x}) - y)^2$? *[Hint: The squared error is unbounded.]*

(e) For regression with squared error, if we train using fewer points (smaller $N - K$) to get g , do you expect $\sigma^2(g^-)$ to be higher or lower? *[Hint: For continuous, non-negative random variables, higher mean often implies higher variance.]*

(f) Conclude that increasing the size of the validation set can result in a better *or* a worse estimate of E_{out}.

The expected validation error for $\mathcal{H}_2$ is illustrated in Figure 4.8, where we used the experimental design in Exercise 4.2, with $Q_f = 10$, $N \quad 40$ and noise level 0.4. The expected validation error equals $E_{\mathrm{out}}(g\ )$, per Equation (4.8).

Figure 4.8: The expected validation error $\mathbb{E}[E_{\mathrm{val}}(g^-)]$ as a function of K; the shaded area is $\mathbb{E}[E_{\mathrm{val}}] \pm \sigma_{\mathrm{val}}$.

The figure clearly shows that there is a price to be paid for setting aside K data points to get this unbiased estimate of E_{out}: when we set aside more data for validation, there are fewer training data points and so g becomes worse; $E_{\mathrm{out}}(g\ )$, and hence the expected validation error, *increases* (the blue curve). As we expect, the uncertainty in E_{val} as measured by σ_{val} (size of the shaded region) is decreasing with K, up to the point where the variance $\sigma^2(g\ )$ gets really bad. This point comes when the number of training data points becomes critically small, as in Exercise 4.7(e). If K is neither too small nor too large, E_{val} provides a good estimate of E_{out}. A rule of thumb in practice is to set $K = \frac{N}{5}$ (set aside 20% of the data for validation).

We have established two conflicting demands on K. It has to be big enough for E_{val} to be reliable, and it has to be small enough so that the training set with $N - K$ points is big enough to get a decent g . Inequality (4.9) quantifies the first demand. The second demand is quantified by the learning curve

discussed in Section 2.3.2 (also the blue curve in Figure 4.8, from right to left), which shows how the expected out-of-sample error goes down as the number of training data points goes up . The fact that more training data lead to a better final hypothesis has been extensively verified empirically, although it is challenging to prove theoretically.

Restoring $\mathcal{D}$. Although the learning curve suggests that taking out K data points for validation and using only $N-K$ for training will cost us in terms of E_{out}, we do not have to pay that price! The purpose of validation is to *estimate* the out-of-sample performance, and E_{val} happens to be a good estimate of $E_{\text{out}}(g^-)$. This does not mean that we have to output g^- as our final hypothesis. The primary goal is to get the best possible hypothesis, so we should output g, the hypothesis trained on the entire set $\mathcal{D}$. The secondary goal is to estimate E_{out}, which is what validation allows us to do. Based on our discussion of learning curves, $E_{\text{out}}(g) \leq E_{\text{out}}(g^-)$, so

Figure 4.9: Using a validation set to estimate E_{out}.

$$E_{\text{out}}(g) \leq E_{\text{out}}(g^-) \leq E_{\text{val}}(g^-) + O\left(\frac{1}{\sqrt{K}}\right). \tag{4.10}$$

The first inequality is subdued because it was not rigorously proved. If we first train with $N-K$ data points, validate with the remaining K data points and then retrain using all the data to get g, the validation error we got will likely still be better at estimating $E_{\text{out}}(g)$ than the estimate using the VC-bound with $E_{\text{in}}(g)$, especially for large hypothesis sets with big d_{VC}.

So far, we have treated the validation set as a way to estimate E_{out}, without involving it in any decisions that affect the learning process. Estimating E_{out} is a useful role by itself a customer would typically want to know how good the final hypothesis is (in fact, the inequalities in (4.10) suggest that the validation error is a pessimistic estimate of E_{out}, so your customer is likely to be pleasantly surprised when he tries your system on new data). However, as we will see next, an important role of a validation set is in fact to guide the learning process. That's what distinguishes a validation set from a test set.

4.3.2 Model Selection

By far, the most important use of validation is for *model selection.* This could mean the choice between a linear model and a nonlinear model, the choice of the order of polynomial in a model, the choice of the value of a regularization

Figure 4.10: Optimistic bias of the validation error when using a validation set for the model selected.

parameter, or any other choice that affects the learning process. In almost every learning situation, there are some choices to be made and we need a principled way of making these choices.

The leap is to realize that validation can be used to estimate the out-of-sample error for more than one model. Suppose we have M models $\mathcal{H}_1, \ldots, \mathcal{H}_M$. Validation can be used to select one of these models. Use the training set $\mathcal{D}_{\text{train}}$ to learn a final hypothesis g^-_m for each model. Now evaluate each model on the validation set to obtain the validation errors $E_1, \cdots, E_M$, where

$$E_m = E_{\text{val}}(g^-_m); \quad m = 1, \ldots, M.$$

The validation errors estimate the out-of-sample error $E_{\text{out}}(g^-_m)$ for each $\mathcal{H}_m$.

Exercise 4.8

Is E_m an unbiased estimate for the out of sample error $E_{\text{out}}(g^-_m)$?

It is now a simple matter to select the model with lowest validation error. Let m^* be the index of the model which achieves the minimum validation error. So for $\mathcal{H}_{m^*}$, $E_{m^*} \leq E_m$ for $m = 1, \ldots, M$. The model $\mathcal{H}_{m^*}$ is the model selected based on the validation errors. Note that E_{m^*} is no longer an unbiased estimate of $E_{\text{out}}(g^-_{m^*})$. Since we *selected* the model with minimum validation error, E_{m^*} will have an optimistic bias. This optimistic bias when selecting between $\mathcal{H}_2$ and $\mathcal{H}_5$ is illustrated in Figure 4.10, using the experimental design described in Exercise 4.2 with $Q_f = 3$, $\sigma^2 = 0.4$ and $N = 35$.

Exercise 4.9

Referring to Figure 4.10, why are both curves increasing with K? Why do they converge to each other with increasing K?

How good is the generalization error for this entire process of model selection using validation? Consider a new model $\mathcal{H}_{\text{val}}$ consisting of the final hypotheses learned from the training data using each model $\mathcal{H}_1, \ldots, \mathcal{H}_M$:

$$\mathcal{H}_{\text{val}} \quad \{g_1^-, g_2^-, \ldots, g_M^-\}.$$

Model selection using the validation set chose one of the hypotheses in $\mathcal{H}_{\text{val}}$ based on its performance on $\mathcal{D}_{\text{val}}$. Since the model $\mathcal{H}_{\text{val}}$ was obtained before ever looking at the data in the validation set, this process is entirely equivalent to learning a hypothesis from $\mathcal{H}_{\text{val}}$ using the data in $\mathcal{D}_{\text{val}}$. The validation errors $E_{\text{val}}(g_m^-)$ are 'in-sample' errors for this learning process and so we may apply the VC bound for finite hypothesis sets, with $|\mathcal{H}_{\text{val}}| \quad M$:

$$E_{\text{out}}(g_{m^*}^-) \leq E_{\text{val}}(g_{m^*}^-) + O\left(\sqrt{\frac{\ln M}{K}}\right). \tag{4.11}$$

What if we didn't use a validation set to choose the model? One alternative would be to use the in-sample errors from each model as the model selection criterion. Specifically, pick the model which gives a final hypothesis with minimum in-sample error. This is equivalent to picking the hypothesis with minimum in-sample error from the grand model which contains all the hypotheses in each of the M original models. If we want a bound on the out-of-sample error for the final hypothesis that results from this selection, we need to apply the VC-penalty for this grand hypothesis set which is the union of the M hypothesis sets (see Problem 2.14). Since this grand hypothesis set can have a huge VC-dimension, the bound in (4.11) will generally be tighter.

The goal of model selection is to select the best model *and* output the best hypothesis from that model. Specifically, we want to select the model m for which $E_{\text{out}}(g_m)$ will be minimum when we retrain with all the data. Model selection using a validation set relies on the leap of faith that if $E_{\text{out}}(g_m)$ is minimum, then $E_{\text{out}}(g_m^-)$ is also minimum. The validation errors E_m estimate $E_{\text{out}}(g_m^-)$, so modulo our leap of faith, the validation set should pick the right model. No matter which model m^* is selected, however, based on the discussion of learning curves in the previous section, we should not output $g_{m^*}^-$ as the final hypothesis. Rather, once m^* is selected using validation, learn using all the data and output g_{m^*}, which satisfies

Figure 4.11: Using a validation set for model selection

$$E_{\text{out}}(g_{m^*}) \leq E_{\text{out}}(g_{m^*}^-) \leq E_{\text{val}}(g_{m^*}^-) + O\left(\sqrt{\frac{\ln M}{K}}\right). \tag{4.12}$$

Again, the first inequality is subdued because we didn't prove it.

Figure 4.12: Model selection between $\mathcal{H}_2$ and $\mathcal{H}_5$ using a validation set. The solid black line uses E_{in} for model selection, which always selects $\mathcal{H}_5$. The dotted line shows the optimal model selection, if we could select the model based on the true out of sample error. This is unachievable, but a useful benchmark. The best performer is clearly the validation set, outputting g_{m^*}. For suitable K, even $g^-_{m^*}$ is better than in sample selection.

Continuing our experiment from Figure 4.10, we evaluate the out-of-sample performance when using a validation set to select between the models $\mathcal{H}_2$ and $\mathcal{H}_5$. The results are shown in Figure 4.12. Validation is a clear winner over using E_{in} for model selection.

Exercise 4.10

(a) From Figure 4.12, $\mathbb{E}[E_{\text{out}}(g^-_{m^*})]$ is initially decreasing. How can this be, if $\mathbb{E}[E_{\text{out}}(g^-_m)]$ is increasing in K for each m?

(b) From Figure 4.12 we see that $\mathbb{E}[E_{\text{out}}(g_{m^*})]$ is initially decreasing, and then it starts to increase. What are the possible reasons for this?

(c) When $K = 1$, $\mathbb{E}[E_{\text{out}}(g^-_{m^*})] < \mathbb{E}[E_{\text{out}}(g_{m^*})]$. How can this be, if the learning curves for both models are decreasing?

Example 4.3. We can use a validation set to select the value of the regularization parameter in the augmented error of (4.6). Although the most important part of a model is the hypothesis set, every hypothesis set has an associated learning algorithm which selects the final hypothesis g. Two models may be different only in their learning algorithm, while working with the same hypothesis set. Changing the value of λ in the augmented error changes the learning algorithm (the criterion by which g is selected) and effectively changes the model.

Based on this discussion, consider the M *different* models corresponding to the same hypothesis set $\mathcal{H}$ but with M different choices for λ in the augmented error. So, we have $(\mathcal{H}, \lambda_1), (\mathcal{H}, \lambda_2), \ldots, (\mathcal{H}, \lambda_M)$ as our M different models. We

may, for example, choose $\lambda_1 \quad 0, \lambda_2 \quad 0.01, \lambda_3 \quad 0.02, \ldots, \lambda_M \quad 10$. Using a validation set to choose one of these M models amounts to determining the value of λ to within a resolution of 0.01. □

We have analyzed validation for model selection based on a finite number of models. If validation is used to choose the value of a parameter, for example λ as in the previous example, then the value of M will depend on the resolution to which we determine that parameter. In the limit, the selection is actually among an infinite number of models since the value of λ can be any real number. What happens to bounds like (4.11) and (4.12) which depend on M? Just as the Hoeffding bound for a finite hypothesis set did not collapse when we moved to infinite hypothesis sets with finite VC-dimension, bounds like (4.11) and (4.12) will not completely collapse either. We can derive VC-type bounds here too, because even though there are an infinite number of models, these models are all very similar; they differ only slightly in the value of λ. As a rule of thumb, what matters is the number of parameters we are trying to set. If we have only one or a few parameters, the estimates based on a decent-sized validation set would be reliable. The more choices we make based on the same validation set, the more 'contaminated' the validation set becomes and the less reliable its estimates will be. The more we use the validation set to fine tune the model, the more the validation set becomes like a *training* set used to 'learn the right *model*'; and we all know how limited a training set is in its ability to estimate E_{out}.

You will be hard pressed to find a serious learning problem in which validation is not used. Validation is a conceptually simple technique, easy to apply in almost any setting, and requires no specific knowledge about the details of a model. The main drawback is the reduced size of the training set, but that can be significantly mitigated through a modified version of validation which we discuss next.

4.3.3 Cross Validation

Validation relies on the following chain of reasoning,

$$\underset{\text{(small } K)}{E_{\text{out}}(g)} \approx \underset{\text{(large } K)}{E_{\text{out}}(g^-)} \approx E_{\text{val}}(g^-),$$

which highlights the dilemma we face in trying to select K. We are going to output g. When K is large, there is a discrepancy between the two out-of-sample errors $E_{\text{out}}(g\ )$ (which E_{val} directly estimates) and $E_{\text{out}}(g)$ (which is the final error when we learn using all the data $\mathcal{D}$). We would like to choose K as small as possible in order to minimize the discrepancy between $E_{\text{out}}(g\ )$ and $E_{\text{out}}(g)$; ideally $K \quad 1$. However, if we make this choice, we lose the reliability of the validation estimate as the bound on the RHS of (4.9) becomes huge. The validation error $E_{\text{val}}(g\ )$ will still be an unbiased estimate of $E_{\text{out}}(g\ )$

(g is trained on $N-1$ points), but it will be so unreliable as to be useless since it is based on only one data point. This brings us to the *cross validation* estimate of out-of-sample error. We will focus on the *leave-one-out* version which corresponds to a validation set of size $K = 1$, and is also the easiest case to illustrate. More popular versions typically use larger K, but the essence of the method is the same.

There are N ways to partition the data into a training set of size $N-1$ and a validation set of size 1. Specifically, let

$$\mathcal{D}_n = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{n-1}, y_{n-1}), \cancel{(\mathbf{x}_n, y_n)}, (\mathbf{x}_{n+1}, y_{n+1}), \ldots, (\mathbf{x}_N, y_N)$$

be the data set $\mathcal{D}$ after leaving out data point $(\mathbf{x}_n, y_n)$, which has been shaded in red. Denote the final hypothesis learned from $\mathcal{D}_n$ by g_n^-. Let e_n be the error made by g_n^- on its validation set which is just a single data point $\{(\mathbf{x}_n, y_n)\}$:

$$\mathsf{e}_n = E_{\text{val}}(g_n^-) = \mathsf{e}\left(g_n^-(\mathbf{x}_n), y_n\right).$$

The cross validation estimate is the average value of the e_n's,

$$E_{\text{cv}} = \frac{1}{N}\sum_{n=1}^{N} \mathsf{e}_n.$$

Figure 4.13: Illustration of leave one out cross validation for a linear fit using three data points. The average of the three red errors obtained by the linear fits leaving out one data point at a time is E_{cv}.

Figure 4.13 illustrates cross validation on a simple example. Each e_n is a wild, yet unbiased estimate for the corresponding $E_{\text{out}}(g_n^-)$, which follows after setting $K = 1$ in (4.8). With cross validation, we have N functions $g_1^-, \ldots, g_N^-$ together with the N error estimates $\mathsf{e}_1, \ldots, \mathsf{e}_N$. The hope is that these N errors *together* would be almost equivalent to estimating E_{out} on a reliable validation set of size N, while at the same time we managed to use $N-1$ points to obtain each g_n^-. Let's try to understand why E_{cv} is a good estimator of E_{out}.

First and foremost, E_{cv} is an unbiased estimator of '$E_{\text{out}}(g\ )$'. We have to be a little careful here because we don't have a single hypothesis g , as we did when using a single validation set. Depending on the $(\mathbf{x}_n, y_n)$ that was taken out, each g_n^- can be a different hypothesis. To understand the sense in which E_{cv} estimates E_{out}, we need to revisit the concept of the learning curve.

Ideally, we would like to know $E_{\text{out}}(g)$. The final hypothesis g is the result of learning on a random data set $\mathcal{D}$ of size N. It is almost as useful to know the *expected* performance of your model when you learn on a data set of size N; the hypothesis g is just one such instance of learning on a data set of size N. This expected performance averaged over data sets of size N, when viewed as a function of N, is exactly the learning curve shown in Figure 4.2. More formally, for a given model, let

$$\bar{E}_{\text{out}}(N) \quad \mathbb{E}_{\mathcal{D}}[E_{\text{out}}(g)]$$

be the expectation (over data sets $\mathcal{D}$ of size N) of the out-of-sample error produced by the model. The expected value of E_{cv} is exactly $\bar{E}_{\text{out}}(N-1)$. This is true because it is true for each individual validation error e_n:

$$\begin{aligned}
\mathbb{E}_{\mathcal{D}}[\mathsf{e}_n] &= \mathbb{E}_{\mathcal{D}_n}\, \mathbb{E}_{(\mathbf{x}_n, y_n)}\, [\mathsf{e}(g_n^-(\mathbf{x}_n), y_n)]\,, \\
&= \mathbb{E}_{\mathcal{D}_n}[E_{\text{out}}(g_n^-)], \\
&= \bar{E}_{\text{out}}(N-1).
\end{aligned}$$

Since this equality holds for each e_n, it also holds for the average. We highlight this result by making it a theorem.

Theorem 4.4. E_{cv} is an unbiased estimate of $\bar{E}_{\text{out}}(N-1)$ (the expectation of the model performance, $\mathbb{E}[E_{\text{out}}]$, over data sets of size $N-1$).

Now that we have our cross validation estimate of E_{out}, there is no need to output any of the g_n^- as our final hypothesis. We might as well squeeze every last drop of performance and retrain using the entire data set $\mathcal{D}$, outputting g as the final hypothesis and getting the benefit of going from $N-1$ to N on the learning curve. In this case, the cross validation estimate will on average be an upper estimate for the out-of-sample error: $E_{\text{out}}(g) \leq E_{\text{cv}}$, so expect to be pleasantly surprised, albeit slightly.

Figure 4.14: Using cross validation to estimate E_{out}

With just simple validation and a validation set of size K 1, we know that the validation estimate will not be reliable. How reliable is the cross validation estimate E_{cv}? We can measure the reliability using the variance of E_{cv}.

Unfortunately, while we were able to pin down the expectation of E_{cv}, the variance is not so easy.

If the N cross validation errors $\mathsf{e}_1, \ldots, \mathsf{e}_N$ were equivalent to N errors on a totally separate validation set of size N, then E_{cv} would indeed be a reliable estimate, for decent-sized N. The equivalence would hold if the individual e_n's were independent of each other. Of course, this is too optimistic. Consider two validation errors $\mathsf{e}_n, \mathsf{e}_m$. The validation error e_n depends on g_n^- which was trained on data containing $(\mathbf{x}_m, y_m)$. Thus, e_n has a dependency on $(\mathbf{x}_m, y_m)$. The validation error e_m is computed using $(\mathbf{x}_m, y_m)$ directly, and so it also has a dependency on $(\mathbf{x}_m, y_m)$. Consequently, there is a possible correlation between e_n and e_m through the data point $(\mathbf{x}_m, y_m)$. That correlation wouldn't be there if we were validating a single hypothesis using N fresh (independent) data points.

How much worse is the cross validation estimate as compared to an estimate based on a truly independent set of N validation errors? A VC-type probabilistic bound, or even computation of the asymptotic variance of the cross validation estimate (Problem 4.23), is challenging. One way to quantify the reliability of E_{cv} is to compute how many fresh validation data points would have a comparable reliability to E_{cv}, and Problem 4.24 discusses one way to do this. There are two extremes for this effective size. On the high end is N, which means that the cross validation errors are essentially independent. On the low end is 1, which means that E_{cv} is only as good as any single one of the individual cross validation errors e_n, i.e., the cross validation errors are totally dependent. While one cannot prove anything theoretically, in practice the reliability of E_{cv} is much closer to the higher end.

Cross validation for model selection. In Figure 4.11, the estimates E_m for the out-of-sample error of model $\mathcal{H}_m$ were obtained using the validation set. Instead, we may use cross validation estimates to obtain E_m: use cross validation to obtain estimates of the out-of-sample error for each model $\mathcal{H}_1, \ldots, \mathcal{H}_M$, and select the model with the smallest cross validation error. Now, train this model selected by cross validation using all the data to output a final hypothesis, making the usual leap of faith that $E_{\text{out}}(g\ )$ tracks $E_{\text{out}}(g)$ well.

Example 4.5. In Figure 4.13, we illustrated cross validation for estimating E_{out} of a linear model $(h(x) \quad ax + b)$ using a simple experiment with three data points generated from a constant target function with noise. We now consider a second model, the constant model $(h(x) \quad b)$. We can also use cross validation to estimate E_{out} for the constant model, illustrated in Figure 4.15.

Figure 4.15: Leave one-out cross validation error for a constant fit.

If we use the in-sample error after fitting all the data (three points), then the linear model wins because it can use its additional degree of freedom to fit the data better. The same is true with the cross validation data sets of size two – the linear model has perfect in-sample error. But, with cross validation, what matters is the error on *the outstanding point* in each of these fits. Even to the naked eye, the average of the cross validation errors is smaller for the constant model which obtained E_{cv} 0.065 versus E_{cv} 0.184 for the linear model. The constant model wins, according to cross validation. The constant model also has lower E_{out} and so cross validation selected the correct model in this example. □

One important use of validation is to estimate the optimal regularization parameter λ, as described in Example 4.3. We can use cross validation for the same purpose as summarized in the algorithm below.

Cross validation for selecting λ:

1: Define M models by choosing different values for λ in the augmented error: $(\mathcal{H}, \lambda_1), (\mathcal{H}, \lambda_2), \ldots, (\mathcal{H}, \lambda_M)$
2: **for** each model m $1, \ldots, M$ **do**
3: Use the cross validation module in Figure 4.14 to estimate $E_{\text{cv}}(m)$, the cross validation error for model m.
4: Select the model m^* with minimum $E_{\text{cv}}(m^*)$.
5: Use model $(\mathcal{H}, \lambda_{m^*})$ and all the data $\mathcal{D}$ to obtain the final hypothesis g_{m^*}. Effectively, you have estimated the optimal λ.

We see from Figure 4.14 that estimating E_{cv} for just a single model requires N rounds of learning on $\mathcal{D}_1, \ldots, \mathcal{D}_N$, each of size N 1. So the cross validation algorithm above requires MN rounds of learning. This is a formidable task. If we could analytically obtain E_{cv}, that would be a big bonus, but analytic results are often difficult to come by for cross validation. One exception is in the case of linear models, where we are able to derive an exact analytic formula for the cross validation estimate.

Analytic computation of E_{cv} for linear models. Recall that for linear regression with weight decay, $\mathbf{w}_{\text{reg}} \quad (\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathrm{Z}^{\mathrm{T}}\mathbf{y}$, and the in-sample predictions are

$$\hat{\mathbf{y}} \quad \mathrm{H}(\lambda)\mathbf{y},$$

where $\mathrm{H}(\lambda) \quad \mathrm{Z}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathrm{Z}^{\mathrm{T}}$. Given H, $\hat{\mathbf{y}}$, and $\mathbf{y}$, it turns out that we can analytically compute the cross validation estimate as:

$$E_{\text{cv}} \quad \frac{1}{N}\sum_{n=1}^{N}\left(\frac{\hat{y}_n - y_n}{1 \quad \mathrm{H}_{nn}(\lambda)}\right)^2. \tag{4.13}$$

Notice that the cross validation estimate is very similar to the in-sample error, $E_{\text{in}} \quad \frac{1}{N}\sum_n(\hat{y}_n - y_n)^2$, differing only by a normalization of each term in the sum by a factor $1/(1 \quad \mathrm{H}_{nn}(\lambda))^2$. One use for this analytic formula is that it can be directly optimized to obtain the best regularization parameter λ. A proof of this remarkable formula is given in Problem 4.26.

Even when we cannot derive such an analytic characterization of cross validation, the technique widely results in good out-of-sample error estimates in practice, and so the computational burden is often worth enduring. Also, as with using a validation set, cross validation applies in almost any setting without requiring specific knowledge about the details of the models.

So far, we have lived in a world of unlimited computation, and all that mattered was out-of-sample error; in reality, computation time can be of consequence, especially with huge data sets. For this reason, leave-one-out cross validation may not be the method of choice.[4] A popular derivative of leave-one-out cross validation is *V-fold cross validation.*[5] In V-fold cross validation, the data are partitioned into V disjoint sets (or folds) $\mathcal{D}_1, \ldots, \mathcal{D}_V$, each of size approximately N/V; each set $\mathcal{D}_v$ in this partition serves as a validation set to compute a validation error for a hypothesis g learned on a training set which is the complement of the validation set, $\mathcal{D} \setminus \mathcal{D}_v$. So, you always validate a hypothesis on data that was *not* used for training that particular hypothesis. The V-fold cross validation error is the average of the V validation errors that are obtained, one from each validation set $\mathcal{D}_v$. Leave-one-out cross validation is the same as N-fold cross validation. The gain from choosing $V \ll N$ is computational. The drawback is that you will be estimating E_{out} for a hypothesis g trained on less data (as compared with leave-one-out) and so the discrepancy between $E_{\text{out}}(g)$ and $E_{\text{out}}(g\ )$ will be larger. A common choice in practice is 10-fold cross validation, and one of the folds is illustrated below.

[4]Stability problems have also been reported in leave one out.

[5]Some authors call it K fold cross validation, but we choose V so as not to confuse with the size of the validation set K.

4.3.4 Theory Versus Practice

Both validation and cross validation present challenges for the mathematical theory of learning, similar to the challenges presented by regularization. The theory of generalization, in particular the VC analysis, forms the foundation for learnability. It provides us with guidelines under which it is possible to make a generalization conclusion with high probability. It is not straightforward, and sometimes not possible, to rigorously carry these conclusions over to the analysis of validation, cross validation, or regularization. What is possible, and indeed quite effective, is to use the theory as a guideline. In the case of regularization, constraining the choice of a hypothesis leads to better generalization, as we would intuitively expect, even if the hypothesis set remains technically the same. In the case of validation, making a choice for few parameters does not overly contaminate the validation estimate of E_{out}, even if the VC guarantee for these estimates is too weak. In the case of cross validation, the benefit of averaging several validation errors is observed, even if the estimates are not independent.

Although these techniques were based on sound theoretical foundation, they are to be considered *heuristics* because they do not have a full mathematical justification in the general case. Learning from data is an empirical task with theoretical underpinnings. We prove what we can prove, but we use the theory as a guideline when we don't have a conclusive proof. In a practical application, heuristics may win over a rigorous approach that makes unrealistic assumptions. The only way to be convinced about what works and what doesn't in a given situation is to try out the techniques and see for yourself. The basic message in this chapter can be summarized as follows.

1. Noise (stochastic or deterministic) affects learning adversely, leading to overfitting.
2. Regularization helps to prevent overfitting by constraining the model, reducing the impact of the noise, while still giving us flexibility to fit the data.
3. Validation and cross validation are useful techniques for estimating E_{out}. One important use of validation is model selection, in particular to estimate the amount of regularization to use.

Example 4.6. We illustrate validation on the handwritten digit classification task of deciding whether a digit is 1 or not (see also Example 3.1) based on the two features which measure the symmetry and average intensity of the digit. The data is shown in Figure 4.16(a).

(a) Digits classification task (b)Error curves

Figure 4.16: (a) The digits data of which 500 are selected as the training set. (b) The data are transformed via the 5th order polynomial transform to a 20 dimensional feature vector. We show the performance curves as we vary the number of these features used for classification.

We have randomly selected 500 data points as the training data and the remaining are used as a test set for evaluation. We considered a nonlinear feature transform to a 5th order polynomial feature space:

$$(1, x_1, x_2) \to (1, x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1^2x_2, \ldots, x_1^5, x_1^4x_2, x_1^3x_2^2, x_1^2x_2^3, x_1x_2^4, x_2^5).$$

Figure 4.16(b) shows the in-sample error as you use more of the transformed features, increasing the dimension from 1 to 20. As you add more dimensions (increase the complexity of the model), the in-sample error drops, as expected. The out-of-sample error drops at first, and then starts to increase, as we hit the approximation-generalization tradeoff. The leave-one-out cross validation error tracks the behavior of the out-of-sample error quite well. If we were to pick a model based on the in-sample error, we would use all 20 dimensions. The cross validation error is minimized between 5-7 feature dimensions; we take 6 feature dimensions as the model selected by cross validation. The table below summarizes the resulting performance metrics:

	E_{in}	E_{out}
No Validation	0%	2.5%
Cross Validation	0.8%	1.5%

Cross validation results in a performance improvement of about 1%, which is a massive relative improvement (40% reduction in error rate).

Exercise 4.11

In this particular experiment, the black curve (E_{cv}) is sometimes below and sometimes above the the red curve (E_{out}). If we repeated this experiment many times, and plotted the average black and red curves, would you expect the black curve to lie above or below the red curve?

It is illuminating to see the actual classification boundaries learned with and without validation. These resulting classifiers, together with the 500 in-sample data points, are shown in the next figure.

20 dim classifier (no validation)
$E_{\text{in}} = 0\%$
$E_{\text{out}} = 2.5\%$

6 dim classifier (LOO-CV)
$E_{\text{in}} = 0.8\%$
$E_{\text{out}} = 1.5\%$

It is clear that the worse out-of-sample performance of the classifier picked without validation is due to the overfitting of a few noisy points in the training data. While the training data is perfectly separated, the shape of the resulting boundary seems highly contorted, which is a symptom of overfitting. Does this remind you of the first example that opened the chapter? There, albeit in a toy example, we similarly obtained a highly contorted fit. As you can see, overfitting is real, and here to stay! □

4.4 Problems

Problem 4.1 Plot the monomials of order i, $\phi_i(x) = x^i$. As you increase the order, does this correspond to the intuitive notion of increasing complexity?

Problem 4.2 Consider the feature transform $\mathbf{z} = [L_0(x), L_1(x), L_2(x)]^{\mathrm{T}}$ and the linear model $h(x) = \mathbf{w}^{\mathrm{T}}\mathbf{z}$. For the hypothesis with $\mathbf{w} = [1, -1, 1]^{\mathrm{T}}$, what is $h(x)$ explicitly as a function of x. What is its degree?

Problem 4.3 The Legendre Polynomials are a family of orthogonal polynomials which are useful for regression. The first two Legendre Polynomials are $L_0(x) = 1$, $L_1(x) = x$. The higher order Legendre Polynomials are defined by the recursion:

$$L_k(x) = \frac{2k-1}{k} x L_{k-1}(x) - \frac{k-1}{k} L_{k-2}(x).$$

(a) What are the first six Legendre Polynomials? Use the recursion to de velop an efficient algorithm to compute $L_0(x), \ldots, L_K(x)$ given x. Your algorithm should run in time linear in K. Plot the first six Legendre polynomials.

(b) Show that $L_k(x)$ is a linear combination of monomials $x^k, x^{k-2}, \ldots$ (ei ther all odd or all even order, with highest order k). Thus,

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

(c) Show that $\frac{x^2-1}{k}\frac{dL_k(x)}{dx} = xL_k(x) - L_{k-1}(x)$. *[Hint: use induction.]*

(d) Use part (c) to show that L_k satisfies *Legendre's differential equation*

$$\frac{d}{dx}(x^2-1)\frac{dL_k(x)}{dx} = k(k+1)L_k(x).$$

This means that the Legendre Polynomials are eigenfunctions of a Her mitian linear differential operator and, from Sturm Liouville theory, they form an orthogonal basis for continuous functions on $[-1, 1]$.

(e) Use the recurrence to show directly the orthogonality property:

$$\int_{-1}^{1} dx\; L_k(x)L_\ell(x) = \begin{cases} 0 & \ell \neq k, \\ \frac{2}{2k+1} & \ell - k. \end{cases}$$

[Hint: use induction on k, with $\ell \leq k$. Use the recurrence for L_k and consider separately the four cases $\ell = k, k-1, k-2$ and $\ell < k-2$. For the case $\ell = k$ you will need to compute the integral $\int_{-1}^{1} dx\; x^2 L_{k-1}^2(x)$. In order to do this, you could use the differential equation in part (c), multiply by xL_k and then integrate both sides (the LHS can be integrated by parts). Now solve the resulting equation for $\int_{-1}^{1} dx\; x^2 L_{k-1}^2(x)$.]

Problem 4.4 LAMi This problem is a detailed version of Exercise 4.2. We set up an experimental framework which the reader may use to study various aspects of overfitting. The input space is $\mathcal{X} = [-1, 1]$, with uniform input probability density, $P(x) = \frac{1}{2}$. We consider the two models $\mathcal{H}_2$ and $\mathcal{H}_{10}$. The target function is a polynomial of degree Q_f, which we write as $f(x) = \sum_{q=0}^{Q_f} a_q L_q(x)$, where $L_q(x)$ are the Legendre polynomials. We use the Legendre polynomials because they are a convenient orthogonal basis for the polynomials on $[-1, 1]$ (see Section 4.2 and Problem 4.3 for some basic information on Legendre polynomials). The data set is $\mathcal{D} = (x_1, y_1), \ldots, (x_N, y_N)$, where $y_n = f(x_n) + \sigma\epsilon_n$ and ϵ_n are iid standard Normal random variates.

For a single experiment, with specified values for Q_f, N, σ, generate a random degree-Q_f target function by selecting coefficients a_q independently from a standard Normal, rescaling them so that $\mathbb{E}_{\mathbf{a},x}\left[f^2\right] = 1$. Generate a data set, selecting $x_1, \ldots, x_N$ independently from $P(x)$ and $y_n = f(x_n) + \sigma\epsilon_n$. Let g_2 and g_{10} be the best fit hypotheses to the data from $\mathcal{H}_2$ and $\mathcal{H}_{10}$ respectively, with respective out of-sample errors $E_{\text{out}}(g_2)$ and $E_{\text{out}}(g_{10})$.

(a) Why do we normalize f? *[Hint: how would you interpret σ?]*

(b) How can we obtain g_2, g_{10}? *[Hint: pose the problem as linear regression and use the technology from Chapter 3.]*

(c) How can we compute E_{out} analytically for a given g_{10}?

(d) Vary Q_f, N, σ and for each combination of parameters, run a large number of experiments, each time computing $E_{\text{out}}(g_2)$ and $E_{\text{out}}(g_{10})$. Averaging these out-of-sample errors gives estimates of the expected out-of sample error for the given learning scenario (Q_f, N, σ) using $\mathcal{H}_2$ and $\mathcal{H}_{10}$. Let

$$\begin{aligned} E_{\text{out}}(\mathcal{H}_2) &= \text{average over experiments}(E_{\text{out}}(g_2)), \\ E_{\text{out}}(\mathcal{H}_{10}) &= \text{average over experiments}(E_{\text{out}}(g_{10})). \end{aligned}$$

Define the overfit measure $E_{\text{out}}(\mathcal{H}_{10}) - E_{\text{out}}(\mathcal{H}_2)$. When is the overfit measure significantly positive (i.e., overfitting is serious) as opposed to significantly negative? Try the choices $Q_f \in \{1, 2, \ldots, 100\}$, $N \in \{20, 25, \ldots, 120\}$, $\sigma^2 \in \{0, 0.05, 0.1, \ldots, 2\}$.

Explain your observations.

(e) Why do we take the average over many experiments? Use the variance to select an acceptable number of experiments to average over.

(f) Repeat this experiment for classification, where the target function is a noisy perceptron, $f = \text{sign}\left(\sum_{q=1}^{Q_f} a_q L_q(x) + \epsilon\right)$. Notice that $a_0 = 0$, and the a_q's should be normalized so that $\mathbb{E}_{\mathbf{a},x}\left[\left(\sum_{q=1}^{Q_f} a_q L_q(x)\right)^2\right] = 1$. For classification, the models $\mathcal{H}_2, \mathcal{H}_{10}$ contain the sign of the 2nd and 10th order polynomials respectively. You may use a learning algorithm for non-separable data from Chapter 3.

Problem 4.5 If $\lambda < 0$ in the augmented error $E_{\text{aug}}(\mathbf{w}) = E_{\text{in}}(\mathbf{w}) + \lambda \mathbf{w}^{\mathrm{T}}\mathbf{w}$, what soft order constraint does this correspond to? *[Hint: $\lambda < 0$ encourages large weights.]*

Problem 4.6 In the augmented error minimization with $\Gamma = \mathrm{I}$ and $\lambda > 0$:

(a) Show that $\|\mathbf{w}_{\text{reg}}\| \leq \|\mathbf{w}_{\text{lin}}\|$, justifying the term weight decay. *[Hint: start by assuming that $\|\mathbf{w}_{\text{reg}}\| > \|\mathbf{w}_{\text{lin}}\|$ and derive a contradiction.]*

In fact a stronger statement holds: $\|\mathbf{w}_{\text{reg}}\|$ is decreasing in λ.

(b) Explicitly verify this for linear models. *[Hint:*

$$\mathbf{w}_{\text{reg}}^{\mathrm{T}}\mathbf{w}_{\text{reg}} = \mathbf{u}^{\mathrm{T}}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\mathrm{I})^{-2}\mathbf{u},$$

where $\mathbf{u} = \mathrm{Z}^{\mathrm{T}}\mathbf{y}$ and Z is the transformed data matrix. Show that $\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\mathrm{I}$ has the same eigenvectors with correspondingly larger eigenvalues as $\mathrm{Z}^{\mathrm{T}}\mathrm{Z}$. Expand $\mathbf{u}$ in the eigenbasis of $\mathrm{Z}^{\mathrm{T}}\mathrm{Z}$. For a matrix A, how are the eigenvectors and eigenvalues of A^{-2} related to those of A?]

Problem 4.7 Show that the in-sample error

$$E_{\text{in}}(\mathbf{w}_{\text{reg}}) = \frac{1}{N}\mathbf{y}^{\mathrm{T}}(\mathrm{I} - \mathrm{H}(\lambda))^2\mathbf{y}$$

from Example 4.2 is an increasing function of λ, where $\mathrm{H}(\lambda) = \mathrm{Z}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z}+\lambda\mathrm{I})^{-1}\mathrm{Z}^{\mathrm{T}}$ and Z is the transformed data matrix.

To do so, let the SVD of $\mathrm{Z} = \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}$ and let $\mathrm{Z}^{\mathrm{T}}\mathrm{Z}$ have eigenvalues $\sigma_1^2, \ldots, \sigma_d^2$. Define the vector $\mathbf{a} = \mathrm{U}^{\mathrm{T}}\mathbf{y}$. Show that

$$E_{\text{in}}(\mathbf{w}_{\text{reg}}) = E_{\text{in}}(\mathbf{w}_{\text{lin}}) + \frac{1}{N}\sum_{i=1}^{d} a_i^2\left(1 - \frac{\sigma_i^2}{\sigma_i^2 + \lambda}\right)^2,$$

and proceed from there.

Problem 4.8 In the augmented error minimization with $\Gamma = \mathrm{I}$ and $\lambda > 0$, assume that E_{in} is differentiable and use gradient descent to minimize E_{aug}:

$$\mathbf{w}(t+1) \leftarrow \mathbf{w}(t) - \eta\nabla E_{\text{aug}}(\mathbf{w}(t)).$$

Show that the update rule above is the same as

$$\mathbf{w}(t+1) \leftarrow (1 - 2\eta\lambda)\mathbf{w}(t) - \eta\nabla E_{\text{in}}(\mathbf{w}(t)).$$

Note: This is the origin of the name 'weight decay': $\mathbf{w}(t)$ decays before being updated by the gradient of E_{in}.

Problem 4.9 In Tikhonov regularization, the regularized weights are given by $\mathbf{w}_{\text{reg}} \quad (\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\Gamma^{\mathrm{T}}\Gamma)^{-1}\mathrm{Z}^{\mathrm{T}}\mathbf{y}$. The Tikhonov regularizer Γ is a $k \times (d+1)$ matrix, each row corresponding to a $d+1$ dimensional vector. Each row of Z corresponds to a $d+1$ dimensional vector (the first component is 1). For each row of Γ, construct a *virtual example* $(\mathbf{z}_i, 0)$ for $i = 1, \ldots, k$, where $\mathbf{z}_i$ is the vector obtained from the ith row of Γ after scaling it by $\sqrt{\lambda}$, and the target value is 0. Add these k virtual examples to the data, to construct an augmented data set, and consider non-regularized regression with this augmented data.

(a) Show that, for the augmented data, $\mathrm{Z}_{\text{aug}} \quad \begin{bmatrix} \mathrm{Z} \\ \sqrt{\lambda} \cdot \Gamma \end{bmatrix}$ and $\mathbf{y}_{\text{aug}} \quad \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix}$

(b) Show that solving the least squares problem with Z_{aug} and $\mathbf{y}_{\text{aug}}$ results in the same regularized weight $\mathbf{w}_{\text{reg}}$, i.e. $\mathbf{w}_{\text{reg}} \quad (\mathrm{Z}_{\text{aug}}^{\mathrm{T}}\mathrm{Z}_{\text{aug}})^{-1}\mathrm{Z}_{\text{aug}}^{\mathrm{T}}\mathbf{y}_{\text{aug}}$.

This result may be interpreted as follows: an equivalent way to accomplish weight-decay-type regularization with linear models is to create a bunch of *virtual examples* all of whose target values are zero.

Problem 4.10 In this problem, you will investigate the relationship between the soft order constraint and the augmented error. The regularized weight $\mathbf{w}_{\text{reg}}$ is a solution to

$$\min E_{\text{in}}(\mathbf{w}) \text{ subject to } \mathbf{w}^{\mathrm{T}}\Gamma^{\mathrm{T}}\Gamma\mathbf{w} \leq C.$$

(a) If $\mathbf{w}_{\text{lin}}^{\mathrm{T}}\Gamma^{\mathrm{T}}\Gamma\mathbf{w}_{\text{lin}} \leq C$, then what is $\mathbf{w}_{\text{reg}}$?

(b) If $\mathbf{w}_{\text{lin}}^{\mathrm{T}}\Gamma^{\mathrm{T}}\Gamma\mathbf{w}_{\text{lin}} > C$, the situation is illustrated below,

The constraint is satisfied in the shaded region and the contours of constant E_{in} are the ellipsoids (why ellipsoids?). What is $\mathbf{w}_{\text{reg}}^{\mathrm{T}}\Gamma^{\mathrm{T}}\Gamma\mathbf{w}_{\text{reg}}$?

(c) Show that with

$$\lambda_C = -\frac{1}{2C}\mathbf{w}_{\text{reg}}^{\mathrm{T}}\nabla E_{\text{in}}(\mathbf{w}_{\text{reg}}),$$

$\mathbf{w}_{\text{reg}}$ minimizes $E_{\text{in}}(\mathbf{w}) + \lambda_C\mathbf{w}^{\mathrm{T}}\Gamma^{\mathrm{T}}\Gamma\mathbf{w}$. *[Hint: use the previous part to solve for $\mathbf{w}_{\text{reg}}$ as an equality constrained optimization problem using the method of Lagrange multipliers.]*

(continued on next page)

(d) Show that the following hold for λ_C:

(i) If $\mathbf{w}_{\text{lin}}^{\text{T}}\Gamma^{\text{T}}\Gamma\mathbf{w}_{\text{lin}} \leq C$ then $\lambda_C = 0$ ($\mathbf{w}_{\text{lin}}$ itself satisfies the constraint).

(ii) If $\mathbf{w}_{\text{lin}}^{\text{T}}\Gamma^{\text{T}}\Gamma\mathbf{w}_{\text{lin}} > C$, then $\lambda_C > 0$ (the penalty term is positive).

(iii) If $\mathbf{w}_{\text{lin}}^{\text{T}}\Gamma^{\text{T}}\Gamma\mathbf{w}_{\text{lin}} > C$, then λ_C is a strictly decreasing function of C. *[Hint: show that $\frac{d\lambda_C}{dC} < 0$ for $C \in [0, \mathbf{w}_{\text{lin}}^{\text{T}}\Gamma^{\text{T}}\Gamma\mathbf{w}_{\text{lin}}]$.]*

Problem 4.11 For the linear model in Exercise 4.2, the target function is a polynomial of degree Q_f; the model is $\mathcal{H}_Q$, with polynomials up to order Q. Assume $Q \geq Q_f$. $\mathbf{w}_{\text{lin}} = (\mathrm{Z}^{\text{T}}\mathrm{Z})^{-1}\mathrm{Z}^{\text{T}}\mathbf{y}$, and $\mathbf{y} = \mathrm{Z}\mathbf{w}_{\text{f}} + \boldsymbol{\epsilon}$, where $\mathbf{w}_{\text{f}}$ is the target function and Z is the matrix containing the transformed data.

(a) Show that $\mathbf{w}_{\text{lin}} = \mathbf{w}_{\text{f}} + (\mathrm{Z}^{\text{T}}\mathrm{Z})^{-1}\mathrm{Z}^{\text{T}}\boldsymbol{\epsilon}$. What is the average function $\bar{g}$? Show that bias $= 0$ (recall that: bias$(\mathbf{x}) = (\bar{g}(\mathbf{x}) - f(\mathbf{x}))^2$).

(b) Show that

$$\text{var} = \frac{\sigma^2}{N}\text{trace}\left(\Sigma_\Phi\, \mathbb{E}_{\mathrm{Z}}\left[\left(\tfrac{1}{N}\mathrm{Z}^{\text{T}}\mathrm{Z}\right)^{-1}\right]\right),$$

where $\Sigma_\Phi = \mathbb{E}[\Phi(x)\Phi^{\text{T}}(x)]$. *[Hints: var $= \mathbb{E}[(g^{(\mathcal{D})} - \bar{g})^2]$; first take the expectation with respect to $\boldsymbol{\epsilon}$, then with respect to $\Phi(x)$, the test point, and the last remaining expectation will be with respect to Z. You will need the cyclic property of the trace.]*

(c) Argue that to first order in $\frac{1}{N}$, var $\approx \dfrac{\sigma^2(Q+1)}{N}$.
[Hint: $\frac{1}{N}\mathrm{Z}^{\text{T}}\mathrm{Z} = \frac{1}{N}\sum_{n=1}^{N}\Phi(x_n)\Phi^{\text{T}}(x_n)$ is the in-sample estimate of Σ_Φ. By the law of large numbers, $\frac{1}{N}\mathrm{Z}^{\text{T}}\mathrm{Z} = \Sigma_\Phi + o(1)$.]

For the well specified linear model, the bias is zero and the variance is increasing as the model gets larger (Q increases), but decreasing in N.

Problem 4.12 Use the setup in Problem 4.11 with $Q \geq Q_f$. Consider regression with weight decay using a linear model $\mathcal{H}$ in the transformed space with input probability distribution such that $\mathbb{E}[\mathbf{z}\mathbf{z}^{\text{T}}] = \mathrm{I}$. The regularized weights are given by $\mathbf{w}_{\text{reg}} = (\mathrm{Z}^{\text{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathrm{Z}^{\text{T}}\mathbf{y}$, where $\mathbf{y} = \mathrm{Z}\mathbf{w}_{\text{f}} + \boldsymbol{\epsilon}$.

(a) Show that $\mathbf{w}_{\text{reg}} = \mathbf{w}_{\text{f}} - \lambda(\mathrm{Z}^{\text{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathbf{w}_{\text{f}} + (\mathrm{Z}^{\text{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathrm{Z}^{\text{T}}\boldsymbol{\epsilon}$.

(b) Argue that, to first order in $\frac{1}{N}$,

$$\begin{aligned}\text{bias} &\approx \frac{\lambda^2}{(\lambda+N)^2}\|\mathbf{w}_{\text{f}}\|^2,\\ \text{var} &\approx \frac{\sigma^2}{N}\,\mathbb{E}\left[\text{trace}(\mathrm{H}^2(\lambda))\right],\end{aligned}$$

where $\mathrm{H}(\lambda) = \mathrm{Z}(\mathrm{Z}^{\text{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathrm{Z}^{\text{T}}$.

If we plot the bias and var, we get a figure that is very similar to Figure 2.3, where the tradeoff was based on fit and com plexity rather than bias and var. Here, the bias is increasing in λ (as expected) and in $\|\mathbf{w}_{\mathrm{f}}\|$; the variance is decreasing in λ. When $\lambda = 0$, $\mathrm{trace}(\mathrm{H}^2(\lambda)) = Q + 1$ and so $\mathrm{trace}(\mathrm{H}^2(\lambda))$ appears to be playing the role of an effective number of parameters.

Problem 4.13 Within the linear regression setting, many attempts have been made to quantify the effective number of parameters in a model. Three possibilities are:

(i) $d_{\mathrm{eff}}(\lambda) = 2\mathrm{trace}(\mathrm{H}(\lambda)) - \mathrm{trace}(\mathrm{H}^2(\lambda))$

(ii) $d_{\mathrm{eff}}(\lambda) = \mathrm{trace}(\mathrm{H}(\lambda))$

(iii) $d_{\mathrm{eff}}(\lambda) = \mathrm{trace}(\mathrm{H}^2(\lambda))$

where $\mathrm{H}(\lambda) = \mathrm{Z}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathrm{Z}^{\mathrm{T}}$ and Z is the transformed data matrix. To obtain d_{eff}, one must first compute $\mathrm{H}(\lambda)$ as though you are doing regression. One can then heuristically use d_{eff} in place of d_{vc} in the VC bound.

(a) When $\lambda = 0$, show that for all three choices, $d_{\mathrm{eff}} = \tilde{d} + 1$, where $\tilde{d}$ is the dimension in the $\mathcal{Z}$ space.

(b) When $\lambda > 0$, show that $0 \le d_{\mathrm{eff}} \le \tilde{d} + 1$ and d_{eff} is decreasing in λ for all three choices. *[Hint: Use the singular value decomposition.]*

Problem 4.14 The observed target values $\mathbf{y}$ can be separated into the true target values $\mathbf{f}$ and the noise $\boldsymbol{\epsilon}$, $\mathbf{y} = \mathbf{f} + \boldsymbol{\epsilon}$. The components of $\boldsymbol{\epsilon}$ are iid with variance σ^2 and expectation 0. For linear regression with weight decay regularization, by taking the expected value of the in sample error in (4.2), show that

$$\begin{aligned}\mathbb{E}_{\boldsymbol{\epsilon}}[E_{\mathrm{in}}] &= \frac{1}{N}\mathbf{f}^{\mathrm{T}}(\mathrm{I} - \mathrm{H}(\lambda))^2\mathbf{f} + \frac{\sigma^2}{N}\mathrm{trace}\left((\mathrm{I} - \mathrm{H}(\lambda))^2\right), \\ &= \frac{1}{N}\mathbf{f}^{\mathrm{T}}(\mathrm{I} - \mathrm{H}(\lambda))^2\mathbf{f} + \sigma^2\left(1 - \frac{d_{\mathrm{eff}}}{N}\right),\end{aligned}$$

where $d_{\mathrm{eff}} = 2\mathrm{trace}(\mathrm{H}(\lambda)) - \mathrm{trace}(\mathrm{H}^2(\lambda))$, as defined in Problem 4.13(i), $\mathrm{H}(\lambda) = \mathrm{Z}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\mathrm{I})^{-1}\mathrm{Z}^{\mathrm{T}}$ and Z is the transformed data matrix.

(continued on next page)

(a) If the noise was not overfit, what should the term involving σ^2 be, and why?

(b) Hence, argue that the degree to which the noise has been overfit is $\sigma^2 d_{\text{eff}}/N$. Interpret the dependence of this result on the parameters d_{eff} and N, to justify the use of d_{eff} as an effective number of parameters.

Problem 4.15 We further investigate d_{eff} of Problems 4.13 and 4.14. We know that $\mathrm{H}(\lambda) = \mathrm{Z}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\Gamma^{\mathrm{T}}\Gamma)^{-1}\mathrm{Z}^{\mathrm{T}}$. When Γ is square and invertible, as is usually the case (for example with weight decay, $\Gamma = \mathrm{I}$), denote $\tilde{\mathrm{Z}} = \mathrm{Z}\Gamma^{-1}$. Let $s_0^2, \ldots, s_d^2$ be the eigenvalues of $\tilde{\mathrm{Z}}^{\mathrm{T}}\tilde{\mathrm{Z}}$ ($s_i^2 > 0$ when Z has full column rank).

(a) For $d_{\text{eff}}(\lambda) = \mathrm{trace}(2\mathrm{H}(\lambda) - \mathrm{H}^2(\lambda))$, show that

$$d_{\text{eff}}(\lambda) = d + 1 - \sum_{i=0}^{d} \frac{\lambda^2}{(s_i^2 + \lambda)^2}.$$

(b) For $d_{\text{eff}}(\lambda) = \mathrm{trace}(\mathrm{H}(\lambda))$, show that $d_{\text{eff}}(\lambda) = d + 1 - \sum_{i=0}^{d} \frac{\lambda}{s_i^2+\lambda}$.

(c) For $d_{\text{eff}}(\lambda) = \mathrm{trace}(\mathrm{H}^2(\lambda))$, show that $d_{\text{eff}}(\lambda) = \sum_{i=0}^{d} \frac{s_i^4}{(s_i^2+\lambda)^2}$.

In all cases, for $\lambda \geq 0$, $0 \leq d_{\text{eff}}(\lambda) \leq d+1$, $d_{\text{eff}}(0) = d+1$ and d_{eff} is decreasing in λ. *[Hint: use the singular value decomposition* $\tilde{\mathrm{Z}} = \mathrm{USV}^{\mathrm{T}}$, *where* U, V *are orthogonal and* S *is diagonal with entries* s_i*.]*

Problem 4.16 For linear models and the general Tikhonov regularizer Γ with penalty term $\frac{\lambda}{N}\mathbf{w}^{\mathrm{T}}\Gamma^{\mathrm{T}}\Gamma\mathbf{w}$ in the augmented error, show that

$$\mathbf{w}_{\text{reg}} = (\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\Gamma^{\mathrm{T}}\Gamma)^{-1}\mathrm{Z}^{\mathrm{T}}\mathbf{y},$$

where Z is the feature matrix.

(a) Show that the in-sample predictions are

$$\hat{\mathbf{y}} = \mathrm{H}(\lambda)\mathbf{y},$$

where $\mathrm{H}(\lambda) = \mathrm{Z}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\Gamma^{\mathrm{T}}\Gamma)^{-1}\mathrm{Z}^{\mathrm{T}}$.

(b) Simplify this in the case $\Gamma = \mathrm{Z}$ and obtain $\mathbf{w}_{\text{reg}}$ in terms of $\mathbf{w}_{\text{lin}}$. This is called uniform weight decay.

Problem 4.17 To model uncertainty in the measurement of the inputs, assume that the true inputs $\hat{\mathbf{x}}_n$ are the observed inputs $\mathbf{x}_n$ perturbed by some noise $\boldsymbol{\epsilon}_n$: the true inputs are given by $\hat{\mathbf{x}}_n = \mathbf{x}_n + \boldsymbol{\epsilon}_n$. Assume that the $\boldsymbol{\epsilon}_n$ are independent of $(\mathbf{x}_n, y_n)$ with covariance matrix $\mathbb{E}[\boldsymbol{\epsilon}_n\boldsymbol{\epsilon}_n^{\mathrm{T}}] = \sigma_x^2\mathrm{I}$ and mean

$\mathbb{E}[\epsilon_n] = \mathbf{0}$. The learning algorithm minimizes the *expected in sample error* $\hat{E}_{\text{in}}$, where the expectation is with respect to the uncertainty in the true $\hat{\mathbf{x}}_n$.

$$\hat{E}_{\text{in}}(\mathbf{w}) = \mathbb{E}_{\epsilon_1 \ldots \epsilon_N}\left[\frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}^{\mathrm{T}}\hat{\mathbf{x}}_n - y_n)^2\right].$$

Show that the weights $\hat{\mathbf{w}}_{\text{lin}}$ which result from minimizing $\hat{E}_{\text{in}}$ are equivalent to the weights which would have been obtained by minimizing $E_{\text{in}} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n - y_n)^2$ for the observed data, with Tikhonov regularization. What are Γ and λ (see Problem 4.16 for the general Tikhonov regularizer)?

One can interpret this result as follows: regularization enforces a robustness to potential measurement errors (noise) in the observed inputs.

Problem 4.18 In a regression setting, assume the target function is linear, so $f(\mathbf{x}) = \mathbf{w}_f^{\mathrm{T}}\mathbf{x}$, and $\mathbf{y} = \mathrm{Z}\mathbf{w}_f + \boldsymbol{\epsilon}$, where the entries in $\boldsymbol{\epsilon}$ are iid with zero mean and variance σ^2. Assume a regularization term $\frac{\lambda}{N}\mathbf{w}^{\mathrm{T}}\mathrm{Z}^{\mathrm{T}}\mathrm{Z}\mathbf{w}$ and that $\mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}] = \mathrm{I}$. In this problem derive the optimal value for λ as follows.

(a) Show that the average function is $\bar{g}(\mathbf{x}) = \frac{1}{1+\lambda}f(\mathbf{x})$. What is the bias?

(b) Show that var is asymptotically $\frac{\sigma^2(d+1)}{N(1+\lambda)^2}$. *[Hint: Problem 4.12.]*

(c) Use the bias and asymptotic variance to obtain an expression for $\mathbb{E}[E_{\text{out}}]$. Optimize this with respect to λ to obtain the optimal regularization parameter. *[Answer: $\lambda^* = \frac{\sigma^2(d+1)}{N\|\mathbf{w}_f\|^2}$].*

(d) Explain the dependence of the optimal regularization parameter on the parameters of the learning problem. *[Hint: write $\lambda^* = \frac{(d+1)/N}{\|\mathbf{w}_f\|^2/\sigma^2}$.]*

Problem 4.19 [The Lasso algorithm] Rather than a soft order constraint on the squares of the weights, one could use the absolute values of the weights:

$$\min E_{\text{in}}(\mathbf{w}) \text{ subject to } \sum_{i=0}^{d}|w_i| \leq C.$$

The model is called the lasso algorithm.

(a) Formulate and implement this as a quadratic program. Use the experimental design in Problem 4.4 to compare the lasso algorithm with the quadratic penalty by giving plots of E_{out} versus regularization parameter.

(b) What is the augmented error? Is it more convenient to optimize?

(c) With $d = 5$ and $N = 3$, compare the weights from the lasso versus the quadratic penalty. *[Hint: Look at the number of non-zero weights.]*

Problem 4.20 In this problem, you will explore a consistency condition for weight decay. Suppose that we make an invertible linear transform of the data,

$$\mathbf{z}_n = \mathrm{A}\mathbf{x}_n, \qquad \tilde{y}_n = \alpha y_n.$$

Intuitively, linear regression should not be affected by a linear transform. This means that the new optimal weights should be given by a corresponding linear transform of the old optimal weights.

(a) Suppose $\mathbf{w}$ minimizes the in sample error for the original problem. Show that for the transformed problem, the optimal weights are

$$\tilde{\mathbf{w}} = \alpha(\mathrm{A}^{\mathrm{T}})^{-1}\mathbf{w}.$$

(b) Suppose the regularization penalty term in the augmented error is $\mathbf{w}^{\mathrm{T}}\mathrm{X}^{\mathrm{T}}\mathrm{X}\mathbf{w}$ for the original data and $\mathbf{w}^{\mathrm{T}}\mathrm{Z}^{\mathrm{T}}\mathrm{Z}\mathbf{w}$ for the transformed data. On the original data, the regularized solution is $\mathbf{w}_{\mathrm{reg}}(\lambda)$. Show that for the transformed problem, the same linear transform of $\mathbf{w}_{\mathrm{reg}}(\lambda)$ gives the corresponding regularized weights for the transformed problem:

$$\tilde{\mathbf{w}}_{\mathrm{reg}}(\lambda) = \alpha(\mathrm{A}^{\mathrm{T}})^{-1}\mathbf{w}_{\mathrm{reg}}(\lambda).$$

Problem 4.21 The Tikhonov smoothness penalty which penalizes derivatives of h is $\Omega(h) = \int dx \left(\frac{\partial^2 h(x)}{\partial x^2}\right)^2$. Show that, for linear models, this reduces to a penalty of the form $\mathbf{w}^{\mathrm{T}}\Gamma^{\mathrm{T}}\Gamma\mathbf{w}$. What is Γ?

Problem 4.22 You have a data set with 100 data points. You have 100 models each with VC dimension 10. You set aside 25 points for validation. You select the model which produced minimum validation error of 0.25. Give a bound on the out of sample error for this selected function.

Suppose you instead trained each model on all the data and selected the func tion with minimum in sample error. The resulting in sample error is 0.15. Give a bound on the out of sample error in this case. *[Hint: Use the bound in Problem 2.14 to bound the VC dimension of the union of all the models.]*

Problem 4.23 This problem investigates the covariance of the leave one out cross validation errors, $\mathrm{Cov}_{\mathcal{D}}[\mathrm{e}_n, \mathrm{e}_m]$. Assume that for well behaved models, the learning process is 'stable', and so the change in the learned hypothesis should be small, '$O\left(\frac{1}{N}\right)$', if a new data point is added to a data set of size N. Write $g_n^- = g^{(N\ \ 2)} + \delta_n$ and $g_m^- = g^{(N\ \ 2)} + \delta_m$, where $g^{(N\ \ 2)}$ is the learned hypothesis on $\mathcal{D}^{(N-2)}$, the data minus the nth and mth data points, and δ_n, δ_m are the corrections after addition of the nth and mth data points respectively.

(a) Show that $\mathrm{Var}_{\mathcal{D}}[E_{\mathrm{cv}}] = \frac{1}{N^2}\sum_{n=1}^{N}\mathrm{Var}_{\mathcal{D}}[\mathrm{e}_n] + \frac{1}{N^2}\sum_{n\neq m}^{N}\mathrm{Cov}_{\mathcal{D}}[\mathrm{e}_n, \mathrm{e}_m]$.

(b) Show $\mathrm{Cov}_{\mathcal{D}}[\mathrm{e}_n, \mathrm{e}_m] = \mathrm{Var}_{\mathcal{D}(N\ \ 2)}[E_{\mathrm{out}}(g^{(N-2)})]+$ higher order in δ_n, δ_m.

(c) Assume that any terms involving δ_n, δ_m are $O(\frac{1}{N})$. Argue that

$$\mathrm{Var}_{\mathcal{D}}[E_{\mathrm{cv}}] = \frac{1}{N}\mathrm{Var}_{\mathcal{D}}[\mathrm{e}_1] + \mathrm{Var}_{\mathcal{D}}[E_{\mathrm{out}}(g)] + O(\frac{1}{N}).$$

Does $\mathrm{Var}_{\mathcal{D}}[\mathrm{e}_1]$ decay to zero with N? What about $\mathrm{Var}_{\mathcal{D}}[E_{\mathrm{out}}(g)]$?

(d) Use the experimental design in Problem 4.4 to study $\mathrm{Var}_{\mathcal{D}}[E_{\mathrm{cv}}]$ and give a log log plot of $\mathrm{Var}_{\mathcal{D}}[E_{\mathrm{cv}}]/\mathrm{Var}_{\mathcal{D}}[\mathrm{e}_1]$ versus N. What is the decay rate?

Problem 4.24 For $d = 3$, generate a random data set with N points as follows. For each point, each dimension of $\mathbf{x}$ has a standard Normal distribution. Similarly, generate a $(d+1)$ dimensional target weight vector $\mathbf{w}_{\mathrm{f}}$, and set $y_n = \mathbf{w}_{\mathrm{f}}^{\mathrm{T}}\mathbf{x}_n + \sigma\epsilon_n$ where ϵ_n is noise (also from a standard Normal distribution) and σ is the noise variance; set σ to 0.5.

Use linear regression with weight decay regularization to estimate $\mathbf{w}_{\mathrm{f}}$ with $\mathbf{w}_{\mathrm{reg}}$. Set the regularization parameter to $0.05/N$.

(a) For $N \in \{d+15, d+25, \ldots, d+115\}$, compute the cross validation errors $\mathrm{e}_1, \ldots, \mathrm{e}_N$ and E_{cv}. Repeat the experiment (say) 10^5 times, maintaining the average and variance over the experiments of e_1, e_2 and E_{cv}.

(b) How should your average of the e_1's relate to the average of the E_{cv}'s; how about to the average of the e_2's? Support your claim using results from your experiment.

(c) What are the contributors to the variance of the e_1's?

(d) If the cross validation errors were truly independent, how should the variance of the e_1's relate to the variance of the E_{cv}'s?

(e) One measure of the effective number of fresh examples used in computing E_{cv} is the ratio of the variance of the e_1's to that of the E_{cv}'s. Explain why, and plot, versus N, the effective number of fresh examples (N_{eff}) as a percentage of N. You should find that N_{eff} is close to N.

(f) If you increase the amount of regularization, will N_{eff} go up or down? Explain your reasoning. Run the same experiment with $\lambda = 2.5/N$ and compare your results from part (e) to verify your conjecture.

Problem 4.25 When using a validation set for model selection, all models were learned on the *same* $\mathcal{D}_{\mathrm{train}}$ of size $N-K$, and validated on the *same* $\mathcal{D}_{\mathrm{val}}$ of size K. We have the VC bound (see Equation (4.12)):

$$E_{\mathrm{out}}(g_{m^*}^{-}) \leq E_{\mathrm{val}}(g_{m^*}^{-}) + O\left(\sqrt{\frac{\ln M}{2K}}\right)$$

(continued on next page)

Suppose that instead, you had no control over the validation process. So M learners, each with their own models present you with the results of their val idation processes on *different* validation sets. Here is what you know about each learner:

> Each learner m reports to you the size of their validation set K_m, and the validation error $E_{\text{val}}(m)$. The learners may have used different data sets, except that they faithfully learned on a training set and validated on a held out validation set which was *only* used for validation purposes.

As the model selector, you have to decide which learner to go with.

(a) Should you select the learner with minimum validation error? If yes, why? If no, why not? *[Hint: think VC-bound.]*

(b) If all models are validated on the same validation set as described in the text, why is it okay to select the learner with the lowest validation error?

(c) After selecting learner m^* (say), show that

$$\mathbb{P}[E_{\text{out}}(m^*) > E_{\text{val}}(m^*) + \epsilon] \leq M e^{-2\epsilon^2 \kappa(\epsilon)},$$

where $\kappa(\epsilon) = -\frac{1}{2\epsilon^2} \ln\left(\frac{1}{M}\sum_{m=1}^{M} e^{-2\epsilon^2 K_m}\right)$ is an "average" validation set size.

(d) Show that with probability at least $1-\delta$, $E_{\text{out}} \leq E_{\text{val}} + \epsilon^*$, for any ϵ^* which satisfies $\epsilon^* \geq \sqrt{\frac{\ln(M/\delta)}{2\kappa(\epsilon^*)}}$.

(e) Show that $\min_m K_m \leq \kappa(\epsilon) \leq \frac{1}{M}\sum_{m=1}^{M} K_m$. Is this bound better or worse than the bound when all models use the same validation set size (equal to the average validation set size $\frac{1}{M}\sum_{m=1}^{M} K_m$)?

Problem 4.26 In this problem, derive the formula for the exact expression for the leave-one out cross validation error for linear regression. Let Z be the data matrix whose rows correspond to the transformed data points $\mathbf{z}_n = \Phi(\mathbf{x}_n)$.

(a) Show that:

$$\mathrm{Z}^{\mathrm{T}}\mathrm{Z} = \sum_{n=1}^{N} \mathbf{z}_n \mathbf{z}_n^{\mathrm{T}}; \qquad \mathrm{Z}^{\mathrm{T}}\mathbf{y} = \sum_{n=1}^{N} \mathbf{z}_n y_n; \qquad \mathrm{H}_{nm}(\lambda) = \mathbf{z}_n^{\mathrm{T}} \mathrm{A}^{-1}(\lambda) \mathbf{z}_m,$$

where $\mathrm{A} = \mathrm{A}(\lambda) = \mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda \Gamma^{\mathrm{T}}\Gamma$ and $\mathrm{H}(\lambda) = \mathrm{Z}\mathrm{A}(\lambda)^{-1}\mathrm{Z}^{\mathrm{T}}$. Hence, show that when $(\mathbf{z}_n, y_n)$ is left out, $\mathrm{Z}^{\mathrm{T}}\mathrm{Z} \to \mathrm{Z}^{\mathrm{T}}\mathrm{Z} - \mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}$, and $\mathrm{Z}^{\mathrm{T}}\mathbf{y} \to \mathrm{Z}^{\mathrm{T}}\mathbf{y} - \mathbf{z}_n y_n$.

(b) Compute $\mathbf{w}_n^-$, the weight vector learned when the nth data point is left out, and show that:

$$\mathbf{w}_n^- = \left(\mathrm{A}^{-1} + \frac{\mathrm{A}^{-1}\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}\mathrm{A}^{-1}}{1 - \mathbf{z}_n^{\mathrm{T}}\mathrm{A}^{-1}\mathbf{z}_n}\right)(\mathrm{Z}^{\mathrm{T}}\mathbf{y} - \mathbf{z}_n y_n).$$

[Hint: use the identity $(\mathrm{A} - \mathbf{x}\mathbf{x}^{\mathrm{T}})^{-1} = \mathrm{A}^{-1} + \frac{\mathrm{A}^{-1}\mathbf{x}\mathbf{x}^{\mathrm{T}}\mathrm{A}^{-1}}{1-\mathbf{x}^{\mathrm{T}}\mathrm{A}^{-1}\mathbf{x}}$*.]*

(c) Using (a) and (b), show that $\mathbf{w}_n^- = \mathbf{w} + \dfrac{\hat{y}_n - y_n}{1 - \mathrm{H}_{nn}}\mathrm{A}^{-1}\mathbf{z}_n$, where $\mathbf{w}$ is the regression weight vector using all the data.

(d) The prediction on the validation point is given by $\mathbf{z}_n^{\mathrm{T}}\mathbf{w}_n^-$. Show that

$$\mathbf{z}_n^{\mathrm{T}}\mathbf{w}_n = \frac{\hat{y}_n - \mathrm{H}_{nn}y_n}{1 - \mathrm{H}_{nn}}.$$

(e) Show that $\mathrm{e}_n = \left(\dfrac{\hat{y}_n - y_n}{1 - \mathrm{H}_{nn}}\right)^2$, and hence prove Equation (4.13).

Problem 4.27 Cross validation gives an accurate estimate of $\bar{E}_{\text{out}}(N-1)$, but it can be quite sensitive, leading to problems in model selection. A common heuristic for 'regularizing' cross validation is to use a measure of error $\sigma_{\text{CV}}(\mathcal{H})$ for the cross validation estimate in model selection.

(a) One choice for σ_{CV} is the standard deviation of the leave-one-out errors divided by $\sqrt{N}$, $\sigma_{\text{CV}} \approx \frac{1}{\sqrt{N}}\sqrt{\text{var}(e_1, \ldots, e_n)}$. Why divide by $\sqrt{N}$?

(b) For linear models, show that $\sqrt{N}\sigma_{\text{CV}} = \frac{1}{N}\sum_{n=1}^{N}\left(\frac{\hat{y}_n - y_n}{1-\mathrm{H}_{nn}}\right)^4 \quad E_{\text{CV}}^2$.

(c) (i) Given the best model $\mathcal{H}^*$, the conservative one-sigma approach se lects the simplest model within $\sigma_{\text{CV}}(\mathcal{H}^*)$ of the best.

(ii) The bound minimizing approach selects the model which minimizes $E_{\text{CV}}(\mathcal{H}) + \sigma_{\text{CV}}(\mathcal{H})$.

Use the experimental design in Problem 4.4 to compare these approaches with the 'unregularized' cross validation estimate as follows. Fix $Q_f = 15$, $Q = 20$, and $\sigma = 1$. Use each of the two methods proposed here as well as traditional cross validation to select the optimal value of the regularization parameter λ in the range $\{0.05, 0.10, 0.15, \ldots, 5\}$ using weight decay regularization, $\Omega(\mathbf{w}) = \frac{\lambda}{N}\mathbf{w}^{\mathrm{T}}\mathbf{w}$. Plot the resulting out-of-sample error for the model selected using each method as a function of N, with N in the range $\{2 \times Q, 3 \times Q, \ldots, 10 \times Q\}$.

What are your conclusions?

Chapter 5

Three Learning Principles

The study of learning from data highlights some general principles that are fascinating concepts in their own right. Having gone through the mathematical analysis and empirical illustrations of the first few chapters, we have a good foundation from which to articulate some of these principles and explain them in concrete terms.

In this chapter, we will discuss three principles. The first one is related to the choice of model and is called Occam's razor. The other two are related to data; sampling bias establishes an important principle about obtaining the data, and data snooping establishes an important principle about handling the data. A genuine understanding of these principles will protect you from the most common pitfalls in learning from data, and allow you to interpret generalization performance properly.

5.1 Occam's Razor

Although it is not an exact quote of Einstein's, it is often attributed to him that "An explanation of the data should be made *as simple as possible, but no simpler*." A similar principle, *Occam's Razor*, dates from the 14th century and is attributed to William of Occam, where the 'razor' is meant to trim down the explanation to the bare minimum that is consistent with the data.

In the context of learning, the penalty for model complexity which was introduced in Section 2.2 is a manifestation of Occam's razor. If $E_{\text{in}}(g) \quad 0$, then the explanation (hypothesis) is consistent with the data. In this case, the most plausible explanation, with the lowest estimate of E_{out} given in the VC bound (2.14), happens when the complexity of the explanation (measured by $d_{\text{vc}}(\mathcal{H})$) is as small as possible. Here is a statement of the underlying principle.

The simplest model that fits the data is also the most plausible.

Applying this principle, we should choose as simple a model as we think we can get away with. Although the principle that simpler is better may be intuitive, it is neither precise nor self-evident. When we apply the principle to learning from data, there are two basic questions to be asked.

1. What does it mean for a model to be simple?
2. How do we know that simpler is better?

Let's start with the first question. There are two distinct approaches to defining the notion of complexity, one based on a family of objects and the other based on an individual object. We have already seen both approaches in our analysis. The VC dimension in Chapter 2 is a measure of complexity, and it is based on the hypothesis set $\mathcal{H}$ as a whole, i.e., based on a family of objects. The regularization term of the augmented error in Chapter 4 is also a measure of complexity, but in this case it is the complexity of an individual object, namely the hypothesis h.

The two approaches to defining complexity are not encountered only in learning from data; they are a recurring theme whenever complexity is discussed. For instance, in information theory, *entropy* is a measure of complexity based on a family of objects, while *minimum description length* is a related measure based on individual objects. There is a reason why this is a recurring theme. The two approaches to defining complexity are in fact related.

When we say a family of objects is complex, we mean that the family is 'big'. That is, it contains a large variety of objects. Therefore, each individual object in the family is *one of many.* By contrast, a simple family of objects is 'small'; it has relatively few objects, and each individual object is *one of few.*

Why is the sheer number of objects an indication of the level of complexity? The reason is that both the number of objects in a family and the complexity of an object are related to how many parameters are needed to specify the object. When you increase the number of parameters in a learning model, you simultaneously increase how diverse $\mathcal{H}$ is and how complex the individual h is. For example, consider 17th order polynomials versus 3rd order polynomials. There is more variety in 17th order polynomials, and at the same time the individual 17th order polynomial is more complex than a 3rd order polynomial.

The most common definitions of object complexity are based on the number of bits needed to describe an object. Under such definitions, an object is simple if it has a short description. Therefore, a simple object is not only intrinsically simple (as it can be described succinctly), but it also has to be one of few, since there are fewer objects that have short descriptions than there are that have long descriptions, as a matter of simple counting.

Exercise 5.1

Consider hypothesis sets $\mathcal{H}_1$ and $\mathcal{H}_{100}$ that contain Boolean functions on 10 Boolean variables, so $\mathcal{X} = \{-1, +1\}^{10}$. $\mathcal{H}_1$ contains all Boolean functions

which evaluate to $+1$ on exactly one input point, and to -1 elsewhere; $\mathcal{H}_{100}$ contains all Boolean functions which evaluate to $+1$ on exactly 100 input points, and to 1 elsewhere.

(a) How big (number of hypotheses) are $\mathcal{H}_1$ and $\mathcal{H}_{100}$?

(b) How many bits are needed to specify one of the hypotheses in $\mathcal{H}_1$?

(c) How many bits are needed to specify one of the hypotheses in $\mathcal{H}_{100}$?

We now address the second question. When Occam's razor says that simpler is better, it doesn't mean simpler is more elegant. It means simpler has a better chance of being right. Occam's razor is about performance, not about aesthetics. If a complex explanation of the data performs better, we will take it.

The argument that simpler has a better chance of being right goes as follows. We are trying to fit a hypothesis to our data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)\}$ (assume y_n's are binary). There are fewer simple hypotheses than there are complex ones. With complex hypotheses, there would be enough of them to shatter $\mathbf{x}_1, \cdots, \mathbf{x}_N$, so it is certain that we can fit the data set regardless of what the labels $y_1, \cdots, y_N$ are, even if these are completely random. Therefore, fitting the data does not mean much. If, instead, we have a simple model with few hypotheses and we still found one that perfectly fits the dichotomy $\mathcal{D} = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)\}$, this is surprising, and therefore it means something.

Occam's Razor has been formally proved under different sets of idealized conditions. The above argument captures the essence of these proofs; if something is less likely to happen, then when it does happen it is more significant. Let us look at an example.

Example 5.1. Suppose that one constructs a physical theory about the resistivity of a metal under various temperatures. In this theory, aside from some constants that need to be determined, the resistivity ρ has a linear dependence on the temperature T. In order to verify that the theory is correct and to obtain the unknown constants, 3 scientists conduct the following three experiments and present their data to you.

Scientist 1 Scientist 2 Scientist 3

It is clear that Scientist 3 has produced the most convincing evidence for the theory. If the measurements are exact, then, Scientist 2 has managed to falsify the theory and we are back to the drawing board. What about Scientist 1? While he has not falsified the theory, has he provided any evidence for it? The answer is no, for we can reverse the question. Suppose that the theory was not correct, what could the data have done to prove him wrong? Nothing, since any two points can be joined by a line. Therefore, the model is not just likely to fit the data in this case, it is certain to do so. This renders the fit totally insignificant when it does happen. □

This example illustrates a concept related to Occam's Razor, which is the *axiom of non-falsifiability.* The axiom asserts that the data should have some chance of falsifying a hypothesis, if we are to conclude that it can provide evidence for the hypothesis. One way to guarantee that every data set has some chance at falsification is for the VC dimension of the hypothesis set to be less than N, the number of data points. This is discussed further in Problem 5.1. Here is another example of the same concept.

Example 5.2. Financial firms try to pick good traders (predictors of whether the market will go up or not). Suppose that each trader is tested on their prediction (up or down) over the next 5 days and those who perform well will be hired. One might think that this process should produce better and better traders on Wall Street. Viewed as a learning problem, consider each trader to be a prediction hypothesis. Suppose that the hiring pool is 'complex'; we are interviewing 2^5 traders who happen to be a diverse set of people such that their predictions over the next 5 days are all different. Necessarily one of these traders gets it all correct, and will be hired. Hiring the trader through this process may or may not be a good thing, since the process will pick someone even if the traders are just flipping coins to make their predictions. A perfect predictor always exists in this group, so finding one doesn't mean much. If we were interviewing only two traders, and one of them made perfect predictions, that would mean something. □

Exercise 5.2

Suppose that for 5 weeks in a row, a letter arrives in the mail that predicts the outcome of the upcoming Monday night football game. You keenly watch each Monday and to your surprise, the prediction is correct each time. On the day after the fifth game, a letter arrives, stating that if you wish to see next week's prediction, a payment of $50.00 is required. Should you pay?

(a) How many possible predictions of win-lose are there for 5 games?

(b) If the sender wants to make sure that at least one person receives correct predictions on all 5 games from him, how many people should he target to begin with?

(c) After the first letter 'predicting' the outcome of the first game, how many of the original recipients does he target with the second letter?

(d) How many letters altogether will have been sent at the end of the 5 weeks?

(e) If the cost of printing and mailing out each letter is $0.50, how much would the sender make if the recipient of 5 correct predictions sent in the $50.00?

(f) Can you relate this situation to the growth function and the credibility of fitting the data?

Learning from data takes Occam's Razor to another level, going beyond "as simple as possible, but no simpler." Indeed, we may opt for 'a simpler fit than possible', namely an imperfect fit of the data using a simple model over a perfect fit using a more complex one. The reason is that the price we pay for a perfect fit in terms of the penalty for model complexity in (2.14) may be too much in comparison to the benefit of the better fit. This idea was illustrated in Figure 3.7, and is a manifestation of overfitting. The idea is also the rationale behind the recommended policy in Chapter 3: *first* try a linear model one of the simplest models in the arena of learning from data.

5.2 Sampling Bias

A vivid example of sampling bias happened in the 1948 US presidential election between Truman and Dewey. On election night, a major newspaper carried out a telephone poll to ask people how they voted. The poll indicated that Dewey won, and the paper was so confident about the small error bar in its poll that it declared Dewey the winner in its headline. When the actual votes were counted, Dewey lost to the delight of a smiling Truman.

©Associated Press

This was not a case of statistical anomaly, where the newspaper was just incredibly unlucky (remember the δ in the VC bound?). It was a case where the sample was doomed from the get-go, regardless of its size. Even if the experiment were repeated, the result would be the same. In 1948, telephones were expensive and those who had them tended to be in an elite group that favored Dewey much more than the average voter did. Since the newspaper did its poll by telephone, it inadvertently used an in-sample distribution that was different from the out-of-sample distribution. That is what sampling bias is.

If the data is sampled in a biased way, learning will produce a similarly biased outcome.

Applying this principle, we should make sure that the training and testing distributions are the same; if not, our results may be invalid, or, at the very least, require careful interpretation.

If you recall, the VC analysis made very few assumptions, but one assumption it did make was that the data set $\mathcal{D}$ is generated from the same distribution that the final hypothesis g is tested on. In practice, we may encounter data sets that were not generated under those ideal conditions. There are some techniques in statistics and in learning to compensate for the 'mismatch' between training and testing, but not in cases where $\mathcal{D}$ was generated with the exclusion of certain parts of the input space, such as the exclusion of households with no telephones in the above example. There is nothing that can be done when this happens, other than to admit that the result will not be reliable statistical bounds like Hoeffding and VC require a match between the training and testing distributions.

There are many examples of how sampling bias can be introduced in data collection. In some cases it is inadvertently introduced by an oversight, as in the case of Dewey and Truman. In other cases, it is introduced because certain types of data are not available. For instance, in our credit example of Chapter 1, the bank created the training set from the database of previous customers and how they performed for the bank. Such a set necessarily excludes those who applied to the bank for credit cards and were rejected, because the bank does not have data on how they *would have performed* if they were accepted. Since future applicants will come from a mixed population including some who would have been rejected in the past, the 'test set' comes from a different distribution than the training set, and we have a case of sampling bias. In this particular case, if no data on the applicants that were rejected is available, nothing much can be done other than to acknowledge that there is a bias in the final predictor that learning will produce, since a representative training set is just not available.

Exercise 5.3

In an experiment to determine the distribution of sizes of fish in a lake, a net might be used to catch a representative sample of fish. The sample is

then analyzed to find out the fractions of fish of different sizes. If the sample is big enough, statistical conclusions may be drawn about the actual distribution in the entire lake. Can you smell ☺ sampling bias?

There are other cases, arguably more common, where sampling bias is introduced by human intervention. It is not that uncommon for someone to throw away training examples they don't like! A Wall Street firm who wants to develop an automated trading system might choose data sets when the market was 'behaving well' to train the system, with the semi-legitimate justification that they don't want the noise to complicate the training process. They will surely achieve that if they get rid of the 'bad' examples, but they will create a system that can be trusted only in the periods when the market does behave well! What happens when the market is not behaving well is anybody's guess. In general, throwing away training examples based on their values, e.g., examples that look like outliers or don't conform to our preconceived ideas, is a fairly common sampling bias trap.

Other biases. Sampling bias has also been called selection bias in the statistics community. We will stick with the more descriptive term sampling bias for two reasons. First, the bias arises in how the data was *sampled*; second, it is less ambiguous because in the learning context, there is another notion of selection bias drifting around – *selection* of a final hypothesis from the learning model based on the data. The performance of the selected hypothesis on the data is optimistically biased, and this could be denoted as a selection bias. We have referred to this type of bias simply as bad generalization.

There are various other biases that have similar flavor. There is even a special type of bias for the research community, called publication bias! This refers to the bias in published scientific results because negative results are often not published in the literature, whereas positive results are. The common theme of all of these biases is that they render the standard statistical conclusions invalid because the basic premise for such conclusions, that the sampling distribution is the same as the overall distribution, does not hold any more. In the field of learning from data, it is sampling bias in the training set that we need to worry about.

5.3 Data Snooping

Data snooping is the most common trap for practitioners in learning from data. The principle involved is simple enough,

If a data set has affected any step in the learning process, its ability to assess the outcome has been compromised.

Applying this principle, if you want an unbiased assessment of your learning performance, you should keep a test set in a vault and never use it for learning in any way. This is basically what we have been talking about all along in training versus testing, but it goes beyond that. Even if a data set has not been 'physically' used for training, it can still affect the learning process, sometimes in subtle ways.

Exercise 5.4

Consider the following approach to learning. By looking at the data, it appears that the data is linearly separable, so we go ahead and use a simple perceptron, and get a training error of zero after determining the optimal set of weights. We now wish to make some generalization conclusions, so we look up the d_{vc} for our learning model and see that it is $d+1$. Therefore, we use this value of d_{vc} to get a bound on the test error.

(a) What is the problem with this bound is it correct?

(b) Do we know the d_{vc} for the learning model that we actually used? It is this d_{vc} that we need to use in the bound.

To avoid the pitfall in the above exercise, it is extremely important that you choose your learning model *before* seeing any of the data. The choice can be based on general information about the learning problem, such as the number of data points and prior knowledge regarding the input space and target function, but not on the actual data set $\mathcal{D}$. Failure to observe this rule will invalidate the VC bounds, and any generalization conclusions will be up in the air. Even a careful person can fall into the traps of data snooping. Consider the following example.

Example 5.3. An investment bank wants to develop a system for forecasting currency exchange rates. It has 8 years worth of historical data on the US Dollar (USD) versus the British Pound (GBP), so it tries to use the data to see if there is any pattern that can be exploited. The bank takes the series of daily changes in the USD/GBP rate, normalizes it to zero mean and unit variance, and starts to develop a system for forecasting the direction of the change. For each day, it tries to predict that direction based on the fluctuations in the previous 20 days. 75% of the data is used for training, and the remaining 25% is set aside for testing the final hypothesis.

The test shows great success. The final hypothesis has a hit rate (percentage of time getting the direction right) of 52.1%. This may seem modest, but in the world of finance you can make a lot of money if you get that hit rate consistently. Indeed, over the 500 test days (2 years worth, as each year has about 250 trading days), the cumulative profit of the system is a respectable 22%.

When the system is used in live trading, the performance deteriorates significantly. In fact, it loses money. Why didn't the good test performance continue on the new data? In this case, there is a simple explanation and it has to do with data snooping. Although the bank was careful to set aside test points that were not used for training in order to properly evaluate the final hypothesis, the test data had in fact affected the training process in a subtle way. When the original series of daily changes was normalized to zero mean and unit variance, *all of the data* was involved in this step. Therefore, the test data that was extracted had already contributed to the choices made by the learning algorithm by contributing to the values of the mean and the variance that were used in normalization. Although this seems like a minor effect, it *is* data snooping. When you plot the cumulative profit on the test set with or without that snooping step, you see how snooping resulted in an over-optimistic expectation compared to the realistic expectation that avoids snooping.

It is not the normalization that was a bad idea. It is the involvement of test data in that normalization, which contaminated this data and rendered its estimate of the final performance inaccurate. □

One of the most common occurrences of data snooping is the reuse of the same data set. If you try learning using first one model and then another and then another on the same data set, you will eventually 'succeed'. As the saying goes, if you torture the data long enough, it will confess ☺. If you try all possible dichotomies, you will eventually fit any data set; this is true whether we try the dichotomies directly (using a single model) or indirectly (using a sequence of models). The effective VC dimension for the series of trials will not be that of the last model that succeeded, but of the entire union of models that could have been used depending on the outcomes of different trials.

Sometimes the reuse of the same data set is carried out by different people. Let's say that there is a public data set that you would like to work on. Before you download the data, you read about how other people did with this data set

using different techniques. You naturally pick the most promising techniques as a baseline, then try to improve on them and introduce your own ideas. Although you haven't even seen the data set yet, you are already guilty of data snooping. Your choice of baseline techniques was affected by the data set, through the actions of others. You may find that your estimates of the performance will turn out to be too optimistic, since the techniques you are using have already proven well-suited to *this particular* data set.

To quantify the damage done by data snooping, one has to assess the penalty for model complexity in (2.14) taking the snooping into consideration. In the public data set case, the effective VC dimension corresponds to a much bigger hypothesis set than the $\mathcal{H}$ that your learning algorithm uses. It covers all hypotheses that were considered (and mostly rejected) by everybody else in the process of coming up with the solutions that they published and that you used as your baseline. This is a potentially huge set with very high VC dimension, hence the generalization guarantees in (2.14) will be much worse than without data snooping.

Not all data sets subjected to data snooping are equally 'contaminated'. The bounds in (1.6) in the case of a choice between a finite number of hypotheses, and in (2.12) in the case of an infinite number, provide guidelines for the level of contamination. The more elaborate the choice made based on a data set, the more contaminated the set becomes and the less reliable it will be in gauging the performance of the final hypothesis.

Exercise 5.5

Assume we set aside 100 examples from $\mathcal{D}$ that will not be used in training, but will be used to select one of three final hypotheses g_1, g_2, g_3 produced by three different learning algorithms that train on the rest on the data. Each algorithm works with a different $\mathcal{H}$ of size 500. We would like to characterize the accuracy of estimating $E_{\text{out}}(g)$ on the selected final hypothesis if we use the same 100 examples to make that estimate.

(a) What is the value of M that should be used in (1.6) in this situation?

(b) How does the level of contamination of these 100 examples compare to the case where they would be used in training rather than in the final selection?

In order to deal with data snooping, there are basically two approaches.

1. Avoid data snooping: A strict discipline in handling the data is required. Data that is going to be used to evaluate the final performance should be 'locked in a safe' and only brought out after the final hypothesis has been decided. If intermediate tests are needed, separate data sets should be used for that. Once a data set has been used, it should be treated as contaminated as far as testing the performance is concerned.

2. Account for data snooping: If you have to use a data set more than once, keep track of the level of contamination and treat the reliability of

your performance estimates in light of this contamination. The bounds (1.6) and (2.12) can provide guidelines for the relative reliability of different data sets that have been used in different roles within the learning process.

Data snooping versus sampling bias. Sampling bias was defined based on how the data was obtained before any learning; data snooping was defined based on how the data affected the learning, in particular how the learning model is selected. These are obviously different concepts. However, there are cases where sampling bias occurs as a consequence of 'snooping' looking at data that you are not supposed to look at. Here is an example.

Consider predicting the performance of different stocks based on historical data. In order to see if a prediction rule is any good, you take all currently traded companies and test the rule on their stock data over the past 50 years. Let us say that you are testing the "buy and hold" strategy, where you would have bought the stock 50 years ago and kept it until now. If you test this 'hypothesis', you will get excellent performance in terms of profit. Well, don't get too excited! You inadvertently biased the results in your favor by picking only *currently traded companies*, which means that the companies that did not make it are not part of your evaluation. When you put your prediction rule to work, it will be used on all companies whether they will survive or not, since you cannot identify which companies today will be the 'currently traded' companies 50 years from now. This is a typical case of sampling bias, since the problem is that the training data is not representative of the test data. However, if we trace the origin of the bias, we did 'snoop' in this case by looking at future data of companies to determine which of these companies to use in our training. Since we are using information in training that we would not have access to in real trading, this is viewed as a form of data snooping.

5.4 Problems

Problem 5.1 The idea of *falsifiability* – that a claim can be rendered false by observed data – is an important principle in experimental science.

> **Axiom of Non-Falsifiability.** *If the outcome of an experiment has no chance of falsifying a particular proposition, then the result of that experiment does not provide evidence one way or another toward the truth of the proposition.*

Consider the proposition "There is $h \in \mathcal{H}$ that approximates f as would be evidenced by finding such an h with in sample error zero on $\mathbf{x}_1, \cdots, \mathbf{x}_N$." We say that the proposition is falsified if no hypothesis in $\mathcal{H}$ can fit the data perfectly.

(a) Suppose that $\mathcal{H}$ shatters $\mathbf{x}_1, \cdots, \mathbf{x}_N$. Show that this proposition is not falsifiable *for any* f.

(b) Suppose that f is random ($f(\mathbf{x}) = \pm 1$ with probability $\frac{1}{2}$, independently on every $\mathbf{x}$), so $E_{\text{out}}(h) = \frac{1}{2}$ for every $h \in \mathcal{H}$. Show that

$$\mathbb{P}[\text{falsification}] \geq 1 - \frac{m_{\mathcal{H}}(N)}{2^N} .$$

(c) Suppose $d_{\text{vc}} = 10$ and $N = 100$. If you obtain a hypothesis h with zero E_{in} on your data, what can you 'conclude' from the result in part (b)?

Problem 5.2 Structural Risk Minimization (SRM) is a useful framework for model selection that is related to Occam's Razor. Define a *structure* – a nested sequence of hypothesis sets:

The SRM framework picks a hypothesis from each $\mathcal{H}_i$ by minimizing E_{in}. That is, $g_i = \underset{h \in \mathcal{H}_i}{\text{argmin}}\, E_{\text{in}}(h)$. Then, the framework selects the final hypothesis by minimizing E_{in} *and* the model complexity penalty Ω. That is, $g^* = \underset{i=1,2,\cdots}{\text{argmin}}(E_{\text{in}}(g_i) + \Omega(\mathcal{H}_i))$. Note that $\Omega(\mathcal{H}_i)$ should be non decreasing in i because of the nested structure.

(a) Show that the in sample error $E_{\text{in}}(g_i)$ is non increasing in i.

(b) Assume that the framework finds $g^* \in \mathcal{H}_i$ with probability p_i. How does p_i relate to the complexity of the target function?

(c) Argue that the p_i's are unknown but $p_0 \leq p_1 \leq p_2 \leq \cdots \leq 1$.

(d) Suppose $g^* = g_i$. Show that

$$\mathbb{P}\left[|E_{\text{in}}(g_i) - E_{\text{out}}(g_i)| > \epsilon \mid g^* = g_i\right] \leq \frac{1}{p_i} \cdot 4m_{\mathcal{H}_i}(2N)e^{-\epsilon^2 N/8}.$$

Here, the conditioning is on selecting g_i as the final hypothesis by SRM. *[Hint: Use the Bayes theorem to decompose the probability and then apply the VC bound on one of the terms]*

You may interpret this result as follows: if you use SRM and end up with g_i, then the generalization bound is a factor $\frac{1}{p_i}$ worse than the bound you would have gotten had you simply started with $\mathcal{H}_i$.

Problem 5.3 In our credit card example, the bank starts with some vague idea of what constitutes a good credit risk. So, as customers $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ arrive, the bank applies its vague idea to approve credit cards for some of these customers. Then, only those who got credit cards are monitored to see if they default or not.

For simplicity, suppose that the first N customers were given credit cards. Now that the bank knows the behavior of these customers, it comes to you to improve their algorithm for approving credit. The bank gives you the data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$.

Before you look at the data, you do mathematical derivations and come up with a credit approval function. You now test it on the data and, to your delight, obtain perfect prediction.

(a) What is M, the size of your hypothesis set?

(b) With such an M, what does the Hoeffding bound say about the probability that the true performance is worse than 2% error for $N = 10000$?

(c) You give your g to the bank and assure them that the performance will be better than 2% error and your confidence is given by your answer to part (b). The bank is thrilled and uses your g to approve credit for new clients. To their dismay, more than half their credit cards are being defaulted on. Explain the possible reason(s) behind this outcome.

(d) Is there a way in which the bank could use your credit approval function to have your probabilistic guarantee? How? *[Hint: The answer is yes!]*

Problem 5.4 The S&P 500 is a set of the largest 500 companies currently trading. Suppose there are $10,000$ stocks currently trading, and there have been $50,000$ stocks which have ever traded over the last 50 years (some of these have gone bankrupt and stopped trading). We wish to evaluate the profitability of various 'buy and hold' strategies using these 50 years of data (roughly $12,500$ trading days).

Since it is not easy to get stock data, we will confine our analysis to today's S&P 500 stocks, for which the data is readily available.

(a) A stock is profitable if it went up on more than 50% of the days. Of your S&P stocks, the most profitable went up on 52% of the days ($E_{\text{in}} = 0.48$).

(i) Since we picked the best among 500, using the Hoeffding bound,

$$\mathbb{P}[|E_{\text{in}} - E_{\text{out}}| > 0.02] \leq 2 \times 500 \times e^{-2\times 12500\times 0.02^2} \approx 0.045.$$

There is a greater than 95% chance this stock is profitable. Where did we go wrong?

(ii) Give a better estimate for the probability that this stock is profitable. *[Hint: What should the correct M be in the Hoeffding bound?]*

(b) We wish to evaluate the profitability of 'buy and hold' for general stock trading. We notice that all of our 500 S&P stocks went up on at least 51% of the days.

(i) We conclude that buying and holding a stocks is a good strategy for general stock trading. Where did we go wrong?

(ii) Can we say *anything* about the performance of buy and hold trading?

Problem 5.5 You think that the stock market exhibits reversal, so if the price of a stock sharply drops you expect it to rise shortly thereafter. If it sharply rises, you expect it to drop shortly thereafter.

To test this hypothesis, you build a trading strategy that buys when the stocks go down and sells in the opposite case. You collect historical data on the current S&P 500 stocks, and your hypothesis gave a good annual return of 12%.

(a) When you trade using this system, do you expect it to perform at this level? Why or why not?

(b) How can you test your strategy so that its performance in sample is more reflective of what you should expect in reality?

Problem 5.6 One often hears "Extrapolation is harder than interpolation." Give a possible explanation for this phenomenon using the principles in this chapter. *[Hint: training distribution versus testing distribution.]*

Chapter 6

Similarity-Based Methods

"It's a *manohorse*", exclaimed the confident little 5 year old boy. We call it the Centaur out of habit, but who can fault the kid's intuition? The 5 year old has never seen this thing before now, yet he came up with a reasonable classification for the beast. He is using the simplest method of learning that we know of – *similarity* – and yet it's effective: the child searches through his history for similar objects (in this case a man and a horse) and builds a classification based on these similar objects.

The method is simple and intuitive, yet when we get into the details, several issues need to be addressed in order to arrive at a technique that is quantitative and fit for a computer. The goal of this chapter is to build exactly such a quantitative framework for similarity based learning.

6.1 Similarity

The 'manohorse' is interesting because it requires a deep understanding of similarity: first, to say that the Centaur is similar to both man and horse; and, second, to decide that there is enough similarity to both objects so that neither can be excluded, warranting a new class. A good measure of similarity allows us to not only classify objects using similar objects, but also detect the arrival of a new class of objects (novelty detection).

A simple classification rule is to give a new input the class of the most similar input in your data. This is the 'nearest neighbor' rule. To implement the nearest neighbor rule, we need to first quantify the similarity between two objects. There are different ways to measure similarity, or equivalently dissimilarity. Consider the following example with 3 digits.

The two 9s should be regarded as very similar. Yet, if we naively measure similarity by the number of black pixels in common, the two 9s have only two in common. On the other hand, the 6 has many more black pixels in common with either 9, even though the 6 should be regarded as dissimilar to both 9s. Before measuring the similarity, one should preprocess the inputs, for example by centering, axis aligning and normalizing the size in the case of an image. One can go further and extract the relevant *features* of the data, for example size (number of black pixels) and symmetry as was done in Chapter 3. These practical considerations regarding the nature of the learning task, though important, are not our primary focus here. We will assume that through domain expertise or otherwise, features have been constructed to identify the important dimensions, and if two inputs differ in these dimensions, then the inputs are likely to be dissimilar. Given this assumption, there are well established ways to measure similarity (or dissimilarity) in different contexts.

6.1.1 Similarity Measures

For inputs $\mathbf{x}, \mathbf{x}'$ which are vectors in $\mathbb{R}^d$, we can measure dissimilarity using the standard Euclidean distance,

$$d(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|.$$

The smaller the distance, the more similar are the objects corresponding to inputs $\mathbf{x}$ and $\mathbf{x}'$.[1] For Boolean features the Euclidean distance is the square root of the well known *Hamming distance*. The Euclidean distance is a special case of a more general distance measure which can be defined for an arbitrary positive semi-definite matrix Q:

$$d(\mathbf{x}, \mathbf{x}') = (\mathbf{x} - \mathbf{x}')^{\mathrm{T}} \mathrm{Q} (\mathbf{x} - \mathbf{x}').$$

A useful special case, known as the *Mahalanobis distance* is to set $\mathrm{Q} = \Sigma^{-1}$, where Σ is the covariance matrix [2] of the data. The Mahalanobis distance metric depends on the data set. The main advantage of the Mahalanobis distance over the standard Euclidean distance is that it takes into account correlations among the data dimensions and scale. A similar effect can be

[1] An axiomatic treatment of similarity that goes beyond metric-based similarity appeared in "Features of Similarity", A. Tversky, *Psychological Review*, 84(4), pp 327–352, 1977.

[2] $\Sigma = \dfrac{1}{N} \displaystyle\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^{\mathrm{T}} - \bar{\mathbf{x}}\bar{\mathbf{x}}^{\mathrm{T}}$, where $\bar{\mathbf{x}} = \dfrac{1}{N} \displaystyle\sum_{i=1}^{N} \mathbf{x}_i$.

accomplished by first using input preprocessing to standardize the data (see Chapter **??**) and then using the standard Euclidean distance. Another useful measure, especially for Boolean vectors, is the *cosine similarity*,

$$\text{CosSim}(\mathbf{x}, \mathbf{x}') = \frac{\mathbf{x} \bullet \mathbf{x}'}{\|\mathbf{x}\|\|\mathbf{x}'\|}.$$

The cosine similarity is the cosine of the angle between the two vectors, $\text{CosSim} \in [-1, 1]$, and larger values indicate greater similarity. When the objects represent sets, then the set similarity or *Jaccard coefficient* is often used. For example, consider two movies which have been watched by two different sets of users S_1, S_2. We may measure how similar these movies are by how similar the two sets S_1 and S_2 are:

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|};$$

$1 - J(S_1, S_2)$ can be used as a measure of distance which conveniently has the properties that a metric formally satisfies, such as the triangle inequality. We focus on the Euclidean distance which is also a metric; many of the algorithms, however, apply to arbitrary similarity measures.

Exercise 6.1

(a) Give two vectors with very high cosine similarity but very low Euclidean distance similarity. Similarly, give two vectors with very low cosine similarity but very high Euclidean distance similarity.

(b) If the origin of the coordinate system changes, which measure of similarity changes? How will this affect your choice of features?

6.2 Nearest Neighbor

Simple rules survive; and, the nearest neighbor technique is perhaps the simplest of all. We will summarize the entire algorithm in a short paragraph. But, before we forge ahead, let's not forget the two basic competing principles laid out in the first five chapters: any learning technique should be expressive enough that we can fit the data and obtain low E_{in}; however, it should be reliable enough that a low E_{in} implies a low E_{out}.

The *nearest neighbor rule* is embarrassingly simple. There is no training phase (or no 'learning' so to speak). The entire algorithm is specified by how one computes the final hypothesis $g(\mathbf{x})$ on a test input $\mathbf{x}$. Recall that the data set is $\mathcal{D} = (\mathbf{x}_1, y_1) \ldots (\mathbf{x}_N, y_N)$, where $y_n = \pm 1$. To classify the test point $\mathbf{x}$, find the nearest point to $\mathbf{x}$ in the data set (the nearest neighbor), and use the classification of this nearest neighbor.

Formally speaking, reorder the data according to distance from $\mathbf{x}$ (breaking ties using the data point's index for simplicity). We write $(\mathbf{x}_{[n]}(\mathbf{x}), y_{[n]}(\mathbf{x}))$

for the nth such reordered data point with respect to $\mathbf{x}$. We will drop the dependence on $\mathbf{x}$ and simply write $(\mathbf{x}_{[n]}, y_{[n]})$ when the context is clear. So,

$$d(\mathbf{x}, \mathbf{x}_{[1]}) \leq d(\mathbf{x}, \mathbf{x}_{[2]}) \leq \cdots \leq d(\mathbf{x}, \mathbf{x}_{[N]})$$

The final hypothesis is

$$g(\mathbf{x}) = y_{[1]}(\mathbf{x})$$

($\mathbf{x}$ is classified by just looking at the label of the nearest data point to $\mathbf{x}$). This simple nearest neighbor rule admits a nice geometric interpretation, shown for two dimensions in Figure 6.1.

Figure 6.1: Nearest neighbor Voronoi tessellation.

The shading illustrates the final hypothesis $g(\mathbf{x})$. Each data point $\mathbf{x}_n$ 'owns' a region defined by the points closer to $\mathbf{x}_n$ than to any other data point. These regions are convex polytopes (convex regions whose faces are hyperplanes), some of which are unbounded; in two dimensions, we get convex polygons. The resulting set of regions defined by such a set of points is called a *Voronoi (or Dirichlet) tessellation* of the space. The final hypothesis g is a Voronoi tessellation with each region inheriting the class of the data point that owns the region. Figure 6.2(a) further illustrates the nearest neighbor classifier for a sample of 500 data points from the digits data described in Chapter 3.

The clear advantage of the nearest neighbor rule is that it is simple and intuitive, easy to implement and there is no training. It is expressive, as it achieves zero in-sample error (as can immediately be deduced from Figure 6.1), and as we will soon see, it is reliable. A practically important cosmetic upside, when presenting to a client for example, is that the classification of a test object is easy to 'explain': just present the similar object on which the classification is based. The main disadvantage is the computational overhead.

VC dimension. The nearest neighbor rule fits within our standard supervised learning framework with a very large hypothesis set. Any set of n labeled points induces a Voronoi tessellation with each Voronoi region assigned to a class; thus any set of n labeled points defines a hypothesis. Let $\mathcal{H}_n$ be the hypothesis set containing all hypotheses which result from some labeled Voronoi tessellation on n points. Let $\mathcal{H} = \cup_{n=1}^{\infty} \mathcal{H}_n$ be the union of all these hypothesis sets; this is the hypothesis set for the nearest neighbor rule. The learning algorithm picks the particular hypothesis from $\mathcal{H}$ which corresponds to the realized labeled data set; this hypothesis is in $\mathcal{H}_N \subset \mathcal{H}$. Since the training error is zero no matter what the size of the data set, the nearest neighbor rule is non-falsifiable [3] and the VC-dimension of this model is infinite. So, from the VC-worst-case analysis, this spells doom. A finite VC-dimension would have been great, and it would have given us one form of reliability, namely that E_{out} is close to E_{in} and so minimizing E_{in} works. The nearest neighbor method is reliable in another sense, and we are going to need some new tools if we are to argue the case.

6.2.1 Nearest Neighbor is 2-Optimal

Using a probabilistic argument, we will show that, under reasonable assumptions, the nearest neighbor rule has an out-of-sample error that is at most twice the *minimum possible out-of-sample error*. The success of the nearest neighbor algorithm relies on the nearby point $\mathbf{x}_{[1]}(\mathbf{x})$ having the same classification as the test point $\mathbf{x}$. This means two things: there is a 'nearby' point in the data set; and, the target function is reasonably smooth so that the classification of this nearest neighbor is indicative of the classification of the test point.

We model the target value, which is ± 1, as noisy and define

$$\pi(\mathbf{x}) = \mathbb{P}[y = +1|\mathbf{x}].$$

A data pair $(\mathbf{x}, y)$ is obtained by first generating $\mathbf{x}$ from the input probability distribution $P(\mathbf{x})$, and then y from the conditional distribution $\pi(\mathbf{x})$. We can relate $\pi(\mathbf{x})$ to a deterministic target function $f(\mathbf{x})$ by observing that if $\pi(\mathbf{x}) \geq \frac{1}{2}$, then the optimal prediction is $f(\mathbf{x}) = +1$; and, if $\pi(\mathbf{x}) < \frac{1}{2}$ then the optimal prediction is $f(\mathbf{x}) = -1$.[4]

[3] Recall that a hypothesis set is non-falsifiable if it can fit any data set.

[4] When $\pi(\mathbf{x}) = \frac{1}{2}$, we break the tie in favor of $+1$.

Exercise 6.2

Let

$$f(\mathbf{x}) = \begin{cases} +1 & \text{if } \pi(\mathbf{x}) \geq \frac{1}{2}, \\ -1 & \text{otherwise.} \end{cases}$$

Show that the probability of error on a test point $\mathbf{x}$ is

$$e(f(\mathbf{x})) = \mathbb{P}[f(\mathbf{x}) \neq y] = \min\{\pi(\mathbf{x}), 1 - \pi(\mathbf{x})\}$$

and $e(f(\mathbf{x})) \leq e(h(\mathbf{x}))$ for any other hypothesis h (deterministic or not).

The error $e(f(\mathbf{x}))$ is the probability that $y \neq f(\mathbf{x})$ on the test point $\mathbf{x}$. The previous exercise shows that $e(f(\mathbf{x})) = \min\{\pi(\mathbf{x}), 1 - \pi(\mathbf{x})\}$, which is the minimum probability of error possible on test point $\mathbf{x}$. Thus, the best possible out-of-sample misclassification error is the expected value of $e(f(\mathbf{x}))$,

$$E^*_{\text{out}} = \mathbb{E}_{\mathbf{x}}[e(f(\mathbf{x}))] = \int \mathrm{d}\mathbf{x} \;\; P(\mathbf{x}) \min\{\pi(\mathbf{x}), 1 - \pi(\mathbf{x})\}.$$

If we are to infer $f(\mathbf{x})$ from a nearest neighbor using $f(\mathbf{x}_{[1]})$, then $\pi(\mathbf{x})$ should not be varying too rapidly and $\mathbf{x}_{[1]}$ should be close to $\mathbf{x}$. We require that $\pi(\mathbf{x})$ should be continuous.[5] To ensure that there is a near enough neighbor $\mathbf{x}_{[1]}$ for *any* test point $\mathbf{x}$, it will suffice that N be large enough. We now show that, if $\pi(\mathbf{x})$ is continuous and N is large enough, then the simple nearest neighbor rule is reliable; specifically, it achieves an out-of-sample error which is at most twice the minimum achievable error (hence the title of this section).

Let $\{\mathcal{D}_N\}$ be a sequence of data sets with sizes $N = 1, 2, \ldots$ generated according to $P(\mathbf{x}, y)$ and let $\{g_N\}$ be the associated nearest neighbor decision rules for each data set. For N large enough, we will show that

$$E_{\text{out}}(g_N) \leq 2E^*_{\text{out}}.$$

So, even though the nearest neighbor rule is non-falsifiable, the test error is at most a factor of 2 worse than optimal (in the asymptotic limit, i.e., for large N). The nearest neighbor rule is reliable; however, E_{in}, which is always zero, will never match E_{out}. We will never have a good theoretical estimate of E_{out} (so we will not know our final performance), but we know that you cannot do much better. The formal statement we can make is:

Theorem 6.1. For any $\delta > 0$, and any continuous noisy target $\pi(\mathbf{x})$, there is a large enough N for which, with probability at least $1 - \delta$, $E_{\text{out}}(g_N) \leq 2E^*_{\text{out}}$.

The intuition behind the factor of 2 is that $f(\mathbf{x})$ makes a mistake only if there is an error on the test point $\mathbf{x}$, whereas the nearest neighbor classifier makes a mistake if there is an error on the test point *or* on the nearest neighbor (which adds up to the factor of 2).

[5] Formally, it is only required that the assumptions hold on a set of probability 1.

More formally, consider a test point $\mathbf{x}$. The nearest neighbor classifier makes an error on $\mathbf{x}$ if $y = 1$ and $y_{[1]} = -1$ or if $y = -1$ and $y_{[1]} = 1$.

$$\begin{aligned}\mathbb{P}[g_N(\mathbf{x}) \neq y] &= \pi(\mathbf{x}) \cdot (1 - \pi(\mathbf{x}_{[1]})) + (1 - \pi(\mathbf{x})) \cdot \pi(\mathbf{x}_{[1]}),\\ &= 2\pi(\mathbf{x}) \cdot (1 - \pi(\mathbf{x})) + \epsilon_N(\mathbf{x}),\\ &= 2\eta(\mathbf{x}) \cdot (1 - \eta(\mathbf{x})) + \epsilon_N(\mathbf{x}),\end{aligned}$$

where we defined $\eta(\mathbf{x}) = \min\{\pi(\mathbf{x}), 1 - \pi(\mathbf{x})\}$ (the minimum probability of error on $\mathbf{x}$), and $\epsilon_N(\mathbf{x}) = (2\pi(\mathbf{x}) - 1) \cdot (\pi(\mathbf{x}) - \pi(\mathbf{x}_{[1]}))$. Observe that

$$|\epsilon_N(\mathbf{x})| \leq |\pi(\mathbf{x}) - \pi(\mathbf{x}_{[1]})|,$$

because $0 \leq \pi(\mathbf{x}) \leq 1$. To get $E_{\text{out}}(g_N)$, we take the expectation with respect to $\mathbf{x}$ and use $E^*_{\text{out}} = \mathbb{E}[\eta(\mathbf{x})]\ \mathbb{E}[\eta(\mathbf{x})^2] \geq \mathbb{E}[\eta(\mathbf{x})]^2 = {E^*_{\text{out}}}^2$:

$$\begin{aligned}E_{\text{out}}(g_N) &= \mathbb{E}\left[\mathbb{P}[g_N(\mathbf{x}) \neq y]\right]\\ &= 2\,\mathbb{E}\left[\eta(\mathbf{x})\right] - 2\,\mathbb{E}\left[\eta^2(\mathbf{x})\right] + \mathbb{E}_{\mathbf{x}}[\epsilon_N(\mathbf{x})]\\ &\leq 2E^*_{\text{out}}(1 - E^*_{\text{out}}) + \mathbb{E}_{\mathbf{x}}[\epsilon_N(\mathbf{x})].\end{aligned}$$

Under very general conditions, $\mathbb{E}_{\mathbf{x}}[\epsilon_N(\mathbf{x})] \to 0$ in probability, which implies Theorem 6.1. We don't give a formal proof, but sketch the intuition for why $\epsilon_N(\mathbf{x}) \to 0$. When the data set gets very large ($N \to \infty$), *every* point $\mathbf{x}$ has a nearest neighbor that is close by. That is, $\mathbf{x}_{[1]}(\mathbf{x}) \to \mathbf{x}$ for all $\mathbf{x}$. This is the case if, for example, $P(\mathbf{x})$ has bounded support[6] as shown in Problem 6.9 (more general settings are covered in the literature). By the continuity of $\pi(\mathbf{x})$, if $\mathbf{x}_{[1]} \to \mathbf{x}$ then $\pi(\mathbf{x}_{[1]}) \to \pi(\mathbf{x})$, and since $|\epsilon_N(\mathbf{x})| \leq |\pi(\mathbf{x}_{[1]}) - \pi(\mathbf{x})|$, it follows that $\epsilon_N(\mathbf{x}) \to 0$.[7] The proof also highlights when the nearest neighbor works: a test point must have a nearby neighbor, and the noisy target $\pi(\mathbf{x})$ should be slowly varying so that the y-value at the nearby neighbor is indicative of the y-value at $\mathbf{x}$.

When E^*_{out} is small, $E_{\text{out}}(g)$ is at most twice as bad. When E^*_{out} is large, you are doomed anyway (but on the positive side, you will get a better factor than 2 ☺). It is quite startling that as simple a rule as nearest neighbor is 2-optimal.

At least half the classification power of the data is in the nearest neighbor.

The rate of convergence to 2-optimal depends on how smooth $\pi(\mathbf{x})$ is, but the convergence itself is never in doubt, relying only on the weak requirement that $\pi(\mathbf{x})$ is continuous. The fact that you get a small E_{out} from a model that can fit any data set has led to the notion that the nearest neighbor rule is somehow 'self-regularizing'. Recall that regularization guides the learning

[6] $\mathbf{x} \in \text{supp}(P)$ if for any $\epsilon > 0$, $\mathbb{P}[B_\epsilon(\mathbf{x})] > 0$ ($B_\epsilon(\mathbf{x})$ is the ball of radius ϵ centered on $\mathbf{x}$).

[7] One technicality remains because we need to interchange the order of limit and expectation: $\lim_{N\to\infty} \mathbb{E}_{\mathbf{x}}[\epsilon(\mathbf{x})] = \mathbb{E}_{\mathbf{x}}[\lim_{N\to\infty} \epsilon(\mathbf{x})] = 0$. For the technically inclined, this can be justified under mild measurability assumptions using dominated convergence.

toward simpler hypotheses, and helps to combat noise, especially when there are fewer data points. This self regularizing in the nearest neighbor algorithm occurs naturally because the final hypothesis is indeed simpler when there are fewer data points. The following examples of the nearest neighbor classifier with increasing number of data points illustrates how the boundary gets more complicated only as you increase N.

6.2.2 k-Nearest Neighbors (k-NN)

By considering more than just a single neighbor, one obtains the generalization of the nearest neighbor rule to the k-nearest neighbor rule (k-NN). For simplicity, assume that k is odd. The k-NN rule classifies the test point $\mathbf{x}$ according to the majority class among the k nearest data points to $\mathbf{x}$ (the k nearest neighbors). The nearest neighbor rule is the 1-NN rule. For $k > 1$, the final hypothesis is:

$$g(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{k} y_{[i]}(\mathbf{x})\right).$$

(k is odd and $y_n = \pm 1$). Figure 6.2 gives a comparison of the nearest neighbor rule with the 21-NN rule on the digits data (blue circles are the digit 1 and all other digits are the red x's). When k is large, the hypothesis is 'simpler'; in particular, when $k = N$ the final hypothesis is a constant equal to the majority class in the data. The parameter k controls the complexity of the resulting hypothesis. Smaller values of k result in a more complex hypothesis, with lower in-sample-error. The nearest neighbor rule always achieves zero in-sample error. When $k > 1$ (see Figure 6.2(b)), the in-sample error is not necessarily zero and we observe that the complexity of the resulting classification boundary drops considerably with higher k.

Three Neighbors is Enough. As with the simple nearest neighbor rule, one can analyze the performance of the k-NN rule (for large N). The next exercise guides the reader through this analysis.

(a) 1-NN rule (b) 21-NN rule

Figure 6.2: The 1-NN and 21-NN rules for classifying a random sample of 500 digits (1 (blue circle) vs all other digits). Note, $21 \approx \sqrt{500}$. For the 1-NN rule, the in-sample error is zero, resulting in a complicated decision boundary with islands of red and blue regions. For the 21-NN rule, the in-sample error is not zero and the decision boundary is 'simpler'.

Exercise 6.3

Fix an odd $k \geq 1$. For $N = 1, 2, \ldots$ and data sets $\{\mathcal{D}_N\}$ of size N, let g_N be the k-NN rule derived from $\mathcal{D}_N$, with out-of-sample error $E_{\text{out}}(g_N)$.

(a) Argue that $E_{\text{out}}(g_N) = \mathbb{E}_{\mathbf{x}}[Q_k(\eta(\mathbf{x}))] + \mathbb{E}_{\mathbf{x}}[\epsilon_N(\mathbf{x})]$ for some error term $\epsilon_N(\mathbf{x})$ which converges to zero, and where

$$Q_k(\eta) = \sum_{i=0}^{(k-1)/2} \binom{k}{i} \left(\eta^{i+1}(1-\eta)^{k-i} + (1-\eta)^{i+1}\eta^{k-i} \right),$$

and $\eta(\mathbf{x}) = \min\{\pi(\mathbf{x}), 1 - \pi(\mathbf{x})\}$.

(b) Plot $Q_k(\eta)$ for $\eta \in [0, \frac{1}{2}]$ and $k = 1, 3, 5$.

(c) Show that for large enough N, with probability at least $1 - \delta$,

$$\begin{aligned} k = 3: &\qquad E_{\text{out}}(g_N) \leq E^*_{\text{out}} + 3\,\mathbb{E}\left[\eta^2(\mathbf{x});\right] \\ k = 5: &\qquad E_{\text{out}}(g_N) \leq E^*_{\text{out}} + 10\,\mathbb{E}\left[\eta^3(\mathbf{x})\right]. \end{aligned}$$

(d) *[Hard]* Show that $E_{\text{out}}(g_N)$ is asymptotically $E^*_{\text{out}}(1 + O(k^{-1/2}))$. *[Hint: Use your plot of Q_k to argue that there is some $a(k)$ such that $Q_k \leq \eta(1 + a(k))$, and show that the best such $a(k)$ is $O(1/\sqrt{k})$.]*

For $k = 3$, as N grows, one finds that the out-of-sample error is nearly optimal:

$$E_{\text{out}}(g_N) \leq E^*_{\text{out}} + 3\,\mathbb{E}\left[\eta^2(\mathbf{x})\right].$$

To interpret this result, suppose that $\eta(\mathbf{x})$ is approximately constant. Then,

for $k = 3$, $E_{\text{out}}(g_N) \leq E^*_{\text{out}} + 3(E^*_{\text{out}})^2$ (when N is large). The intuition here is that the nearest neighbor classifier makes a mistake if there is an error on the test $\mathbf{x}$ *or* if there are at least 2 mistakes on the three nearest neighbors. The probability of two errors is $O(\eta^2)$ which is dominated by the probability of error on the test point. The 3-NN rule is approximately optimal, converging to E^*_{out} as $E^*_{\text{out}} \to 0$.

To illustrate further, suppose the optimal classifier gives $E^*_{\text{out}} = 1\%$. Then, for large enough N, 3-NN delivers E_{out} of at most 1.03%, which is essentially optimal. The exercise also shows that going to 5-NN improves the asymptotic performance to 1.001%, which is certainly closer to optimal than 3-NN, but hardly worth the effort. Thus, a useful practical guideline is: $k = 3$ nearest neighbors is often enough.

Approximation Versus Generalization. The parameter k controls the approximation-generalization trade off. Choosing k too small leads to too complex a decision rule, which overfits the data; on the other hand, k too large leads to underfitting. Though $k = 3$ works well when E^*_{out} is small, for general situations we need to choose between the different k: each value of k is a different model. We need to pick the best value of k (the best model). That is, we need model selection.

Exercise 6.4

Consider the task of selecting a nearest neighbor rule. What's wrong with the following logic applied to selecting k? (Limits are as $N \to \infty$.)

> Consider the hypothesis set $\mathcal{H}_{\text{NN}}$ with N hypotheses, the k-NN rules using $k = 1, \ldots, N$. Use the in-sample error to choose a value of k which minimizes E_{in}. Using the generalization error bound in Equation (2.1), conclude that $E_{\text{in}} \to E_{\text{out}}$ because $\log N/N \to 0$. Hence conclude that asymptotically, we will be picking the best value of k, based on E_{in} alone.

[Hints: What value of k will be picked? What will E_{in} be? Does your 'hypothesis set' depend on the data?]

Fortunately, there is powerful theory which gives a guideline for selecting k as a function of N; let $k(N)$ be the choice of k, which is chosen to grow as N grows. Note that $1 \leq k(N) \leq N$.

Theorem 6.2. For $N \to \infty$, if $k(N) \to \infty$ and $k(N)/N \to 0$ then,

$$E_{\text{in}}(g) \to E_{\text{out}}(g) \qquad \text{and} \qquad E_{\text{out}}(g) \to E^*_{\text{out}}.$$

Theorem 6.2 holds under smoothness assumptions on $\pi(\mathbf{x})$ and regularity conditions on $P(\mathbf{x})$ which are very general. The theorem sets $k(N)$ as a function of N and states that as $N \to \infty$, we can recover the optimal classifier. One good choice for $k(N)$ is $\lfloor \sqrt{N} \rfloor$.

We sketch the intuition for Theorem 6.2. For a test point $\mathbf{x}$, consider the distance r to its k-th nearest neighbor. The fraction of points which fall within distance r of $\mathbf{x}$ is k/N which, by assumption, is approaching zero. The only way that the fraction of points within r of $\mathbf{x}$ goes to zero is if r itself tends to zero. That is, *all* the k nearest neighbors to $\mathbf{x}$ are approaching $\mathbf{x}$. Thus, the fraction of these k neighbors with $y = +1$ is approximating $\pi(\mathbf{x})$ and as $k \to \infty$ this fraction will approach $\pi(\mathbf{x})$ by the law of large numbers. The majority vote among these k neighbors will therefore be implementing the optimal classifier f from Exercise 6.2, and hence we will obtain the optimal error E^*_{out}. We can also directly identify the roles played by the two assumptions in the statement of the theorem. The $k/N \to 0$ condition is to ensure that all the k nearest neighbors are close to $\mathbf{x}$; the $k \to \infty$ is to ensure that there are enough close neighbors so that the majority vote is almost always the optimal decision.

Few practitioners would be willing to blindly set $k = \lfloor\sqrt{N}\rfloor$. They would much rather the data spoke for itself. Validation, or cross validation (Chapter 4) are good ways to select k. Validation to select a nearest neighbor classifier would run as follows. Randomly divide the data $\mathcal{D}$ into a training set $\mathcal{D}_{\text{train}}$ and a validation set $\mathcal{D}_{\text{val}}$ of sizes $N - K$ and K respectively (the size of the validation set K is not to be confused with k, the number of nearest neighbors to use). Let $\mathcal{H}_{\text{train}} = \{g^-_1, \ldots, g^-_{N-K}\}$ be $N-K$ hypotheses[8] defined by the k-NN rules obtained from $\mathcal{D}_{\text{train}}$, for $k = 1, \ldots, N-K$. Now use $\mathcal{D}_{\text{val}}$ to select one hypothesis g^- from $\mathcal{H}_{\text{train}}$ (the one with minimum validation error). Similarly, one could use cross validation instead of validation.

Exercise 6.5

Consider using validation to select a nearest neighbor rule (hypothesis g^- from $\mathcal{H}_{\text{train}}$). Let g^-_* be the hypothesis in $\mathcal{H}_{\text{train}}$ with minimum E_{out}.

(a) Show that if $K/\log(N-K) \to \infty$ then validation chooses a good hypothesis, $E_{\text{out}}(g^-) \approx E_{\text{out}}(g^-_*)$. Formally state such a result and show it to be true. *[Hint: The statement has to be probabilistic; use the Hoeffding bound and the fact that choosing g^- amounts to selecting a hypothesis from among $N-K$ using a data set of size K.]*

(b) If also $N - K \to \infty$, then show that $E_{\text{out}}(g^-) \to E^*_{\text{out}}$ (validation results in near optimal performance). *[Hint: Use (a) together with Theorem 6.2 which shows that some value of k is good.]*

Note that the selected g^- is not a nearest neighbor rule on the full data $\mathcal{D}$; it is a nearest neighbor rule using data $\mathcal{D}_{\text{train}}$, and k^- neighbors. Would the performance improve if we used the k^--NN rule on the full data set $\mathcal{D}$?

The previous exercise shows that validation works if the validation set size is appropriately chosen, for example, proportional to N. Figure 6.3 illustrates

[8] The g^- notation was introduced in Chapter 4 to denote a hypothesis built from the data set that is missing some data points.

(a) In-sample error of k-NN rule (b) Out-of-sample error of k-NN rule

Figure 6.3: In and out-of-sample errors for various choices of k for the 1 vs all other digits classification task. (a) shows the in-sample error; (b) the out-of-sample error. N points are randomly selected for training, and the remaining for computing the test error. The performance is averaged over 10,000 such training-testing splits. Observe that the 3-NN is almost as good as anything; the in and out-of sample errors for $k = N^{1/2}$ are a close match, as expected from Theorem 6.2. The cross validation in-sample error is approaching the out-of-sample error. For $k = 1$ the in-sample error is zero, and is not shown.

our main conclusions on the digits data using $k = \lfloor\sqrt{N}\rfloor$, k chosen using 10-fold cross validation and k fixed at $1, 3$.

6.2.3 Improving the Efficiency of Nearest Neighbor

The simplicity of the nearest neighbor rule is both a virtue and a vice. There's no training cost, but we pay for this when predicting on a test input during deployment. For a test point $\mathbf{x}$, we need to compute the k nearest neighbors. This means we need access to all the data, a memory requirement of Nd and a computational complexity of $O(Nd + N\log k)$ (using appropriate data structures to compute N distances and pick the smallest k). This can be excessive as the next exercise illustrates.

Exercise 6.6

We want to select the value of $k = 1, 3, 5, \ldots, 2\lfloor\frac{N+1}{2}\rfloor - 1$ using 10-fold cross validation. Show that the running time is $O(N^3 d + N^3 \log N)$

The basic approach to addressing these issues is to preprocess the data in some way. This can help by either storing the data in a specialized data structure so that the nearest neighbor queries are more efficient, or by reducing the amount of data that needs to be kept (which helps with the memory and computational

requirements when deploying). These are active research areas, and we discuss the basic approaches here.

Data Condensing The goal of data condensing is to reduce the data set size while making sure the retained data set somehow matches the original full data set. We denote the condensed data set by S. We will only consider the case when S is a subset of $\mathcal{D}$.[9]

Fix k, and for the data set S, let g_S be the k-NN rule derived from S and, similarly, g_D is the k-NN rule derived from the full data set $\mathcal{D}$. The condensed data set S is *consistent* with $\mathcal{D}$ if

$$g_S(\mathbf{x}) = g_{\mathcal{D}}(\mathbf{x}) \qquad \forall \mathbf{x} \in \mathbb{R}^d.$$

This is a strong requirement, and there are computationally expensive algorithms which can achieve this (see Problem 6.11). A weaker requirement is that the condensed data only be *training set consistent*, which means that the condensed data and the full training data give the same classifications on the training data points,

$$g_S(\mathbf{x}_n) = g_{\mathcal{D}}(\mathbf{x}_n) \qquad \forall \mathbf{x}_n \in \mathcal{D}.$$

This means that, as far as the training points are concerned, the rule derived from S is the same as the rule derived from $\mathcal{D}$. Note that training set consistent does not mean $E_{\text{in}}(g_S) = 0$. For starters, when $k > 1$, even the full data $\mathcal{D}$ may not achieve $E_{\text{in}}(g_{\mathcal{D}}) = 0$.

The condensed nearest neighbor algorithm (CNN) is a very simple heuristic that results in a training set consistent subset of the data. We illustrate with a data set of 8 points on the right, setting k to 3. Initialize the working set S randomly with k data points from $\mathcal{D}$ (the boxed points illustrated on the right). As long as S is not training set consistent, augment S with a data point not in S as follows. Let $\mathbf{x}_*$ be *any* data point for which $g_S(\mathbf{x}_*) \neq g_{\mathcal{D}}(\mathbf{x}_*)$, such as the solid blue circle in the example. This solid blue point is classified blue by S (in fact, every point is classified blue by S because S only contains three points, two of which are blue); but, it is classified red by $\mathcal{D}$ (because two of its three nearest points in $\mathcal{D}$ are red). So $g_S(\mathbf{x}_*) \neq g_{\mathcal{D}}(\mathbf{x}_*)$. Let $\mathbf{x}'$ be the nearest data point to $\mathbf{x}_*$ which is not in S *and* has class $g_{\mathcal{D}}(\mathbf{x}_*)$. Note that $\mathbf{x}_*$ could be in S. The reader can easily verify that such an $\mathbf{x}'$ must exist because $\mathbf{x}_*$ is inconsistently classified by S. Add $\mathbf{x}'$ to S. Continue adding to S in this way until S is training set consistent. (Note that $\mathcal{D}$ is training set consistent, by definition.)

[9] More generally, if S contains arbitrary points, S is called a condensed set of *prototypes*.

(a) Condensed data for 1-NN (b) Condensed data for 21-NN

Figure 6.4: The condensed data and resulting classifiers from running the CNN heuristic to condense the 500 digits data points used in Figure 6.2. (a) Condensation to 27 data points for 1-NN training set consistent (b) Condensation to 40 data points for 21-NN training set consistent.

The CNN algorithm must terminate after at most N steps, and when it terminates, the resulting set S must be training set consistent. The CNN algorithm delivers good condensation in practice as illustrated by the condensed data in Figure 6.4 as compared with the full data in Figure 6.2.

Exercise 6.7

Show the following properties of the CNN heuristic. Let S be the current set of points selected by the heuristic.

(a) If S is not training set consistent, and if $\mathbf{x}_*$ is a point which is not training set consistent, show that the CNN heuristic will always find a point to add to S.

(b) Show that the point added will 'help' with the classification of $\mathbf{x}_*$ by S; it suffices to show that the new k nearest neighbors to $\mathbf{x}_*$ in S will contain the point added.

(c) Show that after at most $N - k$ iterations the CNN heuristic must terminate with a training set consistent S.

The CNN algorithm will generally not produce a training consistent set of minimum cardinality. Further, the condensed set can vary depending on the order in which data points are considered, and the process is quite computationally expensive; but, at least it is done only once, and it saves every time a new point is to be classified.

We should distinguish data condensing from *data editing*. The former has a purely computational motive, to reduce the size of the data set while keeping

the decision boundary nearly unchanged. The latter edits the data set to improve prediction performance. This typically means remove the points that you think are noisy (see Chapter **??**). One way to determine if a point is noisy is if it disagrees with the majority class in its k-neighborhood. Such data points are removed from the data and the remaining data are used with the 1-NN rule. A very similar effect can be accomplished by just using the k-NN rule with the original data.

Efficient Search for the Nearest Neighbors The other main approach to alleviating the computational burden is to preprocess the data in some way so that nearest neighbors can be found quickly. There is a large and growing volume of literature in this arena. Our goal here is to describe the basic idea with a simple technique.

Suppose we want to search for the nearest neighbor to the point $\mathbf{x}$ (red). Assume that the data is partitioned into clusters (or *branches*). In our illustration, we have two clusters S_1 and S_2. Each cluster has a 'center' $\boldsymbol{\mu}_j$ and a radius r_j. First search the cluster whose center is closest to the query point $\mathbf{x}$. In the example, we search cluster S_1 because $\|\mathbf{x} - \boldsymbol{\mu}_1\| \leq \|\mathbf{x} - \boldsymbol{\mu}_2\|$. Suppose that the nearest neighbor to $\mathbf{x}$ in S_1 is $\hat{\mathbf{x}}_{[1]}$, which is potentially $\mathbf{x}_{[1]}$, the nearest point to $\mathbf{x}$ in all the data. For any $\mathbf{x}_n \in S_2$, by the triangle inequality, $\|\mathbf{x} - \mathbf{x}_n\| \geq \|\mathbf{x} - \boldsymbol{\mu}_2\| - r_2$. Thus, if the *bound* condition

$$\|\mathbf{x} - \hat{\mathbf{x}}_{[1]}\| \leq \|\mathbf{x} - \boldsymbol{\mu}_2\| - r_2$$

holds, then we are done (we don't need to search S_2), and can claim that $\hat{\mathbf{x}}_{[1]}$ is a nearest neighbor ($\mathbf{x}_{[1]} = \hat{\mathbf{x}}_{[1]}$). We gain a computational saving if the bound condition holds. This approach is called a *branch and bound* technique for finding the nearest neighbor. Let's informally consider when we are likely to satisfy the bound condition, and hence obtain computational saving. By the triangle inequality, $\|\mathbf{x} - \hat{\mathbf{x}}_{[1]}\| \leq \|\mathbf{x} - \boldsymbol{\mu}_1\| + r_1$, so, if $\|\mathbf{x} - \boldsymbol{\mu}_1\| + r_1 \leq \|\mathbf{x} - \boldsymbol{\mu}_2\| - r_2$, then the bound condition is certainly satisfied. To interpret this sufficient condition, consider the case when $\mathbf{x}$ is much closer to $\boldsymbol{\mu}_1$, in which case $\|\mathbf{x} - \boldsymbol{\mu}_1\| \approx 0$ and $\|\mathbf{x} - \boldsymbol{\mu}_2\| \approx \|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\|$. Then we want

$$r_1 + r_2 \ll \|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\|$$

to ensure the bound holds. In words, the radii of the clusters (the *within cluster spread*) should be much smaller than the distance between clusters

(the *between cluster spread*). A good clustering algorithm should produce exactly such clusters, with small within cluster spread and large between cluster spread.

There is nothing to prevent us from applying the branch and bound technique recursively within each cluster. So, for example, to search for the nearest neighbor inside S_1, we can assume that S_1 has been partitioned into (say) two clusters S_{11} and S_{12}, and so on.

Exercise 6.8

(a) Give an algorithmic pseudo code for the recursive branch and bound search for the nearest neighbor, assuming that every cluster with more than 2 points is split into 2 sub-clusters.

(b) Assume the sub-clusters of a cluster are balanced, i.e. contain exactly half the points in the cluster. What is the maximum depth of the recursive search for a nearest neighbor. (Assume the number of data points is a power of 2).

(c) Assume balanced sub-clusters and that the bound condition always holds. Show that the time to find the nearest neighbor is $O(d \log N)$.

(d) How would you apply the branch and bound technique to finding the k-nearest neighbors as opposed to the single nearest neighbor?

The computational savings of the branch and bound algorithm depend on how likely it is that the bound condition is met, which depends on how good the partitioning is. We observe that the clusters should be balanced (so that the recursion depth is not large), well separated and have small radii (so the bound condition will often hold). A very simple heuristic for getting such a good partition into M clusters is based on a clustering algorithm we will discuss later. Here is the main idea.

First create a set of M well separated centers for the clusters; we recommend a simple greedy approach. Start by taking an arbitrary data point as a center. Now, as a second center, add the point furthest from the first center; to get each new center, keep adding the point furthest from the current set of centers until you have M centers (the distance of a point from a set is the distance to its nearest neighbor in the set). In the example, the red point is the first (random) center added; the largest blue point is added next and so on. This greedy algorithm can be implemented in $O(MNd)$ time using appropriate data structures. The partition is inferred from the Voronoi regions of each center: all points in a center's

Voronoi region belong to that center's cluster. The center of each cluster is redefined as the average data point in the cluster, and its radius is the maximum distance from the center to a point in the cluster,

$$\boldsymbol{\mu}_j = \frac{1}{|S_j|} \sum_{\mathbf{x}_n \in S_j} \mathbf{x}_n; \qquad r_j = \max_{\mathbf{x}_n \in S_j} \|\mathbf{x}_n - \boldsymbol{\mu}_j\|.$$

We can improve the clusters by obtaining new Voronoi regions defined by these new centers $\boldsymbol{\mu}_j$, and then updating the centers and radii appropriately for these new clusters. Naturally, we can continue this process iteratively (we introduce this algorithm later in the chapter as Lloyd's algorithm for clustering). The main goal here is computational efficiency, and the perfect partition is not necessary. The partition, centers and radii after one refinement are illustrated on the right. Note that though the clusters are disjoint, the spheres enclosing the clusters need not be.

The clusters, together with $\boldsymbol{\mu}_j$ and r_j can be computed in $O(MNd)$ time. If the depth of the partitioning is $O(\log N)$, which is the case if the clusters are nearly balanced, then the total run time to compute all the clusters at every level is $O(MNd \log N)$. In Problem 6.16 you can experiment with the performance gains from such simple branch and bound techniques.

6.2.4 Nearest Neighbor is Nonparametric

There is some debate in the literature on *parametric* versus *nonparametric* learning models. Having covered linear models, and nearest neighbor, we are in a position to articulate some of this discussion. We begin by identifying some of the defining characteristics of parametric versus nonparametric models.

Nonparametric methods have no *explicit* parameters to be learned from data; they are, in some sense, general and expressive or flexible; and, they typically store the training data, which are then used during deployment to predict on a test point. Parametric methods, on the other hand, explicitly learn certain parameters from data. These parameters specify the final hypothesis from a typically 'restrictive' *parameterized* functional form; the learned parameters are sufficient for prediction, i.e. the parameters summarize the data which is then not needed later during deployment.

To further highlight the distinction between these two types of techniques, consider the nearest neighbor rule versus the linear model. The nearest neighbor rule is the prototypical nonparametric method. There are no parameters to be learned. Yet, the nonparametric nearest neighbor method is flexible and

general: the final decision rule can be very complex, or not, molding its shape to the data. The linear model, which we saw in Chapter 3 is the prototypical parametric method. In d dimensions, there are $d+1$ parameters (the weights and bias) which need to be learned from the data. These parameters summarize the data, and after learning, the data can be discarded: prediction only needs the learned parameters. The linear model is quite rigid, in that no matter what the data, you will always get a linear separator as g. Figure 6.5 illustrates this difference in flexibility on a toy example.

(a) Nonparametric NN (b) Parametric linear

Figure 6.5: The decision boundary of the 'flexible' nonparametric nearest neighbor rule molds itself to the data, whereas the 'rigid' parametric linear model will always give a linear separator.

The k-nearest neighbor method would also be considered nonparametric (once the 'parameter' k has been specified). Theorem 6.2 is an example of a general convergence result.[10] Under mild regularity conditions, no matter what the target f is, we can recover it as $N \to \infty$, provided that k is chosen appropriately. That's quite a powerful statement about such a simple learning model applied to learning a general f. Such convergence results under mild assumptions on f are a trademark of nonparametric methods. This has led to the folklore that nonparametric methods are, in some sense, more powerful than their cousins the parametric methods: for the parametric linear model, *only if* the target f happens to be in the linearly parameterized hypothesis set, can one get such convergence to f with larger N.

To complicate the distinction between the two methods, let's look at the non-linear feature transform (e.g. the polynomial transform). As the polynomial order increases, the number of parameters to be estimated increases and $\mathcal{H}$, the hypothesis set, gets more and more expressive. It is possible to

[10] For the technically oriented, it establishes the *universal consistency* of the k-NN rule.

choose the polynomial order to increase with N, but not too quickly so that $\mathcal{H}$ gets more and more expressive, eventually capturing any fixed target and yet the number of parameters grows slowly enough that it is possible to effectively learn them from the data. Such a model has properties of both the non-parametric and parametric settings: you still need to learn parameters (like parametric models) but the hypothesis set is getting more and more expressive so that you do get convergence to f. As a result, such techniques have sometimes been called parametric, and sometimes nonparametric, neither being a satisfactory adjective; we prefer the term *semi-parametric*. Many techniques are semi-parametric, for example Neural Networks (see Chapter **??**).

In applications, the purely parametric model (like the linear model) has become synonymous with 'specialized' in that you have thought very carefully about the features and the form of the model and you are confident that the specialized linear model will work. To take another example from interest rate prediction, a well known specialized parametric model is the multi-factor Vasicek model. If we had no idea what the appropriate model is for interest rate prediction, then rather than commit to some specialized model, one might use a 'generic' nonparametric or semi-parametric model such as k-NN or neural networks. In this setting, the use of the more generic model, by being more flexible, accommodates our inability to pin down the specific form of the target function. We stress that the availability of 'generic' nonparametric and semi-parametric models is not a license to stop thinking about the right features and form of the model; it is just a backstop that is available when the target is not easy to explicitly model parametrically.

6.2.5 Multiclass Data

The nearest neighbor method is an elegant platform to study an issue which, unto now, we have ignored. In our digits data, there are not 2 classes (± 1), but 10 classes $(0, 1, \ldots, 9)$. This is a *multiclass* problem. Several approaches to the multiclass problem decompose it into a sequence of 2 class problems. For example, first classify between '1' and 'not 1'; if you classify as '1' then you are done. Otherwise determine '2' versus 'not 2'; this process continues until you have narrowed down the digit. Several complications arise in determining what is the best sequence of 2 class problems; and, they need not be one versus all (they can be any subset versus its complement).

The nearest neighbor method offers a simple solution. The class of the test point $\mathbf{x}$ is simply the class of its nearest neighbor $\mathbf{x}_{[1]}$. One can also generalize to the k-nearest neighbor rule: the class of $\mathbf{x}$ is the majority class among the k-nearest neigh-

(a) Multiclass digits data (b) 21-NN decision regions

Figure 6.6: 21-NN decision classifier for the multiclass digits data.

bors $\mathbf{x}_{[1]}, \ldots, \mathbf{x}_{[k]}$, breaking ties randomly.

Exercise 6.9

With C classes labeled $1, \ldots, C$, define $\pi_c(\mathbf{x}) = \mathbb{P}[c|\mathbf{x}]$ (the probability to observe class c given $\mathbf{x}$, analogous to $\pi(\mathbf{x})$). Let $\eta(\mathbf{x}) = 1 - \max_c \pi_c(\mathbf{x})$.

(a) Define a target $f(\mathbf{x}) = \text{argmax}_c\, \pi_c(\mathbf{x})$. Show that, on a test point $\mathbf{x}$, f attains the minimum possible error probability of

$$e(f(\mathbf{x})) = \mathbb{P}[f(\mathbf{x}) \neq y] = \eta(\mathbf{x}).$$

(b) Show that for the nearest neighbor rule ($k = 1$), with high probability, the final hypothesis g_N achieves an error on the test point $\mathbf{x}$ that is

$$e(g_N(\mathbf{x})) \overset{N\to\infty}{\longrightarrow} \sum_{c=1}^{C} \pi_c(\mathbf{x})(1 - \pi_c(\mathbf{x})).$$

(c) Hence, show that for large enough N, with high probability,

$$E_{\text{out}}(g_N) \leq 2E^*_{\text{out}} - \frac{C}{C-1}(E^*_{\text{out}})^2.$$

[Hint: Show that $\sum_i a_i^2 \geq a_1^2 + \frac{(1-a_1)^2}{C-1}$ for any $a_1 \geq \cdots \geq a_C \geq 0$ and $\sum_i a_i = 1$,]

The previous exercise shows that even for the multiclass problem, the simple nearest neighbor is at most a factor of 2 from optimal. One can also obtain a universal consistency result as in Theorem 6.2. The result of running a 21-nearest neighbor rule on the digits data is illustrated in Figure 6.6.

The Confusion Matrix. The probability of misclassification which we discussed in the 2-class problem can be generalized to a *confusion matrix* which is a better representation of the performance, especially when there are multiple classes. The confusion matrix tells us not only the probability of error, but also the nature of the error, namely which classes are confused with each other. Analogous to the misclassification error, one has the in-sample and out-of-sample confusion matrices. The in-sample (resp. out-of-sample) confusion matrix is a $C \times C$ matrix which measures how often one class is classified as another. So, the out-of-sample confusion matrix is

$$\begin{aligned} \mathrm{E}_{\mathrm{out}}(c_1, c_2) &= \mathbb{P}[\text{class } c_1 \text{ is classified as } c_2 \text{ by } g] \\ &= \mathbb{E}_{\mathbf{x}}\left[\pi_{c_1}(\mathbf{x})\, [\![g(\mathbf{x}) = c_2]\!]\right] \\ &= \int \mathrm{d}\mathbf{x} \;\; P(\mathbf{x})\pi_{c_1}(\mathbf{x})\, [\![g(\mathbf{x}) = c_2]\!] \,. \end{aligned}$$

where $[\![\cdot]\!]$ is the indicator function which equals 1 when its argument is true. Similarly, we can define the in-sample confusion matrix,

$$E_{\mathrm{in}}(c_1, c_2) = \frac{1}{N} \sum_{n: y_n = c_1} [\![g(\mathbf{x}_n) = c_2]\!] \,,$$

where the sum is over all data points with classification c_1, and the summand counts the number of those cases that g classifies as c_2. The out-of-sample confusion matrix obtained by running the 21-NN rule on the 10-class digits data is shown in Table 6.1. The sum of the diagonal elements gives the probability of correct classification, which is about 42%. From Table 6.1, we can easily identify some of the digits which are commonly confused, for example 8 is often classified as 0. In comparison, for the two class problem of classifying digit 1 versus all others, the success was upwards of 98%. For the multiclass problem, random performance would achieve a success rate of 10%, so the performance is significantly above random; however, it is significantly below the 2-class version; multiclass problems are typically much harder than the two class problem. Though symmetry and intensity are two features that have enough information to distinguish between 1 versus the other digits, they are not powerful enough to solve the multiclass problem. Better features would certainly help. One can also gain by breaking up the problem into a sequence of 2-class tasks and tailoring the features to each 2-class problem using domain knowledge.

6.2.6 Nearest Neighbor for Regression.

With multiclass data, the natural way to combine the classes of the nearest neighbors to obtain the class of a test input is by using some form of majority voting procedure. When the output is a real number ($y \in \mathbb{R}$), the natural way to combine the outputs of the nearest neighbors is using some form of averaging. The simplest way to extend the k-nearest neighbor algorithm to regression

True	Predicted										
	0	1	2	3	4	5	6	7	8	9	
0	**13.5**	0.5	0.5	1	0	0.5	0	0	0.5	0	16.5
1	0.5	**13.5**	0	0	0	0	0	0	0	0	14
2	0.5	0	**3.5**	1	1	1.5	1	1	0	0.5	10
3	2.5	0	1.5	**2**	0.5	0.5	0.5	0.5	0.5	1	9.5
4	0.5	0	1	0.5	**1.5**	0.5	1	2	0	1.5	8.5
5	0.5	0	2.5	1	0.5	**1.5**	1	1	0	0.5	7.5
6	0.5	0	2	1	1	1	**1**	1	0	1	8.5
7	0	0	1.5	0.5	1.5	0.5	1	**3**	0	1	9
8	3.5	0	0.5	1	0.5	0.5	0.5	0	**0.5**	1	8
9	0.5	0	1	1	1	0.5	1	1	0.5	**2**	8.5
	22.5	14	14	9	7.5	7	7	9.5	2	8.5	100

Table 6.1: Out-of-sample confusion matrix for the 21-NN rule on the 10-class digits problem (all numbers are in %). The bold entries along the diagonal are the correct classifications. Practitioners sometimes work with the class conditional confusion matrix which is $\mathbb{P}[c_2|c_1]$, the probability that g classifies c_2 *given* that the true class is c_1. The class conditional confusion matrix is obtained from the confusion matrix by dividing each row by the sum of the elements in the row. Sometimes the class conditional confusion matrix is easier to interpret because the ideal case is a diagonal matrix with all diagonals equal to 100%.

is to take the average of the target values from the k-nearest neighbors:

$$g(\mathbf{x}) = \frac{1}{k}\sum_{i=1}^{k} y_{[i]}(\mathbf{x}).$$

There are no explicit parameters being learned, and so this is a nonparametric regression technique. Figure 6.7 illustrates the k-NN technique for regression using a toy data set generated by the target function in light gray.

Figure 6.7: k-NN for regression on a noisy toy data set with $N = 20$.

Exercise 6.10

You are using k-NN for regression (Figure 6.7).

(a) Show that E_{in} is zero when $k = 1$.

(b) Why is the final hypothesis not smooth, making step-like jumps?

(c) How does the choice of k control how well g approximates f? (Consider the cases k too small and k too large.)

(d) How does the final hypothesis g behave when $x \to \pm\infty$.

Outputting a Probability with k-NN. Recall that for binary classification, we took $\text{sign}(g(\mathbf{x}))$ as the final hypothesis. We can also get a natural extension of the nearest neighbor technique outputting a probability. The final hypothesis is

$$g(\mathbf{x}) = \frac{1}{k}\sum_{i=1}^{k} [\![y_{[i]} = +1]\!] ,$$

which is just the fraction of the k nearest neighbors that are classified $+1$.

For both regression and logistic regression, as with classification, if $N \to \infty$ and $k \to \infty$ with $k/N \to 0$, then, under mild assumptions, you will recover a close approximation to the target function: $g(\mathbf{x}) \to f(\mathbf{x})$ (as N grows). This convergence is true even if the data (y values) are noisy, with bounded noise variance, because the averaging can overcome that noise when k gets large.

6.3 Radial Basis Functions

In the k-nearest neighbor rule, only k neighbors influence the prediction at a test point $\mathbf{x}$. Rather than fix the value of k, the radial basis function (RBF) technique allows all data points to contribute to $g(\mathbf{x})$, but not equally. Data points further from $\mathbf{x}$ contribute less; a radial basis function (or kernel) ϕ quantifies how the contribution decays as the distance increases. The contribution of $\mathbf{x}_n$ to the classification at $\mathbf{x}$ will be proportional to $\phi(\|\mathbf{x} - \mathbf{x}_n\|)$, and so the properties of a typical kernel are that it is positive and non-increasing. The most popular kernel is the Gaussian kernel,

$$\phi(z) = e^{-\frac{1}{2}z^2}.$$

Another common kernel is the window kernel,

$$\phi(z) = \begin{cases} 1 & z \le 1, \\ 0 & z > 1, \end{cases}$$

It is common to normalize the kernel so that it integrates to 1 over $\mathbb{R}^d$. For the Gaussian kernel, the normalization constant is $(2\pi)^{-d/2}$; for the window kernel, the normalization constant is $\pi^{-d/2}\Gamma(\frac{1}{2}d+1)$, where $\Gamma(\cdot)$ is the Gamma function. The normalization constant is not important for RBFs, but for density estimation (see later), the normalization will be needed.

One final ingredient is the scale r which specifies the width of the kernel. The scale determines the 'unit' of length against which distances are measured. If $\|\mathbf{x} - \mathbf{x}_n\|$ is small relative to r, then $\mathbf{x}_n$ will have a significant influence on the value of $g(\mathbf{x})$. We denote the influence of $\mathbf{x}_n$ at $\mathbf{x}$ by $\alpha_n(\mathbf{x})$,

$$\alpha_n(\mathbf{x}) = \phi\left(\frac{\|\mathbf{x} - \mathbf{x}_n\|}{r}\right).$$

The 'radial' in RBF is because the influence only depends on the radial distance $\|\mathbf{x} - \mathbf{x}_n\|$. We compute $g(\mathbf{x})$ as the weighted sum of the y-values:

$$g(\mathbf{x}) = \frac{\sum_{n=1}^{N} \alpha_n(\mathbf{x}) y_n}{\sum_{m=1}^{N} \alpha_m(\mathbf{x})}, \tag{6.1}$$

where the sum in the denominator ensures that the sum of the weights is 1. The hypothesis (6.1) is the direct analog of the nearest neighbor rule for regression. For classification, the final output is $\text{sign}(g(\mathbf{x}))$; and, for logistic regression the output is $\theta(g(\mathbf{x}))$, where $\theta(s) = e^s/(1+e^s)$.

The scale parameter r determines the relative emphasis on nearby points. The smaller the scale parameter, the more emphasis is placed on the nearest points. Figure 6.8 illustrates how the final hypothesis depends on r and the next exercise shows that when $r \to 0$, the RBF is the nearest neighbor rule.

Exercise 6.11

When $r \to 0$, show that for the Gaussian kernel, the RBF final hypothesis is $g(\mathbf{x}) = y_{[1]}$, the same as the nearest neighbor rule.

[Hint: $g(\mathbf{x}) = \dfrac{\sum_{n=1}^{N} y_{[n]}\alpha_{[n]}/\alpha_{[1]}}{\sum_{m=1}^{N} \alpha_{[m]}/\alpha_{[1]}}$ *and show that* $\alpha_{[n]}/\alpha_{[1]} \to 0$ *for* $n \neq 1$*.]*

As r gets large, the kernel width gets larger and more of the data points contribute to the value of $g(\mathbf{x})$. As a result, the final hypothesis gets smoother. The scale parameter r plays a similar role to the number of neighbors k in the k-NN rule; in fact, the RBF technique with the window kernel is sometimes

Figure 6.8: The nonparametric RBF using the same noisy data from Figure 6.7. Too small an r results in a complex hypothesis that overfits. Too large an r results in an excessively smooth hypothesis that underfits.

called the r-nearest neighbor rule, because it uniformly weights all neighbors within distance r of $\mathbf{x}$. One way to choose r is using cross validation. Another is to use Theorem 6.2 as a guide. Roughly speaking, the neighbors within distance r of $\mathbf{x}$ contribute to $g(\mathbf{x})$. The volume of this region is order of r^d, and so the number of points in this region is order of Nr^d; thus, using scale parameter r effectively corresponds to using a number of neighbors k that is order of Nr^d. Since we want $k/N \to \infty$ and $k \to \infty$ as N grows, one effective choice for r is $(\sqrt[2d]{N})^{-1}$, which corresponds to $k \approx \sqrt{N}$.

6.3.1 Radial Basis Function Networks

There are two ways to interpret the RBF in equation (6.1). The first is as a weighted sum of y-values as presented in equation (6.1), where the weights are α_n. This corresponds to centering a single kernel, or a bump, at $\mathbf{x}$. This bump decays as you move away from $\mathbf{x}$ and the value the bump attains at $\mathbf{x}_n$ determines the weight α_n (see Figure 6.9(a)). The alternative view is to rewrite Equation (6.1) as

$$g(\mathbf{x}) = \sum_{n=1}^{N} w_n(\mathbf{x}) \phi\left(\frac{\|\mathbf{x} - \mathbf{x}_n\|}{r}\right), \tag{6.2}$$

where $w_n(\mathbf{x}) = y_n / \sum_{i=1}^{N} \phi\left(\|\mathbf{x} - \mathbf{x}_i\|/r\right)$. This version corresponds to centering a bump at every $\mathbf{x}_n$. The bump at $\mathbf{x}_n$ has a height $w_n(\mathbf{x})$. As you move away from $\mathbf{x}_n$, the bump decays, and the value the bump attains at $\mathbf{x}$ is the contribution of $\mathbf{x}_n$ to $g(\mathbf{x})$ (see Figure 6.9(b)).

With the second interpretation, $g(\mathbf{x})$ is the sum of N bumps of different heights, where each bump is centered on a data point. The width of each bump is determined by r. The height of the bump centered on $\mathbf{x}_n$ is $w_n(\mathbf{x})$, which varies depending on the test point. If we fix the bump heights to w_n, independent of the test point, then the same bumps centered on the data points apply to every test point $\mathbf{x}$ and the functional form simplifies. Define

(a) Bump centered on $\mathbf{x}$ (b) Bumps centered on $\mathbf{x}_n$

Figure 6.9: Different views of the RBF. (a) $g(\mathbf{x})$ is a weighted sum of y_n with weights α_n determined by a bump centered on $\mathbf{x}$. (b) $g(\mathbf{x})$ is a sum of bumps, one on each $\mathbf{x}_n$ having height w_n.

$\Phi_n(\mathbf{x}) = \phi(\|\mathbf{x} - \mathbf{x}_n\|/r)$. Then our hypotheses have the form

$$h(\mathbf{x}) = \sum_{n=1}^{N} w_n \Phi_n(\mathbf{x}), \tag{6.3}$$

where now w_n are constants (parameters) that we get to choose. We switched over to $h(\mathbf{x})$ because as of yet, we do not know what the final hypothesis g is until we specify the weights $w_1, \ldots, w_N$, whereas in (6.2), the weights $w_n(\mathbf{x})$ are specified. Observe that (6.3) is just a linear model $h(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{z}$ with the nonlinear transform to an N-dimensional space given by

$$\mathbf{z} = \mathbf{\Phi}(\mathbf{x}) = \begin{bmatrix} \Phi_1(\mathbf{x}) \\ \vdots \\ \Phi_N(\mathbf{x}) \end{bmatrix}.$$

The nonlinear transform is determined by the kernel ϕ and the data points $\mathbf{x}_1, \ldots, \mathbf{x}_n$. Since the nonlinear transform is obtained by placing a kernel bump on each data point, these nonlinear transforms are often called *local* basis functions. Notice that the nonlinear transform is not known ahead of time. It is only known once the data points are seen.

In the nonparametric RBF, the weights $w_n(\mathbf{x})$ were specified by the data and there was nothing to learn. Now that w_n are parameters, we need to learn their values (the new technique is parametric). As we did with linear models, the natural way to set the parameters w_n is to fit the data by minimizing the in-sample error. Since we have N parameters, we should be able to fit the data exactly. Figure 6.10 illustrates the parametric RBF as compared to the nonparametric RBF, highlighting some of the differences between taking $g(\mathbf{x})$ as the weighted sum of y-values scaled by a bump at $\mathbf{x}$ versus the sum of fixed bumps at each data point $\mathbf{x}_n$.

(a) Nonparametric RBF, small r (b) Nonparametric RBF, large r

(c) Parametric RBF, small r (b) Parametric RBF, large r

Figure 6.10: (a),(b) The nonparametric RBF; (c),(d) The parametric RBF. The toy data set has 3 points. The width r controls the smoothness of the hypothesis. The nonparametric RBF is step-like with large $|x|$ behavior determined by the target values at the peripheral data; the parametric RBF is bump-like with large x behavior that converges to zero.

Exercise 6.12

(a) For the Gaussian kernel, what is $g(\mathbf{x})$ as $\|\mathbf{x}\| \to \infty$ for the nonparametric RBF versus for the parametric RBF with fixed w_n?

(b) Let Z be the square feature matrix defined by $\mathrm{Z}_{nj} = \Phi_j(\mathbf{x}_n)$. Assume Z is invertible. Show that $g(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{\Phi}(\mathbf{x})$, with $\mathbf{w} = \mathrm{Z}^{-1}\mathbf{y}$ exactly interpolates the data points. That is, $g(\mathbf{x}_n) = y_n$, giving $E_{\mathrm{in}}(g) = 0$.

(c) Does the nonparametric RBF always have $E_{\mathrm{in}} = 0$?

The parametric RBF has N parameters $w_1, \ldots, w_N$, ensuring we can always fit the data. When the data has stochastic or deterministic noise, this means we will overfit. The root of the problem is that we have too many parameters, one for each bump. Solution: restrict the number of bumps to $k \ll N$. If we restrict the number of bumps to k, then only k weights $w_1, \ldots, w_k$ need to be learned. Naturally, we will choose the weights that best fit the data.

Where should the bumps be placed? Previously, with N bumps, this was

not an issue since we placed one bump on each data point. Though we cannot place a bump on each data point, we can still try to choose the bump centers so that they represent the data points as closely as possible. Denote the centers of the bumps by $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$. Then, a hypothesis has the parameterized form

$$h(\mathbf{x}) = w_0 + \sum_{j=1}^{k} w_j \phi\left(\frac{\|\mathbf{x} - \boldsymbol{\mu}_j\|}{r}\right) = \mathbf{w}^{\mathrm{T}} \boldsymbol{\Phi}(\mathbf{x}), \tag{6.4}$$

where $\boldsymbol{\Phi}(\mathbf{x})^{\mathrm{T}} = [1, \Phi_1(\mathbf{x}), \ldots, \Phi_k(\mathbf{x})]$ and $\Phi_j(\mathbf{x}) = \phi(\|\mathbf{x} - \boldsymbol{\mu}_j\|/r)$. Observe that we have added back the bias term w_0.[11] For the nonparametric RBF or the parametric RBF with N bumps, we did not need the bias term. However, when you only have a small number of bumps and no bias term, the learning gets distorted if the y-values have non-zero mean (as is also the case if you do linear regression without the constant term).

The unknowns that one has to learn by fitting the data are the weights $w_0, w_1, \ldots, w_k$ and the bump centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ (which are vectors in $\mathbb{R}^d$). The model we have specified is called the *radial basis function network (RBF-network)*. We can get the RBF-network model for classification by taking the sign of the output signal, and for logistic regression we pass the output signal through the sigmoid $\theta(s) = e^s/(1 + e^s)$.

A useful graphical representation of the model as a 'feed forward' network is illustrated. There are several important features that are worth discussing. First, the hypothesis set is very similar to a linear model except that the transformed features $\Phi_j(\mathbf{x})$ can depend on the data set (through the choice of the $\boldsymbol{\mu}_j$ which are chosen to fit the data). In the linear model of Chapter 3, the transformed features were fixed ahead of time. Because the $\boldsymbol{\mu}_j$ appear inside a nonlinear function, this is our first model that is not linear in its parameters. It is linear in the w_j, but nonlinear in the $\boldsymbol{\mu}_j$. It turns out that allowing the the basis functions to depend on the data adds significant power to this model over the linear model. We discuss this issue in more detail later, in Chapter **??**.

Other than the parameters w_j and $\boldsymbol{\mu}_j$ which are chosen to fit the data, there are two high-level parameters k and r which specify the nature of the hypothesis set. These parameters control two aspects of the hypothesis set that were discussed in Chapter 5, namely the size of a hypothesis set (quantified by k, the number of bumps) and the complexity of a single hypothesis (quantified by r, which determines how 'wiggly' an individual hypothesis is).

[11] An alternative notation for the scale parameter $1/r$ is γ, and for the bias term w_0 is b (often used in the context of SVM, see Chapter **??**).

It is important to choose a good value of k to avoid overfitting (too high a k) or underfitting (too low a k). A good strategy for choosing k is using cross validation. For a given k, a good simple heuristic for setting r is

$$r = \frac{R}{k^{1/d}},$$

where $R = \max_{i,j} \|\mathbf{x}_i - \mathbf{x}_j\|$ is the 'diameter' of the data. The rationale for this choice of r is that k spheres of radius r have a volume equal to $c_d k r^d = c_d R^d$ which is about the volume of the data (c_d is a constant depending only on d). So, the k bumps can 'cover' the data with this choice of r.

6.3.2 Fitting the Data

To complete our specification of the RBF-network model, we need to discuss how to fit the data. For a given k and r, to obtain our final hypothesis g, we need to determine the centers $\boldsymbol{\mu}_j$ and the weights w_j. In addition to the weights, we appear to have an abundance of parameters in the centers $\boldsymbol{\mu}_j$. Fortunately, we will determine the centers by choosing them to represent the inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$ without reference to the $y_1, \ldots, y_N$. The heavy lifting to fit the y_n will be done by the weights, so if there is overfitting, it is mostly due to the flexibility to choose the weights, not the centers.

For a given set of centers, $\boldsymbol{\mu}_j$, the hypothesis is linear in the w_j; we can therefore use our favorite algorithm from Chapter 3 to fit the weights *if the centers are fixed*. The algorithm is summarized as follows.

Fitting the RBF-network to the data (given k, r):

1: Using the inputs X, determine k centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$.

2: Compute the $N \times (k+1)$ feature matrix Z

$$\mathrm{Z}_{n,0} = 1 \qquad \mathrm{Z}_{n,j} = \Phi_j(\mathbf{x}_n) = \phi\left(\frac{\|\mathbf{x}_n - \boldsymbol{\mu}_j\|}{r}\right).$$

Each row of Z is the RBF-feature corresponding to $\mathbf{x}_n$ (where we have the dummy bias coordinate $\mathrm{Z}_{n,0} = 1$).

3: Fit the *linear* model $\mathrm{Z}\mathbf{w}$ to $\mathbf{y}$ to determine the weights $\mathbf{w}^*$.

We briefly elaborate on step 3 in the above algorithm.

- For classification, you can use one of the algorithms for linear models from Chapter 3 to find a set of weights $\mathbf{w}$ that separate the positive points in Z from the negative points; the linear programming algorithm in Problem 3.6 would also work well.
- For regression, you can use the analytic pseudo-inverse algorithm

$$\mathbf{w}^* = \mathrm{Z}^\dagger \mathbf{y} = (\mathrm{Z}^\mathrm{T}\mathrm{Z})^{-1}\mathrm{Z}^\mathrm{T}\mathbf{y},$$

where the last expression is valid when Z has full column rank. Recall also the discussion about overfitting from Chapter 4. It is prudent to use regularization when there is stochastic or deterministic noise. With regularization parameter λ, the solution becomes $\mathbf{w}^* = (\mathrm{Z}^{\mathrm{T}}\mathrm{Z}+\lambda\mathrm{I})^{-1}\mathrm{Z}^{\mathrm{T}}\mathbf{y}$; the regularization parameter can be selected with cross validation.

- For logistic regression, you can obtain $\mathbf{w}^*$ by minimizing the logistic regression cross entropy error, using, for example, gradient descent. More details can be found in Chapter 3.

In all cases, the meat of the RBF-network is in determining the centers $\boldsymbol{\mu}_j$ and computing the radial basis feature matrix Z. From that point on, it is a straightforward linear model.

(a) $k = 4$ (b) $k = 10$ (c) $k = 10$, regularized

Figure 6.11: The parametric RBF-network using the same noisy data from Figure 6.7. (a) Small k. (b) Large k results in overfitting. (c) Large k with regularized linear fitting of weights ($\lambda = 0.05$). The centers μ_j are the gray shaded circles and $r = 1/k$. The target function is the light gray curve.

Figure 6.11 illustrates the RBF-network on the noisy data from Figure 6.7. We (arbitrarily) selected the centers μ_j by partitioning the data points into k equally sized groups and taking the average data point in each group as a center. When $k = 4$, one recovers a decent fit to f even though our centers were chosen somewhat arbitrarily. When $k = 10$, there is clearly overfitting, and regularization is called for.

Given the centers, fitting a RBF-network reduces to a linear problem, for which we have good algorithms. We now address this first step, namely how to determine these centers. Determining the optimal bump centers is the hard part, because this is where the model becomes essentially nonlinear. Luckily, the physical interpretation of this hypothesis set as a set of bumps centered on the $\boldsymbol{\mu}_j$ helps us. When there were N bumps (the nonparametric case), the natural choice was to place one bump on top of each data point. Now that there are only $k \ll N$ bumps, we should still choose the bump centers to closely represent the inputs in the data (we only need to match the $\mathbf{x}_n$, so we do not need to know the y_n). This is an *unsupervised* learning problem; in fact, a very important one.

One way to formulate the task is to require that no $\mathbf{x}_n$ be far away from a bump center. The $\mathbf{x}_n$ should cluster around the centers, with each center $\boldsymbol{\mu}_j$ representing one cluster. This is known as the *k-center problem*, known to be NP-hard. Nevertheless, we need a computationally efficient way to partition the data into k clusters and pick representative centers $\{\boldsymbol{\mu}_j\}_{j=1}^k$, one center for each cluster. One thing we learn from Figure 6.11 is that even for somewhat arbitrarily selected centers, the final result is still quite good, and so we don't have to find the absolutely optimal way to represent the data using k centers; a decent approximation will do. The k-means clustering algorithm is one way to pick a set of centers that are a decent representation of the data.

6.3.3 Unsupervised k-Means Clustering

Clustering is one of the classic unsupervised learning tasks. The k-means clustering algorithm is a simple yet powerful tool for obtaining clusters and cluster centers. We illustrate a clustering of 500 of the digits data into ten clusters. It is instructive to compare the clusters obtained here in an unsupervised manner with the classification regions of the k-NN rule in Figure 6.6(b). The clusters seem to mimic to some extent the classification regions, indicating that there is significant information in the input data alone.

We already saw the basic algorithm when we discussed partitioning for the branch and bound approach to finding the nearest neighbor. The goal of k-means clustering is to partition the input data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ into k sets $S_1, \ldots, S_k$ and select centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ for each cluster. The centers are representative of the data if every data point in cluster S_j is close to its corresponding center $\boldsymbol{\mu}_j$. For cluster S_j with center $\boldsymbol{\mu}_j$, define the squared error measure E_j to quantify the quality of the cluster,

$$E_j = \sum_{\mathbf{x}_n \in S_j} \|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2.$$

The error E_j measures how well the center $\boldsymbol{\mu}_j$ approximates the points in S_j. The k-means error function just sums this cluster error over all clusters,

$$E_{\text{in}}(S_1, \ldots, S_k; \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k) = \sum_{j=1}^{k} E_j = \sum_{n=1}^{N} \|\mathbf{x}_n - \boldsymbol{\mu}(\mathbf{x}_n)\|^2, \tag{6.5}$$

where $\boldsymbol{\mu}(\mathbf{x}_n)$ is the center of the cluster to which $\mathbf{x}_n$ belongs. We seek the partition $S_1, \ldots, S_k$ and centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ that minimize the k-means error.

Exercise 6.13

(a) Fix the clusters to $S_1, \ldots, S_k$. Show that the centers which minimize $E_{\text{in}}(S_1, \ldots, S_k; \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k)$ are the centroids of the clusters:

$$\boldsymbol{\mu}_j = \frac{1}{|S_j|} \sum_{\mathbf{x}_n \in S_j} \mathbf{x}_n.$$

(b) Fix the centers to $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$. Show that the clusters which minimize $E_{\text{in}}(S_1, \ldots, S_k; \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k)$ are obtained by placing into S_j all points for which the closest center is $\boldsymbol{\mu}_j$, breaking ties arbitrarily:

$$S_j = \{\mathbf{x}_n : \|\mathbf{x}_n - \boldsymbol{\mu}_j\| \leq \|\mathbf{x}_n - \boldsymbol{\mu}_\ell\| \text{ for } \ell = 1, \ldots, k\}.$$

Minimizing the k-means error is an NP-hard problem. However, the previous exercise says that if we fix a partition, then the optimal centers are easy to obtain. Similarly, if we fix the centers, then the optimal partition is easy to obtain. This suggests a very simple iterative algorithm which is known as *Lloyd's algorithm*. The algorithm starts with candidate centers and iteratively improves them until convergence.

Lloyd's Algorithm for k-Means Clustering:

1: Initialize $\boldsymbol{\mu}_j$ (e.g. using the greedy approach on page 16).
2: Construct S_j to be all points closest to $\boldsymbol{\mu}_j$.
3: Update each $\boldsymbol{\mu}_j$ to equal the centroid of S_j.
4: Repeat steps 2 and 3 until E_{in} stops decreasing.

Lloyd's algorithm was the algorithm used to cluster the digits data into the ten clusters that were shown in the previous figure.

Exercise 6.14

Show that steps 2 and 3 in Lloyd's algorithm can never increase E_{in}, and hence that the algorithm must eventually stop iterating. *[Hint: There are only a finite number of different partitions]*

Lloyd's algorithm produces a partition which is 'locally optimal' in an unusual sense: given the centers, there is no better way to assign the points to clusters defined by the centers; and, given the partition, there is no better choice for the centers. However, the algorithm need not find the optimal partition as one might be able to improve the k-means error by simultaneously changing the centers and the cluster memberships. Lloyd's algorithm falls into a class of algorithms known as *E-M* (expectation-maximization) algorithms. It minimizes a complex error function by separating the variables to be optimized into two sets. If one set is known, then it is easy to optimize the other set, which is the basis for an iterative algorithm, such as with Lloyd's algorithm. It

(a) 10- center RBF-network (b) 300-center RBF-network

Figure 6.12: The RBF-network model for classifying the digits data ('1' versus 'not 1') with (a) 10 centers and (b) 300 centers. The centers are first obtained by the k-means clustering algorithm. The resulting classifier is obtained after fitting the weights using the linear regression algorithm for classification (see Chapter 3). The figures highlight the possibility of overfitting with too many centers.

is an on-going area of theoretical research to find algorithms which can guarantee near optimal clustering. There are some algorithms which achieve this kind of accuracy and run in $O(n2^{O(d)})$ time (curse of dimensionality). For our practical purposes, we do not need the best clustering *per se*; we just need a good clustering that is representative of the data, because we still have some flexibility to fit the data by choosing the weights w_j. Lloyd's algorithm works just fine in practice, and it is quite efficient. In Figure 6.12, we show the RBF-network classifier for predicting '1' versus 'not 1' when using 10 centers versus 300 centers. In both cases, the cluster centers were determined using Lloyd's algorithm. Just as with linear models, one can overfit with RBF-networks. To get the best classifier, one should use regularization and validation to combat overfitting.

Selecting the Number of Clusters k. We introduced clustering as a means to determine the centers of the RBF-network in supervised learning. The ultimate goal is out-of-sample prediction, so we need to choose k (the number of centers) to give enough flexibility without overfitting. In our supervised setting, we can minimize a cross validation error to choose k. However, more generally, clustering is a stand alone unsupervised task. With our digits data, a natural choice for k is ten, one cluster for each digit. Algorithms to choose k using only the unlabeled data without knowing the number of classes in the supervised task are also part of on-going research.

6.3.4 Justifications and Extensions of RBFs

We 'derived' RBFs as a natural extension to the k-nearest neighbor algorithm; a soft version of k-nearest neighbors, if you will. There are other ways to derive them. RBF-networks arise naturally through Tikhonov regularization for fitting nonlinear functions to the data – Tikhonov regularization penalizes curvature in the final hypothesis (see Problem 6.20). RBF-networks also arise naturally from noisy interpolation theory where one tries to achieve minimum expected in-sample error under the assumption that the $\mathbf{x}$-values are noisy; this is similar to regularization where one asks g to be nearly a constant equal to y_n in the neighborhood of $\mathbf{x}_n$.

Our RBF-network had k identical bumps. A natural way to extend this model is to allow the bumps to have different shapes. This can be accomplished by choosing different scale parameters for each bump (so r_j instead of r). Further, the bumps need not be spherically symmetric. For the Gaussian kernel, the hypothesis for this more general RBF-network has the form:

$$h(\mathbf{x}) = \sum_{j=1}^{k} w_j e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_j)^{\mathrm{T}}\Sigma_j^{-1}(\mathbf{x}-\boldsymbol{\mu}_j)}. \tag{6.6}$$

The weights $\{w_j\}_{j=1}^k$, the centers $\{\boldsymbol{\mu}_j\}_{j=1}^k$ (vectors in $\mathbb{R}^d$) and the shapes $\{\Sigma_j\}_{j=1}^k$ ($d \times d$ positive definite symmetric covariance matrices) all need to be learned from the data. Intuitively, each bump j still represents a cluster of data centered at $\boldsymbol{\mu}_j$, but now the scale r is replaced by a 'scale matrix' Σ_j which captures the scale and correlations of points in the cluster. For the Gaussian kernel, the $\boldsymbol{\mu}_j$ are the cluster centroids, and the Σ_j are the cluster covariance matrices. We can fit the w_j using techniques for linear models, once the $\boldsymbol{\mu}_j$ and Σ_j are given. As with the simpler RBF-network, the location and shape parameters can be learned in an unsupervised way; an E-M algorithm similar to Lloyd's algorithm can be applied (Section 6.4.1 elaborates on the details). For Gaussian kernels, this unsupervised learning task is called learning a Gaussian Mixture Model (GMM).

6.4 Probability Density Estimation

Clustering was our first unsupervised method that tried to organize the input data. A cluster contains points that are similar to each other. The probability density of $\mathbf{x}$ is a generalization of clustering to a finer representation. Clusters can be thought of as regions of high probability. The basic task in probability density estimation is to estimate, for a given $\mathbf{x}$, how likely it is that you would generate inputs *similar* to $\mathbf{x}$. To answer this question we need to look at what fraction of the inputs in the data are similar to $\mathbf{x}$. Since similarity plays an important role, many of the similarity-based techniques in supervised learning have counterparts in the unsupervised task of probability density estimation. Vast tomes have been written on this topic, so we only skim its surface.

The Histogram. Given data $\mathbf{x}_1, \ldots, \mathbf{x}_N$ generated independently from $P(\mathbf{x})$, the task is to learn the entire probability density P. Since $P(\mathbf{x})$ is the density of points at $\mathbf{x}$, the natural estimate is to use the in-sample density of points around $\mathbf{x}$. The *histogram* density estimate illustrated on the right uses a partition of $\mathcal{X}$ into uniform disjoint hypercubes (multidimensional intervals). Every point $\mathbf{x}$ falls into one bin and each bin has volume V. In one dimension, V is just the length of the interval. The estimate $\hat{P}(\mathbf{x})$ is the density of points in the hypercube that contains $\mathbf{x}$, normalized so that the integral is 1.

$$\hat{P}(\mathbf{x}) = \frac{1}{N} \cdot \frac{N(\mathbf{x})}{V},$$

where $N(\mathbf{x})$ is the number of data points in the hypercube containing $\mathbf{x}$. The density of points in this hypercube containing $\mathbf{x}$ is $N(\mathbf{x})/V$. The normalization by $1/N$ is to ensure that $\int d\mathbf{x}\ \hat{P}(\mathbf{x}) = 1$. The histogram estimate $\hat{P}(\mathbf{x})$ is not continuous. The main parameter which controls the quality of the estimate is V (or the number of bins, which is inversely related to V). Under mild assumptions, one recovers the true density as $V \to 0$ and $N \cdot V \to \infty$. This ensures that you are only using points very similar to $\mathbf{x}$ to estimate the density at $\mathbf{x}$ and there are enough such points being used. One choice is $V = 1/\sqrt{N}$.

Nearest Neighbor Density Estimation. Rather than using uniform hypercubes, one can use the distance to the kth nearest neighbor to determine the volume of the region containing $\mathbf{x}$. Let $r_{[k]}(\mathbf{x}) = \|\mathbf{x} - \mathbf{x}_{[k]}\|$ be the distance from $\mathbf{x}$ to its kth nearest neighbor, and $V_{[k]}(\mathbf{x})$ the volume of the spheroid centered at $\mathbf{x}$ of radius $r_{[k]}(\mathbf{x})$. In d-dimensions, $V_{[k]}$ and $r_{[k]}$ are related by

$$V_{[k]} = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)} {r_{[k]}}^d.$$

The density of points in this spheroid is $k/V_{[k]}$, so we have the estimate

$$\hat{P}(\mathbf{x}) \quad = \quad c \cdot \frac{k}{V_{[k]}(\mathbf{x})},$$

where c is chosen by normalizing $\hat{P}$, so that $\int \mathrm{d}\mathbf{x}\ \ \hat{P}(\mathbf{x}) = 1$. The 3-NN density estimate using 6 points from a uniform distribution is illustrated to the right. As is evident, the nearest neighbor density estimate is continuous but not smooth, having sharp spikes and troughs.

(a) 31-NN density estimate (b) Density contours

Figure 6.13: The 31-NN density estimate for 500 digits data. (a) The probability density surface showing the spike behavior of the estimate. (b) Density contours (red is high, blue is low) from which we can visually identify the clusters in the data as regions of locally high density.

The nearest neighbor density estimate only works with a bounded input space because otherwise the estimate is not normalizable. In fact the tail of the nearest neighbor density estimate is decaying as $\|\mathbf{x}\|^{-d}$ which is fat-tailed. The nearest neighbor density estimate is nonparametric, and the usual convergence theorem holds. If $k \to \infty$ and $k/N \to 0$, then under mild assumptions on P, $\hat{P}$ converges to P (where the discrepancy between P and $\hat{P}$ is measured by the integrated squared error, $\int \mathrm{d}\mathbf{x} \;\; (P(\mathbf{x}) - \hat{P}(\mathbf{x}))^2$). Figure 6.13 illustrates the nearest neighbor density estimate using 500 randomly sampled digits data.

Parzen Windows: RBF Density Estimation. Perhaps the most common density estimation technique is the Parzen window. Recall that the kernel ϕ is a bump function which is positive. For density estimation, it is convenient to normalize ϕ so that $\int d\mathbf{x}\ \phi(\|\mathbf{x}\|) = 1$. The normalized Gaussian kernel is

$$\phi(z) = \frac{1}{(2\pi)^{d/2}} e^{-\frac{1}{2}z^2}.$$

One can verify that $\hat{P}(\mathbf{x}) = \phi(\|\mathbf{x}\|)$ is a probability density, and for any $r > 0$,

$$\hat{P}(\mathbf{x}) = \frac{1}{r^d} \cdot \phi\left(\frac{\|\mathbf{x}\|}{r}\right)$$

is also a density (r is the width of the bump). The Parzen window is similar to the nonparametric RBF in supervised learning: you have a bump with weight $\frac{1}{N}$ on each data point, and $\hat{P}(\mathbf{x})$ is a sum of the bumps:

(a) RBF density estimate (b) Density contours

Figure 6.14: Gaussian kernel Parzen window with $r = 0.02$ using 500 digits data. (a) The probability density surface is smooth but bumpy around the periphery of the data; the tail is exponential, dropping very quickly to zero away from the data. Compare this with the spiky more slowly decaying density produced by nearest neighbor in Figure 6.13. (b) Density contours (red is high, blue is low) highlighting the clusters in the data as regions of locally high density. Clusters are more readily visible with the smoother Parzen window than the nearest neighbor estimate.

$$\hat{P}(\mathbf{x}) = \frac{1}{Nr^d} \sum_{i=1}^{N} \phi\left(\frac{\|\mathbf{x} - \mathbf{x}_i\|}{r}\right).$$

Since each bump integrates to 1, the scaling by $\frac{1}{N}$ ensures that $\hat{P}(\mathbf{x})$ integrates to 1. The figure to the right illustrates the Gaussian kernel Parzen window with six points from a uniform density. As opposed to the nearest neighbor density, the Parzen window is smooth, and has a thin tail. These properties are inherited from the Gaussian kernel, which is smooth and exponentially decaying. Figure 6.14 shows the Gaussian kernel Parzen window applied to our 500 randomly sampled digits data, with $r = 0.02$. Observe that the Parzen window can be bumpy around the periphery of the data. We can reduce this bumpiness by increasing r, which risks missing the finer structure at the heart of the data.

As with nearest neighbors, the Parzen window is nonparametric, modulo the choice of the kernel-width. If $r \to 0$ and $r^d N \to \infty$ as N grows, then under mild assumptions, $\hat{P}$ converges to P with respect to integrated square error. One choice for r is $1/\sqrt[2d]{N}$. One can also use cross validation to determine a good value of r that maximizes the likelihood of the validation set (recall that we used likelihood to derive the error function for logistic regression).

6.4.1 Gaussian Mixture Models (GMMs)

Just as the RBF-network is the parametric version of the nonparametric RBF, the Gaussian mixture model (GMM) is the parametric version of the Parzen window density estimate. The Parzen window estimate places a Gaussian bump at every data point; the GMM places just k bumps at centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$. The Gaussian kernel is the most commonly used and easiest to handle. In d-dimensions, the Gaussian density with center $\boldsymbol{\mu}$ and covariance matrix Σ is:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})}.$$

Note that the center $\boldsymbol{\mu}$ is also the expected value, $\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}$; and the covariance between two features x_i, x_j is Σ_{ij}, $\mathbb{E}[(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}] = \Sigma$.

To derive the GMM, we use the following simple model of how the data is generated. There are k Gaussian distributions, with respective means $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ and covariance matrices $\Sigma_1, \ldots, \Sigma_k$. To generate a data point $\mathbf{x}$, first pick a Gaussian $j \in \{1, \ldots, k\}$ according to probabilities $\{w_1, \ldots, w_k\}$, where $w_j > 0$ and $\sum_{j=1}^{k} w_j = 1$. After selecting Gaussian j, $\mathbf{x}$ is generated according to a Gaussian distribution with parameters $\boldsymbol{\mu}_j, \Sigma_j$. The probability density of $\mathbf{x}$ given that you picked Gaussian j is

$$P(\mathbf{x}|j) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_j, \Sigma_j) = \frac{1}{(2\pi)^{d/2}|\Sigma_j|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_j)^{\mathrm{T}}\Sigma_j^{-1}(\mathbf{x}-\boldsymbol{\mu}_j)}.$$

By the law of total probability, the unconditional probability density for $\mathbf{x}$ is

$$P(\mathbf{x}) = \sum_{j=1}^{k} P(\mathbf{x}|j)\,\mathbb{P}[j] = \sum_{j=1}^{k} w_j \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_j, \Sigma_j) \tag{6.7}$$

Compare (6.7) with (6.4). The GMM is just an RBF-network with weights w_j and a Gaussian kernel – a sum of Gaussian bumps, hence the name.

Each of the k Gaussian bumps represents a 'cluster' of the data. The probability density in Equation (6.7) puts a bump (Gaussian) of total probability (weight) w_j at the center $\boldsymbol{\mu}_j$; Σ_j determines the 'shape' of the bump. An example of some data generated from a GMM in two dimensions with $k = 3$ is shown on the right. The Gaussians are illustrated by a contour of constant probability. The different shapes of the clusters are controlled by the covariances Σ_j. Equation (6.7) is based on a *generative* model for the data, and the parameters of the model need to be learned from the actual data. These parameters are the mixture weights w_j, the centers $\boldsymbol{\mu}_j$ and the covariance matrices Σ_j.

To determine the unknown parameters in the GMM, we will minimize an in-sample error called the likelihood. We already saw the likelihood when we derived the in-sample error for logistic regression in Chapter 3. Among all the possible GMMs, each determined by a different choice for the parameters in Equation (6.7), we want to select one. Our criterion for choosing is that, for the chosen density estimate $\hat{P}$, the data should have a high probability of being generated. Since the data points are independent, the probability (density) for the data $\mathbf{x}_1, \ldots, \mathbf{x}_N$ if the data were generated according to $\hat{P}(\mathbf{x})$ is

$$\hat{P}(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} \hat{P}(\mathbf{x}_n) = \prod_{n=1}^{N} \left(\sum_{j=1}^{k} w_j \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_j, \Sigma_j) \right).$$

This is the likelihood of a particular GMM specified by parameters $\{w_j, \boldsymbol{\mu}_j \Sigma_j\}$. The method of maximum likelihood selects the parameters which maximize $\hat{P}(\mathbf{x}_1, \ldots, \mathbf{x}_N)$, or equivalently which minimize $-\ln \hat{P}(\mathbf{x}_1, \ldots, \mathbf{x}_N)$. Thus, we may minimize the in-sample error:

$$\begin{aligned} E_{\text{in}}(w_j, \boldsymbol{\mu}_j, \Sigma_j) &= -\ln \hat{P}(\mathbf{x}_1, \ldots, \mathbf{x}_N) \\ &= -\sum_{n=1}^{N} \ln \left(\sum_{j=1}^{k} w_j \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_j, \Sigma_j) \right). \end{aligned} \tag{6.8}$$

To determine $\{w_j, \boldsymbol{\mu}_j, \Sigma_j\}$, we need an algorithm to minimize this in-sample error. Equation (6.8) is general in that we can replace the Gaussian density $\mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_j, \Sigma_j)$ with any other parameterized density and we will obtain a non-Gaussian mixture model. Unfortunately, even for the friendly Gaussian mixture model, the summation inside the logarithm makes it very difficult to minimize the in-sample error, and we are going to need a heuristic. This heuristic is the Expectation Maximization (E-M) algorithm. The E-M algorithm is efficient and produces good results. To illustrate, we run a 10-center GMM on our sample of 500 digits data. The results are shown in Figure 6.15.

The Expectation Maximization (E-M) Algorithm The E-M algorithm was introduced in the late 1970s and is an important heuristic for maximizing the likelihood function. It is based on the notion of a *latent* (hidden, unmeasured, missing) piece of data that would make the optimization much easier. In the context of the Gaussian Mixture Model, suppose we knew which data points came from bump 1, bump 2, ..., bump k. The problem would suddenly become much easier, because we can estimate the center and covariance matrix of each bump using the data from that bump alone; further we can estimate the probabilities w_j by the fraction of data points in bump j. Unfortunately we do not know which data came from which bump, so we start with a guess, and iteratively improve this guess. The general algorithm is

(a) 10-center GMM (b) Density contours

Figure 6.15: The 10-center GMM for 500 digits data. (a) The probability density surface. The bumps are no longer spherical as in the Parzen window. The bump shapes are determined by the covariance matrices. (b) Density contours (red is high, blue is low) with the centers as black dots. The centers identify the 'clusters' (compare with the k-means clusters on page 31).

E-M Algorithm for GMMs:

1: Start with estimates for the bump membership of each $\mathbf{x}_n$.
2: Estimates of $w_j, \boldsymbol{\mu}_j, \Sigma_j$ given the bump memberships.
3: Update the bump memberships given $w_j, \boldsymbol{\mu}_j, \Sigma_j$; iterate to step 2 until convergence.

To mathematically specify the algorithm, we will add one more ingredient, namely that the bump memberships need not be all or nothing. Specifically, at iteration t, let $\gamma_{nj}(t) \geq 0$ be the 'fraction' of data point $\mathbf{x}_n$ that belongs to bump j, with $\sum_{j=1}^{k} \gamma_{nj} = 1$ (the entire point is allocated among all the bumps); you can view γ_{nj} as the probability that $\mathbf{x}_n$ was generated by bump j. The 'number' of data points belonging to bump j is given by

$$N_j = \sum_{n=1}^{N} \gamma_{nj}.$$

The γ_{nj} are the hidden variables that we do not know, but if we did know the γ_{nj}, then we could compute estimates of $w_j, \boldsymbol{\mu}_j, \Sigma_j$:

$$\begin{aligned} w_j &= \frac{N_j}{N}; \\ \boldsymbol{\mu}_j &= \frac{1}{N_j} \sum_{n=1}^{N} \gamma_{nj} \mathbf{x}_n; \\ \Sigma_j &= \frac{1}{N_j} \sum_{n=1}^{N} \gamma_{nj} \mathbf{x}_n \mathbf{x}_n^{\mathrm{T}} - \boldsymbol{\mu}_j \boldsymbol{\mu}_j^{\mathrm{T}}. \end{aligned} \tag{6.9}$$

Intuitively, the weights are the fraction of data belonging to bump j; the means are the average data point belonging to bump j where we take into account that only a fraction of a data point may belong to a bump; and, the covariance matrix is the weighted covariance of the data belonging to the bump. Once we have these new estimates of the parameters, we can update the bump memberships γ_{nj}. To get $\gamma_{nj}(t+1)$, we compute the probability that data point $\mathbf{x}_n$ came from bump j *given* the parameters $w_j, \boldsymbol{\mu}_j, \Sigma_j$. We want

$$\gamma_{nj}(t+1) = \mathbb{P}[j|\mathbf{x}_n].$$

By an application of Bayes rule,

$$\mathbb{P}[j|\mathbf{x}_n] = \frac{P(\mathbf{x}_n|j)\,\mathbb{P}\,[j]}{P(\mathbf{x}_n)} = \frac{\mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_j, \Sigma_j) \cdot w_j}{P(\mathbf{x}_n)}.$$

We won't have to compute $P(\mathbf{x}_n)$ in the denominator because it is independent of j and can be fixed by the normalization condition $\sum_{j=1}^{k} \gamma_{nj} = 1$. We thus arrive at the update for the bump memberships,

$$\gamma_{nj}(t+1) = \frac{w_j \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_j, \Sigma_j)}{\sum_{\ell=1}^{k} w_\ell \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_\ell, \Sigma_\ell)}.$$

After updating the bump memberships, we iterate back to the parameters and continue until convergence. To complete the specification of the algorithm, we need to specify the initial bump memberships. A simple heuristic is to use the k-means clustering algorithm described earlier to obtain a 'hard' partition with $\gamma_{nj} \in \{0, 1\}$, and proceed from there.

Exercise 6.15

What would happen in the E-M algorithm described above if you initialized the bump memberships uniformly to $\gamma_{nj} = 1/k$?

The E-M algorithm is a remarkable example of a learning algorithm that 'bootstraps' itself to a solution. The algorithm starts by guessing some values for unknown quantities that you would like to know. The guess is probably quite wrong, but nevertheless the algorithm makes inferences based on the incorrect guess. These inferences are used to slightly improve the guess. The guess and inferences slightly improve each other in this way until at the end you have bootstrapped yourself to a decent guess at the unknowns, as well as a good inference based on those unknowns.

6.5 Problems

Problem 6.1 Consider the following data set with 7 data points.

$$\left(\begin{bmatrix}1\\0\end{bmatrix},-1\right)\left(\begin{bmatrix}0\\1\end{bmatrix},-1\right)\left(\begin{bmatrix}0\\-1\end{bmatrix},-1\right)\left(\begin{bmatrix}-1\\0\end{bmatrix},-1\right)$$
$$\left(\begin{bmatrix}0\\2\end{bmatrix},+1\right)\left(\begin{bmatrix}0\\-2\end{bmatrix},+1\right)\left(\begin{bmatrix}-2\\0\end{bmatrix},+1\right)$$

(a) Show the decision regions for the 1-NN and 3-NN rules.

(b) Consider the non-linear transform

$$\begin{bmatrix}x_1\\x_2\end{bmatrix}\mapsto\begin{bmatrix}z_1\\z_2\end{bmatrix}=\begin{bmatrix}\sqrt{x_1^2+x_2^2}\\ \arctan(x_2/x_1)\end{bmatrix},$$

which maps $\mathbf{x}$ to $\mathbf{z}$. Show the classification regions in the $\mathbf{x}$-space for the 1-NN and 3-NN rules implemented on the data in the $\mathbf{z}$-space.

Problem 6.2 Use the same data from the previous problem.

(a) Let the mean of all the -1 points be $\boldsymbol{\mu}_{-1}$ and the mean of all the +1 points be $\boldsymbol{\mu}_{+1}$. Suppose the data set were condensed into the two *prototypes* $\{(\boldsymbol{\mu}_{-1},-1),\ (\boldsymbol{\mu}_{+1},+1)\}$ (these points need not be data points, so they are called prototypes). Plot the classification regions for the 1-NN rule using the condensed data. What is the in-sample error?

(b) Consider the following approach to condensing the data. At each step, merge the two closest points of the same class as follows:

$$(\mathbf{x},c)+(\mathbf{x}',c)\to(\tfrac{1}{2}(\mathbf{x}+\mathbf{x}'),c).$$

Again, this method of condensing produces prototypes. Continue condensing until you have two points remaining (of different classes).

Plot the 1-NN rule with the condensed data. What is the in-sample error?

Problem 6.3 Show that the k-nearest neighbor rule with distance defined by $d(\mathbf{x},\mathbf{x}')=(\mathbf{x}-\mathbf{x}')^{\mathrm{T}}\mathrm{Q}(\mathbf{x}-\mathbf{x}')$, where Q is positive semi-definite, is equivalent to the k-nearest neighbor rule with the standard Euclidean distance in some transformed feature space. Explicitly construct this space. What is the dimension of this space. *[Hint: Think about the rank of* Q*.]*

Problem 6.4 For the double semi-circle problem in Problem 3.1, plot the decision regions for the 1-NN and 3-NN rules.

Problem 6.5 Show that each of the Voronoi regions in the Voronoi diagram for any data set is convex. (A set $\mathcal{C}$ is convex if for any $\mathbf{x}, \mathbf{x}' \in \mathcal{C}$ and any $\lambda \in [0,1]$, $\lambda\mathbf{x} + (1-\lambda)\mathbf{x}' \in \mathcal{C}$.)

Problem 6.6 For linear regression with weight decay, $g(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathbf{w}_{\mathrm{reg}}$. Show that

$$g(\mathbf{x}) = \sum_{n=1}^{N} \mathbf{x}^{\mathrm{T}}(\mathrm{Z}^{\mathrm{T}}\mathrm{Z} + \lambda\Gamma^{\mathrm{T}}\Gamma)^{-1}\mathbf{x}_n y_n.$$

A *kernel representation* of a hypothesis g is a representation of the form

$$g(\mathbf{x}) = \sum_{n=1}^{N} K(\mathbf{x}, \mathbf{x}_n) y_n,$$

where $K(\mathbf{x}, \mathbf{x}')$ is the kernel function. What is the kernel function in this case?

One can interpret the kernel representation of the final hypothesis from linear regression as a similarity method, where g is a weighted sum of the target values $\{y_n\}$, weighted by the "similarity" $K(\mathbf{x}, \mathbf{x}_n)$ between the point $\mathbf{x}$ and the data point $\mathbf{x}_n$. Does this look similar ☺ to RBFs?

Problem 6.7 Consider the hypothesis set $\mathcal{H}$ which contains all labeled Voronoi tessellations on K points. Show that $d_{\mathrm{vc}}(\mathcal{H}) = K$.

Problem 6.8 Suppose the target function is deterministic, so $\pi(\mathbf{x}) = 0$ or $\pi(\mathbf{x}) = 1$. The decision boundary implemented by f is defined as follows. The point $\mathbf{x}$ is on the decision boundary if every ball of positive radius centered on $\mathbf{x}$ contains both positive and negative points. Conversely if $\mathbf{x}$ is not on the decision boundary, then some ball around $\mathbf{x}$ contains only points of one classification.

Suppose the decision boundary (a set of points) has probability zero. Show that the simple nearest neighbor rule will asymptotically (in N) converge to optimal error E^*_{out} (with probability 1).

Problem 6.9 Assume that the support of P is the unit cube in d dimensions. Show that for any $\epsilon, \delta > 0$, there is a sufficiently large N for which, with probability at least $1 - \delta$,

$$\sup_{\mathbf{x} \in \mathcal{X}} \|\mathbf{x} - \mathbf{x}_{(1)}(\mathbf{x})\| \leq \epsilon.$$

[Hint: Partition $[0,1]^d$ into disjoint hypercubes of size $\epsilon/\sqrt{d}$. For any $\mathbf{x} \in [0,1]^d$, show that at least one of these hypercubes is completely contained in

$B_\epsilon(\mathbf{x})$. Let $p > 0$ be the minimum P-measure of a hypercube. Since the number of hypercubes is finite, use a union bound to show that with high probability, every hypercube will contain at least k points for any fixed k as $N \to \infty$. Conclude that $\|\mathbf{x} - \mathbf{x}_{(k)}\| \leq \epsilon$ with high probability as $N \to \infty$.]

Problem 6.10 Let $E_{\text{out}}(k) = \lim_{N\to\infty} E_{\text{out}}(g_N(k))$, where $g_N(k)$ is the k-nearest neighbor rule on a data set of size N, where k is odd. Let E^*_{out} is the optimal out-of-sample probability of error. Show that (with probability 1),

$$E^*_{\text{out}} \leq E_{\text{out}}(k) \leq E_{\text{out}}(k-2) \leq \cdots \leq E_{\text{out}}(1) \leq 2E^*_{\text{out}}.$$

Problem 6.11 For the 1-NN rule in two dimensions ($d = 2$) and data set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, consider the Voronoi diagram and let V_n be the Voronoi region containing the point $\mathbf{x}_n$. Two Voronoi regions are adjacent if they have a face in common (in this case an edge). Mark a point $\mathbf{x}_n$ if the classification of every Voronoi region neighboring V_n is the same as y_n. Now condense the data by removing all marked points.

(a) Show that the condensed data is consistent with the full data for the 1-NN rule.

(b) How does the out-of-sample error for the 1-NN rule using condensed data compare with the 1-NN rule on the full data (worst case and on average)?

Problem 6.12 For the 1-NN rule, define the *influence set* S_n of point $\mathbf{x}_n$ as the set of points of the same class as $\mathbf{x}_n$ which are closer to $\mathbf{x}_n$ than to a point of a different class. So $\mathbf{x}_m \in S_n$ if

$$\|\mathbf{x}_n - \mathbf{x}_m\| < \|\mathbf{x}_{m'} - \mathbf{x}_m\| \text{ for all } m' \text{ with } y_{m'} \neq y_m.$$

We do not allow $\mathbf{x}_n$ to be in S_n. Suppose the largest influence set is S_{n^*}, the influence set of $\mathbf{x}_{n^*}$ (break ties according to the data point index). Remove all points in S_{n^*}. Update the remaining influence sets by deleting $\mathbf{x}_{n^*}$ *and* S_{n^*} from them. Repeat this process until all the influence sets are empty. The set of points remaining is a condensed subset of $\mathcal{D}$.

(a) Show that the remaining condensed set is training set consistent.

(b) Do you think this will give a smaller or larger condensed set than the CNN algorithm discussed in the text. Explain your intuition?

(c) Give an efficient algorithm to implement this greedy condensing. What is your run time.

Problem 6.13 Construct a data set with 1,000 points as shown.

The data are generated from a 4-center GMM. The centers are equally spaced on the unit circle, and covariance matrices all equal to $\sigma \mathrm{I}$ with $\sigma = 0.15$. The weight of each Gaussian bump is $\frac{1}{4}$. Points generated from the first and third centers have $y_n = +1$ and the other two centers are -1. To generate a point, first determine a bump (each has probability $\frac{1}{4}$) and then generate a point from the Gaussian for that bump.

(a) Implement the CNN algorithm in the text and the condensing algorithm from the previous problem.

(b) Run your algorithm on your generated data and give a plot of the condensed data in each case.

(c) Repeat this experiment 1,000 times and compute the average sizes of the condensed sets for the two algorithms.

Problem 6.14 Run the condensed nearest neighbor (CNN) algorithm for 3-NN on the digits data. Use all the data for classifying "1" versus "not 1".

(a) Set $N = 500$ and randomly split your data into a training set of size N and use the remaining data as a test/validation set.

(b) Use the 3-NN algorithm with all the training data and evaluate its performance: report E_{in} and E_{test}.

(c) Use the CNN algorithm to condense the data. Evaluate the performance of the 3-NN rule with the condensed data: report E_{in} and E_{out}.

(d) Repeat parts (b) and (c) using 1,000 random training-test splits and report the average E_{in} and E_{out} for the full versus the condensed data.

Problem 6.15 This problem asks you to perform an analysis of the branch and bound algorithm in an idealized setting. Assume the data is partitioned and each cluster with two or more points is partitioned into two sub-clusters of exactly equal size. (The number of data points is a power of 2).

When you run the branch and bound algorithm for a particular test point $\mathbf{x}$, sometimes the bound condition will hold, and sometimes not. If you generated the test point $\mathbf{x}$ randomly, the bound condition will hold with some probability. Assume the bound condition will hold with probability at least $p \geq 0$ at *every* branch, independently of what happened at other branches.

Let $T(N)$ be the expected time to find the nearest neighbor in a cluster with N points. Show: $T(N) = O(d \log N + dN^{\log_2(2-p)})$ (sublinear for $p > 0$).

[Hint: Let $N = 2^k$; show that $T(N) \leq 2d + T(\frac{N}{2}) + (1-p)T(\frac{N}{2})$.]

Problem 6.16

(a) Generate a data set of 10,000 data points uniformly in the unit square $[0,1]^2$ to test the performance of the branch and bound method:

  (i) Construct a 10-partition for the data using the simple greedy heuristic described in the text.

  (ii) Generate 10,000 random query points and compare the running time of obtaining the nearest neighbor using the partition with branch and bound versus the brute force approach which does not use the partition.

(b) Repeat (a) but instead generate the data from a mixture of 10 gaussians with centers randomly distributed in $[0,1]^2$ and identical covariances for each bump equal to $\sigma \mathrm{I}$ where $\sigma = 0.1$.

(c) Explain your observations.

(d) Does your decision to use the branch and bound technique depend on how many test points you will need to evaluate?

Problem 6.17 Using Exercise 6.8, assume that $p = \frac{1}{2}$ (the probability that the bound condition will hold at any branch point). Estimate the asymptotic running time to estimate the out-of-sample error with 10-fold cross validation to select the value of k, and compare with Exercise 6.6.

Problem 6.18 An alternative to the k-nearest neighbor rule is the r-nearest neighbor rule: classify a test point $\mathbf{x}$ using the majority class among all neighbors $\mathbf{x}_n$ within distance r of $\mathbf{x}$. The r-nearest neighbor explicitly enforces that all neighbors contributing to the decision must be close; however, it does not mean that there will be many such neighbors. Assume that the input density is bounded from above and below, $L \leq P(\mathbf{x}) \leq U$.

(a) Show that the expected number of neighbors contributing to the decision for any particular $\mathbf{x}$ is order of Nr^d.

(b) Argue that as N grows, if $Nr^d \to \infty$ and $r \to 0$, then the classifier approaches optimal.

(c) Give one example of such a choice for r (as a function of N, d).

Problem 6.19 For the full RBFN in Equation (6.6) give the details of a 2-stage algorithm to fit the model to the data. The first stage determines the parameters of the bumps (their centers and covariance matrices); and, the second stage determines the weights. *[Hint: For the first stage, think about the E-M algorithm to learn a Gaussian mixture model.]*

Problem 6.20 **[RBFs from regularization]** This problem requires advanced calculus. Let $d = 1$; we wish to minimize a regularized error

$$E_{\text{aug}}(h) = \sum_{i=1}^{N}(h(x_i) - y_i)^2 + \lambda \sum_{k=0}^{\infty} a_k \int_{-\infty}^{\infty} dx \ \left(h^{(k)}(x)\right)^2,$$

where λ is the regularization parameter. Assume $h^{(k)}(x)$ (the kth derivative of h) decays to to 0 as $|x| \to \infty$. Let $\delta(x)$ be the Dirac delta function.

(a) How would you select a_k to penalize how curvy or wiggly h is?

(b) [Calculus of Variations] Show that

$$E_{\text{aug}}(h) = \int_{-\infty}^{\infty} dx \ \left[\sum_{i=1}^{N}(h(x) - y_i)^2 \delta(x - x_i) + \lambda \sum_{k=0}^{\infty} a_k \left(h^{(k)}(x)\right)^2\right].$$

Now find a stationary point of this functional: perturb h by δh and assume that for all k, $\delta h^{(k)} \to 0$ as $|x| \to 0$. Compute the change $\delta E_{\text{aug}}(h) = E_{\text{aug}}(h + \delta h) - E_{\text{aug}}(h)$ and set it to 0. After integrating by parts and discarding all terms of higher order than linear in δh, and setting all boundary terms from the integration by parts to 0, derive the following condition for h to be stationary (since δh is arbitrary),

$$\sum_{i=1}^{N}(h(x) - y_i)\delta(x - x_i) + \lambda \sum_{k=0}^{\infty}(-1)^k a_k h^{(2k)}(x) = 0.$$

(c) [Green's Functions] $\mathrm{L} = \sum_{k=0}^{\infty}(-1)^k a_k \frac{d^k}{dx^k}$ is a linear differential operator. The Green's function $G(x, x')$ for L satisfies $\mathrm{L}G(x, x') = \delta(x - x')$. Show that we can satisfy the stationarity condition by choosing

$$h(x) = \sum_{i=1}^{N} w_i G(x, x_i),$$

with $\mathbf{w} = (\mathrm{G} + \lambda \mathrm{I})^{-1}\mathbf{y}$, where $\mathrm{G}_{ij} = G(x_i, x_j)$. ($h$ resembles an RBF with the Green's function as kernel. If L is translation and rotation invariant then $G(\mathbf{x}, \mathbf{x}') = G(\|\mathbf{x} - \mathbf{x}'\|)$, and we have an RBF.)

(d) [Computing the Green's Function] Solve $\mathrm{L}G(x, x') = \delta(x - x')$ to get G. Define the Fourier transform and its inverse,

$$\hat{G}(f, x') = \int_{-\infty}^{\infty} dx \, e^{2\pi i f x} G(x, x'), \qquad \hat{G}(x, x') = \int_{-\infty}^{\infty} df \, e^{-2\pi i f x} \hat{G}(f, x').$$

Fourier transform both sides of $\mathrm{L}G(x, x') = \delta(x - x')$, integrate by parts, assuming that the boundary terms vanish, and show that

$$G(x, x') = \int_{-\infty}^{\infty} df \frac{e^{2\pi i f (x' - x)}}{Q(f)},$$

where $Q(f) = \sum_{k=0}^{\infty} a_k (2\pi f)^{2k}$ is an even function whose power series expansion is determined by the a_k. If $a_k = 1/(2^k k!)$, what is $Q(f)$. Show that in this case the Green's function is the Gaussian kernel,

$$G(x, x') = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(x-x')^2}.$$

(For regularization that penalizes a particular combination of the derivatives of h, the *optimal* non-parametric regularized fit is a Gaussian kernel RBF.) *[Hint: You may need:* $\int_{-\infty}^{\infty} dt\, e^{-at^2+bt} = \sqrt{\frac{\pi}{a}} e^{b^2/4a}$, $Re(a) > 0$.*]*

Problem 6.21 Develop a linear programming approach to classification with similarity oracle $d(\cdot, \cdot)$ (as in Problem 3.6). Assume RBF-like hypotheses:

$$h(\mathbf{x}) = \text{sign}\left(w_0 + \sum_{i=1}^{N} w_i d(\mathbf{x}, \mathbf{x}_i)\right),$$

where $\mathbf{w}$ is the weight parameter to be determined by fitting the data. Pick the weights that fit the data and minimize the sum of weight sizes $\sum_{i=1}^{N} |w_i|$ (*lasso* regularization where we don't penalize w_0).

(a) Show that to find the weights, one solves the minimization problem:

$$\underset{\mathbf{w}}{\text{minimize}} \sum_{i=0}^{N} |w_i| \quad \text{s.t. } y_n \left(w_0 + \sum_{i=1}^{N} w_i d(\mathbf{x}_n, \mathbf{x}_i)\right) \geq 1.$$

Do you expect overfitting?

(b) Suppose we allow some error in the separation, then

$$y_n \left(w_0 + \sum_{i=1}^{N} w_i d(\mathbf{x}_n, \mathbf{x}_i)\right) \geq 1 - \zeta_n,$$

where $\zeta_n \geq 0$ are slack variables that measure the degree to which the data point $(\mathbf{x}_n, yn)$ has been misclassified. The total error is $\sum_{n=1}^{N} \zeta_n$. If you minimize a combination of the total weight sizes and the error with emphasis C on error, then argue that the optimization problem becomes

$$\begin{aligned} \underset{\mathbf{w},\boldsymbol{\zeta}}{\text{minimize}} \quad & \sum_{n=1}^{N} |w_n| + C \sum_{n=1}^{N} \zeta_n, \\ \text{s.t.} \quad & y_n \left(w_0 + \sum_{i=1}^{N} w_n d(\mathbf{x}_n, \mathbf{x}_i)\right) \geq 1 - \zeta_n, \\ & \zeta_n \geq 0, \end{aligned} \tag{6.10}$$

where the inequalities must hold for $n = 1, \ldots, N$.

The minimization trades off sparsity of the weight vector with the extent of misclassification. To encourage smaller in-sample error, one sets C to be large.

Problem 6.22 Show that the minimization in (6.10) is a linear program:

$$\begin{aligned}
\underset{\mathbf{w},\boldsymbol{\zeta},\boldsymbol{\alpha}}{\text{minimize}} \quad & \sum_{n=1}^{N} \alpha_n + C \sum_{n=1}^{N} \zeta_n, \\
& -\alpha_n \le w_n \le \alpha_n, \\
\text{s.t.} \quad & y_n \left(w_0 + \sum_{i=1}^{N} w_n d(\mathbf{x}_n, \mathbf{x}_i) \right) \ge 1 - \zeta_n, \\
& \zeta_n \ge 0,
\end{aligned}$$

where the inequalities must hold for $n = 1, \ldots, N$. Formulate this linear program in a standard form as in Problem 3.6. You need to specify what the parameters $\mathrm{A}, \mathbf{a}, \mathbf{b}$ are and what the optimization variable $\mathbf{z}$ is.

[Hint: Use auxiliary variables $\alpha_1, \ldots, \alpha_N$ to rewrite $|w_n|$ using linear functions.]

Problem 6.23 Consider a data distribution, $P(\mathbf{x}, y)$ which is a mixture of k Gaussian distributions with means $\{\boldsymbol{\mu}_j\}_{j=1}^k$ and covariance matrices $\{\Sigma_j\}_{j=1}^k$; each Gaussian has probability $p_j > 0$ of being selected, $\sum_{j=1}^k p_j = 1$; each Gaussian generates a positive label with probability π_j. To generate $(\mathbf{x}, y)$, first select a Gaussians using probabilities $p_1, \ldots, p_k$. If Gaussian ℓ is selected, generate $\mathbf{x}$ from this Gaussian distribution, with mean $\boldsymbol{\mu}_\ell$ and covariance Σ_ℓ, and $y = +1$ with probability π_ℓ ($y = -1$ otherwise).

For test point $\mathbf{x}$, show that the classifier with minimum error probability is

$$f(\mathbf{x}) = \text{sign}\left(\sum_{j=1}^{k} w_j e^{-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_j)^{\mathrm{T}} \Sigma_j^{-1} (\mathbf{x} - \boldsymbol{\mu}_j)} \right),$$

where $w_j = p_j(2\pi_j - 1)$. *[Hint: Show that the optimal decision rule can be written $f(\mathbf{x}) = sign(\mathbb{P}[+1|\mathbf{x}] - \mathbb{P}[-1|\mathbf{x}])$. Use Bayes' theorem and simplify.]*

(This is the RBF-network for classification. Since $\boldsymbol{\mu}_j, \Sigma_j, p_j, \pi_j$ are unknown, they must be fit to the data. This problem shows the connection between the RBF-network for classification and a very simple probabilistic model of the data. The Bayesians often view the RBF-network through this lens.)

Problem 6.24 **[Determining the Number of Clusters k]** Use the same input data $(\mathbf{x}_n)$ as in Problem 6.13.

(a) Run Lloyd's k-means clustering algorithm, for $k = 1, 2, 3, 4, \ldots$, outputting the value of the k-means objective $E_{\text{in}}(k)$.

(b) Generate benchmark random data of the same size, uniformly over the smallest axis-aligned square containing the the actual data (this data has no well defined clusters). Run Lloyd's algorithm on this random data for $k = 1, 2, 3, 4, \ldots$. Repeat for several such random data sets to obtain the average k-means error as a function of k, $E_{\text{in}}^{\text{rand}}(k)$.

(c) Compute and plot the *gap* statistic (as a function of k)

$$G(k) = \log E_{\text{in}}^{\text{rand}}(k) - \log E_{\text{in}}(k).$$

(d) Argue that the maximum of the gap statistic is a reasonable choice for the number of clusters to use. What value for k do you get?

(e) Repeat with different choices for σ in Problem 6.13, and plot the value of k chosen by the gap statistic versus σ. Explain your result.

Problem 6.25 **[Novelty Detection]** Novelty corresponds to the arrival of a new cluster. Use the same input data $(\mathbf{x}_n)$ as in Problem 6.13.

(a) Use the method of Problem 6.24 to determine the number of clusters.

(b) Add data (one by one) from a 5th Gaussian centered on the origin with the same covariance matrix as the other four Gaussians. For each new point, recompute the number of clusters using the method of Problem 6.24. Plot the number of clusters versus ℓ the amount of data added.

(c) Repeat (b) and plot, as a function of ℓ, the average increase in the number of clusters from adding ℓ data points.

(d) From your plot in (c), estimate ℓ^*, the number of data points needed to identify the existence of this new cluster.

(e) Vary σ (the variance parameter of the Gaussians) and plot ℓ^* versus σ. Explain your results.

Problem 6.26 Let $V = \{v_1, \ldots, v_M\}$ be a universe of objects. Define the distance between two sets $S_1, S_2 \subseteq V$ by

$$d(S_1, S_2) = 1 - J(S_1, S_2),$$

where $J(S_1, S_2) = |S_1 \cap S_2|/|S_1 \cup S_2|$ is the Jaccard coefficient. Show that $d(\cdot, \cdot)$ is a metric satisfying non-negativity, symmetry and the triangle inequality.

Problem 6.27 Consider maximum likelihood estimation of the parameters of a GMM in one dimension from data $x_1, \ldots, x_N$. The probability density is

$$P(x) = \sum_{k=1}^{K} \frac{w_k}{2\pi\sigma_k^2} \exp\left(-\frac{(x-\mu_k)^2}{2\sigma_k^2}\right).$$

(a) If $K = 1$ what are the maximum likelihood estimate of μ_1 and σ_1^2.

(b) If $K > 1$, show that the maximum likelihood estimate is not well defined; specifically that the maximum of the likelihood function is infinite.

(c) How does the E-M algorithm perform? Under what conditions does the E-M algorithm converge to a well defined estimator?

(d) To address the problem in (b), define the "regularized" likelihood: For each x_n, define the 2ϵ-interval $B_n = [x_n - \epsilon, x_n + \epsilon]$. The ϵ-regularized likelihood is the probability that each $x_n \in B_n$. Intuitively, one does not measure x_n, but rather a 2ϵ-interval in which x_n lies.

(i) Show that

$$\mathbb{P}[x_n \in B_n] = \sum_{k=1}^{K} w_k \left(F_{\mathcal{N}}\left(\frac{x_i + \epsilon - \mu_k}{\sigma_k}\right) - F_{\mathcal{N}}\left(\frac{x_i + \epsilon - \mu_k}{\sigma_k}\right)\right),$$

where $F_{\mathcal{N}}(\cdot)$ is the standard normal distribution function.

(ii) For fixed ϵ, show that the maximum likelihood estimator is now well defined. What about in the limit as $\epsilon \to 0$?

(iii) Give an E-M algorithm to maximize the ϵ-regularized likelihood.

Problem 6.28 Probability density estimation is a very general task in that supervised learning can be posed as probability density estimation. In supervised learning, the task is to learn $f(\mathbf{x})$ from $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$.

(a) Let $P(\mathbf{x}, y)$ be the joint distribution of $(\mathbf{x}, y)$. For the squared error, $E_{\text{out}}(h) = \mathbb{E}_{P(\mathbf{x},y)}[(h(\mathbf{x}) - y)^2]$. Show that among *all* functions, the one which minimizes E_{out} is $f(\mathbf{x}) = \mathbb{E}_{P(\mathbf{x},y)}[y|\mathbf{x}]$.

(b) To estimate f, first estimate $P(\mathbf{x}, y)$ and then compute $\mathbb{E}[y|\mathbf{x}]$. Treat the N data points as N "unsupervised" points $\mathbf{z}_1, \ldots, \mathbf{z}_N$ in $\mathbb{R}^{d+1}$, where $\mathbf{z}_n^{\mathrm{T}} = [\mathbf{x}_n^{\mathrm{T}}, y_n]$. Suppose that you use a GMM to estimate $P(\mathbf{z})$, so the parameters $w_k, \boldsymbol{\mu}_k, \Sigma_k$ have been estimated ($w_k \geq 0$, $\sum_k w_k = 1$), and

$$P(\mathbf{z}) = \sum_{k=1}^{K} \frac{w_k |\mathrm{S}_k|^{1/2}}{(2\pi)^{(d+1)/2}} e^{-\frac{1}{2}(\mathbf{z}-\boldsymbol{\mu}_k)^{\mathrm{T}} \mathrm{S}_k (\mathbf{z}-\boldsymbol{\mu}_k)},$$

where $\mathrm{S}_k = \Sigma_k^{-1}$. Let $\mathrm{S}_k = \begin{bmatrix} \mathrm{A}_k & \mathbf{b}_k \\ \mathbf{b}_k^{\mathrm{T}} & c_k \end{bmatrix}$ and $\boldsymbol{\mu}_k = \begin{bmatrix} \boldsymbol{\alpha}_k \\ \beta_k \end{bmatrix}$. Show that

$$g(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}] = \frac{\sum_{k=1}^{K} \hat{w}_k e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\alpha}_k)^{\mathrm{T}} \Omega_k (\mathbf{x}-\boldsymbol{\alpha}_k)} (\beta_k + \frac{1}{c_k}\mathbf{b}_k^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\alpha}_k))}{\sum_{k=1}^{K} \hat{w}_k e^{\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\alpha}_k)^{\mathrm{T}} \Omega_k (\mathbf{x}-\boldsymbol{\alpha}_k)\right)}},$$

where $\Omega_k = \mathrm{A}_k - \mathbf{b}_k \mathbf{b}_k^{\mathrm{T}}/c_k$ and $\hat{w}_k = w_k \sqrt{|\mathrm{S}_k|/c_k}$. Interpret this functional form in terms of Radial Basis Functions.

(c) If you non-parametric Parzen windows with spherical Gaussian kernel to estimate $P(\mathbf{x}, y)$, show that $g(\mathbf{x})$ is an RBF,

$$g(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}] = \frac{\sum_{n=1}^{N} e^{-\frac{1}{2r^2}\|\mathbf{x}-\mathbf{x}_n\|^2} y_n}{\sum_{n=1}^{N} e^{-\frac{1}{2r^2}\|\mathbf{x}-\mathbf{x}_n\|^2}}.$$

[Hint: This is a special case of the previous part: what are $K, w_k, \boldsymbol{\mu}_k, \mathrm{S}_k$?]

Chapter 7

Neural Networks

Neural networks are a biologically inspired[1] model which has had considerable engineering success in applications ranging from time series prediction to vision. Neural networks are a generalization of the perceptron which uses a feature transform that is *learned* from the data. As such, they are a very powerful and flexible model.

Neural networks are a good candidate model for learning from data because they can efficiently approximate complex target functions and they come with good algorithms for fitting the data. We begin with the basic properties of neural networks, and how to train them on data. We will introduce a variety of useful techniques for fitting the data by minimizing the in-sample error. Because neural networks are a very flexible model, with great approximation power, it is easy to overfit the data; we will study a number of techniques to control overfitting specific to neural networks.

7.1 The Multi-layer Perceptron (MLP)

The perceptron cannot implement simple classification functions. To illustrate, we use the target on the right, which is related to the Boolean XOR function. In this example, f cannot be written as $\text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x})$. However, f is composed of two linear parts. Indeed, as we will soon see, we can decompose f into two simple perceptrons, corresponding to the lines in the figure, and then combine the outputs of these two perceptrons in a simple way to get back f. The two perceptrons are shown next.

[1]The analogy with biological neurons though inspiring should not be taken too far; after all, we build planes with wings that do not flap. In much the same way, neural networks, when applied to learning from data, do not much resemble their biological counterparts.

The target f equals $+1$ when exactly one of h_1, h_2 equals $+1$. This is the Boolean XOR function: $f = \text{XOR}(h_1, h_2)$, where $+1$ represents "TRUE" and -1 represents "FALSE". We can rewrite f using the simpler OR and AND operations: $\text{OR}(h_1, h_2) = +1$ if at least one of h_1, h_2 equal $+1$ and $\text{AND}(h_1, h_2) = +1$ if both h_1, h_2 equal $+1$. Using standard Boolean notation (multiplication for AND, addition for OR, and overbar for negation),

$$f = h_1\overline{h_2} + \overline{h_1}h_2.$$

Exercise 7.1

Consider a target function f whose '$+$' and '$-$' regions are illustrated below.

The target f has three perceptron components h_1, h_2, h_3:

Show that

$$f = \overline{h_1}h_2h_3 + h_1\overline{h_2}h_3 + h_1h_2\overline{h_3}.$$

Is there a systematic way of going from a target which is a decomposition of perceptrons to a Boolean formula like this? *[Hint: consider only the regions of f which are '$+$' and use the* disjunctive normal form *(*OR *of* ANDs*).]*

Exercise 7.1 shows that a complicated target, which is composed of perceptrons, is a disjunction of conjunctions (OR of ANDs) applied to the component

perceptrons. This is a useful insight, because OR and AND can be implemented by the perceptron:

$$\begin{aligned} \text{OR}(x_1, x_2) &= \text{sign}(x_1 + x_2 + 1.5); \\ \text{AND}(x_1, x_2) &= \text{sign}(x_1 + x_2 - 1.5). \end{aligned}$$

This implies that these more complicated targets are ultimately just combinations of perceptrons. To see how to combine the perceptrons to get f, we introduce a graph representation of perceptrons, starting with OR and AND:

A node outputs a value to an arrow. The weight on an arrow multiplies this output and passes the result to the next node. Everything coming to this next node is summed and then transformed by sign$(\cdot)$ to get the final output.

Exercise 7.2

(a) The Boolean OR and AND of two inputs can be extended to more than two inputs: OR$(x_1, \ldots, x_M) = +1$ if any one of the M inputs is $+1$; AND$(x_1, \ldots, x_M) = +1$ if all the inputs equal $+1$. Give graph representations of OR$(x_1, \ldots, x_M)$ and AND$(x_1, \ldots, x_M)$.

(b) Give the graph representation of the perceptron: $h(\mathbf{x}) = \text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x})$.

(c) Give the graph representation of OR$(x_1, \overline{x_2}, x_3)$.

The MLP for a Complex Target. Since $f = h_1\overline{h_2} + \overline{h_1}h_2$, which is an OR of the two inputs $h_1\overline{h_2}$ and $\overline{h_1}h_2$, we first use the OR perceptron, to obtain:

The two inputs $h_1\overline{h_2}$ and $\overline{h_1}h_2$ are ANDs. As such, they can be simulated by the output of two AND perceptrons. To deal with negation of the inputs to the AND, we negate the weights multiplying the negated inputs (as you have done in Exercise 7.1(c)). The resulting graph representation of f is:

The blue and red weights are simulating the required two ANDs. Finally, since $h_1 = \text{sign}(\mathbf{w}_1^{\mathrm{T}}\mathbf{x})$ and $h_2 = \text{sign}(\mathbf{w}_2^{\mathrm{T}}\mathbf{x})$ are perceptrons, we further expand the h_1 and h_2 nodes to obtain the graph representation of f.

The next exercise asks you to compute an explicit algebraic formula for f. The visual graph representation is much neater and easier to generalize.

Exercise 7.3

Use the graph representation to get an explicit formula for f and show that:

$$f(\mathbf{x}) = \text{sign}\Big[\text{sign}(h_1(\mathbf{x}) - h_2(\mathbf{x}) - \tfrac{3}{2}) - \text{sign}(h_1(\mathbf{x}) - h_2(\mathbf{x}) + \tfrac{3}{2}) + \tfrac{3}{2}\Big],$$

where $h_1(\mathbf{x}) = \text{sign}(\mathbf{w}_1^{\mathrm{T}}\mathbf{x})$ and $h_2(\mathbf{x}) = \text{sign}(\mathbf{w}_2^{\mathrm{T}}\mathbf{x})$

Let's compare the graph form of f with the graph form of the simple perceptron, shown to the right. More layers of nodes are used between the input and output to implement f, as compared to the simple perceptron, hence we call it a multi-layer layer perceptron (MLP),. The additional layers are called *hidden layers*.

Notice that the layers feed forward into the next layer only (there are no backward pointing arrows and no jumps to other layers). The input (leftmost) layer is not counted as a layer, so in this example, there are 3 layers (2 hidden layers with 3 nodes each, and an output layer with 1 node). The simple perceptron has no hidden layers, just an input and output. The addition of hidden layers is what allowed us to implement the more complicated target.

Exercise 7.4

For the target function in Exercise 7.1, give the MLP in graphical form, as well as the explicit algebraic form.

If f can be decomposed into perceptrons using an OR of ANDs, then it can be implemented by a 3-layer perceptron. If f is not strictly decomposable into perceptrons, but the decision boundary is smooth, then a 3-layer perceptron can come arbitrarily close to implementing f. A 'proof by picture' illustration for a disc target function follows:

The formal proof is somewhat analogous to the theorem in calculus which says that any continuous function on a compact set can be approximated arbitrarily closely using step functions. The perceptron is the analog of the step function.

We have thus found a generalization of the simple perceptron that looks much like the simple perceptron itself, except for the addition of more layers. We gained the ability to model more complex target functions by adding more nodes (*hidden units*) in the hidden layers – this corresponds to allowing more perceptrons in the decomposition of f. In fact, a suitably large 3-layer MLP can closely approximate just about any target function, and fit any data set, so it is a very powerful learning model. Use it with care. If your MLP is too large you may lose generalization ability.

Once you fix the size of the MLP (number of hidden layers and number of hidden units in each layer), you learn the weights on every link (arrow) by fitting the data. Let's consider the simple perceptron,

$$h(\mathbf{x}) = \theta(\mathbf{w}^{\mathrm{T}}\mathbf{x}).$$

When $\theta(s) = \text{sign}(s)$, learning the weights was already a hard combinatorial problem and had a variety of algorithms, including the pocket algorithm, for fitting data (Chapter 3). The combinatorial optimization problem is even harder with the MLP, for the same reason, namely that the $\text{sign}(\cdot)$ function is not smooth; a smooth, differentiable approximation to $\text{sign}(\cdot)$ will allow us to use analytic methods, rather than purely combinatorial methods, to find the optimal weights. We therefore approximate, or 'soften' the $\text{sign}(\cdot)$ function by using the $\tanh(\cdot)$ function. The MLP is sometimes called a (hard) threshold neural network because the transformation function is a hard threshold at zero. Here, we choose $\theta(x) = \tanh(x)$ which is in-between linear and the hard threshold: nearly linear for $x \approx 0$ and nearly ± 1 for $|x|$ large. The $\tanh(\cdot)$ function is another example of a *sigmoid* (because its shape looks like a flattened out 's'), related to the sigmoid we used for logistic regression.[2] Such networks are called sigmoidal neural networks. Just as we could use the weights learned from linear regression for classification, we could use weights learned using the sigmoidal neural network with $\tanh(\cdot)$ activation function for classification by replacing the output activation function with $\text{sign}(\cdot)$.

Exercise 7.5

Given $\mathbf{w}_1$ and $\epsilon > 0$, find $\mathbf{w}_2$ such that $|\text{sign}(\mathbf{w}_1^{\mathrm{T}}\mathbf{x}_n) - \tanh(\mathbf{w}_2^{\mathrm{T}}\mathbf{x}_n)| \leq \epsilon$ for $\mathbf{x}_n \in \mathcal{D}$. *[Hint: For large enough α, $\text{sign}(x) \approx \tanh(\alpha x)$.]*

The previous example shows that the $\text{sign}(\cdot)$ function can be closely approximated by the $\tanh(\cdot)$ function. A concrete illustration of this is shown in the figure to the right. The figure shows how the in-sample error E_{in} varies with one of the weights in $\mathbf{w}$ on an example problem for the perceptron (blue curve) as compared to the sigmoidal version (red curve). The sigmoidal approximation captures the general shape of the error, so that if we minimize the sigmoidal in-sample error, we get a good approximation to minimizing the in-sample classification error.

[2] In logistic regression, we used the sigmoid because we wanted a probability as the output. Here, we use the 'soft' $\tanh(\cdot)$ because we want a friendly objective function to optimize.

7.2 Neural Networks

The neural network is our 'softened' MLP. Let's begin with a graph representation of a *feed-forward neural network* (the only kind we will consider).

The graph representation depicts a function in our hypothesis set. While this graphical view is aesthetic and intuitive, with information 'flowing' from the inputs on the far left, along links and through hidden nodes, ultimately to the output $h(\mathbf{x})$ on the far right, it will be necessary to algorithmically describe the function being computed. Things are going to get messy, and this calls for a very systematic notation; bear with us.

7.2.1 Notation

There are layers labeled by $\ell = 0, 1, 2, \ldots, L$. In our example above, $L = 3$, i.e. we have three layers (the input layer $\ell = 0$ is usually not considered a layer and is meant for feeding in the inputs). The layer $\ell = L$ is the output layer, which determines the value of the function. The layers in between, $0 < \ell < L$, are the hidden layers. We will use superscript$^{(\ell)}$ to refer to a particular layer. Each layer ℓ has 'dimension' $d^{(\ell)}$, which means that it has $d^{(\ell)} + 1$ nodes, labeled $0, 1, \ldots, d^{(\ell)}$. Every layer has one special node, which is called the *bias* node (labeled 0). This bias node is set to have an output 1, which is analogous to the fictitious $x_0 = 1$ convention that we had for linear models.

Every arrow represents a weight or connection strength from a node in a layer to a node in the *next higher* layer. Notice that the bias nodes have no incoming weights. There are no other connection weights.[3] A node with an incoming weight indicates that some signal is fed into this node. Every such node with an input has a *transformation function* θ. If $\theta(s) = \text{sign}(s)$, then we have the MLP for classification. As we mentioned before, we will be using a soft version of the MLP with $\theta(x) = \tanh(x)$ to approximate the $\text{sign}(\cdot)$ function. The $\tanh(\cdot)$ is a soft threshold or sigmoid, and we already saw a related sigmoid

[3] In a more general setting, weights can connect any two nodes, in addition to going backward (i.e., one can have cycles). Such networks are called recurrent neural networks, and we do not consider them here.

when we discussed logistic regression in Chapter 3. Ultimately, when we do classification, we replace the output sigmoid by the hard threshold $\text{sign}(\cdot)$. As a comment, if we were doing regression instead, our entire discussion goes through with the output transformation being replaced by the identity function (no transformation) so that the output is a real number. If we were doing logistic regression, we would replace the output $\tanh(\cdot)$ sigmoid by the logistic regression sigmoid.

The neural network model $\mathcal{H}_{\text{nn}}$ is specified once you determine the *architecture* of the neural network, that is the dimension of each layer $\mathbf{d} = [d^{(0)}, d^{(1)}, \ldots, d^{(L)}]$ (L is the number of layers). A hypothesis $h \in \mathcal{H}_{\text{nn}}$ is specified by selecting weights for the links. Let's zoom into a node in hidden layer ℓ, to see what weights need to be specified.

A node has an incoming signal s and an output x. The weights on links into the node from the previous layer are $w^{(\ell)}$, so the weights are indexed by the layer into which they go. Thus, the output of the nodes in layer $\ell - 1$ is multiplied by weights $w^{(\ell)}$. We use subscripts to index the nodes in a layer. So, $w_{ij}^{(\ell)}$ is the weight *into* node j in layer ℓ *from* node i in the previous layer, the signal going into node j in layer ℓ is $s_j^{(\ell)}$, and the output of this node is $x_j^{(\ell)}$. There are some special nodes in the network. The zero nodes in every layer are constant nodes, set to output 1. They have no incoming weight, but they have an outgoing weight. The nodes in the input layer $\ell = 0$ are for the input values, and have no incoming weight or transformation function.

For the most part, we only need to deal with the network on a layer by layer basis, so we introduce vector and matrix notation for that. We collect all the input signals to nodes $1, \ldots, d^{(\ell)}$ in layer ℓ in the vector $\mathbf{s}^{(\ell)}$. Similarly, collect the output from nodes $0, \ldots, d^{(\ell)}$ in the vector $\mathbf{x}^{(\ell)}$; note that $\mathbf{x}^{(\ell)} \in \{1\} \times \mathbb{R}^{d^{(\ell)}}$ because of the bias node 0. There are links connecting the outputs of all nodes in the previous layer to the inputs of layer ℓ. So, into layer ℓ, we have a $(d^{(\ell-1)} + 1) \times d^{(\ell)}$ matrix of weights $\mathrm{W}^{(\ell)}$. The (i, j)-entry of $\mathrm{W}^{(\ell)}$ is $w_{ij}^{(\ell)}$ going from node i in the previous layer to node j in layer ℓ.

layer ℓ parameters

signals in	$\mathbf{s}^{(\ell)}$	$d^{(\ell)}$ dimensional input vector
outputs	$\mathbf{x}^{(\ell)}$	$d^{(\ell)}+1$ dimensional output vector
weights in	$\mathrm{W}^{(\ell)}$	$(d^{(\ell-1)}+1)\times d^{(\ell)}$ dimensional matrix
weights out	$\mathrm{W}^{(\ell+1)}$	$(d^{(\ell)}+1)\times d^{(\ell+1)}$ dimensional matrix

After you fix the weights $\mathrm{W}^{(\ell)}$ for $\ell = 1,\ldots,L$, you have specified a particular neural network hypothesis $h \in \mathcal{H}_{\mathrm{nn}}$. We collect all these weight matrices into a single weight parameter $\mathbf{w} = \{\mathrm{W}^{(1)}, \mathrm{W}^{(2)}, \ldots, \mathrm{W}^{(L)}\}$, and sometimes we will write $h(\mathbf{x};\mathbf{w})$ to explicitly indicate the dependence of the hypothesis on $\mathbf{w}$.

7.2.2 Forward Propagation

The neural network hypothesis $h(\mathbf{x})$ is computed by the *forward propagation* algorithm. First observe that the inputs and outputs of a layer are related by the transformation function,

$$\mathbf{x}^{(\ell)} = \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix}. \tag{7.1}$$

where $\theta(\mathbf{s}^{(\ell)})$ is a vector whose components are $\theta(s_j^{(\ell)})$. To get the input vector into layer ℓ, we compute the weighted sum of the outputs from the previous layer, with weights specified in $\mathrm{W}^{(\ell)}$: $s_j^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{ij}^{(\ell)} x_i^{(\ell-1)}$. This process is compactly represented by the matrix equation

$$\mathbf{s}^{(\ell)} = (\mathrm{W}^{(\ell)})^{\mathrm{T}}\mathbf{x}^{(\ell-1)}. \tag{7.2}$$

All that remains is to initialize the input layer to $\mathbf{x}^{(0)} = \mathbf{x}$ (so $d^{(0)} = d$, the input dimension)[4] and use Equations (7.2) and (7.1) in the following chain,

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{\mathrm{W}^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} \xrightarrow{\mathrm{W}^{(2)}} \mathbf{s}^{(2)} \xrightarrow{\theta} \mathbf{x}^{(2)} \cdots \longrightarrow \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x}).$$

[4]Recall that the input vectors are also augmented with $x_0 = 1$.

Forward propagation to compute $h(\mathbf{x})$:

$\mathbf{x}^{(0)} \leftarrow \mathbf{x}$ **[Initialization]**

for $\ell = 1$ to L **do** **[Forward Propagation]**

$\quad \mathbf{s}^{(\ell)} \leftarrow (\mathrm{W}^{(\ell)})^{\mathrm{T}}\mathbf{x}^{(\ell-1)}$

$\quad \mathbf{x}^{(\ell)} \leftarrow \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix}$

$h(\mathbf{x}) = \mathbf{x}^{(L)}$ **[Output]**

After forward propagation, the output vector $\mathbf{x}^{(\ell)}$ at every layer $l = 0, \ldots, L$ has been computed.

Exercise 7.6

Let V and Q be the number of nodes and weights in the neural network,

$$V = \sum_{\ell=0}^{L} d^{(\ell)}, \qquad Q = \sum_{\ell=1}^{L} d^{(\ell)}(d^{(\ell-1)} + 1).$$

In terms of V and Q, how many computations are made in forward propagation (additions, multiplications and evaluations of θ).

[Answer: $O(Q)$ multiplications and additions, and $O(V)$ θ-evaluations.]

If we want to compute E_{in}, all we need is $h(\mathbf{x}_n)$ and y_n. For the sum of squares,

$$\begin{aligned} E_{\text{in}}(\mathbf{w}) &= \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n;\mathbf{w}) - y_n)^2 \\ &= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n^{(L)} - y_n)^2. \end{aligned}$$

We now discuss how to minimize E_{in} to obtain the learned weights. It will be a direct application of gradient descent, with a special algorithm that computes the gradient efficiently.

7.2.3 Backpropagation Algorithm

We studied an algorithm for getting to a local minimum of a smooth in-sample error surface in Chapter 3, namely gradient descent: initialize the weights to $\mathbf{w}(0)$ and for $t = 1, 2, \ldots$ update the weights by taking a step in the negative gradient direction,

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E_{\text{in}}(\mathbf{w}(t))$$

we called this (batch) *gradient descent*. To implement gradient descent, we need the gradient.

Exercise 7.7

For the sigmoidal perceptron, $h(\mathbf{x}) = \tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x})$, let the in-sample error be $E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) - y_n)^2$. Show that

$$\nabla E_{\text{in}}(\mathbf{w}) = \frac{2}{N}\sum_{n=1}^{N}(\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) - y_n)(1 - \tanh^2(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n))\mathbf{x}_n.$$

If $\mathbf{w} \to \infty$, what happens to the gradient; how this is related to why it is hard to optimize the perceptron.

We now consider the sigmoidal multi-layer neural network with $\theta(x) = \tanh(x)$. Since $h(\mathbf{x})$ is smooth, we can apply gradient descent to the resulting error function. To do so, we need the gradient $\nabla E_{\text{in}}(\mathbf{w})$. Recall that the weight vector $\mathbf{w}$ contains all the weight matrices $\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(L)}$, and we need the derivatives with respect to all these weights. Unlike the sigmoidal perceptron in Exercise 7.7, for the multilayer sigmoidal network there is no simple closed form expression for the gradient. Consider an in-sample error which is the sum of the point-wise errors over the data points (as is the squared in-sample error),

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}\mathsf{e}_n.$$

where $\mathsf{e}_n = \mathsf{e}(h(\mathbf{x}_n), y_n)$. For the squared error, $\mathsf{e}(h, y) = (h - y)^2$. To compute the gradient of E_{in}, we need its derivative with respective to each weight matrix:

$$\frac{\partial E_{\text{in}}}{\partial \mathrm{W}^{(\ell)}} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial \mathsf{e}_n}{\partial \mathrm{W}^{(\ell)}}, \tag{7.3}$$

The basic building block in (7.3) is the partial derivative of the error on a data point e, with respect to the $\mathrm{W}^{(\ell)}$. A quick and dirty way to get $\frac{\partial \mathsf{e}}{\partial \mathrm{W}^{(\ell)}}$ is

to use the numerical finite difference approach. The complexity of obtaining the partial derivatives with respect to every weight is $O(Q^2)$, where Q is the number of weights (see Problem 7.6). From (7.3), we have to compute these derivatives for every data point, so the numerical approach is computationally prohibitive. We now derive an elegant dynamic programming algorithm known as *backpropagation*.[5] Backpropagation allows us to compute the partial derivatives with respect to every weight efficiently, using $O(Q)$ computation. We describe backpropagation for getting the partial derivative of the error e, but the algorithm is general and can be used to get the partial derivative of any function of the output $h(\mathbf{x})$ with respect to the weights.

Backpropagation is based on several applications of the chain rule to write partial derivatives in layer ℓ using partial derivatives in layer $(\ell + 1)$. To describe the algorithm, we define the *sensitivity vector* for layer ℓ, which is the sensitivity (gradient) of the error e with respect to the input signal $\mathbf{s}^{(\ell)}$ that goes into layer ℓ. We denote the sensitivity by $\boldsymbol{\delta}^{(\ell)}$,

$$\boldsymbol{\delta}^{(\ell)} = \frac{\partial \mathsf{e}}{\partial \mathbf{s}^{(\ell)}}.$$

The sensitivity quantifies how e changes with $\mathbf{s}^{(\ell)}$. Using the sensitivity, we can write the partial derivatives with respect to the weights $\mathrm{W}^{(\ell)}$ as

$$\frac{\partial \mathsf{e}}{\partial \mathrm{W}^{(\ell)}} = \mathbf{x}^{(\ell-1)} (\boldsymbol{\delta}^{(\ell)})^{\mathrm{T}}. \tag{7.4}$$

We will derive this formula later, but for now let's examine it closely. The partial derivatives on the left form a matrix with dimensions $(d^{(\ell-1)}+1) \times d^{(\ell)}$ and the 'outer product' of the two vectors on the right give exactly such a matrix. The partial derivatives have contributions from two components. (i) The output vector of the layer from which the weights originate; the larger the output, the more sensitive e is to the weights in the layer. (ii) The sensitivity vector of the layer into which the weights go; the larger the sensitivity vector, the more sensitive e is to the weights in that layer.

The outputs $\mathbf{x}^{(\ell)}$ for every layer $\ell \geq 0$ can be computed by a forward propagation. So to get the partial derivatives, it suffices to obtain the sensitivity vectors $\boldsymbol{\delta}^{(\ell)}$ for every layer $\ell \geq 1$ (remember that there is no input signal to layer $\ell = 0$). It turns out that the sensitivity vectors can be obtained by running a slightly modified version of the neural network *backwards*, and hence the name backpropagation. In forward propagation, each layer outputs the vector $\mathbf{x}^{(\ell)}$ and in backpropagation, each layer outputs (backwards) the vector $\boldsymbol{\delta}^{(\ell)}$. In forward propagation, we compute $\mathbf{x}^{(\ell)}$ from $\mathbf{x}^{(\ell-1)}$ and in backpropagation, we compute $\boldsymbol{\delta}^{(\ell)}$ from $\boldsymbol{\delta}^{(\ell+1)}$. The basic idea is illustrated in the following figure.

[5]Dynamic programming is an elegant algorithmic technique in which one builds up a solution to a complex problem using the solutions to related but simpler problems.

As you can see in the figure, the neural network is slightly modified only in that we have changed the transformation function for the nodes. In forward propagation, the transformation was the sigmoid $\theta(\cdot)$. In backpropagation, the transformation is *multiplication by* $\theta'(\mathbf{s}^{(\ell)})$, where $\mathbf{s}^{(\ell)}$ is the input to the node. So the transformation function is now different for each node, and it depends on the input to the node, which depends on $\mathbf{x}$. This input was computed in the forward propagation. For the $\tanh(\cdot)$ transformation function, $\tanh'(\mathbf{s}^{(\ell)}) = \mathbf{1} - \tanh^2(\mathbf{s}^{(\ell)}) = \mathbf{1} - \mathbf{x}^{(\ell)} \otimes \mathbf{x}^{(\ell)}$, where $\otimes$ denotes component-wise multiplication.

In the figure, layer $(\ell+1)$ outputs (backwards) the sensitivity vector $\boldsymbol{\delta}^{(\ell+1)}$, which gets multiplied by the weights $\mathrm{W}^{(\ell+1)}$, summed and passed into the nodes in layer ℓ. Nodes in layer ℓ multiply by $\theta'(\mathbf{s}^{(\ell)})$ to get $\boldsymbol{\delta}^{(\ell)}$. Using $\otimes$, a shorthand notation for this backpropagation step is:

$$\boldsymbol{\delta}^{(\ell)} = \theta'(\mathbf{s}^{(\ell)}) \otimes \left[\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}, \tag{7.5}$$

where the vector $\left[\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$ contains components $1, \ldots, d^{(\ell)}$ of the vector $\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}$ (excluding the bias component which has index 0). This formula is not surprising. The sensitivity of e to inputs of layer ℓ is proportional to the slope of the activation function in layer ℓ (bigger slope means a small change in $\mathbf{s}^{(\ell)}$ will have a larger effect on $\mathbf{x}^{(\ell)}$), the size of the weights going out of the layer (bigger weights mean a small change in $\mathbf{s}^{(\ell)}$ will have more impact on $\mathbf{s}^{(\ell+1)}$) and the sensitivity in the next layer (a change in layer ℓ affects the inputs to layer $\ell+1$, so if e is more sensitive to layer $\ell+1$, then it will also be more sensitive to layer ℓ).

We will derive this backward recursion later. For now, observe that if we know $\boldsymbol{\delta}^{(\ell+1)}$, then you can get $\boldsymbol{\delta}^{(\ell)}$. We use $\boldsymbol{\delta}^{(L)}$ to seed the backward process, and we can get that explicitly because $\mathsf{e} = (\mathbf{x}^{(L)} - y)^2 = (\theta(\mathbf{s}^{(L)}) - y)^2$.

Therefore,

$$\begin{aligned}
\boldsymbol{\delta}^{(L)} &= \frac{\partial \mathsf{e}}{\partial \mathbf{s}^{(L)}} \\
&= \frac{\partial}{\partial \mathbf{s}^{(L)}} (\mathbf{x}^{(L)} - y)^2 \\
&= 2(\mathbf{x}^{(L)} - y) \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \\
&= 2(\mathbf{x}^{(L)} - y)\theta'(\mathbf{s}^{(L)}).
\end{aligned}$$

When the output transformation is $\tanh(\cdot)$, $\theta'(\mathbf{s}^{(L)}) = 1 - (x^{(L)})^2$ (classification); when the output transformation is the identity (regression), $\theta'(\mathbf{s}^{(L)}) = 1$. Now, using (7.5), we can compute all the sensitivities:

$$\boldsymbol{\delta}^{(1)} \longleftarrow \boldsymbol{\delta}^{(2)} \ \cdots \ \longleftarrow \boldsymbol{\delta}^{(L-1)} \longleftarrow \boldsymbol{\delta}^{(L)}.$$

Note that since there is only one output node, $\mathbf{s}^L$ is a scalar, and so too is $\boldsymbol{\delta}^{(L)}$. The algorithm box below summarizes backpropagation.

Backpropagation to compute sensitivities $\boldsymbol{\delta}^{(\ell)}$.
Input: a data point $(\mathbf{x}, y)$.

0: Run forward propagation on $\mathbf{x}$ to compute and save:

$$\begin{array}{ll} \mathbf{s}^{(\ell)} & \text{for } \ell = 1, \ldots, L; \\ \mathbf{x}^{(\ell)} & \text{for } \ell = 0, \ldots, L. \end{array}$$

1: $\delta^{(L)} \leftarrow 2(x^{(L)} - y)\theta'(s^{(L)})$ **[Initialization]**

$$\theta'(s^{(L)}) = \begin{cases} 1 - (x^{(L)})^2 & \theta(s) = \tanh(s); \\ 1 & \theta(s) = s. \end{cases}$$

2: **for** $\ell = L - 1$ to 1 **do** **[Back-Propagation]**

3: Let $\theta'(\mathbf{s}^{(\ell)}) = \left[1 - \mathbf{x}^{(\ell)} \otimes \mathbf{x}^{(\ell)}\right]_1^{d^{(\ell)}}$.

4: Compute the sensitivity $\boldsymbol{\delta}^{(\ell)}$ from $\boldsymbol{\delta}^{(\ell+1)}$:

$$\boldsymbol{\delta}^{(\ell)} \leftarrow \theta'(\mathbf{s}^{(\ell)}) \otimes \left[\mathrm{W}^{(\ell+1)} \boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$$

In step 3, we assumed tanh-hidden node transformations. If the hidden unit transformation functions are not $\tanh(\cdot)$, then the derivative in step 3 should be updated accordingly. Using forward propagation, we compute $\mathbf{x}^{(\ell)}$ for $\ell = 0, \ldots, L$ and using backpropagation, we compute $\boldsymbol{\delta}^{(\ell)}$ for $\ell = 1, \ldots, L$. Finally, we get the partial derivative of the error on a *single* data point using Equation (7.4). Nothing illuminates the moving parts better than working an example from start to finish.

Example 7.1. Consider the following neural network.

There is a single input, and the weight matrices are:

$$\mathrm{W}^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}; \quad \mathrm{W}^{(2)} = \begin{bmatrix} 0.2 \\ 1 \\ -3 \end{bmatrix}; \quad \mathrm{W}^{(3)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

For the data point $x = 2, y = 1$, forward propagation gives:

$\mathbf{x}^{(0)}$	$\mathbf{s}^{(1)}$	$\mathbf{x}^{(1)}$	$\mathbf{s}^{(2)}$	$\mathbf{x}^{(2)}$	$\mathbf{s}^{(3)}$	$\mathbf{x}^{(3)}$
$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 0.1 & 0.3 \\ 0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.7 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix}$	$[-1.48]$	$\begin{bmatrix} 1 \\ -0.90 \end{bmatrix}$	$[-0.8]$	-0.66

We show above how $\mathbf{s}^{(1)} = (\mathrm{W}^{(1)})^{\mathrm{T}}\mathbf{x}^{(0)}$ is computed. Backpropagation gives

$\boldsymbol{\delta}^{(3)}$	$\boldsymbol{\delta}^{(2)}$	$\boldsymbol{\delta}^{(1)}$
$[-1.855]$	$[(1 - 0.9^2) \cdot 2 \cdot -1.855] = [-0.69]$	$\begin{bmatrix} -0.44 \\ 0.88 \end{bmatrix}$

We have explicitly shown how $\boldsymbol{\delta}^{(2)}$ is obtained from $\boldsymbol{\delta}^{(3)}$. It is now a simple matter to combine the output vectors $\mathbf{x}^{(\ell)}$ with the sensitivity vectors $\boldsymbol{\delta}^{(\ell)}$ using (7.4) to obtain the partial derivatives that are needed for the gradient:

$$\frac{\partial \mathsf{e}}{\partial \mathrm{W}^{(1)}} = \mathbf{x}^{(0)}(\boldsymbol{\delta}^{(1)})^{\mathrm{T}} = \begin{bmatrix} -0.44 & 0.88 \\ -0.88 & 1.75 \end{bmatrix}; \quad \frac{\partial \mathsf{e}}{\partial \mathrm{W}^{(2)}} = \begin{bmatrix} -0.69 \\ -0.42 \\ -0.53 \end{bmatrix}; \quad \frac{\partial \mathsf{e}}{\partial \mathrm{W}^{(3)}} = \begin{bmatrix} -1.85 \\ 1.67 \end{bmatrix}.$$

□

Exercise 7.8

Repeat the computations in Example 7.1 for the case when the output transformation is the identity. You should compute $\mathbf{s}^{(\ell)}, \mathbf{x}^{(\ell)}, \boldsymbol{\delta}^{(\ell)}$ and $\partial \mathsf{e}/\partial \mathrm{W}^{(\ell)}$

Let's derive (7.4) and (7.5), which are the core equations of backpropagation. There's nothing to it but repeated application of the chain rule. If you wish to trust our math, you won't miss much by moving on.

Begin safe skip: If you trust our math, you can skip this part without compromising the logical sequence. A similar green box will tell you when to rejoin.

To begin, let's take a closer look at the partial derivative, $\partial \mathsf{e}/\partial \mathrm{W}^{(\ell)}$. The situation is illustrated in Figure 7.1.

Figure 7.1: Chain of dependencies from $\mathrm{W}^{(\ell)}$ to $\mathbf{x}^{(L)}$.

We can identify the following chain of dependencies by which $\mathrm{W}^{(\ell)}$ influences the output $\mathbf{x}^{(L)}$, and hence the error e.

$$\mathrm{W}^{(\ell)} \longrightarrow \mathbf{s}^{(\ell)} \longrightarrow \mathbf{x}^{(\ell)} \longrightarrow \mathbf{s}^{(\ell+1)} \cdots \longrightarrow \mathbf{x}^{(L)} = h.$$

To derive (7.4), we drill down to a single weight and use the chain rule. For a single weight $w_{ij}^{(\ell)}$, a change in $w_{ij}^{(\ell)}$ only affects $\mathbf{s}_j^{(\ell)}$ and so by the chain rule,

$$\frac{\partial \mathsf{e}}{\partial w_{ij}^{(\ell)}} = \frac{\partial \mathbf{s}_j^{(\ell)}}{\partial w_{ij}^{(\ell)}} \cdot \frac{\partial \mathsf{e}}{\partial \mathbf{s}_j^{(\ell)}} = \mathbf{x}_i^{(\ell-1)} \cdot \boldsymbol{\delta}_j^{(\ell)},$$

where the last equality follows because $\mathbf{s}_j^{(\ell)} = \sum_{\alpha=0}^{d^{(\ell-1)}} w_{\alpha j}^{(\ell)} \mathbf{x}_\alpha^{(\ell-1)}$ and by definition of $\boldsymbol{\delta}_j^{(\ell)}$. We have derived the component form of (7.4).

We now derive the component form of (7.5). Since e depends on $\mathbf{s}^{(\ell)}$ only through $\mathbf{x}^{(\ell)}$ (see Figure 7.1), by the chain rule, we have:

$$\boldsymbol{\delta}_j^{(\ell)} = \frac{\partial \mathsf{e}}{\partial \mathbf{s}_j^{(\ell)}} = \frac{\partial \mathsf{e}}{\partial \mathbf{x}_j^{(\ell)}} \cdot \frac{\partial \mathbf{x}_j^{(\ell)}}{\partial \mathbf{s}_j^{(\ell)}} = \theta'\left(\mathbf{s}_j^{(\ell)}\right) \cdot \frac{\partial \mathsf{e}}{\partial \mathbf{x}_j^{(\ell)}}.$$

To get the partial derivative $\partial \mathsf{e}/\partial \mathbf{x}^{(\ell)}$, we need to understand how e changes due to changes in $\mathbf{x}^{(\ell)}$. Again, from Figure 7.1, a change in $\mathbf{x}^{(\ell)}$ only affects

$\mathbf{s}^{(\ell+1)}$ and hence e. Because a particular component of $\mathbf{x}^{(\ell)}$ can affect *every* component of $\mathbf{s}^{(\ell+1)}$, we need to sum these dependencies using the chain rule:

$$\frac{\partial \mathsf{e}}{\partial \mathbf{x}_j^{(\ell)}} = \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial \mathbf{s}_k^{(\ell+1)}}{\partial \mathbf{x}_j^{(\ell)}} \cdot \frac{\partial \mathsf{e}}{\partial \mathbf{s}_k^{(\ell+1)}} = \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \boldsymbol{\delta}_k^{(\ell+1)}.$$

Putting all this together, we have arrived at the component version of (7.5)

$$\boldsymbol{\delta}_j^{(\ell)} = \theta'(\mathbf{s}_j^{(\ell)}) \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \boldsymbol{\delta}_k^{(\ell+1)}, \tag{7.6}$$

Intuitively, the first term comes from the impact of $\mathbf{s}^{(\ell)}$ on $\mathbf{x}^{(\ell)}$; the summation is the impact of $\mathbf{x}^{(\ell)}$ on $\mathbf{s}^{(\ell+1)}$, and the impact of $\mathbf{s}^{(\ell+1)}$ on h is what gives us back the sensitivities in layer $(\ell+1)$, resulting in the backward recursion.

End safe skip: Those who skipped are now rejoining us to discuss how backpropagation gives us ∇E_{in}.

Backpropagation works with a data point $(\mathbf{x}, y)$ and weights $\mathbf{w} = \{\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(L)}\}$. Since we run one forward and backward propagation to compute the outputs $\mathbf{x}^{(\ell)}$ and the sensitivities $\boldsymbol{\delta}^{(\ell)}$, the running time is order of the number of weights in the network. We compute once for each data point $(\mathbf{x}_n, y_n)$ to get $\nabla E_{\text{in}}(\mathbf{x}_n)$ and, using the sum in (7.3), we aggregate these single point gradients to get the full *batch* gradient ∇E_{in}. We summarize the algorithm below.

Algorithm to Compute $E_{\text{in}}(\mathbf{w})$ and $\mathbf{g} = \nabla E_{\text{in}}(\mathbf{w})$.
Input: $\mathbf{w} = \{\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(L)}\}$; $\mathcal{D} = (\mathbf{x}_1, y_1) \ldots (\mathbf{x}_N, y_n)$.
Output: error $E_{\text{in}}(\mathbf{w})$ and gradient $\mathbf{g} = \{\mathrm{G}^{(1)}, \ldots, \mathrm{G}^{(L)}\}$.

```
Initialize: E_in = 0 and G^(ℓ) = 0 · W^(ℓ) for ℓ = 1, ..., L.
for Each data point (x_n, y_n), n = 1, ..., N, do
    Compute x^(ℓ) for ℓ = 0, ..., L.        [forward propagation]
    Compute δ^(ℓ) for ℓ = L, ..., 1.        [backpropagation]
    E_in ← E_in + (1/N)(x^(L) − y_n)^2.
    for ℓ = 1, ..., L do
        G^(ℓ)(x_n) = [x^(ℓ−1)(δ^(ℓ))^T]
        G^(ℓ) ← G^(ℓ) + (1/N) G^(ℓ)(x_n)
```

($\mathrm{G}^{(\ell)}(\mathbf{x}_n)$ is the gradient on data point $\mathbf{x}_n$). The weight update for a single iteration of fixed learning rate gradient descent is $\mathrm{W}^{(\ell)} \leftarrow \mathrm{W}^{(\ell)} - \eta \mathrm{G}^{(\ell)}$, for $\ell = 1, \ldots, L$. We do all this for *one* iteration of gradient descent, a costly computation for just one little step.

In Chapter 3, we discussed *stochastic gradient descent (SGD)* as a more efficient alternative to the batch mode. Rather than wait for the aggregate gradient $\mathrm{G}^{(\ell)}$ at the end of the iteration, one immediately updates the weights as each data point is sequentially processed using the single point gradient in step 7 of the algorithm: $\mathrm{W}^{(\ell)} = \mathrm{W}^{(\ell)} - \eta \mathrm{G}^{(\ell)}(\mathbf{x}_n)$. In this sequential version, you still run a forward and backward propagation for each data point, but make N updates to the weights. A comparison of batch gradient descent with SGD is shown to the right. We used 500 training examples from the digits data and a 2-layer neural network with 5 hidden units and learning rate $\eta = 0.01$. The SGD curve is erratic because one is not minimizing the total error at each iteration, but the error on a specific data point. One method to dampen this erratic behavior is to decrease the learning rate as the minimization proceeds.

The speed at which you minimize E_{in} can depend heavily on the optimization algorithm you use. SGD appears significantly better than plain vanilla gradient descent, but we can do much better – even SGD is not very efficient. In Section 7.5, we discuss some other powerful methods (for example, conjugate gradients) that can significantly improve upon gradient descent and SGD, by making more effective use of the gradient.

Initialization and Termination. Choosing the initial weights and deciding when to stop the gradient descent can be tricky, as compared with logistic regression, because the in-sample error is not convex anymore. From Exercise 7.7, if the weights are initialized too large so that $\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) \approx \pm 1$, then the gradient will be close to zero and the algorithm won't get anywhere. This is especially a problem if you happen to initialize the weights to the wrong sign. It is usually best to initialize the weights to *small* random values where $\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) \approx 0$ so that the algorithm has the flexibility to move the weights easily to fit the data. One good choice is to initialize using Gaussian random weights, $w_i \sim N(0, \sigma_w^2)$ where σ_w^2 is small. But how small should σ_w^2 be? A simple heuristic is that we want $|\mathbf{w}^{\mathrm{T}}\mathbf{x}_n|^2$ to be small. Since $\mathbb{E}_{\mathbf{w}}\left[|\mathbf{w}^{\mathrm{T}}\mathbf{x}_n|^2\right] = \sigma_w^2 \|\mathbf{x}_n\|^2$, we should choose σ_w^2 so that $\sigma_w^2 \cdot \max_n \|\mathbf{x}_n\|^2 \ll 1$.

Exercise 7.9

What can go wrong if you just initialize all the weights to exactly zero?

When do we stop? It is risky to rely solely on the size of the gradient to stop. As illustrated on the right, you might stop prematurely when the iteration reaches a relatively flat region (which is more common than you might suspect). A combination of stopping criteria is best in practice, for example stopping only when there is marginal error improvement coupled with small error, plus an upper bound on the number of iterations.

7.2.4 Regression for Classification

In Chapter 3, we mentioned that you could use the weights resulting from linear regression as perceptron weights for classification, and you can do the same with neural networks. Specifically, fit the classification data ($y_n = \pm 1$) as though it were a regression problem. This means you use the identity function as the output node transformation, instead of $\tanh(\cdot)$. This can be a great help because of the 'flat regions' which the network is susceptible to when using gradient descent, which happens often in training. The reason for these flat periods in the optimization is the exceptionally flat nature of the tanh function when its argument gets large. If for whatever reason the weights get large toward the beginning of the training, then the error surface begins to look flat, because the tanh has been saturated. Now, gradient descent cannot make any progress and you might think you are at a minimum, when in fact you are far from a minimum. The problem of a flat error surface is considerably mitigated when the output transformation is the identity because you can recover from an initial bad move if it happens to take you to large weights (the linear output never saturates). For a concrete example of a prematurely flat in-sample error, see the figure in Example 7.2 on page 25.

7.3 Approximation versus Generalization

A large enough MLP with 2 hidden layers can approximate smooth decision functions arbitrarily well. It turns out that a single hidden layer suffices.[6] A neural network with a single hidden layer having m hidden units ($d^{(1)} = m$) implements a function of the form

$$h(\mathbf{x}) = \theta\left(w_{01}^{(2)} + \sum_{j=1}^{m} w_{j1}^{(2)} \theta\left(\sum_{i=0}^{d} w_{ij}^{(1)} x_i\right)\right).$$

[6] Though one hidden layer is enough, it is not necessarily the most efficient way to fit the data; for example a much smaller 2-hidden-layer network may exist.

This is a cumbersome representation for such a simple network. A simplified notation for this special case is much more convenient. For the second-layer weights, we will just use $w_0, w_1, \ldots, w_m$ and we will use $\mathbf{v}_j$ to denote the jth column of the first layer weight matrix $\mathrm{W}^{(1)}$, for $j = 1 \ldots m$. With this simpler notation, the hypothesis becomes much more pleasant looking:

$$h(\mathbf{x}) = \theta\left(w_0 + \sum_{j=1}^{m} w_j \theta\left(\mathbf{v}_j^{\mathrm{T}}\mathbf{x}\right)\right).$$

Neural Network versus Nonlinear Transforms. Recall the linear model from Chapter 3, with nonlinear transform $\mathbf{\Phi}(\mathbf{x})$ that transforms $\mathbf{x}$ to $\mathbf{z}$:

$$\mathbf{x} \to \mathbf{z} = \mathbf{\Phi}(\mathbf{x}) = [1, \phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_m(\mathbf{x})]^{\mathrm{T}}.$$

The linear model with nonlinear transform is a hypothesis of the form

$$h(\mathbf{x}) = \theta\left(w_0 + \sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right).$$

The $\phi_j(\cdot)$ are called basis functions. On face value, the neural network and the linear model look nearly identical, by setting $\theta(\mathbf{v}_j^{\mathrm{T}}\mathbf{x}) = \phi_j(\mathbf{x})$. There is a subtle difference, though, and this difference has a big practical impact. With the nonlinear transform, the basis functions $\phi_j(\cdot)$ are fixed ahead of time before you look at the data. With the neural network, the 'basis function' $\theta(\mathbf{v}_j^{\mathrm{T}}\mathbf{x})$ has a parameter $\mathbf{v}_j$ inside, and we can tune $\mathbf{v}_j$ *after* seeing the data. First, this has a computational impact because the parameter $\mathbf{v}_j$ appears *inside* the nonlinearity $\theta(\cdot)$; the model is no longer linear in its parameters. We saw a similar effect with the centers of the radial basis function network in Chapter 6. Models which are nonlinear in their parameters pose a significant computational challenge when it comes to fitting to data. Second, it means that we can *tune the basis functions to the data.* Tunable basis functions, although computationally harder to fit to data, do give us considerably more flexibility to fit the data than do fixed basis functions. With m tunable basis functions one has roughly the same approximation power to fit the data as with m^d fixed basis functions. For large d, tunable basis functions have considerably more power.

Exercise 7.10

It is no surprise that adding nodes in the hidden layer gives the neural network more approximation ability, because you are adding more parameters.

How many weight parameters are there in a neural network with architecture specified by $\mathbf{d} = [d^{(0)}, d^{(1)}, \ldots, d^{(L)}]$, a vector giving the number of nodes in each layer? Evaluate your formula for a 2 hidden layer network with 10 hidden nodes in each hidden layer.

Approximation Capability of the Neural Network. It is possible to quantify how the approximation ability of the neural network grows as you increase m, the number of hidden units. Such results fall into the field known as functional approximation theory, a field which, in the context of neural networks, has produced some interesting results. Usually one starts by making some assumption about the smoothness (or complexity) of the target function f. On the theoretical side, you have lost some generality as compared with, for example, the VC-analysis. However, in practice, such assumptions are OK because target functions are often smooth. If you assume that the data are generated by a target function with complexity[7] at most C_f, then a variety of bounds exist on how small an in-sample error is achievable with m hidden units. For regression with squared error, one can achieve in-sample error

$$E_{\text{in}}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n) - y_n)^2 \leq \frac{(2RC_f)^2}{m},$$

where $R = \max_n \|\mathbf{x}_n\|$ is the 'radius' of the data. The in-sample error decreases inversely with the number of hidden units. For classification, a similar result with slightly worse dependence on m exists. With high probability,

$$E_{\text{in}} \leq E^*_{\text{out}} + O(C_f/\sqrt{m}),$$

where E^*_{out} is the out-of-sample error of the optimal classifier that we discussed in Chapter 6. The message is that E_{in} can be made small by choosing a large enough hidden layer.

Generalization and the VC-Dimension. For sufficiently large m, we can get E_{in} to be small, so what remains is to ensure that $E_{\text{in}} \approx E_{\text{out}}$. We need to look at the VC-dimension. For the two layer hard-threshold neural network (MLP) where $\theta(x) = \text{sign}(x)$, we show a simple bound on the VC dimension:

$$d_{\text{VC}} \leq (\text{const}) \cdot md\log(md). \tag{7.7}$$

For a general sigmoid neural network, d_{VC} can be infinite. For the $\tanh(\cdot)$ sigmoid, with $\text{sign}(\cdot)$ output node, one can show that $d_{\text{VC}} = O(VQ)$ where V is the number of hidden nodes and Q is the number of weights; for the two layer case

$$d_{\text{VC}} = O(md(m+d)).$$

The $\tanh(\cdot)$ network has higher VC-dimension than the 2-layer MLP, which is not surprising because $\tanh(\cdot)$ can approximate $\text{sign}(\cdot)$ by choosing large enough weights. So every dichotomy that can be implemented by the MLP can also be implemented by the $\tanh(\cdot)$ neural network.

[7] We do not describe details of how the complexity of a target can be measured. One measure is the size of the 'high frequency' components of f in its Fourier transform. Another more restrictive measure is the number of bounded derivatives f has.

To derive (7.7), we will actually show a more general result. Consider the hypothesis set illustrated by the network on the right. Hidden node i in the hidden layer implements a function $h_i \in \mathcal{H}_i$ which maps $\mathbb{R}^d$ to $\{+1, -1\}$; the output node implements a function $h_c \in \mathcal{H}_C$ which maps $\mathbb{R}^m$ to $\{+1, -1\}$. This output node combines the outputs of the hidden layer nodes to implement the hypothesis

$$h(\mathbf{x}) = h_C(h_1(\mathbf{x}), \ldots, h_m(\mathbf{x})).$$

(For the 2-layer MLP, all the hypothesis sets are perceptrons.)

Suppose the VC-dimension of $\mathcal{H}_i$ is d_i and the VC-dimension of $\mathcal{H}_C$ is d_c. Fix $\mathbf{x}_1, \ldots, \mathbf{x}_N$, and the hypotheses $h_1, \ldots, h_m$ implemented by the hidden nodes. The hypotheses $h_1, \ldots, h_m$ are now fixed basis functions defining a transform to $\mathbb{R}^m$,

$$\mathbf{x}_1 \to \mathbf{z}_1 = \begin{bmatrix} h_1(\mathbf{x}_1) \\ \vdots \\ h_m(\mathbf{x}_1) \end{bmatrix} \quad \cdots \quad \mathbf{x}_N \to \mathbf{z}_N = \begin{bmatrix} h_1(\mathbf{x}_N) \\ \vdots \\ h_m(\mathbf{x}_N) \end{bmatrix}.$$

The transformed points are binary vectors in $\mathbb{R}^m$. Given $h_1, \ldots, h_m$, the points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ are transformed to an arrangement of points $\mathbf{z}_1, \ldots, \mathbf{z}_N$ in $\mathbb{R}^m$. Using our flexibility to choose $h_1, \ldots, h_m$, we now upper bound the number of possible *different* arrangements $\mathbf{z}_1, \ldots, \mathbf{z}_N$ we can get.

The first components of all the $\mathbf{z}_n$ are given by $h_1(\mathbf{x}_1), \ldots, h_1(\mathbf{x}_N)$, which is a dichotomy of $\mathbf{x}_1, \ldots, \mathbf{x}_N$ implemented by h_1. Since the VC-dimension of $\mathcal{H}_1$ is d_1, there are at most N^{d_1} such dichotomies.[8] That is, there are at most N^{d_1} different ways of choosing assignments to *all* the first components of the $\mathbf{z}_n$. Similarly, an assignment to all the ith components can be chosen in at most N^{d_i} ways. Thus, the total number of possible arrangements for $\mathbf{z}_1, \ldots, \mathbf{z}_N$ is at most

$$\prod_{i=1}^{m} N^{d_i} = N^{\sum_{i=1}^{m} d_i}.$$

Each of these arrangements can be dichotomized in at most N^{d_c} ways, since the VC-dimension of $\mathcal{H}_C$ is d_c. Each such dichotomy for a particular arrangement gives one dichotomy of the data $\mathbf{x}_1, \ldots, \mathbf{x}_N$. Thus, the maximum number of different dichotomies we can implement on $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is upper bounded by the product: the number of possible arrangements times the number of ways

[8] Recall that for any hypothesis set with VC-dimension d_{VC} and any $N \geq d_{\text{VC}}$, $m(N)$ (the maximum number of implementable dichotomies) is bounded by $(eN/d_{\text{VC}})^{d_{\text{VC}}} \leq N^{d_{\text{VC}}}$ (for the sake of simplicity we assume that $d_{\text{VC}} \geq 2$).

of dichotomizing a particular arrangement. We have shown that

$$m(N) \leq N^{d_c} \cdot N^{\sum_{i=1}^m d_i} = N^{d_c + \sum_{i=1}^m d_i}.$$

Let $D = d_c + \sum_{i=1}^m d_i$. After some algebra (left to the reader), if $N \geq 2D \log_2 D$, then $m(N) < 2^N$, from which we conclude that $d_{\text{VC}} \leq 2D \log_2 D$. For the 2-layer MLP, $d_i = d + 1$ and $d_c = m + 1$, and so we have that $D = d_c + \sum_{i=1}^m d_i = m(d+2) + 1 = O(md)$. Thus, $d_{\text{VC}} = O(md \log(md))$. Our analysis looks very crude, but it is almost tight: it is possible to shatter $\Omega(md)$ points with m hidden units (see Problem 7.16), and so the upper bound can be loose by at most a logarithmic factor. Using the VC-dimension, the generalization error bar from Chapter 2 is $O(\sqrt{(d_{\text{VC}} \log N)/N})$ which for the 2-layer MLP is $O(\sqrt{(md \log(md) \log N)/N})$.

We will get good generalization if m is not too large and we can fit the data if m is large enough. A balance is called for. For example, choosing $m = \frac{1}{d}\sqrt{N}$ as $N \to \infty$, $E_{\text{out}} \to E_{\text{in}}$ and $E_{\text{in}} \to E^*_{\text{out}}$. That is, $E_{\text{out}} \to E^*_{\text{out}}$ (the optimal performance) as N grows, and m grows sub-linearly with N. In practice the 'asymptotic' regime is a luxury and one does not simply set $m \approx \sqrt{N}$. These theoretical results are a good guideline, but the best out-of-sample performance usually results when you control overfitting using validation (to select the number of hidden units) and regularization to prevent overfitting.

We conclude with a note on where neural networks sit in the parametric-nonparametric debate. There are explicit parameters to be learned, so parametric seems right. But distinctive features of nonparametric models also stand out: the neural network is generic and flexible and can realize optimal performance when N grows. Neither parametric nor nonparametric captures the whole story. We choose to label neural networks as *semi-parametric*.

7.4 Regularization and Validation

The multi-layer neural network is powerful, and, coupled with gradient descent (a good algorithm to minimize E_{in}), we have a recipe for overfitting. We discuss some practical techniques to help.

7.4.1 Weight Based Complexity Penalties

As with linear models, one can regularize the learning using a complexity penalty by minimizing an augmented error (penalized in-sample error). The squared weight decay regularizer is popular, having augmented error:

$$E_{\text{aug}}(\mathbf{w}) \quad = \quad E_{\text{in}}(\mathbf{w}) + \frac{\lambda}{N} \sum_{\ell,i,j} (w_{ij}^{(\ell)})^2$$

The regularization parameter λ is selected via validation, as discussed in Chapter 4. To apply gradient descent, we need $\nabla E_{\text{aug}}(\mathbf{w})$. The penalty term adds

to the gradient a term proportional to weights,

$$\frac{\partial E_{\text{aug}}(\mathbf{w})}{\partial \mathrm{W}^{(\ell)}} = \frac{\partial E_{\text{in}}(\mathbf{w})}{\partial \mathrm{W}^{(\ell)}} + \frac{2\lambda}{N}\mathrm{W}^{(\ell)}.$$

We know how to obtain $\partial E_{\text{in}}/\partial \mathrm{W}^{(\ell)}$ using backpropagation. The penalty term adds a component to the weight update that is in the negative direction of $\mathbf{w}$, i.e. towards zero weights – hence the term weight decay.

Another similar regularizer is *weight elimination*, having augmented error:

$$E_{\text{aug}}(\mathbf{w}, \lambda) = E_{\text{in}}(\mathbf{w}) + \frac{\lambda}{N}\sum_{\ell,i,j} \frac{(w_{ij}^{(\ell)})^2}{1 + (w_{ij}^{(\ell)})^2}.$$

For a small weight, the penalty term is much like weight decay, and will decay that weight to zero. For a large weight, the penalty term is approximately a constant, and contributes little to the gradient. Small weights decay faster than large weights, and the effect is to 'eliminate' those smaller weights.

Exercise 7.11

For weight elimination, show that $\dfrac{\partial E_{\text{aug}}}{\partial w_{ij}^{(\ell)}} = \dfrac{\partial E_{\text{in}}}{\partial w_{ij}^{(\ell)}} + \dfrac{2\lambda}{N} \cdot \dfrac{w_{ij}^{(\ell)}}{(1 + (w_{ij}^{(\ell)})^2)^2}$.

Argue that weight elimination shrinks small weights faster than large ones.

7.4.2 Early Stopping

Another method for regularization, which on face value does not look like regularization is early stopping. An iterative method such as gradient descent does not explore your full hypothesis set all at once. With more iterations, more of your hypothesis set is explored. This means that by using fewer iterations, you explore a smaller hypothesis set and should get better generalization.[9]

Consider fixed-step gradient descent with step size η. At the first step, we start at weights $\mathbf{w}_0$, and take a step of size η to $\mathbf{w}_1 = \mathbf{w}_0 - \eta\frac{\mathbf{g}_0}{\|\mathbf{g}_0\|}$. Because we have taken a step in the direction of the negative gradient, we have 'looked at' all the hypotheses in the shaded region shown on the right.

This is because a step in the negative gradient leads to the sharpest decrease in $E_{\text{in}}(\mathbf{w})$, and so $\mathbf{w}_1$ minimizes $E_{\text{in}}(\mathbf{w})$ among all weights with $\|\mathbf{w} - \mathbf{w}_0\| \le \eta$. We indirectly searched the entire hypothesis set

$$\mathcal{H}_1 = \{\mathbf{w} : \ \|\mathbf{w} - \mathbf{w}_0\| \le \eta\},$$

and picked the hypothesis $\mathbf{w}_1 \in \mathcal{H}_1$ with minimum in-sample error.

[9] If we are to be sticklers for correctness, the hypothesis set explored could depend on the data set and so we cannot directly apply the VC analysis which requires the hypothesis set to be fixed ahead of time. Since we are just illustrating the main idea, we will brush such technicalities under the rug.

Now consider the second step, as illustrated to the right, which moves to $\mathbf{w}_2$. We indirectly explored the hypothesis set of weights with $\|\mathbf{w} - \mathbf{w}_1\| \leq \eta$, picking the best. Since $\mathbf{w}_1$ was already the minimizer of E_{in} over $\mathcal{H}_0$, this means that $\mathbf{w}_2$ is the minimizer of E_{in} among all hypotheses in $\mathcal{H}_2$, where

$$\mathcal{H}_2 = \mathcal{H}_1 \cup \{\mathbf{w} : \ \|\mathbf{w} - \mathbf{w}_1\| \leq \eta\}.$$

Note that $\mathcal{H}_1 \subset \mathcal{H}_2$. Similarly, we define hypothesis set

$$\mathcal{H}_3 = \mathcal{H}_2 \cup \{\mathbf{w} : \ \|\mathbf{w} - \mathbf{w}_2\| \leq \eta\},$$

and in the 3rd iteration, we pick weights $\mathbf{w}_3$ than minimize E_{in} over $\mathbf{w} \in \mathcal{H}_3$. We can continue this argument as gradient descent proceeds, and define a nested sequence of hypothesis sets

$$\mathcal{H}_1 \subset \mathcal{H}_2 \subset \mathcal{H}_3 \subset \mathcal{H}_4 \subset \cdots .$$

As t increases, $E_{\text{in}}(\mathbf{w}_t)$ is decreasing, and $d_{\text{vc}}(\mathcal{H}_t)$ is increasing. So, we expect to see the approximation-generalization trade-off which was illustrated in Figure 2.3 (reproduced here with iteration t a proxy for d_{vc}):

The figure suggests it may be better to stop early at some t^*, well before reaching a minimum of E_{in}. Indeed, this picture is observed in practice.

Example 7.2. We revisit the digits task of classifying '1' versus all other digits, with 70 randomly selected data points and a small sigmoidal neural network with a single hidden unit and $\tanh(\cdot)$ output node. The figure below shows the in-sample error and test error versus iteration number.

The curves reinforce our theoretical discussion: the test error initially decreases as the approximation gain overcomes the worse generalization error bar; then, the test error increases as the generalization error bar begins to dominate the approximation gain, and overfitting becomes a serious problem. □

In the previous example, despite using a parsimonious neural network with just a single hidden node, overfitting was an issue because the data are noisy and the target function is complex, so both stochastic and deterministic noise are significant. We need to regularize.

In the example, it is better to stop early at t^* and *constrain* the learning to the smaller hypothesis set $\mathcal{H}_{t^*}$. In this sense, early stopping is a form of regularization. Early stopping is related to weight decay, as illustrated to the right. You initialize $\mathbf{w}_0$ near zero; if you stop early at $\mathbf{w}_{t^*}$ you have stopped at weights closer to $\mathbf{w}_0$, i.e., smaller weights. Early stopping indirectly achieves smaller weights, which is what weight decay directly achieves. To determine when to stop training, use a validation set to monitor the validation error at iteration t as you minimize the training-set error. Report the weights $\mathbf{w}_{t^*}$ that have minimum validation error when you are done training.

After selecting t^*, it is tempting to use all the data to train for t^* iterations. Unfortunately, adding back the validation data and training for t^* iterations can lead to a completely different set of weights. The validation estimate of performance only holds for $\mathbf{w}_{t^*}$ (the weights you should output). This appears to go against the wisdom of the decreasing learning curve from Chapter 4: if you learn with more data, you get a better final hypothesis.[10]

[10] Using all the data to train to an in-sample error of $E_{\text{train}}(w_{t^*})$ is also not recommended. Further, an in-sample error of $E_{\text{train}}(w_{t^*})$ may not even be achievable with all the data.

Exercise 7.12

Why does outputting $\mathbf{w}_{t^*}$ rather than training with all the data for t^* iterations *not* go against the wisdom that learning with more data is better.

[Hint: "More data is better" applies to a **fixed** *model $(\mathcal{H}, \mathcal{A})$. Early stopping is model selection on a nested hypothesis sets $\mathcal{H}_1 \subset \mathcal{H}_2 \subset \cdots$ determined by $\mathcal{D}_{\text{train}}$. What happens if you were to use the full data $\mathcal{D}$?]*

When using early stopping, the usual trade-off exists for choosing the size of the validation set: too large and there is little data to train on; too small and the validation error will not be reliable.A rule of thumb is to set aside a fraction of the data (one-tenth to one-fifth) for validation.

Exercise 7.13

Suppose you run gradient descent for 1000 iterations. You have 500 examples in $\mathcal{D}$, and you use 450 for $\mathcal{D}_{\text{train}}$ and 50 for $\mathcal{D}_{\text{val}}$. You output the weight from iteration 50, with $E_{\text{val}}(\mathbf{w}_{50}) = 0.05$ and $E_{\text{train}}(\mathbf{w}_{50}) = 0.04$.

(a) Is $E_{\text{val}}(\mathbf{w}_{50}) = 0.05$ an unbiased estimate of $E_{\text{out}}(\mathbf{w}_{50})$?

(b) Use the Hoeffding bound to get a bound for E_{out} using E_{val} plus an error bar. Your bound should hold with probability at least 0.1.

(c) Can you bound E_{out} using E_{train} or do you need more information?

Example 7.2 also illustrates another common problem with the sigmoidal output node: gradient descent often hits a flat region where E_{in} decreases very little.[11] You might stop training, thinking you found a local minimum. This 'early stopping' by mistake is sometimes called the 'self-regularizing' property of sigmoidal neural networks. Accidental regularization due to misinterpreted convergence is unreliable. Validation is much better.

7.4.3 Experiments With Digits Data

Let's put theory to practice on the digits task (to classify '1' versus all other digits). We learn on 500 randomly chosen data points using a sigmoidal neural network with one hidden layer and 10 hidden nodes. There are 41 weights (tunable parameters), so more than 10 examples per degree of freedom, which is quite reasonable. We use identity output transformation $\theta(s) = s$ to reduce the possibility of getting stuck at a flat region of the error surface. At the end of training, we use the output transformation $\theta(s) = \text{sign}(s)$ for actually classifying data. After more than *2 million* iterations of gradient descent, we manage to get close to a local minimum. The result is shown in Figure 7.2.
It doesn't take a genius to see the overfitting. Figure 7.2 attests to the approximation capabilities of a moderately sized neural network. Let's try weight

[11] The linear output transformation function helps avoid such excessively flat regions.

Figure 7.2: 10 hidden unit neural network trained with gradient descent on 500 examples from the digits data (no regularization). Blue circles are the digit '1' and the red x's are the other digits. Overfitting is rampant.

decay to fight the overfitting. We minimize $E_{\text{aug}}(\mathbf{w}, \lambda) = E_{\text{in}}(\mathbf{w}) + \frac{\lambda}{N}\mathbf{w}^{\text{T}}\mathbf{w}$, with $\lambda = 0.01$. We get a much more believable separator, shown below.

As a final illustration, let's try early stopping with a validation set of size 50 (one-tenth of the data); so the training set will now have size 450.

The training dynamics of gradient descent are shown in Figure 7.3(a). The linear output transformation function has helped as there are no extremely flat periods in the training error. The classification boundary with early stopping at t^* is shown in Figure 7.3(b). The result is similar to weight decay. In both cases, the regularized classification boundary is more believable. Ultimately, the quantitative statistics are what matters, and these are summarized below.

(a) Training dynamics (b) Final hypothesis

Figure 7.3: Early stopping with 500 examples from the digits data. (a) Training and validation errors for gradient descent with a training set of size 450 and validation set of size 50. (b) The 'regularized' final hypothesis obtained by early stopping at t^*, the minimum validation error.

	E_{train}	E_{val}	E_{in}	E_{out}
No Regularization	–	–	0.2%	3.1%
Weight Decay	–	–	1.0%	2.1%
Early Stopping	1.1%	2.0%	1.2%	2.0%

7.5 Beefing Up Gradient Descent

Gradient descent is a simple method to minimize E_{in} that has problems converging, especially with flat error surfaces. One solution is to minimize a friendlier error instead, which is why training with a linear output node helps. Rather than change the error measure, there is plenty of room to improve the algorithm itself. Gradient descent takes a step of size η in the negative gradient direction. How should we determine η and is the negative gradient the best direction in which to move?

Exercise 7.14

Consider the error function $E(\mathbf{w}) = (\mathbf{w} - \mathbf{w}^*)^{\mathrm{T}}\mathrm{Q}(\mathbf{w} - \mathbf{w}^*)$, where Q is an arbitrary positive definite matrix. Set $\mathbf{w} = \mathbf{0}$.

Show that the gradient $\nabla E(\mathbf{w}) = -\mathrm{Q}\mathbf{w}^*$. What weights minimize $E(\mathbf{w})$. Does gradient descent move you in the direction of these optimal weights?

Reconcile your answer with the claim in Chapter 3 that the gradient is the best direction in which to take a step. *[Hint: How big was the step?]*

The previous exercise shows that the negative gradient is not necessarily the best direction for a large step, and we would like to take larger steps to increase

the efficiency of the optimization. The next figure shows two algorithms: our old friend gradient descent and our soon-to-be friend conjugate gradient descent. Both algorithms are minimizing E_{in} for a 5 hidden unit neural network fitting 200 digits data. The performance difference is dramatic.

We now discuss methods for 'beefing up' gradient descent, but only scratch the surface of this important topic known as numerical optimization. The two main steps in an iterative optimization procedure are to determine:

1. Which direction should one search for a local optimum?
2. How large a step should one take in that direction?

7.5.1 Choosing the Learning Rate η

In gradient descent, the learning rate η multiplies the negative gradient to give the move $-\eta\nabla E_{\text{in}}$. The size of the step taken is proportional to η. The optimal step size (and hence learning rate η) depends on how wide or narrow the error surface is near the minimum.

wide: use large η. narrow: use small η.

When the surface is wider, we can take larger steps without overshooting; since $\|\nabla E_{\text{in}}\|$ is small, we need a large η. Since we do not know ahead of time how wide the surface is, it is easy to choose an inefficient value for η.

Variable learning rate gradient descent. A simple heuristic that adapts the learning rate to the error surface works well in practice. If the error drops, increase η; if not, the step was too large, so reject the update and decrease η. For little extra effort, we get a significant boost to gradient descent.

Variable Learning Rate Gradient Descent:

1: Initialize $\mathbf{w}(0)$, and η_0 at $t = 0$. Set $\alpha > 1$ and $\beta < 1$.
2: **while** stopping criterion has not been met **do**
3: Let $\mathbf{g}(t) = \nabla E_{\text{in}}(\mathbf{w}(t))$, and set $\mathbf{v}(t) = -\mathbf{g}(t)$.
4: **if** $E_{\text{in}}(\mathbf{w}(t) + \eta_t \mathbf{v}(t)) < E_{\text{in}}(\mathbf{w}(t))$ **then**
5: accept: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta_t \mathbf{v}(t)$; $\eta_{t+1} = \alpha \eta_t$.
6: **else**
7: reject: $\mathbf{w}(t+1) = \mathbf{w}(t)$; $\eta_{t+1} = \beta \eta_t$.
8: Iterate to the next step, $t \leftarrow t + 1$.

It is usually best to go with a conservative increment parameter, for example $\alpha \approx 1.05 - 1.1$, and a bit more aggressive decrement parameter, for example $\beta \approx 0.5 - 0.8$. This is because, if the error doesn't drop, then one is in an unusual situation and more drastic action is called for.

After a little thought, one might wonder why we need a learning rate at all. Once the direction in which to move, $\mathbf{v}(t)$, has been determined, why not simply continue along that direction until the error stops decreasing? This leads us to *steepest descent* – gradient descent with *line search*.

Steepest Descent. Gradient descent picks a descent direction $\mathbf{v}(t) = -\mathbf{g}(t)$ and updates the weights to $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \mathbf{v}(t)$. Rather than pick η arbitrarily, we will choose the optimal η that minimizes $E_{\text{in}}(\mathbf{w}(t+1))$. Once you have the direction to move, make the best of it by moving along the line $\mathbf{w}(t) + \eta \mathbf{v}(t)$ and stopping when E_{in} is minimum (hence the term line search). That is, choose a step size η^*, where

$$\eta^*(t) = \operatorname*{argmin}_{\eta} E_{\text{in}}(\mathbf{w}(t) + \eta \mathbf{v}(t)).$$

Steepest Descent (Gradient Descent + Line Search):

1: Initialize $\mathbf{w}(0)$ and set $t = 0$;
2: **while** stopping criterion has not been met **do**
3: Let $\mathbf{g}(t) = \nabla E_{\text{in}}(\mathbf{w}(t))$, and set $\mathbf{v}(t) = -\mathbf{g}(t)$.
4: Let $\eta^* = \operatorname{argmin}_{\eta} E_{\text{in}}(\mathbf{w}(t) + \eta \mathbf{v}(t))$.
5: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta^* \mathbf{v}(t)$.
6: Iterate to the next step, $t \leftarrow t + 1$.

The line search in step 4 is a one dimensional optimization problem. Line search is an important step in most optimization algorithms, so an efficient algorithm is called for. Write $E(\eta)$ for $E_{\text{in}}(\mathbf{w}(t) + \eta \mathbf{v}(t))$. The goal is to find a minimum of $E(\eta)$. We give a simple algorithm based on binary search.

Line Search. The idea is to find an interval on the line which is guaranteed to contain a local minimum. Then, rapidly narrow the size of this interval while maintaining as an invariant the fact that it contains a local minimum.

The basic invariant is a U-arrangement:

$$\eta_1 < \eta_2 < \eta_3 \quad \text{with}$$
$$E(\eta_2) < \min\{E(\eta_1), E(\eta_3)\}.$$

Since E is continuous, there must be a local minimum in the interval $[\eta_1, \eta_3]$. Now, consider the midpoint of the interval,

$$\bar{\eta} = \frac{1}{2}(\eta_1 + \eta_3),$$

hence the name *bisection algorithm*. Suppose that $\bar{\eta} < \eta_2$ as shown. If $E(\bar{\eta}) < E(\eta_2)$ then $\{\eta_1, \bar{\eta}, \eta_2\}$ is a new, smaller U-arrangement; and, if $E(\bar{\eta}) > E(\eta_2)$, then $\{\bar{\eta}, \eta_2, \eta_3\}$ is the new smaller U-arrangement. In either case, the bisection process can be iterated with the new U-arrangement. If $\bar{\eta}$ happens to equal η_2, perturb $\bar{\eta}$ slightly to resolve the degeneracy. We leave it to the reader to determine how to obtain the new smaller U-arrangement for the case $\bar{\eta} > \eta_2$.

An efficient algorithm to find an initial U-arrangement is to start with $\eta_1 = 0$ and $\eta_2 = \epsilon$ for some step ϵ. If $E(\eta_2) < E(\eta_1)$, consider the sequence

$$\eta = 0, \epsilon, 2\epsilon, 4\epsilon, 8\epsilon, \ldots$$

(each time the step doubles). At some point, the error must increase. When the error increases for the first time, the last three steps give a U-arrangement. If, instead, $E(\eta_1) < E(\eta_2)$, consider the sequence

$$\eta = \epsilon, 0, -\epsilon, -2\epsilon, -4\epsilon, -8\epsilon, \ldots$$

(the step keeps doubling but in the reverse direction). Again, when the error increases for the first time, the last three steps give a U-arrangement.[12]

Exercise 7.15

Show that $|\eta_3 - \eta_1]$ decreases exponentially in the bisection algorithm. *[Hint: show that two iterations at least halve the interval size.]*

The bisection algorithm continues to bisect the interval and update to a new U-arrangement until the length of the interval $|\eta_3 - \eta_1|$ is small enough, at which

[12] We do not worry about $E(\eta_1) = E(\eta_2)$ – such ties can be broken by small perturbations.

point you can return the midpoint of the interval as the approximate local minimum. Usually 20 iterations of bisection are enough to get an acceptable solution. A better quadratic interpolation algorithm is given in Problem 7.8, which only needs about 4 iterations in practice.

Example 7.3. We illustrate these three heuristics for improving gradient descent on our digit recognition (classifying '1' versus other digits). We use 200 data points and a neural network with 5 hidden units. We show the performance of gradient descent, gradient descent with variable learning rate, and steepest descent (line search) in Figure 7.4. The table below summarizes the in-sample error at various points in the optimization.

	Optimization Time		
Method	10 sec	1,000 sec	50,000 sec
Gradient Descent	0.079	0.0206	0.00144
Stochastic Gradient Descent	**0.0213**	**0.00278**	0.000022
Variable Learning Rate	0.039	0.014	0.00010
Steepest Descent	0.043	0.0189	**0.000012**

Figure 7.4: Gradient descent, variable learning rate and steepest descent using digits data and a 5 hidden unit 2-layer neural network with linear output. For variable learning rate, $\alpha = 1.1$ and $\beta = 0.8$.

Note that SGD is quite competitive. The figure illustrates why it is hard to know when to stop minimizing. A flat region 'trapped' all the methods, even though we used a linear output node transform. It is very hard to differentiate between a flat region (which is typically caused by a very steep valley that leads to inefficient zig-zag behavior) and a true local minimum. □

7.5.2 Conjugate Gradient Minimization

Conjugate gradient is a queen among optimization methods because it leverages a simple principle. Don't undo what you have already accomplished. When you end a line search, because the error cannot be further decreased by moving back or forth along the search direction, it must be that the new gradient and the previous line search direction are orthogonal. What this means is that you have succeeded in setting one of the components of the gradient to zero, namely the component along the search direction $\mathbf{v}(t)$ (see the figure). If the next search direction is the negative of the new gradient, it will be orthogonal to the previous search direction.

You are at a local minimum when the gradient is zero, and setting one component to zero is certainly a step in the right direction. As you move along the next search direction (for example the new negative gradient), the gradient will change and may not remain orthogonal to the previous search direction, a task you laboriously accomplished in the previous line search. The conjugate gradient algorithm chooses the next direction $\mathbf{v}(t+1)$ so that the gradient along this direction, *will remain perpendicular to the previous search direction* $\mathbf{v}(t)$. This is called the conjugate direction, hence the name. After a line search along this new direction $\mathbf{v}(t+1)$ to minimize E_{in}, you will have set *two* components of the gradient to zero. First, the gradient remained perpendicular to the previous search direction $\mathbf{v}(t)$. Second, the gradient will be orthogonal to $\mathbf{v}(t+1)$ because of the line search (see the figure). The gradient along the new direction $\mathbf{v}(t+1)$ is shown by the blue arrows in the figure. Because $\mathbf{v}(t+1)$ is conjugate to $\mathbf{v}(t)$, observe how the gradient as we move along $\mathbf{v}(t+1)$ remains orthogonal to the previous direction $\mathbf{v}(t)$.

Exercise 7.16

Why does the new search direction pass through the optimal weights?

We made progress! Now two components of the gradient are zero. In two dimensions, this means that the gradient itself must be zero and we are done.

In higher dimension, if we could continue to set a component of the gradient to zero with each line search, maintaining all previous components at zero, we will eventually set every component of the gradient to zero and be at a local minimum. Our discussion is true for an idealized quadratic error function. In general, conjugate gradient minimization implements our idealized expectations approximately. Nevertheless, it works like a charm because the idealized setting is a good approximation once you get close to a local minimum, and this is where algorithms like gradient descent become ineffective.

Now for the details. The algorithm constructs the current search direction as a linear combination of the previous search direction and the current gradient,

$$\mathbf{v}(t) = -\mathbf{g}(t) + \mu_t \cdot \mathbf{v}(t-1),$$

where

$$\mu_t = \frac{\mathbf{g}(t+1)^{\mathrm{T}}(\mathbf{g}(t+1) - \mathbf{g}(t))}{\mathbf{g}(t)^{\mathrm{T}}\mathbf{g}(t)}.$$

The term $\mu_t \cdot \mathbf{v}(t-1)$ is called the *momentum* term because it asks you to keep moving in the same direction you were moving in. The multiplier μ_t is called the momentum parameter. The full conjugate gradient descent algorithm is summarized in the following algorithm box.

Conjugate Gradient Descent:

1: Initialize $\mathbf{w}(0)$ and set $t = 0$; set $\mathbf{v}(-1) = \mathbf{0}$
2: **while** stopping criterion has not been met **do**
3: Let $\mathbf{v}(t) = -\mathbf{g}(t) + \mu_t \mathbf{v}(t-1)$, where

$$\mu_t = \frac{\mathbf{g}(t+1)^{\mathrm{T}}(\mathbf{g}(t+1) - \mathbf{g}(t))}{\mathbf{g}(t)^{\mathrm{T}}\mathbf{g}(t)}.$$

4: Let $\eta^* = \operatorname{argmin}_\eta E_{\text{in}}(\mathbf{w}(t) + \eta \mathbf{v}(t))$.
5: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta^* \mathbf{v}(t)$;
6: Iterate to the next step, $t \leftarrow t+1$;

The only difference between conjugate gradient descent and steepest descent is in step 3 where the line search direction is different from the negative gradient. Contrary to intuition, the negative gradient direction is not always the best direction to move, because it can undo some of the good work you did before.

In practice, for error surfaces that are not exactly quadratic, the $\mathbf{v}(t)$'s are only approximately conjugate and it is recommended that you 'restart' the algorithm by setting μ_t to zero every so often (for example every d iterations). That is, every d iterations you throw in a steepest descent iteration.

Example 7.4. Continuing with the digits example, we compare conjugate gradient and the previous champion steepest descent in the next table and Figure 7.5.

Figure 7.5: Steepest descent versus conjugate gradient descent using 200 examples of the digits data and a 2-layer sigmoidal neural network with 5 hidden units.

	Optimization Time		
Method	10 sec	1,000 sec	50,000 sec
Steepest Descent	0.043	0.0189	1.2×10^{-5}
Conjugate Gradients	**0.0200**	$\mathbf{1.13 \times 10^{-6}}$	$\mathbf{2.73 \times 10^{-9}}$

The performance difference is dramatic. □

7.6 Deep Learning: Networks with Many Layers

Universal approximation says that a single hidden layer with enough hidden units can approximate any target function. But, that may not be a natural way to represent the target function. Often, many layers more closely mimics human learning. Let's get our feet wet with the digit recognition problem to classify '1' versus '5'. A natural first step is to decompose the two digits into basic components, just as one might break down a face into two eyes, a nose, a mouth, two ears, etc. Here is one attempt for a prototypical '1' and '5'.

Indeed, we could plausibly argue that every '1' should contain a ϕ_1, ϕ_2 and ϕ_3; and, every '5' should contain a ϕ_3, ϕ_4, ϕ_5, ϕ_6 and perhaps a little ϕ_1. We have deliberately used the notation ϕ_i which we used earlier for the coordinates of the feature transform Φ. These basic shapes are *features* of the input, and, for example, we would like ϕ_1 to be large (close to 1) if its corresponding feature is in the input image and small (close to -1) if not.

Exercise 7.17

The basic shape ϕ_3 is in both the '1' and the '5'. What other digits do you expect to contain each basic shape $\phi_1 \cdots \phi_6$. How would you select additional basic shapes if you wanted to distinguish between all the digits. (What properties should useful basic shapes satisfy?)

We can build a classifier for '1' versus '5' from these basic shapes. Remember how, at the beginning of the chapter, we built a complex Boolean function from the 'basic' functions AND and OR? Let's mimic that process here. The complex function we are building is the digit classifier and the basic functions are our features. Assume, for now, that we have feature functions ϕ_i which compute the presence $(+1)$ or absence (-1) of the corresponding feature. Take a close look at the following network and work it through from input to output.

Ignoring details like the exact values of the weights, node z_1 answers the question "is the image a '1'?" and similarly node z_5 answers "is the image a '5'?" Let's see why. If they have done their job correctly when we feed in a '1', ϕ_1, ϕ_2, ϕ_3 compute $+1$, and ϕ_4, ϕ_5, ϕ_6 compute -1. Combining $\phi_1, \ldots, \phi_6$ with the signs of the weights on outgoing edges, all the inputs to z_1 will be positive hence z_1 outputs $+1$; all but one of the inputs into z_5 are negative, hence z_5 outputs -1. A similar analysis holds if you feed in the '5'. The final

node combines z_1 and z_5 to the final output. At this point, it is useful to fill in all the blanks with an exercise.

Exercise 7.18

Since the input $\mathbf{x}$ is an image it is convenient to represent it as a matrix $[x_{ij}]$ of its pixels which are black ($x_{ij} = 1$) or white ($x_{ij} = 0$). The basic shape ϕ_k identifies a set of these pixels which are black.

(a) Show that feature ϕ_k can be computed by the neural network node

$$\phi_k(\mathbf{x}) = \tanh\left(w_0 + \sum_{ij} w_{ij} x_{ij}\right).$$

(b) What are the inputs to the neural network node?

(c) What do you choose as values for the weights? *[Hint: consider separately the weights of the pixels for those $x_{ij} \in \phi_k$ and those $x_{ij} \notin \phi_k$.]*

(d) How would you choose w_0? (Not all digits are written identically, and so a basic shape may not always be exactly represented in the image.)

(e) Draw the final network, filling in as many details as you can.

You may have noticed, that the output of z_1 is all we need to solve our problem. This would not be the case if we were solving the full multi-class problem with nodes $z_0, \ldots, z_9$ corresponding to all ten digits. Also, we solved our problem with relative ease – our 'deep' network has just 2 hidden layers. In a more complex problem, like face recognition, the process would start just as we did here, with basic shapes. At the next level, we would constitute more complicated shapes from the basic shapes, but we would not be home yet. These more complicated shapes would constitute still more complicated shapes until at last we had realistic objects like eyes, a mouth, ears, etc. There would be a hierarchy of 'basic' features until we solve our problem at the very end.

Now for the punch line and crux of our story. The punch line first. Shine your floodlights back on the network we constructed, and scrutinize what the different layers are doing. The first layer constructs a low-level representation of basic shapes; the next layer builds a higher level representation from these basic shapes. As we progress up more layers, we get more complex representations in terms of simpler parts from the previous layer: an 'intelligent' decomposition of the problem, starting from simple and getting more complex, until finally the problem is solved. This is the promise of the deep network, that it provides some human-like insight into how the problem is being solved based on a hierarchy of more complex representations for the input. While we might attain a solution of similar accuracy with a single hidden layer, we would gain no such insight. The picture is rosy for our intuitive digit recognition problem, but here is the crux of the matter: for a complex learning problem, how do we automate all of this in a computer algorithm?

Figure 7.6: Greedy deep learning algorithm. (a) First layer weights are learned. (b) First layer is fixed and second layer weights are learned. (c) First two layers are fixed and third layer weights are learned. (d) Learned weights can be used as a starting point to fine-tune the entire network.

7.6.1 A Greedy Deep Learning Algorithm

Historically, the shallow (single hidden layer) neural network was favored over the deep network because deep networks are hard to train, suffer from many local minima and, relative to the number of tunable parameters, they have a very large tendency to overfit (composition of nonlinearities is typically much more powerful than a linear combination of nonlinearities). Recently, some simple heuristics have shown good performance empirically and have brought deep networks back into the limelight. Indeed, the current best algorithm for digit recognition is a deep neural network trained with such heuristics.

The greedy heuristic has a general form. Learn the first layer weights $\mathrm{W}^{(1)}$ and fix them.[13] The output of the first hidden layer is a nonlinear transformation of the inputs $\mathbf{x}_n \to \mathbf{x}_n^{(1)}$. These outputs $\mathbf{x}_n^{(1)}$ are used to train the second layer weights $\mathrm{W}^{(2)}$, *while keeping the first layer weights fixed.* This is the essence of the greedy algorithm, to 'greedily' pick the first layer weights, fix them, and then move on to the second layer weights. One ignores the possibility that better first layer weights might exist if one takes into account what the second layer is doing. The process continues with the outputs $\mathbf{x}^{(2)}$ used to learn the weights $\mathrm{W}^{(3)}$, and so on.

[13] Recall that we use the superscript $(\cdot)^{(\ell)}$ to denote the layer ℓ.

Greedy Deep Learning Algorithm:

1: **for** $\ell = 1, \ldots, L$ **do**
2: $\mathrm{W}^{(1)} \cdots \mathrm{W}^{(\ell-1)}$ are given from previous iterations.
3: Compute layer $\ell - 1$ outputs $\mathbf{x}_n^{(\ell-1)}$ for $n = 1, \ldots, N$.
4: Use $\{\mathbf{x}_n^{(\ell-1)}\}$ to learn weights W^{ℓ} by training a *single* hidden layer neural network. ($\mathrm{W}^{(1)} \cdots \mathrm{W}^{(\ell-1)}$ are fixed.)

We have to clarify step 4 in the algorithm. The weights $\mathrm{W}^{(\ell)}$ and V are learned, though V is not needed in the algorithm. To learn the weights, we minimize an error (which will depend on the output of the network), and that error is not yet defined. To define the error, we must first define the output and then how to compute the error from the output.

Unsupervised Auto-encoder. One approach is to take to heart the notion that the hidden layer gives a high-level representation of the inputs. That is, we should be able to reconstruct all the important aspects of the input from the hidden layer output . A natural test is to reconstruct the input itself: the output will be $\hat{\mathbf{x}}_n$, a prediction of the input $\mathbf{x}_n$; and, the error is the difference between the two. For example, using squared error,

$$\mathsf{e}_n = \|\hat{\mathbf{x}}_n - \mathbf{x}_n\|^2.$$

When all is said and done, we obtain the weights without using the targets y_n and the hidden layer gives an encoding of the inputs, hence the name unsupervised auto-encoder. This is reminiscent of the radial basis function network in Chapter 6, where we used an unsupervised technique to learn the centers of the basis functions, which provided a representative set of inputs as the centers. Here, we go one step further and dissect the input-space itself into pieces that are representative of the learning problem. At the end, the targets have to be brought back into the picture (usually in the output layer).

Supervised Deep Network. The previous approach adheres to the philosophical goal that the hidden layers provide an 'intelligent' hierarchical representation of the inputs. A more direct approach is to train the two-layer network on the targets. In this case the output is the predicted target $\hat{y}_n$ and the error measure $\mathsf{e}_n(y_n, \hat{y}_n)$ would be computed in the usual way (for example squared error, cross entropy error, etc.).

In practice, there is no verdict on which method is better, with the unsupervised auto-encoder camp being slightly more crowded than the supervised camp. Try them both and see what works for your problem, that's usually the best way. Once you have your error measure, you just reach into your optimization toolbox and minimize the error using your favorite method (gradient descent, stochastic gradient descent, conjugate gradient descent, ...). A common tactic is to use the unsupervised auto-encoder first to set the weights and then fine tune the whole network using supervised learning. The idea is that the unsupervised pass gets you to the right local minimum of the full network. But, no matter which camp you belong to, you still need to choose the architecture of the deep network (number of hidden layers and their sizes), and there is no magic potion for that. You will need to resort to old tricks like validation, or a deep understanding ☺ of the problem (our hand made network for the '1' versus '5' task suggests a deep network with six hidden nodes in the first hidden layer and two in the second).

Exercise 7.19

Previously, for our digit problem, we used symmetry and intensity. How do these features relate to deep networks? Do we still need them?

Example 7.5. Deep Learning For Digit Recognition. Let's revisit the digits classification problem '1' versus '5' using a deep network architecture

$$[d^{(0)}, d^{(1)}, d^{(2)}, d^{(3)}] = [256, 6, 2, 1].$$

(The same architecture we constructed by hand earlier, with 16×16 input pixels and 1 output.) We will use gradient descent to train the two layer networks in the greedy algorithm. A convenient matrix form for the gradient of the two layer network is given in Problem 7.7. For the unsupervised auto-encoder the target output is the input matrix X. for the supervised deep network, the target output is just the target vector $\mathbf{y}$. We used the supervised approach with 1,000,000 gradient descent iterations for each supervised greedy step using a sample of 1500 examples from the digits data. Here is a look at what the 6 hidden units in the first hidden layer learned. For each hidden node in the first hidden layer, we show the pixels corresponding to the top 20 incoming weights.

Real data is not as clean as our idealized analysis. Don't be surprised. Nevertheless, we can discern that ϕ_2 has picked out the pixels (shapes) in the typical '1' that are unlikely to be in a typical '5'. The other features seem to focus on the '5' and to some extent match our hand constructed features. Let's not dwell on whether the representation captures human intuition; it does to some extent. The important thing is that this result is automatic and purely data driven (other than our choice of the network architecture); and, what matters is out-of-sample performance. For different architectures, we ran more than 1000 validation experiments selecting 500 random training points each time and the remaining data as a test set.

Deep Network Architecture	E_{in}	E_{test}
$[256, 3, 2, 1]$	0	0.170%
$[256, 6, 2, 1]$	0	0.187%
$[256, 12, 2, 1]$	0	0.187%
$[256, 24, 2, 1]$	0	0.183%

E_{in} is always zero because there are so many parameters, even with just 3 hidden units in the first hidden layer. This smells of overfitting. But, the test performance is impressive at 99.8% accuracy, which is all we care about. Our hand constructed features of symmetry and intensity were good, but not quite this good. □

7.7 Problems

Problem 7.1 Implement the decision function below using a 3-layer perceptron.

Problem 7.2 A set of M hyperplanes will generally divide the space into some number of regions. Every point in $\mathbb{R}^d$ can be labeled with an M dimensional vector that determines which side of each plane it is on. Thus, for example if $M = 3$, then a point with a vector $(-1, +1, +1)$ is on the -1 side of the first hyperplane, and on the +1 side of the second and third hyperplanes. A region is defined as the set of points with the *same* label.

(a) Prove that the regions with the same label are convex.

(b) Prove that M hyperplanes can create at most 2^M distinct regions.

(c) **[hard]** What is the maximum number of regions created by M hyperplanes in d dimensions?

[Answer: $\sum_{i=0}^{d} \binom{M}{d}$ *.]*

[Hint: Use induction and let $B(M, d)$ be the number of regions created by M $(d-1)$-dimensional hyperplanes in d-space. Now consider adding the $(M+1)$th hyperplane. Show that this hyperplane intersects at most $B(M, d-1)$ of the $B(M, d)$ regions. For each region it intersects, it adds exactly one region, and so $B(M+1, d) \leq B(M, d) + B(M, d-1)$. (Is this recurrence familiar?) Evaluate the boundary conditions: $B(M, 1)$ and $B(1, d)$, and proceed from there. To see that the $M+1$th hyperplane only intersects $B(M, d-1)$ regions, argue as follows. Treat the $M+1$th hyperplane as a $(d-1)$-dimensional space, and project the initial M hyperplanes into this space to get M hyperplanes in a $(d-1)$-dimensional space. These M hyperplanes can create at most $B(M, d-1)$ regions in this space. Argue that this means that the $M+1$th hyperplane is only intersecting at most $B(M, d-1)$ of the regions created by the M hyperplanes in d-space.]

Problem 7.3 Suppose that a target function f (for classification) is represented by a number of hyperplanes, where the different regions defined by the hyperplanes (see Problem 7.2) could be either classified $+1$ or -1, as with the 2-dimensional examples we considered in the text. Let the hyperplanes be $h_1, h_2, \ldots, h_M$, where $h_m(\mathbf{x}) = \text{sign}(\mathbf{w}_m \bullet \mathbf{x})$. Consider all the regions that are classified $+1$, and let one such region be r^+. Let $\mathbf{c} = (c_1, c_2, \ldots, c_M)$ be the label of any point in the region (all points in a given region have the same label); the label $c_m = \pm 1$ tells which side of h_m the point is on. Define the AND-term corresponding to region r by

$$t_r = h_1^{c_1} h_2^{c_2} \ldots h_M^{c_M}, \text{ where } h_m^{c_m} = \begin{cases} h_m & \text{if } c_m = +1, \\ \bar{h}_m & \text{if } c_m = -1. \end{cases}$$

Show that $f = t_{r_1} + t_{r_2} + \cdots + t_{r_k}$, where $r_1, \ldots, r_k$ are all the positive regions. (We use multiplication for the AND and addition for the OR operators.)

Problem 7.4 Referring to Problem 7.3, any target function which can be decomposed into hyperplanes $h_1, \ldots, h_M$ can be represented by $f = t_{r_1} + t_{r_2} + \cdots + t_{r_k}$, where there are k positive regions.

What is the structure of the 3-layer perceptron (number of hidden units in each layer) that will implement this function, proving the following theorem:

Theorem. Any decision function whose ± 1 regions are defined in terms of the regions created by a set of hyperplanes can be implemented by a 3-layer perceptron.

Problem 7.5 **[Hard]** State and prove a version of a *Universal Approximation Theorem*:

Theorem. Any target function f (for classification) defined on $[0,1]^d$, whose classification boundary surfaces are smooth, can arbitrarily closely be approximated by a 3-layer perceptron.

[Hint: Decompose the unit hypercube into ϵ-hypercubes ($\frac{1}{\epsilon^d}$ of them); The volume of these ϵ-hypercubes which intersects the classification boundaries must tend to zero (why? – use smoothness). Thus, the function which takes on the value of f on any ϵ-hypercube that does not intersect the boundary and an arbitrary value on these boundary ϵ-hypercubes will approximate f arbitrarily closely, as $\epsilon \to 0$.]

Problem 7.6 The finite difference approximation to obtaining the gradient is based on the following formula from calculus:

$$\frac{\partial h}{\partial w_{ij}^{(\ell)}} = \frac{h(w_{ij}^{(\ell)} + \epsilon) - h(w_{ij}^{(\ell)} - \epsilon)}{2\epsilon} + O(\epsilon^2),$$

where $h(w_{ij}^{(\ell)} + \epsilon)$ to denotes the function value when all weights are held at their values in $\mathbf{w}$ except for the weight $w_{ij}^{(\ell)}$, which is perturbed by ϵ. To get the gradient, we need the partial derivative with respect to each weight.

Show that the computational complexity of obtaining all these partial derivatives $O(W^2)$. *[Hint: you have to do two forward propagations for each weight.]*

Problem 7.7 Consider the 2-layer network below, with output *vector* $\hat{\mathbf{y}}$. This is the two layer network used for the greedy deep network algorithm.

Collect the input vectors $\mathbf{x}_n$ (together with a column of ones) as rows in the input data matrix X, and similarly form Z from $\mathbf{z}_n$. The target matrices Y and $\hat{\mathrm{Y}}$ are formed from $\mathbf{y}_n$ and $\hat{\mathbf{y}}_n$ respectively. Assume a linear output node and the hidden layer activation is $\theta(\cdot)$.

(a) Show that the in-sample error is

$$E_{\text{in}} = \tfrac{1}{N}\text{trace}\left((\mathrm{Y} - \hat{\mathrm{Y}})(\mathrm{Y} - \hat{\mathrm{Y}})^{\mathrm{T}}\right),$$

where

$$\begin{aligned} \hat{\mathrm{Y}} &= \mathrm{ZV} \\ \mathrm{Z} &= [\mathbf{1}, \theta(\mathrm{XW})] \end{aligned} \quad \text{and} \quad \begin{array}{ccc} \mathrm{X} & \text{is} & N \times (d+1) \\ \mathrm{W} & \text{is} & (d+1) \times d^{(1)} \\ \mathrm{Z} & \text{is} & N \times (d^{(1)}+1) \\ \mathrm{V} = \left[\begin{smallmatrix} \mathrm{V}_0 \\ \mathrm{V}_1 \end{smallmatrix}\right] & \text{is} & (d^{(1)}+1) \times \text{dim}(\mathbf{y}) \\ \mathrm{Y}, \hat{\mathrm{Y}} & \text{are} & N \times \text{dim}(\mathbf{y}) \end{array}$$

(It is convenient to decompose V into its first row V_0 corresponding to the biases and its remaining rows V_1; $\mathbf{1}$ is the $N \times 1$ vector of ones.)

(b) derive the gradient matrices:

$$\begin{aligned} \frac{\partial E_{\text{in}}}{\partial \mathrm{V}} &= 2\mathrm{Z}^{\mathrm{T}}\mathrm{ZV} - 2\mathrm{Z}^{\mathrm{T}}\mathrm{Y} \\ \frac{\partial E_{\text{in}}}{\partial \mathrm{W}}; &= 2\mathrm{X}^{\mathrm{T}}\left[\theta'(\mathrm{XW}) \otimes (\theta(\mathrm{XW})\mathrm{V}_1\mathrm{V}_1^{\mathrm{T}} + \mathbf{1}\mathrm{V}_0\mathrm{V}_1^{\mathrm{T}} - \mathrm{Y}\mathrm{V}_1^{\mathrm{T}})\right], \end{aligned}$$

where $\otimes$ denotes element-wise multiplication. Some of the matrix derivatives of functions involving the trace from the appendix may be useful.

Problem 7.8 Quadratic Interpolation for Line Search

Assume that a U-arrangement has been found, as illustrated below.

Instead of using bisection to construct the point $\bar{\eta}$, quadratic interpolation fits a quadratic curve $E(\eta) = a\eta^2 + b\eta + c$ to the three points and uses the minimum of this quadratic interpolation as $\bar{\eta}$.

(a) Show that the minimum of the quadratic interpolant for a U-arrangement is within the interval $[\eta_1, \eta_3]$.

(b) Let $e_1 = E(\eta_1)$, $e_2 = E(\eta_2)$, $e_3 = E(\eta_3)$. Obtain the quadratic function that interpolates the three points $\{(\eta_1, e_1), (\eta_2, e_2), (\eta_3, e_3)\}$. Show that the minimum of this quadratic interpolant is given by:

$$\bar{\eta} = \frac{1}{2}\left[\frac{(e_1 - e_2)(\eta_1^2 - \eta_3^2) - (e_1 - e_3)(\eta_1^2 - \eta_2^2)}{(e_1 - e_2)(\eta_1 - \eta_3) - (e_1 - e_3)(\eta_1 - \eta_2)}\right]$$

[Hint: $e_1 = a\eta_1^2 + b\eta_1 + c$, $e_2 = a\eta_2^2 + b\eta_2 + c$, $e_3 = a\eta_3^2 + b\eta_3 + c$. Solve for a, b, c and the minimum of the quadratic is given by $\bar{\eta} = -b/2a$.]

(c) Depending on whether $E(\bar{\eta})$ is less than $E(\eta_2)$, and on whether $\bar{\eta}$ is to the left or right of η_2, there are 4 cases.

In each case, what is the smaller U-arrangement?

(d) What if $\bar{\eta} = \eta_2$, a degenerate case?

Note: in general the quadratic interpolations converge very rapidly to a locally optimal η. In practice, 4 iterations are more than sufficient.

Problem 7.9 [Convergence of Monte-Carlo Minimization]

Suppose the global minimum $\mathbf{w}^*$ is in the unit cube and the error surface is quadratic near $\mathbf{w}^*$. So, near $\mathbf{w}^*$,

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathrm{H}(\mathbf{w} - \mathbf{w}^*)$$

where the Hessian H is a positive definite and symmetric.

(a) If you uniformly sample $\mathbf{w}$ in the unit cube, show that

$$P[E \leq E(\mathbf{w}^*) + \epsilon] = \int_{\mathbf{x}^T \mathrm{H} \mathbf{x} \leq 2\epsilon} d^d\mathbf{x} = \frac{S_d(2\epsilon)}{\sqrt{\det \mathrm{H}}},$$

where $S_d(r)$ is the volume of the $d-$dimensional sphere of radius r,

$$S_d(r) = \pi^{d/2} r^d / \Gamma(\tfrac{d}{2} + 1).$$

[Hints: $P[E \leq E(\mathbf{w}^*) + \epsilon] = P\left[\frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathrm{H}(\mathbf{w} - \mathbf{w}^*) \leq \epsilon\right]$. *Suppose the orthogonal matrix* A *diagonalizes* H*:* $\mathrm{A}^{\mathrm{T}}\mathrm{H}\mathbf{A} = diag[\lambda_1^2, \ldots, \lambda_d^2]$. *Change variables to* $\mathbf{u} = \mathrm{A}^{\mathrm{T}}\mathbf{x}$ *and use* $\det \mathrm{H} = \lambda_1^2 \lambda_2^2 \cdots \lambda_d^2$*.]*

(b) Suppose you sample M times and choose the weights with minimum error, $\mathbf{w}_{\min}$. Show that

$$P[E(\mathbf{w}_{\min}) > E(\mathbf{w}^*) + \epsilon] \approx \left(1 - \frac{1}{\pi d}\left(\mu \frac{\epsilon}{\sqrt{d}}\right)^d\right)^N,$$

where $\mu \approx \sqrt{8e\pi/\bar{\lambda}}$ and $\bar{\lambda}$ is the geometric mean of the eigenvalues of H. (You may use $\Gamma(x+1) \approx x^x e^{-x}\sqrt{2\pi x}$.)

(c) Show that if $N \sim (\frac{\sqrt{d}}{\mu\epsilon})^d \log \frac{1}{\eta}$, then with probability at least $1 - \eta$, $E(\mathbf{w}_{\min}) \leq E(\mathbf{w}^*) + \epsilon$.
(You may use $\log(1-a) \approx -a$ for small a and $(\pi d)^{1/d} \approx 1$.)

Problem 7.10 For a neural network with at least 1 hidden layer and $\tanh(\cdot)$ transformations in each non-input node, what is the gradient (with respect to the weights) if all the weights are set to zero.

Is it a good idea to initialize the weights to zero?

Problem 7.11 **[Optimal Learning Rate]** Suppose that we are in the vicinity of a local minimum, $\mathbf{w}^*$, of the error surface, or that the error surface is quadratic. The expression for the error function is then given by

$$E(\mathbf{w}_t) = E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w}_t - \mathbf{w}^*)\mathrm{H}(\mathbf{w}_t - \mathbf{w}^*) \tag{7.8}$$

from which it is easy to see that the gradient is given by $\mathbf{g}_t = \mathrm{H}(\mathbf{w}_t - \mathbf{w}^*)$. The weight updates are then given by $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\mathrm{H}(\mathbf{w}_t - \mathbf{w}^*)$, and subtracting $\mathbf{w}^*$ from both sides, we see that

$$\epsilon_{t+1} = (\mathrm{I} - \eta\mathrm{H})\epsilon_t \tag{7.9}$$

Since H is symmetric, one can form an orthonormal basis with its eigenvectors. Projecting ϵ_t and ϵ_{t+1} onto this basis, we see that in this basis, each component

decouples from the others, and letting $\epsilon(\alpha)$ be the αth component in this basis, we see that

$$\epsilon_{t+1}(\alpha) = (1 - \eta\lambda_\alpha)\epsilon_t(\alpha) \tag{7.10}$$

so we see that each component exhibits linear convergence with its own coefficient of convergence $k_\alpha = 1 - \eta\lambda_\alpha$. The worst component will dominate the convergence so we are interested in choosing η so that the k_α with largest magnitude is minimized. Since H is positive definite, all the λ_α's are positive, so it is easy to see that one should choose η so that $1 - \eta\lambda_{min} = 1 - \Delta$ and $1 - \eta\lambda_{max} = 1 + \Delta$, or one should choose. Solving for the optimal η, one finds that

$$\eta_{opt} = \frac{2}{\lambda_{min} + \lambda_{max}} \qquad k_{opt} = \frac{1-c}{1+c} \tag{7.11}$$

where $c = \lambda_{min}/\lambda_{max}$ is the condition number of H, and is an important measure of the stability of H. When $c \approx 0$, one usually says that H is ill-conditioned. Among other things, this affects the one's ability to numerically compute the inverse of H.

Problem 7.12 **[Hard]** With a variable learning rate, suppose that $\eta_t \to 0$ satisfying $\sum_t 1/\eta_t = \infty$ and $\sum_t 1/\eta_t^2 < \infty$, for example one could choose $\eta_t = 1/(t+1)$. Show that gradient descent will converge to a local minimum.

Problem 7.13 [Finite Difference Approximation to Hessian]

(a) Consider the function $E(w_1, w_2)$. Show that the finite difference approximation to the second order partial derivatives are given by

$$\frac{\partial^2 E}{\partial w_1^2} = \frac{E(w_1+2h,w_2)+E(w_1-2h,w_2)-2E(w_1,w_2)}{4h^2}$$

$$\frac{\partial^2 E}{\partial w_2^2} = \frac{E(w_1,w_2+2h)+E(w_1,w_2-2h)-2E(w_1,w_2)}{4h^2}$$

$$\frac{\partial^2 E}{\partial w_2^2} = \frac{E(w_1+h,w_2+h)+E(w_1-h,w_2-h)-E(w_1+h,w_2-h)-E(w_1-h,w_2+h)}{4h^2}$$

(b) Give an algorithm to compute the finite difference approximation to the Hessian matrix for $E_{\text{in}}(\mathbf{w})$, the in-sample error for a multilayer neural network with weights $\mathbf{w} = [\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(\ell)}]$.

(c) Compute the asymptotic running time of your algorithm in terms of the number of weights in your network and then number of data points.

Problem 7.14 Suppose we take a fixed step in some direction, we ask what the optimal direction for this fixed step assuming that the quadratic model for the error surface is accurate:

$$E_{\text{in}}(\mathbf{w}_t + \delta\mathbf{w}) = E_{\text{in}}(\mathbf{w}_t) + \mathbf{g}_t^{\mathrm{T}}\Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^{\mathrm{T}}\mathrm{H}_t\Delta\mathbf{w}.$$

So we want to minimize $E_{\text{in}}(\Delta\mathbf{w})$ with respect to $\Delta\mathbf{w}$ subject to the constraint that the step size is η, i.e., that $\Delta\mathbf{w}^{\text{T}}\Delta\mathbf{w} = \eta^2$.

(a) Show that the Lagrangian for this constrained minimization problem is:

$$\mathcal{L} = E_{\text{in}}(\mathbf{w}_t) + \mathbf{g}_t^{\text{T}}\Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^{\text{T}}(\text{H}_t + 2\alpha\text{I})\Delta\mathbf{w}, \tag{7.12}$$

where α is the Lagrange multiplier.

(b) Solve for $\Delta\mathbf{w}$ and α and show that they satisfy the two equations:

$$\begin{aligned} \Delta\mathbf{w} &= -(\text{H}_t + 2\alpha\text{I})^{-1}\mathbf{g}_t, \\ \Delta\mathbf{w}^{\text{T}}\Delta\mathbf{w} &= \eta^2. \end{aligned}$$

(c) Show that α satisfies the implicit equation:

$$\alpha = -\frac{1}{2\eta^2}(\Delta\mathbf{w}^{\text{T}}\mathbf{g}_t + \Delta\mathbf{w}^{\text{T}}\text{H}_t\Delta\mathbf{w}).$$

Argue that the second term is $\theta(1)$ and the first is $O(\sim \|\mathbf{g}_t\|/\eta)$. So, α is large for a small step size η.

(d) Assume that α is large. Show that, To leading order in $\frac{1}{\eta}$

$$\alpha = \frac{\|\mathbf{g}_t\|}{2\eta}.$$

Therefore α is large, consistent with expanding $\Delta\mathbf{w}$ to leading order in $\frac{1}{\alpha}$. *[Hint: expand $\Delta\mathbf{w}$ to leading order in $\frac{1}{\alpha}$.]*

(e) Using (d), show that $\Delta\mathbf{w} = -\left(\text{H}_t + \frac{\|\mathbf{g}_t\|}{\eta}\text{I}\right)^{-1}\mathbf{g}_t$.

Problem 7.15 The outer-product Hessian approximation is $\text{H} = \sum_{n=1}^{N}\mathbf{g}_n\mathbf{g}_n^{\text{T}}$. Let $\text{H}_k = \sum_{n=1}^{k}\mathbf{g}_n\mathbf{g}_n^{\text{T}}$ be the partial sum to k, and let H_k^{-1} be its inverse.

(a) Show that $\text{H}_{k+1}^{-1} = \text{H}_k^{-1} - \dfrac{\text{H}_k^{-1}\mathbf{g}_{k+1}\mathbf{g}_{k+1}^{\text{T}}\text{H}_k^{-1}}{1 + \mathbf{g}_{k+1}^{\text{T}}\text{H}_k^{-1}\mathbf{g}_{k+1}}$.

[Hints: $\text{H}_{k+1} = \text{H}_k + \mathbf{g}_{k+1}\mathbf{g}_{k+1}^{\text{T}}$*; and,* $(\text{A}+\mathbf{z}\mathbf{z}^{\text{T}})^{-1} = \text{A}^{-1} - \frac{\text{A}^{-1}\mathbf{z}\mathbf{z}^{\text{T}}\text{A}^{-1}}{1+\mathbf{z}^{\text{T}}\text{A}^{-1}\mathbf{z}}$*.]*

(b) Use part (a) to give an $O(NW^2)$ algorithm to compute H_t^{-1}, the same time it takes to compute H. (W is the number of dimensions in $\mathbf{g}$).

Note: typically, this algorithm is initialized with $\text{H}_0 = \epsilon\text{I}$ for some small ϵ. So the algorithm actually computes $(\text{H} + \epsilon\text{I})^{-1}$; the results are not very sensitive to the choice of ϵ, as long as ϵ is small.

Problem 7.16 In the text, we computed an upper bound on the VC-dimension of the 2-layer perceptron is $d_{\text{VC}} = O(md\log(md))$ where m is the number of hidden units in the hidden layer. Prove that this bound is essentially tight by showing that $d_{\text{VC}} = \Omega(md)$. To do this, show that it is possible to find md points that can be shattered when m is even as follows.

Consider any set of N points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in general position with $N = md$. N points in d dimensions are in general position if no subset of $d+1$ points lies on a $d-1$ dimensional hyperplane. Now, consider any dichotomy on these points with r of the points classified $+1$. Without loss of generality, relabel the points so that $\mathbf{x}_1, \ldots, \mathbf{x}_r$ are $+1$.

(a) Show that without loss of generality, you can assume that $r \leq N/2$. For the rest of the problem you may therefore assume that $r \leq N/2$.

(b) Partition these r positive points into groups of size d. The last group may have fewer than d points. Show that the number of groups is at most $\frac{N}{2}$. Label these groups $\mathcal{D}_i$ for $i = 1 \ldots q \leq N/2$.

(c) Show that for any subset of k points with $k \leq d$, there is a hyperplane containing those points and no others.

(d) By the previous part, let $\mathbf{w}_i, b_i$ be the hyperplane through the points in group $\mathcal{D}_i$, and containing no others. So

$$\mathbf{w}_i^{\text{T}}\mathbf{x}_n + b_i = 0$$

if and only if $\mathbf{x}_n \in \mathcal{D}_i$. Show that it is possible to find h small enough so that for $\mathbf{x}_n \in \mathcal{D}_i$,

$$|\mathbf{w}_i^{\text{T}}\mathbf{x}_n + b_i| < h,$$

and for $\mathbf{x}_n \notin \mathcal{D}_i$

$$|\mathbf{w}_i^{\text{T}}\mathbf{x}_n + b_i| > h.$$

(e) Show that for $\mathbf{x}_n \in \mathcal{D}_i$,

$$\text{sign}(\mathbf{w}_i^{\text{T}}\mathbf{x}_n + b_i + h) + \text{sign}(-\mathbf{w}_i^{\text{T}}\mathbf{x}_n - b_i + h) = 2,$$

and for $\mathbf{x}_n \notin \mathcal{D}_i$

$$\text{sign}(\mathbf{w}_i^{\text{T}}\mathbf{x}_n + b_i + h) + \text{sign}(-\mathbf{w}_i^{\text{T}}\mathbf{x}_n - b_i + h) = 0$$

(f) Use the results so far to construct a 2-layer MLP with $2r$ hidden units which implements the dichotomy (which was arbitrary). Complete the argument to show that $d_{\text{VC}} \geq md$.

Chapter 8

Support Vector Machines

Linear models are powerful. The nonlinear transform and the neural network (a cascade of linear models) are tools that increase their expressive power. Increasing the expressive power has a price: overfitting and computation time. Can we get the expressive power without paying the price? The answer is yes. Our new model, the Support Vector Machine (SVM), uses a 'safety cushion' when separating the data. As a result, SVM is more robust to noise; this helps combat overfitting. In addition, SVM can work seamlessly with a new powerful tool called the *kernel*: a computationally efficient way to use high (even infinite!) dimensional nonlinear transforms. When we combine the safety cushion of SVM with the 'kernel trick', the result is an efficient powerful nonlinear model with automatic regularization. The SVM model is popular because it performs well in practice and is easy to use. This chapter presents the mathematics and algorithms that make SVM work.

8.1 The Optimal Hyperplane

Let us revisit the perceptron model from Chapter 1. The illustrations below on a toy data set should help jog your memory. In 2 dimensions, a perceptron attempts to separate the data with a line, which is possible in this example.

As you can see, many lines separate the data and the Perceptron Learning Algorithm (PLA) finds one of them. Do we care about which one PLA finds? All separators have $E_{\text{in}} = 0$, so the VC analysis in Chapter 2 gives the same E_{out}-bound for every separator. Well, the VC bound may say one thing, but surely our intuition says that the rightmost separator is preferred ☺.

Let's try to pin down an argument that supports our intuition. In practice, there are measurement errors – *noise*. Place identical shaded regions around each data point, with the radius of the region being the amount of possible measurement error. The true data point can lie anywhere within this 'region of uncertainty' on account of the measurement error. A separator is 'safe' with respect to the measurement error if it classifies the *true* data points correctly. That is, no matter where in its region of uncertainty the true data point lies, it is still on the correct side of the separator. The figure below shows the largest measurement errors which are safe for each separator.

A separator that can tolerate more measurement error is safer. The rightmost separator tolerates the largest error, whereas for the leftmost separator, even a small error in some data points could result in a misclassification. In Chapter 4, we saw that noise (for example measurement error) is the main cause of overfitting. Regularization helps us combat noise and avoid overfitting. In our example, the rightmost separator is more robust to noise without compromising E_{in}; it is better 'regularized'. Our intuition is well justified.

We can also quantify noise tolerance from the viewpoint of the separator. Place a cushion on each side of the separator. We call such a separator with a cushion *fat*, and we say that it separates the data if no data point lies within its cushion. Here is the largest cushion we can place around each of our three candidate separators.

To get the thickest cushion, keep extending the cushion equally on both sides of the separator until you hit a data point. The thickness reflects the amount of noise the separator can tolerate. If any data point is perturbed by at most the thickness of either side of the cushion, it will still be on the correct side of the separator. The maximum thickness (noise tolerance) possible for a separator is called its *margin*. Our intuition tells us to pick the fattest separator possible, the one with maximum margin. In this section, we will address three important questions:

1. Can we efficiently find the fattest separator?
2. Why is a fat separator better than a thin one?
3. What should we do if the data is not separable?

The first question relates to the algorithm; the second relates to E_{out}; and, the third relates to E_{in} (we will also elaborate on this question in Section 8.4).

Our discussion was in 2 dimensions. In higher dimensions, the separator is a hyperplane and our intuition still holds. Here is a warm-up.

Exercise 8.1

Assume $\mathcal{D}$ contains two data points $(\mathbf{x}_+, +1)$ and $(\mathbf{x}_-, -1)$. Show that:

(a) No hyperplane can tolerate noise radius greater than $\frac{1}{2}\|\mathbf{x}_+ - \mathbf{x}_-\|$.

(b) There is a hyperplane that tolerates a noise radius $\frac{1}{2}\|\mathbf{x}_+ - \mathbf{x}_-\|$.

8.1.1 Finding the Fattest Separating Hyperplane

Before we proceed to the mathematics, we fix a convention that we will use throughout the chapter. Recall that a hyperplane is defined by its weights $\mathbf{w}$.

Isolating the bias. We explicitly split the weight vector $\mathbf{w}$ as follows. The bias weight w_0 is taken out, and the rest of the weights $w_1, \ldots, w_d$ remain in $\mathbf{w}$. The reason is that the mathematics will treat these two types of weights differently. To avoid confusion, we will relabel w_0 to b (for bias), but continue to use $\mathbf{w}$ for $(w_1, \ldots, w_d)$.

Previous Chapters

$\mathbf{x} \in \{1\} \times \mathbb{R}^d;\ \mathbf{w} \in \mathbb{R}^{d+1}$

$$\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}; \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}.$$

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x})$$

This Chapter

$\mathbf{x} \in \mathbb{R}^d;\ b \in \mathbb{R},\ \mathbf{w} \in \mathbb{R}^d$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}; \quad \begin{aligned} b &= \text{bias} \\ \mathbf{w} &= \begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix}. \end{aligned}$$

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x} + b)$$

A maximum-margin separating hyperplane has two defining properties.

1. It separates the data.
2. It has the thickest cushion among hyperplanes that separate the data.

To find a separating hyperplane with maximum margin, we first re-examine the definition of a *separating* hyperplane and reshape the definition into an equivalent, more convenient one. Then, we discuss how to compute the margin of any given separating hyperplane (so that we can find the one with maximum margin). As we observed in our earlier intuitive discussion, the margin is obtained by extending the cushion until you hit a data point. That is, the margin is the distance from the hyperplane to the *nearest* data point. We thus need to become familiar with the geometry of hyperplanes; in particular, how to compute the distance from a data point to the hyperplane.

Separating hyperplanes. The hyperplane h, defined by $(b, \mathbf{w})$, separates the data if and only if for $n = 1, \ldots, N$,

$$y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) > 0. \tag{8.1}$$

The signal $y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)$ is positive for each data point. However, the magnitude of the signal is not meaningful by itself since we can make it arbitrarily small or large *for the same hyperplane* by rescaling the weights and the bias. This is because $(b, \mathbf{w})$ is the *same* hyperplane as $(b/\rho, \mathbf{w}/\rho)$ for any $\rho > 0$. By rescaling the weights, we can control the size of the signal for our data points. Let us pick a particular value of ρ,

$$\rho = \min_{n=1,\ldots,N} y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b),$$

which is positive because of (8.1). Now, rescale the weights to obtain the *same* hyperplane $(b/\rho, \mathbf{w}/\rho)$. For these rescaled weights,

$$\min_{n=1,\ldots,N} y_n\left(\frac{\mathbf{w}}{\rho}^{\mathrm{T}}\mathbf{x}_n + \frac{b}{\rho}\right) = \frac{1}{\rho}\min_{n=1,\ldots,N} y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) = \frac{\rho}{\rho} = 1.$$

Thus, for any separating hyperplane, it is always possible to choose weights so that all the signals $y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)$ are of magnitude greater than or equal to 1, with equality satisfied by at least one $(\mathbf{x}_n, y_n)$. This motivates our new definition of a separating hyperplane.

Definition 8.1 (Separating Hyperplane)**.** The hyperplane h separates the data if and only if it can be represented by weights $(b, \mathbf{w})$ that satisfy

$$\min_{n=1,\ldots,N} y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) = 1. \tag{8.2}$$

The conditions (8.1) and (8.2) are equivalent. *Every* separating hyperplane can be accommodated under Definition 8.1. All we did is constrain the way we *algebraically* represent such a hyperplane by choosing a (data dependent) normalization for the weights, to ensure that the magnitude of the signal is meaningful. Our normalization in (8.2) will be particularly convenient for deriving the algorithm to find the maximum-margin separator. The next exercise gives a concrete example of re-normalizing the weights to satisfy (8.2).

Exercise 8.2

Consider the data below and a 'hyperplane' $(b, \mathbf{w})$ that separates the data.

$$\mathrm{X} = \begin{bmatrix} 0 & 0 \\ 2 & 2 \\ 2 & 0 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ +1 \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} 1.2 \\ -3.2 \end{bmatrix} \quad b = -0.5$$

(a) Compute $\rho = \min_{n=1,\ldots,N} y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)$.

(b) Compute the weights $\frac{1}{\rho}(b, \mathbf{w})$ and show that they satisfy (8.2).

(c) Plot both hyperplanes to show that they are the *same* separator.

Margin of a hyperplane. To compute the margin of a separating hyperplane, we need to compute the distance from the hyperplane to the *nearest* data point. As a start, let us compute the distance from an arbitrary point $\mathbf{x}$ to a separating hyperplane $h = (b, \mathbf{w})$ that satisfies (8.2). Denote this distance by $\mathsf{dist}(\mathbf{x}, h)$. Referring to the figure on the right, $\mathsf{dist}(\mathbf{x}, h)$ is the length of the perpendicular from $\mathbf{x}$ to h. Let $\mathbf{x}'$ be any point on the hyperplane, which means $\mathbf{w}^{\mathrm{T}}\mathbf{x}' + b = 0$. Let $\mathbf{u}$ be a unit vector that is normal to the hyperplane h.

Then, $\mathsf{dist}(\mathbf{x}, h) = |\mathbf{u}^{\mathrm{T}}(\mathbf{x} - \mathbf{x}')|$, the projection of the vector $(\mathbf{x} - \mathbf{x}')$ onto $\mathbf{u}$. We now argue that $\mathbf{w}$ is normal to the hyperplane, and so we can take $\mathbf{u} = \mathbf{w}/\|\mathbf{w}\|$. Indeed, any vector lying on the hyperplane can be expressed by $(\mathbf{x}'' - \mathbf{x}')$ for some $\mathbf{x}', \mathbf{x}''$ on the hyperplane, as shown. Then, using $\mathbf{w}^{\mathrm{T}}\mathbf{x} = -b$ for points on the hyperplane,

$$\mathbf{w}^{\mathrm{T}}(\mathbf{x}'' - \mathbf{x}') = \mathbf{w}^{\mathrm{T}}\mathbf{x}'' - \mathbf{w}^{\mathrm{T}}\mathbf{x}' = -b + b = 0.$$

Therefore, $\mathbf{w}$ is orthogonal to *every* vector in the hyperplane, hence it is the normal vector as claimed. Setting $\mathbf{u} = \mathbf{w}/\|\mathbf{w}\|$, the distance from $\mathbf{x}$ to h is

$$\mathsf{dist}(\mathbf{x}, h) = |\mathbf{u}^{\mathrm{T}}(\mathbf{x} - \mathbf{x}')| = \frac{|\mathbf{w}^{\mathrm{T}}\mathbf{x} - \mathbf{w}^{\mathrm{T}}\mathbf{x}'|}{\|\mathbf{w}\|} = \frac{|\mathbf{w}^{\mathrm{T}}\mathbf{x} + b|}{\|\mathbf{w}\|},$$

where we used $\mathbf{w}^{\mathrm{T}}\mathbf{x}' = -b$ in the last step, since $\mathbf{x}'$ is on h. You can now see why we separated the bias b from the weights $\mathbf{w}$: the distance calculation treats these two parameters differently.

We are now ready to compute the margin of a separating hyperplane. Consider the data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$, and hyperplane $h = (b, \mathbf{w})$ that satisfies the separating condition (8.2). Since $y_n = \pm 1$,

$$|\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b| = |y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)| = y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b),$$

where the last equality follows because $(b, \mathbf{w})$ separates the data, which implies that $y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)$ is positive. So, the distance of a data point to h is

$$\mathsf{dist}(\mathbf{x}_n, h) = \frac{y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)}{\|\mathbf{w}\|}.$$

Therefore, the data point that is nearest to the hyperplane has distance

$$\min_{n=1,\ldots,N} \mathsf{dist}(\mathbf{x}_n, h) = \frac{1}{\|\mathbf{w}\|} \cdot \min_{n=1,\ldots,N} y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) = \frac{1}{\|\mathbf{w}\|},$$

where the last equality follows because $(b, \mathbf{w})$ separates the data and satisfies (8.2). This simple expression for the distance of the nearest data point to the hyperplane is the entire reason why we chose to normalize $(b, \mathbf{w})$ as we did, by requiring (8.2).[1] For any separating hyperplane satisfying (8.2), the margin is $1/\|\mathbf{w}\|$. If you hold on a little longer, you are about to reap the full benefit, namely a simple algorithm for finding the optimal (fattest) hyperplane.

The fattest separating hyperplane. The maximum-margin separating hyperplane $(b^*, \mathbf{w}^*)$ is the one that satisfies the separation condition (8.2) with minimum weight-norm (since the margin is the inverse of the weight-norm). Instead of minimizing the weight-norm, we can equivalently minimize $\frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$, which is analytically more friendly. Therefore, to find this optimal hyperplane, we need to solve the following optimization problem.

$$\begin{aligned} \underset{b,\mathbf{w}}{\text{minimize:}} \quad & \tfrac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \\ \text{subject to:} \quad & \min_{n=1,\ldots,N} y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) = 1. \end{aligned} \tag{8.3}$$

The constraint ensures that the hyperplane separates the data as per (8.2). Observe that the bias b does not appear in the quantity being minimized, but it is involved in the constraint (again, b is treated differently than $\mathbf{w}$). To make the optimization problem easier to solve, we can replace the single constraint

[1] Some terminology: the parameters $(b, \mathbf{w})$ of a separating hyperplane that satisfy (8.2) are called the canonical representation of the hyperplane. For a separating hyperplane in its canonical representation, the margin is just the inverse norm of the weights.

$\min_n y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) = 1$ with N 'looser' constraints $y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) \geq 1$ for $n = 1, \ldots, N$ and solve the optimization problem:

$$\begin{aligned} \underset{b,\mathbf{w}}{\text{minimize:}} \quad & \tfrac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \\ \text{subject to:} \quad & y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) \geq 1 \qquad (n = 1, \cdots, N). \end{aligned} \tag{8.4}$$

The constraint in (8.3) implies the constraints in (8.4), which means that the constraints in (8.4) are looser. Fortunately, *at the optimal solution*, the constraints in (8.4) become equivalent to the constraint in (8.3) as long as there are both positive and negative examples in the data. After solving (8.4), we will show that the constraint of (8.3) is automatically satisfied. This means that we will also have solved (8.3).

To do that, we will use a proof by contradiction. Suppose that the solution $(b^*, \mathbf{w}^*)$ of (8.4) has

$$\rho^* = \min_n y_n(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n + b^*) > 1,$$

and therefore is not a solution to (8.3). Consider the rescaled hyperplane $(b, \mathbf{w}) = \frac{1}{\rho^*}(b^*, \mathbf{w}^*)$, which satisfies the constraints in (8.4) by construction. For $(b, \mathbf{w})$, we have that $\|\mathbf{w}\| = \frac{1}{\rho^*}\|\mathbf{w}^*\| < \|\mathbf{w}^*\|$ (unless $\mathbf{w}^* = \mathbf{0}$), which means that $\mathbf{w}^*$ cannot be optimal for (8.4) unless $\mathbf{w}^* = \mathbf{0}$. It is not possible to have $\mathbf{w}^* = \mathbf{0}$ since this would not correctly classify both the positive and negative examples in the data.

We will refer to this fattest separating hyperplane as *the optimal hyperplane*. To get the optimal hyperplane, all we have to do is solve the optimization problem in (8.4).

Example 8.2. The best way to get a handle on what is going on is to carefully work through an example to see how solving the optimization problem in (8.4) results in the optimal hyperplane $(b^*, \mathbf{w}^*)$. In two dimensions, a hyperplane is specified by the parameters (b, w_1, w_2). Let us consider the toy data set that was the basis for the figures on page 8-2. The data matrix and target values, together with the separability constraints from (8.4) are summarized below. The inequality on a particular row is the separability constraint for the corresponding data point in that row.

$$\mathrm{X} = \begin{bmatrix} 0 & 0 \\ 2 & 2 \\ 2 & 0 \\ 3 & 0 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ +1 \\ +1 \end{bmatrix} \qquad \begin{aligned} -b &\geq 1 \quad \text{(i)} \\ -(2w_1 + 2w_2 + b) &\geq 1 \quad \text{(ii)} \\ 2w_1 + b &\geq 1 \quad \text{(iii)} \\ 3w_1 + b &\geq 1 \quad \text{(iv)} \end{aligned}$$

Combining (i) and (iii) gives

$$w_1 \geq 1.$$

Combining (ii) and (iii) gives

$$w_2 \leq -1.$$

This means that $\frac{1}{2}(w_1^2 + w_2^2) \geq 1$ with equality when $w_1 = 1$ and $w_2 = -1$. One can easily verify that

$$(b^* = -1, w_1^* = 1, w_2^* = -1)$$

satisfies all four constraints, minimizes $\frac{1}{2}(w_1^2 + w_2^2)$, and therefore gives the optimal hyperplane. The optimal hyperplane is shown in the following figure.

<u>Optimal Hyperplane</u>

$$g(\mathbf{x}) = \text{sign}(x_1 - x_2 - 1)$$

$$\text{margin:} \quad \frac{1}{\|\mathbf{w}^*\|} = \frac{1}{\sqrt{2}} \approx 0.707.$$

Data points (i), (ii) and (iii) are boxed because their separation constraints are exactly met: $y_n(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n + b^*) = 1$.

For data points which meet their constraints exactly, $\mathsf{dist}(\mathbf{x}_n, g) = \frac{1}{\|\mathbf{w}^*\|}$. These data points sit on the boundary of the cushion and play an important role. They are called *support vectors*. In a sense, the support vectors are 'supporting' the cushion and preventing it from expanding further. □

Exercise 8.3

For separable data that contain both positive and negative examples, and a separating hyperplane h, define the positive-side margin $\rho_+(h)$ to be the distance between h and the nearest data point of class $+1$. Similarly, define the negative-side margin $\rho_-(h)$ to be the distance between h and the nearest data point of class -1. Argue that if h is the optimal hyperplane, then $\rho_+(h) = \rho_-(h)$. That is, the thickness of the cushion on either side of the optimal h is equal.

We make an important observation that will be useful later. In Example 8.2, what happens to the optimal hyperplane if we removed data point (iv), the non-support vector? Nothing! The hyperplane remains a separator with the *same* margin. Even though we removed a data point, a larger margin cannot be achieved since all the support vectors that previously prevented the margin from expanding are still in the data. So the hyperplane remains optimal. Indeed, to compute the optimal hyperplane, only the support vectors are needed; the other data could be thrown away.

Quadratic Programming (QP). For bigger data sets, manually solving the optimization problem in (8.4) as we did in Example 8.2 is no longer feasible.

The good news is that (8.4) belongs to a well-studied family of optimization problems known as quadratic programming (QP). Whenever you minimize a (convex) *quadratic* function, subject to *linear* inequality constraints, you can use quadratic programming. Quadratic programming is such a well studied area that excellent, publicly available solvers exist for many numerical computing platforms. We will not need to know how to solve a quadratic programming problem; we will only need to know how to take any given problem and convert it into a standard form which is then input to a QP-solver. We begin by describing the standard form of a QP-problem:

$$\begin{aligned} \underset{\mathbf{u}\in\mathbb{R}^L}{\text{minimize:}} \quad & \tfrac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} + \mathbf{p}^{\mathrm{T}}\mathbf{u} \\ \text{subject to:} \quad & \mathbf{a}_m^{\mathrm{T}}\mathbf{u} \geq c_m \qquad (m = 1, \cdots, M). \end{aligned} \tag{8.5}$$

The variables to be optimized are the components of the vector $\mathbf{u}$ which has dimension L. All the other parameters Q, $\mathbf{p}$, $\mathbf{a}_m$ and c_m for $m = 1, \ldots, M$ are user-specified. The quantity being minimized contains a quadratic term $\frac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u}$ and a linear term $\mathbf{p}^{\mathrm{T}}\mathbf{u}$. The coefficients of the quadratic terms are in the $L \times L$ matrix Q: $\frac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} = \sum_{i=1}^{L}\sum_{j=1}^{L} q_{ij}u_iu_j$. The coefficients of the linear terms are in the $L \times 1$ vector $\mathbf{p}$: $\mathbf{p}^{\mathrm{T}}\mathbf{u} = \sum_{i=1}^{L} p_iu_i$. There are M *linear* inequality constraints, each one being specified by an $L \times 1$ vector $\mathbf{a}_m$ and scalar c_m. For the QP-problem to be convex, the matrix Q needs to be positive semi-definite. It is more convenient to specify the $\mathbf{a}_m$ as rows of an $M \times L$ matrix A and the c_m as components of an $M \times 1$ vector $\mathbf{c}$:

$$\mathrm{A} = \begin{bmatrix} \text{—}\mathbf{a}_1^{\mathrm{T}}\text{—} \\ \vdots \\ \text{—}\mathbf{a}_M^{\mathrm{T}}\text{—} \end{bmatrix} \qquad \mathbf{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_M \end{bmatrix}.$$

Using the matrix representation, the QP-problem in (8.5) is written as

$$\begin{aligned} \underset{\mathbf{u}\in\mathbb{R}^L}{\text{minimize:}} \quad & \tfrac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} + \mathbf{p}^{\mathrm{T}}\mathbf{u} \\ \text{subject to:} \quad & \mathrm{A}\mathbf{u} \geq \mathbf{c}. \end{aligned} \tag{8.6}$$

The M inequality constraints are all contained in the single vector constraint $\mathrm{A}\mathbf{u} \geq \mathbf{c}$ (which must hold for each component). We will write

$$\mathbf{u}^* \leftarrow \mathsf{QP}(\mathrm{Q}, \mathbf{p}, \mathrm{A}, \mathbf{c})$$

to denote the process of running a QP-solver on the input $(\mathrm{Q}, \mathbf{p}, \mathrm{A}, \mathbf{c})$ to get an optimal solution $\mathbf{u}^*$ for (8.6).[2]

[2] A quick comment about the input to QP-solvers. Some QP-solvers take the reverse inequality constraints $\mathrm{A}\mathbf{u} \leq \mathbf{c}$, which simply means you need to negate A and $\mathbf{c}$ in (8.6). An equality constraint is two inequality constraints ($a = b \iff a \geq b$ and $a \leq b$). Some solvers will accept separate upper and lower bound constraints on each component of $\mathbf{u}$. All these different variants of quadratic programming can be represented by the standard problem with linear inequality constraints in (8.6). However, special types of constraints like equality and upper/lower bound constraints can often be handled in a more numerically stable way than the general linear inequality constraint.

We now show that our optimization problem in (8.4) is indeed a QP-problem. To do so, we must identify Q, $\mathbf{p}$, A and $\mathbf{c}$. First, observe that the quantities that need to be solved for (the optimization variables) are $(b, \mathbf{w})$; collecting these into $\mathbf{u}$, we identify the optimization variable $\mathbf{u} = \left[\begin{smallmatrix} b \\ \mathbf{w} \end{smallmatrix}\right] \in \mathbb{R}^{d+1}$. The dimension of the optimization problem is $L = d + 1$. The quantity we are minimizing is $\frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$. We need to write this in the form $\frac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} + \mathbf{p}^{\mathrm{T}}\mathbf{u}$. Observe that

$$\mathbf{w}^{\mathrm{T}}\mathbf{w} = \begin{bmatrix} b & \mathbf{w}^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} 0 & \mathbf{0}_d^{\mathrm{T}} \\ \mathbf{0}_d & \mathrm{I}_d \end{bmatrix} \begin{bmatrix} b \\ \mathbf{w}^{\mathrm{T}} \end{bmatrix} = \mathbf{u}^{\mathrm{T}} \begin{bmatrix} 0 & \mathbf{0}_d^{\mathrm{T}} \\ \mathbf{0}_d & \mathrm{I}_d \end{bmatrix} \mathbf{u},$$

where I_d is the $d \times d$ identity matrix and $\mathbf{0}_d$ the d-dimensional zero vector. We can identify the quadratic term with $\mathrm{Q} = \left[\begin{smallmatrix} 0 & \mathbf{0}_d^{\mathrm{T}} \\ \mathbf{0}_d & \mathrm{I}_d \end{smallmatrix}\right]$, and there is no linear term, so $\mathbf{p} = \mathbf{0}_{d+1}$. As for the constraints in (8.4), there are N of them in (8.4), so $M = N$. The n-th constraint is $y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) \geq 1$, which is equivalent to

$$\begin{bmatrix} y_n & y_n\mathbf{x}_n^{\mathrm{T}} \end{bmatrix} \mathbf{u} \geq 1,$$

and that corresponds to setting $\mathbf{a}_n^{\mathrm{T}} = y_n \begin{bmatrix} 1 & \mathbf{x}_n^{\mathrm{T}} \end{bmatrix}$ and $c_n = 1$ in (8.5). So, the matrix A contains rows which are very related to an old friend from Chapter 3, the data matrix X augmented with a column of 1s. In fact, the n-th row of A is just the n-th row of the data matrix but multiplied by its label y_n. The constraint vector $\mathbf{c} = \mathbf{1}_N$, an N-dimensional vector of ones.

Exercise 8.4

Let Y be an $N \times N$ diagonal matrix with diagonal entries $\mathrm{Y}_{nn} = y_n$ (a matrix version of the target vector $\mathbf{y}$). Let X be the data matrix augmented with a column of 1s. Show that

$$\mathrm{A} = \mathrm{YX}.$$

We summarize below the algorithm to get an optimal hyperplane, which reduces to a QP-problem in $d + 1$ variables with N constraints.

Linear Hard-Margin SVM with QP

1: Let $\mathbf{p} = \mathbf{0}_{d+1}$ ($(d+1)$-dimensional zero vector) and $\mathbf{c} = \mathbf{1}_N$ (N-dimensional vector of ones). Construct matrices Q and A, where

$$\mathrm{Q} = \begin{bmatrix} 0 & \mathbf{0}_d^{\mathrm{T}} \\ \mathbf{0}_d & \mathrm{I}_d \end{bmatrix}, \qquad \mathrm{A} = \underbrace{\begin{bmatrix} y_1 & -y_1\mathbf{x}_1^{\mathrm{T}}- \\ \vdots & \vdots \\ y_N & -y_N\mathbf{x}_N^{\mathrm{T}}- \end{bmatrix}}_{\text{signed data matrix}}.$$

2: Calculate $\begin{bmatrix} b^* \\ \mathbf{w}^* \end{bmatrix} = \mathbf{u}^* \leftarrow \mathsf{QP}(\mathrm{Q}, \mathbf{p}, \mathrm{A}, \mathbf{c})$.

3: Return the hypothesis $g(\mathbf{x}) = \mathrm{sign}(\mathbf{w}^{*\mathrm{T}}\mathbf{x} + b^*)$.

The learning model we have been discussing is known as the *linear hard-margin support vector machine* (linear hard-margin SVM). Yes, it sounds like something out of a science fiction book ☺. Don't worry, it is just a name that describes an algorithm for constructing an optimal linear model, and in this algorithm only a few of the data points have a role to play in determining the final hypothesis - these few data points are called *support vectors*, as mentioned earlier. The margin being hard means that no data is allowed to lie inside the cushion. We can relax this condition, and we will do so later in this chapter.

Exercise 8.5

Show that the matrix Q described in the linear hard-margin SVM algorithm above is positive semi-definite (that is $\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} \geq 0$ for any $\mathbf{u}$).

The result means that the QP-problem is convex. Convexity is useful because this makes it 'easy' to find an optimal solution. In fact, standard QP-solvers can solve our convex QP-problem in $O((N+d)^3)$.

Example 8.3. We can explicitly construct the QP-problem for our toy example in Example 8.2. We construct Q, $\mathbf{p}$, A, and $\mathbf{c}$ as follows.

$$\mathrm{Q} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathrm{A} = \begin{bmatrix} -1 & 0 & 0 \\ -1 & -2 & -2 \\ 1 & 2 & 0 \\ 1 & 3 & 0 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

A standard QP-solver gives $(b^*, w_1^*, w_2^*) = (-1, 1, -1)$, the same solution we computed manually, but obtained in less than a millisecond. □

In addition to standard QP-solvers that are publicly available, there are also specifically tailored solvers for the linear SVM, which are often more efficient for learning from *large-scale* data, characterized by a large N or d. Some packages are based on a version of stochastic gradient descent (SGD), a technique for optimization that we introduced in Chapter 3, and other packages use more sophisticated optimization techniques that take advantage of special properties of SVM (such as the redundancy of non-support vectors).

Example 8.4 (Comparison of SVM with PLA)**.** We construct a toy data set and use it to compare SVM with the perceptron learning algorithm. To generate the data, we randomly generate $x_1 \in [0, 1]$ and $x_2 \in [-1, 1]$ with the target function being $+1$ above the x_1-axis; $f(\mathbf{x}) = \mathrm{sign}(x_2)$. In this experiment, we used the version of PLA that updates the weights using the misclassified point $\mathbf{x}_n$ with lowest index n. The histogram in Figure 8.1 shows what will typically happen with PLA. Depending on the ordering of the data, PLA can sometimes get lucky and beat the SVM, and sometimes it can be much worse. The SVM (maximum-margin) classifier does not depend on the (random) ordering of the data. □

(a) Data and SVM separator (b) Histogram of E_{out}(PLA)

Figure 8.1: (a) The SVM classifier from data generated using $f(\mathbf{x}) = \text{sign}(x_2)$ (the blue region is $f(\mathbf{x}) = +1$). The margin (cushion) is inside the gray lines and the three support vectors are enclosed in boxes. (b) A histogram of E_{out} for PLA classifiers using random orderings of the data.

Exercise 8.6

Construct a toy data set with $N = 20$ using the method in Example 8.4.

(a) Run the SVM algorithm to obtain the maximum margin separator $(b, \mathbf{w})_{\text{SVM}}$ and compute its E_{out} and margin.

(b) Construct an ordering of the data points that results in a hyperplane with bad E_{out} when PLA is run on it. *[Hint: Identify a positive and negative data point for which the perpendicular bisector separating these two points is a bad separator. Where should these two points be in the ordering? How many iterations will PLA take?]*

(c) Create a plot of your two separators arising from SVM and PLA.

8.1.2 Is a Fat Separator Better?

The SVM (optimal hyperplane) is a linear model, so it cannot fit certain simple functions as discussed in Chapter 3. But, on the positive side, the SVM also inherits the good generalization capability of the simple linear model since the VC dimension is bounded by $d + 1$. Does the support vector machine gain any more generalization ability by maximizing the margin, providing a safety cushion so to speak? Now that we have the algorithm that gets us the optimal hyperplane, we are in a position to shed some light on this question.

We already argued intuitively that the optimal hyperplane is robust to noise. We now show this link to regularization more explicitly. We have seen an optimization problem similar to the one in (8.4) before. Recall the soft-

order constraint when we discussed regularization in Chapter 4,

$$\begin{aligned} \underset{\mathbf{w}}{\text{minimize:}} \quad & E_{\text{in}}(\mathbf{w}) \\ \text{subject to:} \quad & \mathbf{w}^{\mathrm{T}}\mathbf{w} \leq C. \end{aligned}$$

Ultimately, this led to weight-decay regularization. In regularization, we minimize E_{in} given a budget C for $\mathbf{w}^{\mathrm{T}}\mathbf{w}$. For the optimal hyperplane (SVM), we minimize $\mathbf{w}^{\mathrm{T}}\mathbf{w}$ subject to a budget on E_{in} (namely $E_{\text{in}} = 0$). In a sense, the optimal hyperplane is doing automatic weight-decay regularization.

	optimal hyperplane	regularization
minimize:	$\mathbf{w}^{\mathrm{T}}\mathbf{w}$	E_{in}
subject to:	$E_{\text{in}} = 0$	$\mathbf{w}^{\mathrm{T}}\mathbf{w} \leq C$

Both methods are trying to fit the data with small weights. That is the gist of regularization. This link to regularization is interesting, but does it help? Yes, both in theory and practice. We hope to convince you of three things.

(i) A larger-margin separator yields better performance in practice. We will illustrate this with a simple empirical experiment.

(ii) Fat hyperplanes generalize better than thin hyperplanes. We will show this by bounding the number of points that fat hyperplanes can shatter. Our bound is less than $d + 1$ for fat enough hyperplanes.

(iii) The out-of-sample error can be small, even if the dimension d is large. To show this, we bound the cross validation error E_{cv} (recall that E_{cv} is a proxy for E_{out}). Our bound does not explicitly depend on the dimension.

(i) Larger Margin is Better. We will use the toy learning problem in Example 8.4 to study how separators with different margins perform. We randomly generate a separable data set of size $N = 20$. There are infinitely many separating hyperplanes. We sample these separating hyperplanes randomly.

For each random separating hyperplane h, we can compute $E_{\text{out}}(h)$ and the margin $\rho(h)/\rho(\text{SVM})$ (normalized by the maximum possible margin achieved by the SVM). We can now plot how $E_{\text{out}}(h)$ depends on the *fraction* of the available margin that h uses. For each data set, we sample 50,000 random separating hyperplanes and then average the results over several thousands of data sets. Figure 8.2 shows the dependence of E_{out} versus margin.

Figure 8.2 clearly suggests that when choosing a random separating hyperplane, it is better to choose one with larger margin. The SVM, which picks *the* hyperplane with largest margin, is doing a much better job than one of the (typical) random separating hyperplanes. Notice that once you get to large enough margin, increasing the margin further can hurt E_{out}: it is possible to slightly improve the performance among random hyperplanes by slightly sacrificing on the maximum margin and perhaps improving in other ways. Let's now build some theoretical evidence for why large margin separation is good.

Figure 8.2: Dependence of E_{out} versus the margin ρ for a random separating hyperplane. The shaded histogram in the background shows the relative frequencies of a particular margin when selecting a random hyperplane.

(ii) Fat Hyperplanes Shatter Fewer Points. The VC dimension of a hypothesis set is the maximum number of points that can be shattered. A smaller VC dimension gives a smaller generalization error bar that links E_{in} to E_{out} (see Chapter 2 for details). Consider the hypothesis set $\mathcal{H}_\rho$ containing *all* fat hyperplanes of width (margin) at least ρ. A dichotomy on a data set $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)$ can be implemented by a hypothesis $h \in \mathcal{H}_\rho$ if $y_n = h(\mathbf{x}_n)$ and none of the $\mathbf{x}_n$ lie inside the margin of h. Let $d_{\text{vc}}(\rho)$ be the maximum number of points that $\mathcal{H}_\rho$ can shatter.[3] Our goal is to show that restricting the hypothesis set to fat separators can decrease the number of points that can be shattered, that is, $d_{\text{vc}}(\rho) < d + 1$. Here is an example.

Figure 8.3: Thin hyperplanes can implement all 8 dichotomies (the other 4 dichotomies are obtained by negating the weights and bias).

Thin (zero thickness) separators can shatter the three points shown above. As we increase the thickness of the separator, we will soon not be able to implement the rightmost dichotomy. Eventually, as we increase the thickness even

[3]Technically a hypothesis in $\mathcal{H}_\rho$ is not a classifier because inside its margin the output is not defined. Nevertheless we can still compute the maximum number of points that can be shattered and this 'VC dimension' plays a similar role to d_{vc} in establishing a generalization error bar for the model, albeit using more sophisticated analysis.

more, the only dichotomies we can implement will be the constant dichotomies.

Figure 8.4: Only 4 of the 8 dichotomies can be separated by hyperplanes with thickness ρ. The dashed lines show the thickest possible separator for each non-separable dichotomy.

In Figure 8.4, there is no ρ-thick hyperplane that can separate the right two dichotomies. This example illustrates why thick hyperplanes implement fewer of the possible dichotomies. Note that the hyperplanes only 'look' thick because the data are close together. If we moved the data further apart, then even a thick hyperplane can implement all the dichotomies. What matters is the thickness of the hyperplane *relative* to the spacing of the data.

Exercise 8.7

Assume that the data is restricted to lie in a unit sphere.

(a) Show that $d_{\text{vc}}(\rho)$ is non-increasing in ρ.

(b) In 2 dimensions, show that $d_{\text{vc}}(\rho) < 3$ for $\rho > \frac{\sqrt{3}}{2}$. *[Hint: Show that for any 3 points in the unit disc, there must be two that are within distance $\sqrt{3}$ of each other. Use this fact to construct a dichotomy that cannot be implemented by any ρ-thick separator.]*

The exercise shows that for a bounded input space, thick separators can shatter fewer than $d+1$ points. In general, we can prove the following result.

Theorem 8.5 (VC dimension of Fat Hyperplanes)**.** Suppose the input space is the ball of radius R in $\mathbb{R}^d$, so $\|\mathbf{x}\| \leq R$. Then,

$$d_{\text{vc}}(\rho) \leq \lceil R^2/\rho^2 \rceil + 1,$$

where $\lceil R^2/\rho^2 \rceil$ is the smallest integer greater than or equal to R^2/ρ^2.

We also know that the VC dimension is at most $d+1$, so we can pick the better of the two bounds, and we gain when R/ρ is small. The most important fact about this margin-based bound is that it does not explicitly depend on the dimension d. This means that if we transform the data to a high, even infinite, dimensional space, as long as we use fat enough separators, we obtain

good generalization. Since this result establishes a crucial link between the margin and good generalization, we give a simple, though somewhat technical, proof.

Begin safe skip: The proof is cute but not essential. A similar green box will tell you when to rejoin.

Proof. Fix $\mathbf{x}_1, \ldots, \mathbf{x}_N$ that are shattered by hyperplanes with margin ρ. We will show that when N is even, $N \leq R^2/\rho^2 + 1$. When N is odd, a similar but more careful analysis (see Problem 8.8) shows that $N \leq R^2/\rho^2 + 1 + \frac{1}{N}$. In both cases, $N \leq \lceil R^2/\rho^2 \rceil + 1$.

Assume N is even. We need the following geometric fact (which is proved in Problem 8.7). There exists a (balanced) dichotomy $y_1, \ldots, y_n$ such that

$$\sum_{n=1}^{N} y_n = 0, \text{ and } \qquad \left\| \sum_{n=1}^{N} y_n \mathbf{x}_n \right\| \leq \frac{NR}{\sqrt{N-1}}. \tag{8.7}$$

(For random y_n, $\sum_n y_n \mathbf{x}_n$ is a random walk whose distance from the origin doesn't grow faster than $R\sqrt{N}$.) The dichotomy satisfying (8.7) is separated with margin at least ρ (since $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is shattered). So, for some $(\mathbf{w}, b)$,

$$\rho \|\mathbf{w}\| \leq y_n(\mathbf{w}^{\mathrm{T}} \mathbf{x}_n + b), \qquad \text{for } n = 1, \ldots, N.$$

Summing over n, using $\sum_n y_n = 0$ and the Cauchy-Schwarz inequality,

$$N\rho\|\mathbf{w}\| \leq \mathbf{w}^{\mathrm{T}} \sum_{n=1}^{N} y_n \mathbf{x}_n + b \sum_{n=1}^{N} y_n = \mathbf{w}^{\mathrm{T}} \sum_{n=1}^{N} y_n \mathbf{x}_n \leq \|\mathbf{w}\| \left\| \sum_{n=1}^{N} y_n \mathbf{x}_n \right\|.$$

By the bound in (8.7), the RHS is at most $\|\mathbf{w}\| NR/\sqrt{N-1}$, or:

$$\rho \leq \frac{R}{\sqrt{N-1}} \implies N \leq \frac{R^2}{\rho^2} + 1.$$

■

End safe skip: Welcome back for a summary.

Combining Theorem 8.5 with $d_{\mathrm{vc}}(\rho) \leq d_{\mathrm{vc}}(0) = d + 1$, we have that

$$d_{\mathrm{vc}}(\rho) \leq \min\left(\lceil R^2/\rho^2 \rceil, d\right) + 1.$$

The bound suggests that ρ can be used to control model complexity. The separating hyperplane ($E_{\mathrm{in}} = 0$) with the maximum margin ρ will have the smallest $d_{\mathrm{vc}}(\rho)$ and hence smallest generalization error bar.

Unfortunately, there is a complication. The correct width ρ to use is not known to us ahead of time (what if we chose a higher ρ than possible?). The optimal hyperplane algorithm fixes ρ only after seeing the data. Giving yourself the option to use a smaller ρ but settling on a higher ρ when you see the data means the data is being snooped. One needs to modify the VC analysis to take into account this kind of data snooping. The interested reader can find the details related to these technicalities in the literature.

(iii) Bounding the Cross Validation Error. In Chapter 4, we introduced the leave-one-out cross validation error E_{cv} which is the average of the leave-one-out errors e_n. Each e_n is computed by leaving out the data point $(\mathbf{x}_n, y_n)$ and learning on the remaining data to obtain g_n^-. The hypothesis g_n^- is evaluated on $(\mathbf{x}_n, y_n)$ to give e_n, and

$$E_{\text{cv}} = \frac{1}{N}\sum_{n=1}^{N} \mathsf{e}_n.$$

An important property of E_{cv} is that it is an unbiased estimate of the expected out-of-sample error for a data set of size $N-1$. (see Chapter 4 for details on validation). Hence, E_{cv} is a very useful proxy for E_{out}. We can get a surprisingly simple bound for E_{cv} using the number of support vectors. Recall that a support vector lies on the very edge of the margin of the optimal hyperplane. We illustrate a toy data set in Figure 8.5(a) with the support vectors highlighted in boxes.

(a) All data (b) Only support vectors

Figure 8.5: (a) Optimal hyperplane with support vectors enclosed in boxes. (b) The same hyperplane is obtained using only the support vectors.

The crucial observation as illustrated in Figure 8.5(b) is that if any (or all) the data points other than the support vectors are removed, the resulting separator produced by the SVM is unchanged. You are asked to give a formal proof of this in Problem 8.9, but Figure 8.5 should serve as ample justification. This observation has an important implication: $\mathsf{e}_n = 0$ for any data point $(\mathbf{x}_n, y_n)$ that is not a support vector. This is because removing that data point results in the same separator, and since $(\mathbf{x}_n, y_n)$ was classified correctly before removal, it will remain correct after removal. For the support vectors, $\mathsf{e}_n \leq 1$ (binary

classification error), and so

$$E_{\text{cv}}(\text{SVM}) = \frac{1}{N}\sum_{n=1}^{N} \mathsf{e}_n \leq \frac{\#\text{ support vectors}}{N}. \tag{8.8}$$

Exercise 8.8

(a) Evaluate the bound in (8.8) for the data in Figure 8.5.

(b) If one of the four support vectors in a gray box are removed, does the classifier change?

(c) Use your answer in (b) to improve your bound in (a).

The support vectors in gray boxes are non-essential and the support vector in the black box is essential. One can improve the bound in (8.8) to use only essential support vectors. The number of support vectors is unbounded, but the number of essential support vectors is at most $d+1$ (usually much less).

In the interest of full disclosure, and out of fairness to the PLA, we should note that a bound on E_{cv} can be obtained for PLA as well, namely

$$E_{\text{cv}}(\text{PLA}) \leq \frac{R^2}{N\rho^2},$$

where ρ is the margin of the thickest hyperplane that separates the data, and R is an upper bound on $\|\mathbf{x}_n\|$ (see Problem 8.11). The table below provides a summary of what we know based on our discussion so far.

Algorithm For Selecting Separating Hyperplane		
General	PLA	SVM (Optimal Hyperplane)
$d_{\text{VC}} = d+1$		$d_{\text{VC}}(\rho) \leq \min\left(\left\lceil \frac{R^2}{\rho^2} \right\rceil, d\right) + 1$
	$E_{\text{cv}} \leq \dfrac{R^2}{N\rho^2}$	$E_{\text{cv}} \leq \dfrac{\#\text{ support vectors}}{N}$

In general, all you can conclude is the VC bound based on a VC dimension of $d+1$. In high dimensions, this bound can be a very loose. For PLA or SVM, we have additional bounds that do not explicitly depend on the dimension d. If the margin is large, or if the number of support vectors is small (even in infinite dimensions), we are still in good shape.

8.1.3 Non-Separable Data

Our entire discussion so far assumed that the data is linearly separable and focused on separating the data with maximum safety cushion. What if the

(a) Few noisy data. (b) Nonlinearly separable.

Figure 8.6: Non-separable data (reproduction of Figure 3.1).

data is not linearly separable? Figure 8.6 (reproduced from Chapter 3) illustrates the two types of non-separability. In Figure 8.6(a), two noisy data points render the data non-separable. In Figure 8.6(b), the target function is inherently nonlinear.

For the learning problem in Figure 8.6(a), we prefer the linear separator, and need to tolerate the few noisy data points. In Chapter 3, we modified the PLA into the pocket algorithm to handle this situation. Similarly, for SVMs, we will modify the hard-margin SVM to the *soft-margin* SVM in Section 8.4. Unlike the hard margin SVM, the soft-margin SVM allows data points to violate the cushion, or even be misclassified.

To address the other situation in Figure 8.6(b), we introduced the nonlinear transform in Chapter 3. There is nothing to stop us from using the nonlinear transform with the optimal hyperplane, which we will do here.

To render the data separable, we would typically transform into a higher dimension. Consider a transform $\Phi\colon \mathbb{R}^d \to \mathbb{R}^{\tilde{d}}$. The transformed data are

$$\mathbf{z}_n = \mathbf{\Phi}(\mathbf{x}_n).$$

After transforming the data, we solve the hard-margin SVM problem in the $\mathcal{Z}$ space, which is just (8.4) written with $\mathbf{z}_n$ instead of $\mathbf{x}_n$:

$$\begin{aligned} \underset{\tilde{b},\tilde{\mathbf{w}}}{\text{minimize:}} \quad & \frac{1}{2}\tilde{\mathbf{w}}^{\mathrm{T}}\tilde{\mathbf{w}} \\ \text{subject to:} \quad & y_n\left(\tilde{\mathbf{w}}^{\mathrm{T}}\mathbf{z}_n + \tilde{b}\right) \geq 1 \qquad (n = 1, \cdots, N), \end{aligned} \tag{8.9}$$

where $\tilde{\mathbf{w}}$ is now in $\mathbb{R}^{\tilde{d}}$ instead of $\mathbb{R}^d$ (recall that we use tilde for objects in $\mathcal{Z}$ space). The optimization problem in (8.9) is a QP-problem with $\tilde{d}+1$

(a) SVM with Φ_2 (b) SVM with Φ_3

Figure 8.7: Nonlinear separation using the SVM with 2nd and 3rd order polynomial transforms. The margin is shaded in yellow, and the support vectors are boxed. The dimension of Φ_3 is nearly double that of Φ_2, yet the resulting SVM separator is not severely overfitting with Φ_3.

optimization variables and N constraints. To solve this optimization problem, we can use the standard hard-margin algorithm in the algorithm box on page 8-10, after we replace $\mathbf{x}_n$ with $\mathbf{z}_n$ and d with $\tilde{d}$. The algorithm returns an optimal solution $\tilde{b}^*, \tilde{\mathbf{w}}^*$ and the final hypothesis is

$$g(\mathbf{x}) = \text{sign}(\tilde{\mathbf{w}}^{*\mathrm{T}}\Phi(\mathbf{x}) + \tilde{b}^*).$$

In Chapter 3, we introduced a general kth-order polynomial transform Φ_k. For example, the second order polynomial transform is

$$\Phi_2(\mathbf{x}) = (x_1, x_2, x_1^2, x_1x_2, x_2^2).$$

Figure 8.7 shows the result of using the SVM with the 2nd and 3rd order polynomial transforms Φ_2, Φ_3 for the data in Figure 8.6(b). Observe that the 'margin' does not have constant width in the $\mathcal{X}$ space; it is in the $\mathcal{Z}$ space that the width of the separator is uniform. Also, in Figure 8.7(b) you can see that some blue points near the top left are not support vectors, even though they appear closer to the separating surface than the red support vectors near the bottom right. Again this is because it is distances to the separator in the $\mathcal{Z}$ space that matter. Lastly, the dimensions of the two feature spaces are $\tilde{d}_2 = 5$ and $\tilde{d}_3 = 9$ (almost double for the 3rd order polynomial transform). However, the number of support vectors increased from 5 to only 6, and so the bound on E_{cv} did not nearly double.

The benefit of the nonlinear transform is that we can use a sophisticated boundary to lower E_{in}. However, you pay a price to get this sophisticated boundary in terms of a larger d_{vc} and a tendency to overfit. This trade-off is highlighted in the table below for PLA, to compare with SVM.

	perceptrons	perceptron + nonlinear transform
complexity control	**small** $d_{\mathbf{vc}}$	large d_{vc}
boundary	linear	**sophisticated**

We observe from Figure 8.7(b) that the 3rd order polynomial SVM does not show the level of overfitting we might expect when we nearly double the number of free parameters. We can have our cake and eat it too: we enjoy the benefit of high-dimensional transforms in terms of getting sophisticated boundaries, and yet we don't pay too much in terms of E_{out} because d_{vc} and E_{cv} can be controlled in terms of quantities not directly linked to $\tilde{d}$. The table illustrating the trade-offs for the SVM is:

	SVM	SVM + nonlinear transform
complexity control	**smaller 'effective'** $d_{\mathbf{vc}}$	**not too large 'effective'** $d_{\mathbf{vc}}$
boundary	linear	**sophisticated**

You now have the support vector machine at your fingertips: a very powerful, easy to use linear model which comes with automatic regularization. We could be done, but instead, we are going to introduce you to a very powerful tool that can be used to implement nonlinear transforms called a *kernel*. The kernel will allow you to fully exploit the capabilities of the SVM. The SVM has a potential robustness to overfitting even after transforming to a much higher dimension that opens up a new world of possibilities: what about using infinite dimensional transforms? Yes, infinite! It is certainly not feasible within our current technology to use an infinite dimensional transform; but, by using the 'kernel trick', not only can we make infinite dimensional transforms feasible, we can also make them efficient. Stay tuned; it's exciting stuff.

8.2 Dual Formulation of the SVM

The promise of infinite dimensional nonlinear transforms certainly whets the appetite, but we are going to have to earn our cookie ☹. We are going to introduce the *dual* SVM problem which is equivalent to the original *primal* problem (8.9) in that solving both problems gives the same optimal hyperplane, but the dual problem is often easier. It is this dual (no pun intended) view of the SVM that will enable us to exploit the kernel trick that we have touted so loudly. But, the dual view also stands alone as an important formulation of the SVM that will give us further insights into the optimal hyperplane. The connection between primal and dual optimization problems is a heavily studied subject in optimization theory, and we only introduce those few pieces that we need for the SVM.

Note that (8.9) is a QP-problem with $\tilde{d}+1$ variables $(\tilde{b}, \tilde{\mathbf{w}})$ and N constraints. It is computationally difficult to solve (8.9) when $\tilde{d}$ is large, let alone

infinite. The dual problem will also be a QP-problem, but with N variables and $N+1$ constraints – the computational burden no longer depends on $\tilde{d}$, which is a big saving when we move to very high $\tilde{d}$.

Deriving the dual is not going to be easy, but on the positive side, we have already seen the main tool that we will need (in Section 4.2 when we talked about regularization). Regularization introduced us to the constrained minimization problem

$$\begin{aligned} \underset{\mathbf{w}}{\text{minimize:}} \quad & E_{\text{in}}(\mathbf{w}) \\ \text{subject to:} \quad & \mathbf{w}^{\mathrm{T}}\mathbf{w} \leq C, \end{aligned}$$

where C is a user-specified parameter. We showed that for a given C, there is a λ such that minimizing the augmented error $E_{\text{aug}}(\mathbf{w}) = E_{\text{in}}(\mathbf{w}) + \lambda\mathbf{w}^{\mathrm{T}}\mathbf{w}$ gives the same regularized solution. We viewed the *Lagrange multiplier* λ as the user-specified parameter instead of C, and minimized the augmented error instead of solving the constrained minimization problem. Here, we are also going to use Lagrange multipliers, but in a slightly different way. The Lagrange multipliers will arise as *variables* that correspond to the constraints, and we need to formulate a new optimization problem (which is the dual problem) to solve for those variables.

8.2.1 Lagrange Dual for a QP-Problem

Let us illustrate the concept of the dual with a simplified version of the standard QP-problem, using just one inequality constraint. All the concepts will easily generalize to the case with more than one constraint. So, consider the QP-problem

$$\begin{aligned} \underset{\mathbf{u}\in\mathbb{R}^L}{\text{minimize:}} \quad & \tfrac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} + \mathbf{p}^{\mathrm{T}}\mathbf{u} \\ \text{subject to:} \quad & \mathbf{a}^{\mathrm{T}}\mathbf{u} \geq c \end{aligned} \tag{8.10}$$

Here is a closely related optimization problem.

$$\underset{\mathbf{u}\in\mathbb{R}^L}{\text{minimize:}} \quad \tfrac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} + \mathbf{p}^{\mathrm{T}}\mathbf{u} + \max_{\alpha\geq 0} \alpha\left(c - \mathbf{a}^{\mathrm{T}}\mathbf{u}\right). \tag{8.11}$$

The variable $\alpha \geq 0$ multiplies the constraint $(c-\mathbf{a}^{\mathrm{T}}\mathbf{u})$.[4] To obtain the objective in (8.11) from (8.10), we add what amounts to a penalty term that encourages $(c-\mathbf{a}^{\mathrm{T}}\mathbf{u})$ to be negative and satisfy the constraint. The optimization problem in (8.10) is equivalent to the one in (8.11) as long as there is at least one solution that satisfies the constraint in (8.10). The advantage in (8.11) is that the minimization with respect to $\mathbf{u}$ is *unconstrained*, and the price is a slightly more complex objective that involves this 'Lagrangian penalty term'. The next exercise proves the equivalence.

[4]The parameter α is called a Lagrange multiplier. In the optimization literature, the Lagrange multiplier would typically be denoted by λ. Historically, the SVM literature has used α, which is the convention we follow.

Exercise 8.9

Let $\mathbf{u}_0$ be optimal for (8.10), and let $\mathbf{u}_1$ be optimal for (8.11).

(a) Show that $\max_{\alpha \geq 0} \alpha\,(c - \mathbf{a}^{\mathrm{T}}\mathbf{u}_0) = 0$. *[Hint: $c - \mathbf{a}^{\mathrm{T}}\mathbf{u}_0 \leq 0$.]*

(b) Show that $\mathbf{u}_1$ is feasible for (8.10). To show this, suppose to the contrary that $c - \mathbf{a}^{\mathrm{T}}\mathbf{u}_1 > 0$. Show that the objective in (8.11) is infinite, whereas $\mathbf{u}_0$ attains a finite objective of $\frac{1}{2}\mathbf{u}_0^{\mathrm{T}}\mathrm{Q}\mathbf{u}_0 + \mathbf{p}^{\mathrm{T}}\mathbf{u}_0$, which contradicts the optimality of $\mathbf{u}_1$.

(c) Show that $\frac{1}{2}\mathbf{u}_1^{\mathrm{T}}\mathrm{Q}\mathbf{u}_1 + \mathbf{p}^{\mathrm{T}}\mathbf{u}_1 = \frac{1}{2}\mathbf{u}_0^{\mathrm{T}}\mathrm{Q}\mathbf{u}_0 + \mathbf{p}^{\mathrm{T}}\mathbf{u}_0$, and hence that $\mathbf{u}_1$ is optimal for (8.10) and $\mathbf{u}_0$ is optimal for (8.11).

(d) Let $\mathbf{u}^*$ be any optimal solution for (8.11) with $\max_{\alpha \geq 0} \alpha(c - \mathbf{a}^{\mathrm{T}}\mathbf{u}^*)$ attained at α^*. Show that

$$\alpha^*(c - \mathbf{a}^{\mathrm{T}}\mathbf{u}^*) = 0. \tag{8.12}$$

Either the constraint is exactly satisfied with $c - \mathbf{a}^{\mathrm{T}}\mathbf{u}^* = 0$, or $\alpha^* = 0$.

Exercise 8.9 shows that as long as the original problem in (8.10) is feasible, we can obtain a solution by solving (8.11) instead. Let us analyze (8.11) in further detail. Our goal is to simplify it. Introduce the Lagrangian function $\mathcal{L}(\mathbf{u}, \alpha)$, defined by

$$\mathcal{L}(\mathbf{u}, \boldsymbol{\alpha}) = \tfrac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} + \mathbf{p}^{\mathrm{T}}\mathbf{u} + \alpha\,(c - \mathbf{a}^{\mathrm{T}}\mathbf{u})\,. \tag{8.13}$$

In terms of $\mathcal{L}$, the optimization problem in (8.11) is

$$\min_{\mathbf{u}} \max_{\alpha \geq 0} \mathcal{L}(\mathbf{u}, \alpha). \tag{8.14}$$

For convex quadratic programming, when $\mathcal{L}(\mathbf{u}, \alpha)$ has the special form in (8.13) and $c - \mathbf{a}^{\mathrm{T}}\mathbf{u} \leq 0$ is feasible, a profound relationship known as *strong duality* has been proven to hold:

$$\min_{\mathbf{u}} \max_{\alpha \geq 0} \mathcal{L}(\mathbf{u}, \alpha) = \max_{\alpha \geq 0} \min_{\mathbf{u}} \mathcal{L}(\mathbf{u}, \alpha). \tag{8.15}$$

An interested reader can consult a standard reference in convex optimization for a proof. The impact for us is that a solution to the optimization problem on the RHS of (8.15) gives a solution to the problem on the LHS, which is the problem we want to solve. This helps us because on the RHS, one is first minimizing with respect to an *unconstrained* $\mathbf{u}$, and that we can do analytically. This analytical step considerably reduces the complexity of the problem. Solving the problem on the RHS of (8.15) is known as solving the *Lagrange dual problem* (dual problem for short). The original problem is called the *primal* problem.

We briefly discuss what to do when there are many constraints, $\mathbf{a}_m^{\mathrm{T}}\mathbf{u} \geq c_m$ for $m = 1, \ldots, M$. Not much changes. All we do is introduce a Lagrange

multiplier $\alpha_m \geq 0$ for *each* constraint and add the penalty term $\alpha_m(c_m - \mathbf{a}_m^{\mathrm{T}}\mathbf{u})$ into the Lagrangian. Then, just as before, we solve the dual problem. A simple example will illustrate all the mechanics.

Example 8.6. Let's minimize $u_1^2 + u_2^2$ subject to $u_1 + 2u_2 \geq 2$ and $u_1, u_2 \geq 0$. We first construct the Lagrangian,

$$\mathcal{L}(\mathbf{u}, \boldsymbol{\alpha}) = u_1^2 + u_2^2 + \alpha_1(2 - u_1 - 2u_2) - \alpha_2 u_1 - \alpha_3 u_2,$$

where, as you can see, we have added a penalty term for each of the three constraints, and each penalty term has an associated Lagrange multiplier. To solve the dual optimization problem, we first need to minimize $\mathcal{L}(\mathbf{u}, \boldsymbol{\alpha})$ with respect to the unconstrained $\mathbf{u}$. We then need to maximize with respect to $\boldsymbol{\alpha} \geq \mathbf{0}$. To do the unconstrained minimization with respect to $\mathbf{u}$, we use the standard first derivative condition from calculus:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial u_1} = 0 \quad &\Longrightarrow \quad u_1 = \frac{\alpha_1 + \alpha_2}{2}; \\ \frac{\partial \mathcal{L}}{\partial u_2} = 0 \quad &\Longrightarrow \quad u_2 = \frac{2\alpha_1 + \alpha_3}{2}. \end{aligned} \tag{$*$}$$

Plugging these values for u_1 and u_2 back into $\mathcal{L}$ and collecting terms, we have a function only of $\boldsymbol{\alpha}$, which we have to maximize with respect to $\alpha_1, \alpha_2, \alpha_3$:

$$\begin{array}{ll} \text{maximize:} & \mathcal{L}(\boldsymbol{\alpha}) = -\frac{5}{4}\alpha_1^2 - \frac{1}{4}\alpha_2^2 - \frac{1}{4}\alpha_3^2 - \frac{1}{2}\alpha_1\alpha_2 - \alpha_1\alpha_3 + 2\alpha_1 \\ \text{subject to:} & \alpha_1, \alpha_2, \alpha_3 \geq 0 \end{array}$$

Exercise 8.10

Do the algebra. Derive $(*)$ and plug it into $\mathcal{L}(\mathbf{u}, \boldsymbol{\alpha})$ to obtain $\mathcal{L}(\boldsymbol{\alpha})$.

By going to the dual, all we accomplished is to obtain another QP-problem! So, in a sense, we have not solved the problem at all. What did we gain? The new problem is easier to solve. This is because the constraints for the dual problem are simple ($\boldsymbol{\alpha} \geq \mathbf{0}$). In our case, all terms involving α_2 and α_3 are at most zero, and we can attain zero by setting $\alpha_2 = \alpha_3 = 0$. This leaves us with $-\frac{5}{4}\alpha_1^2 + 2\alpha_1$, which is maximized when $\alpha_1 = \frac{4}{5}$. So the final solution is $\alpha_1 = \frac{4}{5}$, $\alpha_2 = \alpha_3 = 0$, and plugging these into $(*)$ gives

$$u_1 = \tfrac{2}{5} \qquad \text{and} \qquad u_2 = \tfrac{4}{5}.$$

The optimal value for the objective is $u_1^2 + u_2^2 = \frac{4}{5}$. □

We summarize our discussion in the following theorem, which states the dual formulation of a QP-problem. In the next section, we will show how this dual formulation is applied to the SVM QP-problem. The theorem looks formidable, but don't be intimidated. Its application to the SVM QP-problem is conceptually no different from our little example above.

Theorem 8.7 (KKT)**.** For a feasible convex QP-problem in *primal* form,

$$\begin{aligned} \underset{\mathbf{u}\in\mathbb{R}^L}{\text{minimize:}} \quad & \tfrac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} + \mathbf{p}^{\mathrm{T}}\mathbf{u} \\ \text{subject to:} \quad & \mathbf{a}_m^{\mathrm{T}}\mathbf{u} \geq c_m \qquad (m = 1, \cdots, M), \end{aligned}$$

define the Lagrange function

$$\mathcal{L}(\mathbf{u}, \boldsymbol{\alpha}) = \tfrac{1}{2}\mathbf{u}^{\mathrm{T}}\mathrm{Q}\mathbf{u} + \mathbf{p}^{\mathrm{T}}\mathbf{u} + \sum_{m=1}^{M} \alpha_m \left(c_m - \mathbf{a}_m^{\mathrm{T}}\mathbf{u}\right).$$

The solution $\mathbf{u}^*$ is optimal for the primal if and only if $(\mathbf{u}^*, \boldsymbol{\alpha}^*)$ is a solution to the dual optimization problem

$$\max_{\boldsymbol{\alpha}\geq\mathbf{0}} \min_{\mathbf{u}} \mathcal{L}(\mathbf{u}, \boldsymbol{\alpha}).$$

The optimal $(\mathbf{u}^*, \boldsymbol{\alpha}^*)$ satisfies the Karush-Kühn-Tucker (KKT) conditions:

(i) *Primal and dual constraints:*

$$\mathbf{a}_m^{\mathrm{T}}\mathbf{u}^* \geq c_m \qquad \text{and} \qquad \alpha_m \geq 0 \qquad (m = 1, \cdots, M).$$

(ii) *Complementary slackness:*

$$\alpha_m^* \left(\mathbf{a}_m^{\mathrm{T}}\mathbf{u}^* - c_m\right) = 0.$$

(iii) *Stationarity with respect to* $\mathbf{u}$*:*

$$\nabla_{\mathbf{u}}\mathcal{L}(\mathbf{u}, \boldsymbol{\alpha})\big|_{\mathbf{u}=\mathbf{u}^*, \boldsymbol{\alpha}=\boldsymbol{\alpha}^*} = \mathbf{0}.$$

The three KKT conditions in Theorem 8.7 characterize the relationship between the optimal $\mathbf{u}^*$ and $\boldsymbol{\alpha}^*$ and can often be used to simplify the Lagrange dual problem. The constraints are inherited from the constraints in the definitions of the primal and dual optimization problems. Complementary slackness is a condition you derived in (8.12) which says that at the optimal solution, the constraints fall into two types: those which are on the boundary and are exactly satisfied (the *active* constraints) and those which are in the interior of the feasible set. The interior constraints must have Lagrange multipliers equal to zero. The stationarity with respect to $\mathbf{u}$ is the necessary and sufficient condition for a convex program to solve the first part of the dual problem, namely $\min_{\mathbf{u}} \mathcal{L}(\mathbf{u}, \boldsymbol{\alpha})$. Next, we focus on the hard-margin SVM, and use the KKT conditions, in particular, stationarity with respect to $\mathbf{u}$, to simplify the Lagrange dual problem.

8.2.2 Dual of the Hard-Margin SVM

We now apply the KKT theorem to the convex QP-problem for hard-margin SVM (8.9). The mechanics of our derivation are not more complex than in

Example 8.6, though the algebra is more cumbersome. The steps we followed in Example 8.6 are a helpful guide to keep in mind.

The hard-margin optimization problem only applies when the data is linearly separable. This means that the optimization problem is convex and feasible, so the KKT theorem applies. For your convenience, we reproduce the hard-margin SVM QP-problem here,

$$\begin{aligned} \underset{b,\mathbf{w}}{\text{minimize:}} \quad & \tfrac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \\ \text{subject to:} \quad & y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) \geq 1 \qquad (n = 1, \cdots, N). \end{aligned} \tag{8.16}$$

The optimization variable is $\mathbf{u} = \left[\begin{smallmatrix} b \\ \mathbf{w} \end{smallmatrix}\right]$. The first task is to construct the Lagrangian. There are N constraints, so we add N penalty terms, and introduce a Lagrange multiplier α_n for each penalty term. The Lagrangian is

$$\begin{aligned} \mathcal{L}(b, \mathbf{w}, \boldsymbol{\alpha}) &= \tfrac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + \sum_{n=1}^{N} \alpha_n \left(1 - y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)\right) \\ &= \tfrac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} - \sum_{n=1}^{N} \alpha_n y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n - b\sum_{n=1}^{N} \alpha_n y_n + \sum_{n=1}^{N} \alpha_n. \end{aligned} \tag{8.17}$$

We must first minimize $\mathcal{L}$ with respect to $(b, \mathbf{w})$ and then maximize with respect to $\boldsymbol{\alpha} \geq \mathbf{0}$. To minimize with respect to $(b, \mathbf{w})$, we need the derivatives of $\mathcal{L}$. The derivative with respect to b is just the coefficient of b because b appears linearly in the Lagrangian. To differentiate the terms involving $\mathbf{w}$, we need some vector calculus identities from the linear algebra appendix: $\frac{\partial}{\partial \mathbf{w}}\mathbf{w}^{\mathrm{T}}\mathbf{w} = 2\mathbf{w}$ and $\frac{\partial}{\partial \mathbf{w}}\mathbf{w}^{\mathrm{T}}\mathbf{x} = \mathbf{x}$. The reader may now verify that

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{n=1}^{N} \alpha_n y_n \qquad \text{and} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n.$$

Setting these derivatives to zero gives

$$\sum_{n=1}^{N} \alpha_n y_n = 0; \tag{8.18}$$

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n. \tag{8.19}$$

Something strange has happened (highlighted in red), which did not happen in Example 8.6. After setting the derivatives to zero in Example 8.6, we were able to solve for $\mathbf{u}$ in terms of the Lagrange multipliers $\boldsymbol{\alpha}$. Here, the stationarity condition for b does not allow us to solve for b directly ☹. Instead, we got a *constraint* that $\boldsymbol{\alpha}$ must satisfy at the final solution. Don't let this unsettle you. The KKT theorem will allow us to ultimately recover b ☺. This constraint on $\boldsymbol{\alpha}$ is not surprising. Any choice for $\boldsymbol{\alpha}$ which does not satisfy (8.18) would

allow $\mathcal{L} \to -\infty$ by appropriately choosing b. Since we *maximize* over $\boldsymbol{\alpha}$, we must choose $\boldsymbol{\alpha}$ to satisfy the constraint and prevent $\mathcal{L} \to -\infty$.

We proceed as in Example 8.6 by plugging the stationarity constraints back into the Lagrangian to get a function solely in terms of $\boldsymbol{\alpha}$. Observe that the constraint in (8.18) means that the term involving b in the Lagrangian (8.17) reduces to zero. The terms in the Lagrangian (8.17) involving the weights $\mathbf{w}$ simplify when we use the expression in (8.19):

$$\begin{aligned}
& \tfrac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} - \sum_{n=1}^{N} \alpha_n y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n \\
= \quad & \frac{1}{2} \underbrace{\sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n^{\mathrm{T}}}_{\mathbf{w}^{\mathrm{T}}} \underbrace{\sum_{m=1}^{N} \alpha_m y_m \mathbf{x}_m}_{\mathbf{w}} - \sum_{n=1}^{N} \alpha_n y_n \underbrace{\sum_{m=1}^{N} \alpha_m y_m \mathbf{x}_m^{\mathrm{T}}}_{\mathbf{w}^{\mathrm{T}}} \mathbf{x}_n \\
= \quad & \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} y_n y_m \alpha_n \alpha_m \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_m - \sum_{n=1}^{N} \sum_{m=1}^{N} y_n y_m \alpha_n \alpha_m \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_m \\
= \quad & -\frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} y_n y_m \alpha_n \alpha_m \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_m. \qquad (8.20)
\end{aligned}$$

After we use (8.18) and (8.20) in (8.17), the Lagrangian reduces to a simpler function of just the variable $\boldsymbol{\alpha}$:

$$\mathcal{L}(\boldsymbol{\alpha}) = -\frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} y_n y_m \alpha_n \alpha_m \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_m + \sum_{n=1}^{N} \alpha_n.$$

We must now maximize $\mathcal{L}(\boldsymbol{\alpha})$ subject to $\boldsymbol{\alpha} \geq \mathbf{0}$, and we cannot forget the constraint (8.18) which we inherited from the stationarity condition on b. We can equivalently *minimize* the negative of $\mathcal{L}(\boldsymbol{\alpha})$, and so we have the following optimization problem to solve.

$$\begin{aligned}
\underset{\boldsymbol{\alpha} \in \mathbb{R}^N}{\text{minimize:}} \quad & \frac{1}{2} \sum_{m=1}^{N} \sum_{n=1}^{N} y_n y_m \alpha_n \alpha_m \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_m - \sum_{n=1}^{N} \alpha_n \qquad (8.21) \\
\text{subject to:} \quad & \sum_{n=1}^{N} y_n \alpha_n = 0 \\
& \alpha_n \geq 0 \qquad (n = 1, \cdots, N).
\end{aligned}$$

If you made it to here, you deserve a pat on the back for surviving all the algebra ☺. The result is similar to what happened in Example 8.6. We have not solved the problem; we have reduced it to another QP-problem. The next exercise guides you through the steps to put (8.21) into the standard QP form.

Exercise 8.11

(a) Show that the problem in (8.21) is a standard QP-problem:

$$\begin{aligned} \underset{\boldsymbol{\alpha}\in\mathbb{R}^N}{\text{minimize:}} \quad & \tfrac{1}{2}\boldsymbol{\alpha}^{\mathrm{T}}\mathrm{Q}_{\mathrm{D}}\boldsymbol{\alpha} - \mathbf{1}_N^{\mathrm{T}}\boldsymbol{\alpha} \\ \text{subject to:} \quad & \mathrm{A}_{\mathrm{D}}\boldsymbol{\alpha} \geq \mathbf{0}_{N+2}, \end{aligned} \tag{8.22}$$

where Q_{D} and A_{D} (D for dual) are given by:

$$\mathrm{Q}_{\mathrm{D}} = \begin{bmatrix} y_1 y_1 \mathbf{x}_1^{\mathrm{T}}\mathbf{x}_1 & \dots & y_1 y_N \mathbf{x}_1^{\mathrm{T}}\mathbf{x}_N \\ y_2 y_1 \mathbf{x}_2^{\mathrm{T}}\mathbf{x}_1 & \dots & y_2 y_N \mathbf{x}_2^{\mathrm{T}}\mathbf{x}_N \\ \vdots & \vdots & \vdots \\ y_N y_1 \mathbf{x}_N^{\mathrm{T}}\mathbf{x}_1 & \dots & y_N y_N \mathbf{x}_N^{\mathrm{T}}\mathbf{x}_N \end{bmatrix} \text{ and } \mathrm{A}_{\mathrm{D}} = \begin{bmatrix} \mathbf{y}^{\mathrm{T}} \\ -\mathbf{y}^{\mathrm{T}} \\ \mathrm{I}_{N\times N} \end{bmatrix}.$$

[Hint: Recall that an equality corresponds to two inequalities.]

(b) The matrix Q_{D} of quadratic coefficients is $[\mathrm{Q}_{\mathrm{D}}]_{mn} = y_m y_n \mathbf{x}_m^{\mathrm{T}}\mathbf{x}_n$. Show that $\mathrm{Q}_{\mathrm{D}} = \mathrm{X}_s\mathrm{X}_s^{\mathrm{T}}$, where X_s is the 'signed data matrix',

$$\mathrm{X}_s = \begin{bmatrix} — \; y_1\mathbf{x}_1^{\mathrm{T}} \; — \\ — \; y_2\mathbf{x}_2^{\mathrm{T}} \; — \\ \vdots \\ — y_N\mathbf{x}_N^{\mathrm{T}} — \end{bmatrix}.$$

Hence, show that Q_{D} is positive semi-definite. This implies that the QP-problem is convex.

Example 8.8. Let us illustrate all the machinery of the dual formulation using the data in Example 8.2 (the toy problem from Section 8.1). For your convenience, we reproduce the data, and we compute $\mathrm{Q}_{\mathrm{D}}, \mathrm{A}_{\mathrm{D}}$ using Exercise 8.11:

$$\mathrm{X} = \begin{bmatrix} 0 & 0 \\ 2 & 2 \\ 2 & 0 \\ 3 & 0 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ +1 \\ +1 \end{bmatrix}; \quad \mathrm{Q}_{\mathrm{D}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 8 & -4 & -6 \\ 0 & -4 & 4 & 6 \\ 0 & -6 & 6 & 9 \end{bmatrix}, \quad \mathrm{A}_{\mathrm{D}} = \begin{bmatrix} -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

We can now write down the optimization problem in (8.21) and manually solve it to get the optimal $\boldsymbol{\alpha}$. The masochist reader is invited to do so in Problem 8.3. Instead, we can use a QP-solver to solve (8.22) in under a millisecond:

$$\boldsymbol{\alpha}^* \leftarrow \mathsf{QP}(\mathrm{Q}_{\mathrm{D}}, -\mathbf{1}, \mathrm{A}_{\mathrm{D}}, \mathbf{0}) \qquad \text{gives} \qquad \boldsymbol{\alpha}^* = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ 1 \\ 0 \end{bmatrix}.$$

Now that we have $\boldsymbol{\alpha}^*$, we can compute the optimal weights using (8.19),

$$\mathbf{w}^* = \sum_{n=1}^{N} y_n \alpha_n^* \mathbf{x}_n = -\tfrac{1}{2}\begin{bmatrix}0\\0\end{bmatrix} - \tfrac{1}{2}\begin{bmatrix}2\\2\end{bmatrix} + \begin{bmatrix}2\\0\end{bmatrix} = \begin{bmatrix}1\\-1\end{bmatrix}.$$

As expected, these are the same optimal weights we got in Example 8.2. What about b? This is where things get tricky. Recall the complementary slackness KKT condition in Theorem 8.7. It asserts that if $\alpha_n^* > 0$, then the corresponding constraint must be exactly satisfied. In our example, $\alpha_1 = \frac{1}{2} > 0$, so $y_1(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_1 + b^*) = 1$. Since $\mathbf{x}_1 = \mathbf{0}$ and $y_1 = -1$, this means $b^* = -1$, exactly as in Example 8.2. So, $g(\mathbf{x}) = \mathrm{sign}(x_1 - x_2 - 1)$. □

On a practical note, the $N \times N$ matrix $\mathrm{Q_D}$ is often dense (containing lots of non-zero elements). If $N = 100,000$, storing $\mathrm{Q_D}$ uses more than 10GB of RAM. Thus, for large-scale applications, specially tailored QP packages that dynamically compute $\mathrm{Q_D}$ and use specific properties of SVM are often used to solve (8.22) efficiently.

8.2.3 Recovering the SVM from the Dual Solution

Now that we have solved the dual problem using quadratic programming to obtain the optimal solution $\boldsymbol{\alpha}^*$, what remains is to compute the optimal hyperplane $(b^*, \mathbf{w}^*)$. The weights are easy, and are obtained using the stationary condition in (8.19):

$$\mathbf{w}^* = \sum_{n=1}^{N} y_n \alpha_n^* \mathbf{x}_n. \tag{8.23}$$

Assume that the data contains at least one positive and one negative example (otherwise the classification problem is trivial). Then, at least one of the $\{\alpha_n^*\}$ will be strictly positive. Let $\alpha_s^* > 0$ (s for *support* vector). The KKT complementary slackness condition in Theorem 8.7 tells us that the constraint corresponding to this non-zero α_s^* is equality. This means that

$$y_s\left(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_s + b^*\right) = 1.$$

We can solve for b^* in the above equation to get

$$\begin{aligned} b^* &= y_s - \mathbf{w}^{*\mathrm{T}}\mathbf{x}_s \\ &= y_s - \sum_{n=1}^{N} y_n \alpha_n^* \mathbf{x}_n^{\mathrm{T}}\mathbf{x}_s. \end{aligned} \tag{8.24}$$

Exercise 8.12

If all the data is from one class, then $\alpha_n^* = 0$ for $n = 1, \ldots, N$.

(a) What is $\mathbf{w}^*$?

(b) What is b^*?

Equations (8.23) and (8.24) connect the optimal $\boldsymbol{\alpha}^*$, which we get from solving the dual problem, to the optimal hyperplane $(b^*, \mathbf{w}^*)$ which solves (8.4). Most importantly, the optimal hypothesis is

$$\begin{aligned} g(\mathbf{x}) &= \text{sign}(\mathbf{w}^{*\mathrm{T}}\mathbf{x} + b^*) \\ &= \text{sign}\left(\sum_{n=1}^{N} y_n \alpha_n^* \mathbf{x}_n^{\mathrm{T}}\mathbf{x} + b^*\right) \\ &= \text{sign}\left(\sum_{n=1}^{N} y_n \alpha_n^* \mathbf{x}_n^{\mathrm{T}}(\mathbf{x} - \mathbf{x}_s) + y_s\right). \end{aligned} \tag{8.25}$$

Recall that $(\mathbf{x}_s, y_s)$ is *any* support vector which is defined by the condition $\alpha_s^* > 0$. There is a summation over n in Equations (8.23), (8.24) and (8.25), but only the terms with $\alpha_n^* > 0$ contribute to the summations. We can therefore get a more efficient representation for $g(\mathbf{x})$ using only the positive α_n^*:

$$g(\mathbf{x}) = \text{sign}\left(\sum_{\alpha_n^* > 0} y_n \alpha_n^* \mathbf{x}_n^{\mathrm{T}}\mathbf{x} + b^*\right), \tag{8.26}$$

where b^* is given by (8.24) (the summation (8.24) can also be restricted to only those $\alpha_n^* > 0$). So, $g(\mathbf{x})$ is determined by only those examples $(\mathbf{x}_n, y_n)$ for which $\alpha_n^* > 0$. We summarize our long discussion about the dual in the following algorithm box.

Hard-Margin SVM with Dual QP

1: Construct $\mathrm{Q_D}$ and $\mathrm{A_D}$ as in Exercise 8.11

$$\mathrm{Q_D} = \begin{bmatrix} y_1 y_1 \mathbf{x}_1^{\mathrm{T}}\mathbf{x}_1 & \dots & y_1 y_N \mathbf{x}_1^{\mathrm{T}}\mathbf{x}_N \\ y_2 y_1 \mathbf{x}_2^{\mathrm{T}}\mathbf{x}_1 & \dots & y_2 y_N \mathbf{x}_2^{\mathrm{T}}\mathbf{x}_N \\ \vdots & \vdots & \vdots \\ y_N y_1 \mathbf{x}_N^{\mathrm{T}}\mathbf{x}_1 & \dots & y_N y_N \mathbf{x}_N^{\mathrm{T}}\mathbf{x}_N \end{bmatrix} \text{ and } \mathrm{A_D} = \begin{bmatrix} \mathbf{y}^{\mathrm{T}} \\ -\mathbf{y}^{\mathrm{T}} \\ \mathrm{I}_{N\times N} \end{bmatrix}.$$

2: Use a QP-solver to optimize the dual problem:

$$\boldsymbol{\alpha}^* \leftarrow \mathsf{QP}(\mathrm{Q_D}, -\mathbf{1}_N, \mathrm{A_D}, \mathbf{0}_{N+2}).$$

3: Let s be a support vector for which $\alpha_s^* > 0$. Compute b^*,

$$b^* = y_s - \sum_{\alpha_n^* > 0} y_n \alpha_n^* \mathbf{x}_n^{\mathrm{T}}\mathbf{x}_s.$$

4: Return the final hypothesis

$$g(\mathbf{x}) = \text{sign}\left(\sum_{\alpha_n^* > 0} y_n \alpha_n^* \mathbf{x}_n^{\mathrm{T}}\mathbf{x} + b^*\right).$$

The *support vectors* are the examples $(\mathbf{x}_s, y_s)$ for which $\alpha_s^* > 0$. The support vectors play two important roles. First, they serve to identify the data points on boundary of the optimal fat-hyperplane. This is because of the complementary slackness condition in the KKT theorem:

$$y_s\left(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_s + b^*\right) = 1.$$

This condition identifies the support vectors as being closest to, in fact on, the boundary of the optimal fat-hyperplane. The leads us to an interesting geometric interpretation: the dual SVM identifies the support vectors on the boundary of the optimal fat-hyperplane, and uses *only* those support vectors to construct the final classifier. We already highlighted these support vectors in Section 8.1, where we used the term support vector to highlight the fact that these data points are 'supporting' the cushion, preventing it from expanding further.

Exercise 8.13

KKT complementary slackness gives that if $\alpha_n^* > 0$, then $(\mathbf{x}_n, y_n)$ is on the boundary of the optimal fat-hyperplane and $y_n\left(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n + b^*\right) = 1$.

Show that the reverse is not true. Namely, it is possible that $\alpha_n^* = 0$ and yet $(\mathbf{x}_n, y_n)$ is on the boundary satisfying $y_n\left(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n + b^*\right) = 1$.

[Hint: Consider a toy data set with two positive examples at $(0,0)$ *and* $(1,0)$*, and one negative example at* $(0,1)$*.]*

The previous exercise says that from the dual, we identify a subset of the points on the boundary of the optimal fat-hyperplane which are called support vectors. All points on the boundary are support vector *candidates*, but only those with $\alpha_n^* > 0$ (and contribute to $\mathbf{w}^*$) are the support vectors.

The second role played by the support vectors is to determine the final hypothesis $g(\mathbf{x})$ through (8.26). The SVM dual problem directly identifies the data points relevant to the final hypothesis. Observe that *only* these support vectors are needed to compute $g(\mathbf{x})$.

Example 8.9. In Example 8.8 we computed $\boldsymbol{\alpha}^* = (\frac{1}{2}, \frac{1}{2}, 1, 0)$ for our toy problem. Since α_1^* is positive, we can choose $(\mathbf{x}_1, y_1) = (\mathbf{0}, -1)$ as our support vector to compute b^* in (8.24). The final hypothesis is

$$\begin{aligned} g(\mathbf{x}) &= \operatorname{sign}\left(-\tfrac{1}{2}\cdot\left[\begin{smallmatrix}2 & 2\end{smallmatrix}\right]\cdot\left[\begin{smallmatrix}x_1\\ x_2\end{smallmatrix}\right] + \left[\begin{smallmatrix}2 & 0\end{smallmatrix}\right]\cdot\left[\begin{smallmatrix}x_1\\ x_2\end{smallmatrix}\right] - 1\right) \\ &= \operatorname{sign}\left(x_1 - x_2 - 1\right), \end{aligned}$$

computed now for the third time ☺. □

We used the fact that only the support vectors are needed to compute the final hypothesis to derive the upper bound on E_{cv} given in (8.8); this upper bound depends only on the number of support vectors. Actually, our bound

on E_{cv} is based on the number of support vector candidates. Since not all support vector candidates are support vectors, one usually gets a much better bound on E_{cv} by instead using the number of support vectors. That is,

$$E_{\text{cv}} \leq \frac{\text{number of } \alpha_n^* > 0}{N}. \tag{8.27}$$

To prove this bound, it is necessary to show that if one throws out any data point with $\alpha_n = 0$, the final hypothesis is not changed. The next exercise asks you to to do exactly this.

Exercise 8.14

Suppose that we removed a data point $(\mathbf{x}_n, y_n)$ with $\alpha_n^* = 0$.

(a) Show that the previous optimal solution $\boldsymbol{\alpha}^*$ remains feasible for the new dual problem (8.21) (after removing α_n^*) .

(b) Show that if there is any other feasible solution for the new dual that has a lower objective value than $\boldsymbol{\alpha}^*$, this would contradict the optimality of $\boldsymbol{\alpha}^*$ for the original dual problem.

(c) Hence, show that $\boldsymbol{\alpha}^*$ (minus α_n^*) is optimal for the new dual.

(d) Hence, show that the optimal fat-hyperplane did not change.

(e) Prove the bound on E_{cv} in (8.27).

In practice, there typically aren't many support vectors, so the bound in (8.27) can be quite good. Figure 8.8 shows a data set with 50 random data points and the resulting optimal hyperplane with 3 support vectors (in black boxes). The support vectors are identified from the dual solution ($\alpha_n^* > 0$). Figure 8.8

Figure 8.8: Support vectors identified from the SVM dual (3 black boxes).

supports the fact that there are often just a few support vectors. This means that the optimal $\boldsymbol{\alpha}^*$ is usually *sparse*, containing many zero elements and a few non-zeros. The sparsity property means that the representation of $g(\mathbf{x})$ in (8.26) can be computed efficiently using only these few support vectors. If there are many support vectors, it is usually more efficient to compute $(b^*, \mathbf{w}^*)$ ahead of time, and use $g(\mathbf{x}) = \text{sign}(\mathbf{w}^{*\mathrm{T}}\mathbf{x} + b^*)$ for prediction.

8.3 Kernel Trick for SVM

We advertised the kernel as a way to use nonlinear transforms into high dimensional spaces efficiently. We are now going to deliver on that promise for SVM. In order to couple the kernel with SVM, we need to view SVM from the dual formulation. And that is why we expended considerable effort to understand this alternative dual view of SVM. The kernel, together with the dual formulation, will allow us to efficiently run SVM with transforms to high or even infinite dimensional spaces.

8.3.1 Kernel Trick via Dual SVM

Let's start by revisiting the procedure for solving nonlinear SVM from the dual formulation based on a nonlinear transform $\Phi\colon \mathcal{X} \to \mathcal{Z}$, which can be done by replacing $\mathbf{x}$ by $\mathbf{z} = \Phi(\mathbf{x})$ in the algorithm box on page 8-30. First, calculate the coefficients for the dual problem that includes the $\mathrm{Q_D}$ matrix; then solve the dual problem to identify the non-zero Lagrange multipliers α_n^* and the corresponding support vectors $(\mathbf{x}_n, y_n)$; finally, use one of the support vectors to calculate b^*, and return the hypothesis $g(\mathbf{x})$ based on b^*, the support vectors, and their Lagrange multipliers.

Throughout the procedure, the only step that may depend on $\tilde{d}$, which is the dimension of $\Phi(\mathbf{x})$, is in calculating the $\mathcal{Z}$-space inner product

$$\Phi(\mathbf{x})^{\mathrm{T}}\Phi(\mathbf{x}').$$

This inner product is needed in the formulation of $\mathrm{Q_D}$ and in the expression for $g(\mathbf{x})$. The 'kernel trick' is based on the following idea: If the transform *and* the inner product can be jointly and efficiently computed in a way that is independent of $\tilde{d}$, the whole nonlinear SVM procedure can be carried out without computing/storing each $\Phi(\mathbf{x})$ explicitly. Then, the procedure can work efficiently for a large or even an infinite $\tilde{d}$.

So the question is, can we effectively do the transform and compute the inner product in an efficient way irrespective of $\tilde{d}$? Let us first define a function that combines both the transform and the inner product:

$$K_\Phi(\mathbf{x}, \mathbf{x}') \equiv \Phi(\mathbf{x})^{\mathrm{T}}\Phi(\mathbf{x}'). \tag{8.28}$$

This function is called a *kernel function*, or just *kernel*. The kernel takes as input two vectors in the $\mathcal{X}$ space and outputs the inner product that would

be computed in the $\mathcal{Z}$ space (for the transform Φ). In its explicit form (8.28), it appears that the kernel transforms the inputs $\mathbf{x}$ and $\mathbf{x}'$ to the $\mathcal{Z}$ space and then computes the inner product. The efficiency of this process would certainly depend on the dimension of the $\mathcal{Z}$ space, which is $\tilde{d}$. The question is whether we can compute $K_\Phi(\mathbf{x}, \mathbf{x}')$ more efficiently.

For two cases of specific nonlinear transforms Φ, we are going to demonstrate that their corresponding kernel functions K_Φ can indeed be efficiently computed, with a cost proportional to d instead of $\tilde{d}$. (For simplicity, we will use K instead of K_Φ when Φ is clear from the context.)

Polynomial Kernel. Consider the second-order polynomial transform:

$$\Phi_2(\mathbf{x}) \quad = \quad (1, x_1, x_2, \ldots, x_d, x_1x_1, x_1x_2, \ldots, x_dx_d),$$

where $\tilde{d} = 1 + d + d^2$ (the identical features x_ix_j and x_jx_i are included separately for mathematical convenience as you will see below). A direct computation of $\Phi_2(\mathbf{x})$ takes $O(\tilde{d})$ in time, and thus a direct computation of

$$\Phi_2(\mathbf{x})^{\mathrm{T}}\Phi_2(\mathbf{x}') \quad = \quad 1 + \sum_{i=1}^{d} x_ix_i' + \sum_{i=1}^{d}\sum_{j=1}^{d} x_ix_jx_i'x_j'$$

also takes time $O(\tilde{d})$. We can simplify the double summation by reorganizing the terms into a product of two separate summations:

$$\sum_{i=1}^{d}\sum_{j=1}^{d} x_ix_jx_i'x_j' = \sum_{i=1}^{d} x_ix_i' \times \sum_{j=1}^{d} x_jx_j' = (\mathbf{x}^{\mathrm{T}}\mathbf{x}')^2.$$

Therefore, we can calculate $\Phi_2(\mathbf{x})^{\mathrm{T}}\Phi_2(\mathbf{x}')$ by an equivalent function

$$K(\mathbf{x}, \mathbf{x}') = 1 + (\mathbf{x}^{\mathrm{T}}\mathbf{x}') + (\mathbf{x}^{\mathrm{T}}\mathbf{x}')^2.$$

In this instance, we see that K can be easily computed in time $O(d)$, which is asymptotically faster than $\tilde{d}$. So, we have demonstrated that, for the polynomial transform, the kernel K is a mathematical and computational shortcut that allows us to combine the transform *and* the inner product into a single more efficient function.

If the kernel K is efficiently computable for some specific Φ, as is the case for our polynomial transform, then whenever we need to compute the inner product of transformed inputs in the $\mathcal{Z}$ space, we can use the *kernel trick* and instead compute the kernel function of those inputs in the $\mathcal{X}$ space. Any learning technique that uses inner products can benefit from this kernel trick.

In particular, let us look back at the SVM dual problem transformed into the $\mathcal{Z}$ space. We obtain the dual problem in the $\mathcal{Z}$ space by replacing every instance of $\mathbf{x}_n$ in (8.21) with $\mathbf{z}_n = \Phi(\mathbf{x}_n)$. After we do this, the dual

optimization problem becomes:

$$\begin{aligned} \underset{\boldsymbol{\alpha}\in\mathbb{R}^N}{\text{minimize:}} \quad & \frac{1}{2}\sum_{m=1}^{N}\sum_{n=1}^{N} y_n y_m \alpha_n \alpha_m \mathbf{z}_n^{\mathrm{T}}\mathbf{z}_m - \sum_{n=1}^{N}\alpha_n \\ \text{subject to:} \quad & \sum_{n=1}^{N} y_n\alpha_n = 0 \\ & \alpha_n \geq 0 \qquad (n = 1, \cdots, N). \end{aligned} \tag{8.29}$$

Now, recall the steps to obtain our final hypothesis. We solve the dual to obtain the optimal $\boldsymbol{\alpha}^*$. For a support vector $(\mathbf{z}_s, y_s)$ with $\alpha_s^* > 0$, define $b^* = y_s - \sum_{\alpha_n^*>0} y_n\alpha_n^*\mathbf{z}_n^{\mathrm{T}}\mathbf{z}_s$. Then, the final hypothesis from (8.25) is

$$g(\mathbf{x}) = \text{sign}\left(\sum_{n=1}^{N} y_n\alpha_n^*\mathbf{z}_n^{\mathrm{T}}\mathbf{z} + b^*\right),$$

where $\mathbf{z} = \Phi(\mathbf{x})$. In the entire dual formulation, $\mathbf{z}_n$ and $\mathbf{z}$ only appear as inner products. If we use the kernel trick to replace every inner product with the kernel function, then the *entire* process of solving the dual to getting the final hypothesis will be in terms of the kernel. We need never visit the $\mathcal{Z}$ space to explicitly construct $\mathbf{z}_n$ or $\mathbf{z}$. The algorithm box below summarizes these steps.

Hard-Margin SVM with Kernel

1: Construct $\mathrm{Q_D}$ from the kernel K, and $\mathrm{A_D}$:

$$\mathrm{Q_D} = \begin{bmatrix} y_1y_1\mathrm{K}_{11} & \cdots & y_1y_N\mathrm{K}_{1N} \\ y_2y_1\mathrm{K}_{21} & \cdots & y_2y_N\mathrm{K}_{2N} \\ \vdots & \vdots & \vdots \\ y_Ny_1\mathrm{K}_{N1} & \cdots & y_Ny_N\mathrm{K}_{NN} \end{bmatrix} \quad \text{and} \quad \mathrm{A_D} = \begin{bmatrix} \mathbf{y}^{\mathrm{T}} \\ -\mathbf{y}^{\mathrm{T}} \\ \mathrm{I}_{N\times N} \end{bmatrix},$$

where $\mathrm{K}_{mn} = K(\mathbf{x}_m, \mathbf{x}_n)$. (K is called the *Gram* matrix.)

2: Use a QP-solver to optimize the dual problem:

$$\boldsymbol{\alpha}^* \leftarrow \mathsf{QP}(\mathrm{Q_D}, -\mathbf{1}_N, \mathrm{A_D}, \mathbf{0}_{N+2}).$$

3: Let s be any support vector for which $\alpha_s^* > 0$. Compute

$$b^* = y_s - \sum_{\alpha_n^*>0} y_n\alpha_n^* K(\mathbf{x}_n, \mathbf{x}_s).$$

4: Return the final hypothesis

$$g(\mathbf{x}) = \text{sign}\left(\sum_{\alpha_n^*>0} y_n\alpha_n^* K(\mathbf{x}_n, \mathbf{x}) + b^*\right).$$

In the algorithm, the dimension of the $\mathcal{Z}$ space has disappeared, and the running time depends on the efficiency of the kernel, and not $\tilde{d}$. For our polynomial kernel this means the efficiency is determined by d.

An efficient kernel function relies on a carefully constructed *specific* transform to allow fast computation. If we consider another 2nd-order transform

$$\Phi(\mathbf{x}) \quad = \quad (3, 1 \cdot x_1, 4 \cdot x_2, \ldots, 1 \cdot x_d, 5 \cdot x_1 x_1, 9 \cdot x_1 x_2, \ldots, 2 \cdot x_d x_d)$$

that multiplies each component of the original transform by some arbitrary coefficient, it would be difficult to derive an efficient kernel, even though our coefficients are related to the magic number π ☺. However, one can have certain combinations of coefficients that would still make K easy to compute. For instance, set parameters $\gamma > 0$ and $\zeta > 0$, and consider the transform

$$\Phi(\mathbf{x}) \quad = \quad \Big(\zeta \cdot 1, \sqrt{2\gamma\zeta} \cdot x_1, \sqrt{2\gamma\zeta} \cdot x_2, \ldots, \sqrt{2\gamma\zeta} \cdot x_d, \\ \gamma \cdot x_1 x_1, \gamma \cdot x_1 x_2, \ldots, \gamma \cdot x_d x_d\Big),$$

then $K(\mathbf{x}, \mathbf{x}') = (\zeta + \gamma \mathbf{x}^{\mathrm{T}} \mathbf{x}')^2$, which is also easy to compute. The resulting kernel K is often called the second-order polynomial kernel.

At first glance, the freedom to choose (γ, ζ) and still get an efficiently-computable kernel looks useful and also harmless. Multiplying each feature in the $\mathcal{Z}$ space by different constants does not change the expressive power of linear classifiers in the $\mathcal{Z}$ space. Nevertheless, changing (γ, ζ) changes the *geometry* in the $\mathcal{Z}$ space, which affects distances and hence the *margin* of a hyperplane. Thus, different (γ, ζ) could result in a different optimal hyperplane in the $\mathcal{Z}$ space since the margins of all the hyperplanes have changed, and this may give a different quadratic separator in the $\mathcal{X}$ space. The following figures show what happens on some artificial data when you vary γ with ζ fixed at 1.

We see that the three quadratic curves are different, and so are their support vectors denoted by the squares. It is difficult, even after some 'forbidden' visual snooping ☺, to decide which curve is better. One possibility is to use the E_{cv} bound based on the number of support vectors – but keep in mind that the E_{cv} bound can be rather loose in practice. Other possibilities

include using the other validation tools that we have introduced in Chapter 4 to choose γ or other parameters in the kernel function. The choice of kernels and kernel parameters is quite important for ensuring a good performance of nonlinear SVM. Some simple guidelines for popular kernels will be discussed in Section 8.3.2.

The derivation of the second-order polynomial kernel above can be extended to the popular *degree-Q polynomial kernel*

$$K(\mathbf{x}, \mathbf{x}') = (\zeta + \gamma \mathbf{x}^{\mathrm{T}} \mathbf{x}')^Q ,$$

where $\gamma > 0$, $\zeta > 0$, and $Q \in \mathbb{N}$.[5] Then, with the kernel trick, hard-margin SVM can learn sophisticated polynomial boundaries of different orders by using exactly the same procedure and just plugging in different polynomial kernels. So, we can efficiently use very high-dimensional kernels while at the same time implicitly control the model complexity by maximizing the margin. The side figure shows a 10-th order polynomial found by kernel-SVM with margin 0.1.

Gaussian-RBF Kernel. Another popular kernel is called the Gaussian-RBF kernel,[6] which has the form

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2\right)$$

for some $\gamma > 0$. Let us take $\gamma = 1$ and take $\mathbf{x}$ to be a scalar $x \in \mathbb{R}$ in order to understand the transform Φ implied by the kernel. In this case,

$$\begin{aligned} K(x, x') &= \exp\left(-\|x - x'\|^2\right) \\ &= \exp\left(-(x)^2\right) \cdot \exp(2xx') \cdot \exp\left(-(x')^2\right) \\ &= \exp\left(-(x)^2\right) \cdot \left(\sum_{k=0}^{\infty} \frac{2^k (x)^k (x')^k}{k!}\right) \cdot \exp\left(-(x')^2\right), \end{aligned}$$

which is equivalent to an inner product in a feature space defined by the nonlinear transform

$$\Phi(x) = \exp(-x^2) \cdot \left(1, \sqrt{\frac{2^1}{1!}} x, \sqrt{\frac{2^2}{2!}} x^2, \sqrt{\frac{2^3}{3!}} x^3, \ldots\right).$$

We got this nonlinear transform by splitting each term in the Taylor series of $K(x, x')$ into identical terms involving x and x'. Note that in this case, Φ is

[5] We stick to the notation Q for the order of a polynomial, not to be confused with the matrix Q in quadratic programming.

[6] RBF comes from the radial basis function introduced in Chapter 6. We use the parameter γ in place of the scale parameter $1/r$, which is common in the context of SVM.

an *infinite-dimensional transform*. Thus, a direct computation of $\Phi(x)^{\mathrm{T}}\Phi(x')$ is not possible in this case. Nevertheless, with the kernel trick, hard-margin SVM can find a hyperplane in the infinite dimensional $\mathcal{Z}$ space with model complexity under control if the margin is large enough.

The parameter γ controls the the width of the Gaussian kernel. Different choices for the width γ correspond to different geometries in the infinite-dimensional $\mathcal{Z}$ space, much like how different choices for (γ, ζ) in the polynomial kernel correspond to different geometries in the polynomial $\mathcal{Z}$ space. The following figures show the classification results of three Gaussian-RBF kernels only differing in their choice of γ.

$\exp(-1\|\mathbf{x}-\mathbf{x}'\|^2)$ $\exp(-10\|\mathbf{x}-\mathbf{x}'\|^2)$ $\exp(-100\|\mathbf{x}-\mathbf{x}'\|^2)$

When the Gaussian functions are narrow (large γ), we clearly see that even the protection of a large margin cannot suppress overfitting. However, for a reasonably small γ, the sophisticated boundary discovered by SVM with the Gaussian-RBF kernel looks quite good. Again, this demonstrates that kernels and kernel parameters need to be carefully chosen to get reasonable performance.

Exercise 8.15

Consider two finite-dimensional feature transforms Φ_1 and Φ_2 and their corresponding kernels K_1 and K_2.

(a) Define $\Phi(\mathbf{x}) = (\Phi_1(\mathbf{x}), \Phi_2(\mathbf{x}))$. Express the corresponding kernel of Φ in terms of K_1 and K_2.

(b) Consider the matrix $\Phi_1(\mathbf{x})\Phi_2(\mathbf{x})^{\mathrm{T}}$ and let $\Phi(\mathbf{x})$ be the vector representation of the matrix (say, by concatenating all the rows). Express the corresponding kernel of Φ in terms of K_1 and K_2.

(c) Hence, show that if K_1 and K_2 are kernels, then so are $K_1 + K_2$ and K_1K_2.

The results above can be used to construct the general polynomial kernels and (when extended to the infinite-dimensional transforms) to construct the general Gaussian-RBF kernels.

8.3.2 Choice of Kernels

Three kernels are popular in practice: linear, polynomial and Gaussian-RBF. The linear kernel, which corresponds to a special polynomial kernel with $Q = 1$, $\gamma = 1$, and $\zeta = 0$, corresponds to the identity transform. Solving the SVM dual problem (8.22) with the linear kernel is equivalent to solving the linear hard-margin SVM (8.4). Many special SVM packages utilize the equivalence to find the optimal $(b^*, \mathbf{w}^*)$ or $\boldsymbol{\alpha}^*$ more efficiently. One particular advantage of the linear hard-margin SVM is that the value of w_i^* can carry some explanation on how the prediction $g(\mathbf{x})$ is made, just like our familiar perceptron. One particular disadvantage of linear hard-margin SVM is the inability to produce a sophisticated boundary, which may be needed if the data is not linearly separable.

The polynomial kernel provides two controls of complexity: an explicit control from the degree Q and an implicit control from the large-margin concept. The kernel can perform well with a suitable choice of (γ, ζ) and Q. Nevertheless, choosing a good combination of three parameters is not an easy task. In addition, when Q is large, the polynomial kernel evaluates to a value with either very big or very small magnitude. The large range of values introduces numerical difficulty when solving the dual problem. Thus, the polynomial kernel is typically used only with degree $Q \leq 10$, and even then, only when $\zeta + \gamma \mathbf{x}^{\mathrm{T}} \mathbf{x}'$ are scaled to reasonable values by appropriately choosing (ζ, γ).

The Gaussian-RBF kernel can lead to a sophisticated boundary for SVM while controlling the model complexity using the large-margin concept. One only needs to specify one parameter, the width γ, and its numerical range is universally chosen in the interval $[0, 1]$. This often makes it preferable to the polynomial and linear kernels. On the down side, because the corresponding transform of the Gaussian-RBF kernel is an infinite-dimensional one, the resulting hypothesis can only be expressed by the support vectors rather than the actual hyperplane, making it difficult to interpret the prediction $g(\mathbf{x})$. Interestingly, when coupling SVM with the Gaussian-RBF kernel, the hypothesis contains a linear combination of Gaussian functions centered at $\mathbf{x}_n$, which can be viewed as a special case of the RBF Network introduced in e-Chapter 6.

In addition to the above three kernels, there are tons of other kernel choices. One can even design new kernels that better suit the learning task at hand. Note, however, that the kernel is defined as a shortcut function for computing inner products. Thus, similar to what's shown in Exercise 8.11, the *Gram* matrix defined by

$$\mathrm{K} = \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & K(\mathbf{x}_1, \mathbf{x}_2) & \dots & K(\mathbf{x}_1, \mathbf{x}_N) \\ K(\mathbf{x}_2, \mathbf{x}_1) & K(\mathbf{x}_2, \mathbf{x}_2) & \dots & K(\mathbf{x}_2, \mathbf{x}_N) \\ \dots & \dots & \dots & \dots \\ K(\mathbf{x}_N, \mathbf{x}_1) & K(\mathbf{x}_N, \mathbf{x}_2) & \dots & K(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$$

must be positive semi-definite. It turns out that this condition is not only necessary but also sufficient for K to be a valid kernel function that corresponds

(a) linear classifier (b) Gaussian-RBF kernel

Figure 8.9: For (a) a noisy data set that linear classifier appears to work quite well, (b) using the Gaussian-RBF kernel with the hard-margin SVM leads to overfitting.

to the inner product in some nonlinear $\mathcal{Z}$ space. This requirement is formally known as Mercer's condition.

$K(\mathbf{x}, \mathbf{x}')$ is a valid kernel function if and only if the kernel matrix K is always symmetric PSD for any given $\{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$.

The condition is often used to rule out invalid kernel functions. On the other hand, proving a kernel function to be a valid one is a non-trivial task, even when using Mercer's condition.

8.4 Soft-margin SVM

The hard-margin SVM assumes that the data is separable in the $\mathcal{Z}$ space. When we transform $\mathbf{x}$ to a high-dimensional $\mathcal{Z}$ space, or an infinite-dimensional one using the Gaussian-RBF kernel for instance, we can easily overfit the data.[7] Figure 8.9(b) depicts this situation. The hard-margin SVM coupled with the Gaussian-RBF kernel insists on fitting the two 'outliers' that are misclassified by the linear classifier, and results in an unnecessarily complicated decision boundary that should be ringing your 'overfitting alarm bells'.

For the data in Figure 8.9(b), we should use a simple linear hyperplane rather than the complex Gaussian-RBF separator. This means that we will need to accept some errors, and the hard-margin SVM cannot accommodate

[7] It can be shown that the Gaussian-RBF kernel can separate any data set (with no repeated points $\mathbf{x}_n$).

that since it is designed for perfect classification of separable data. One remedy is to consider a 'soft' formulation: try to get a large-margin hyperplane, but allow small violation of the margins or even some classification errors. As we will see, this formulation controls the SVM model's sensitivity to outliers.

The most common formulation of the (linear) *soft-margin SVM* is as follows. Introduce an 'amount' of margin violation $\xi_n \geq 0$ for each data point $(\mathbf{x}_n, y_n)$ and require that

$$y_n \left(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b\right) \geq 1 - \xi_n.$$

According to our definition of separation in (8.2), $(\mathbf{x}_n, y_n)$ is separated if $y_n \left(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b\right) \geq 1$, so ξ_n captures by how much $(\mathbf{x}_n, y_n)$ fails to be separated. In terms of margin, recall that if $y_n \left(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b\right) = 1$ in the hard-margin SVM, then the data point is on the boundary of the margin. So, ξ_n captures how far into the margin the data point can go. The margin is not hard anymore, it is 'soft', allowing a data point to penetrate it. Note that if a point $(\mathbf{x}_n, y_n)$ satisfies $y_n \left(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b\right) > 1$, then the margin is not violated and the corresponding ξ_n is defined to be zero. Ideally, we would like the total sum of margin violations to be small, so we modify the hard-margin SVM to the soft-margin SVM by allowing margin violations but adding a penalty term to discourage large violations. The result is the soft-margin optimization problem:

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + C\sum_{n=1}^{N}\xi_n \\ \text{subject to} \quad & y_n \left(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b\right) \geq 1 - \xi_n \text{ for } n = 1, 2, \ldots, N; \\ & \xi_n \geq 0 \text{ for } n = 1, 2, \ldots, N. \end{aligned} \tag{8.30}$$

We solve this optimization problem to obtain $(b, \mathbf{w})$. Compared with (8.4), the new terms are highlighted in red. We get a large margin by minimizing the term $\frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$ in the objective function. We get small violations by minimizing the term $\sum_{n=1}^{N}\xi_n$. By minimizing the sum of these two terms, we get a compromise between our two goals, and that compromise will favor one or the other of our goals (large margin versus small margin violations) depending on the user-defined penalty parameter denoted by C. When C is large, it means we care more about violating the margin, which gets us closer to the hard-margin SVM. When C is small, on the other hand, we care less about violating the margin. By choosing C appropriately, we get a large-margin hyperplane (small effective d_{VC}) with a small amount of margin violations.

The new optimization problem in (8.30) looks more complex than (8.4). Don't panic. We can solve this problem using quadratic programming, just as we did with the hard-margin SVM (see Exercise 8.16 for the explicit solution). Figure 8.10 shows the result of solving (8.30) using the data from Figure 8.6(a). When C is small, we obtain a classifier with very large margin but with many cases of margin violations (some cross the boundary and some don't); when

(a) $C = 1$ (b) $C = 500$

Figure 8.10: The linear soft-margin optimal hyperplane algorithm from Exercise 8.16 on the non-separable data in Figure 8.6(a).

C is large, we obtain a classifier with smaller margin but with less margin violation.

Exercise 8.16

Show that the optimization problem in (8.30) is a QP-problem.

(a) Show that the optimization variable is $\mathbf{u} = \begin{bmatrix} b \\ \mathbf{w} \\ \boldsymbol{\xi} \end{bmatrix}$, where $\boldsymbol{\xi} = \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_N \end{bmatrix}$.

(b) Show that $\mathbf{u}^* \leftarrow \mathsf{QP}(\mathrm{Q}, \mathbf{p}, \mathrm{A}, \mathbf{c})$, where

$$\mathrm{Q} = \begin{bmatrix} 0 & \mathbf{0}_d^{\mathrm{T}} & \mathbf{0}_N^{\mathrm{T}} \\ \mathbf{0}_d & \mathrm{I}_d & \mathbf{0}_{d\times N} \\ \mathbf{0}_N & \mathbf{0}_{N\times d} & \mathbf{0}_{N\times N} \end{bmatrix}, \ \mathbf{p} = \begin{bmatrix} \mathbf{0}_{d+1} \\ C\cdot\mathbf{1}_N \end{bmatrix}, \ \mathrm{A} = \begin{bmatrix} \mathrm{YX} & \mathrm{I}_N \\ \mathbf{0}_{N\times(d+1)} & \mathrm{I}_N \end{bmatrix}, \ \mathbf{c} = \begin{bmatrix} \mathbf{1}_N \\ \mathbf{0}_N \end{bmatrix},$$

and YX is the signed data matrix from Exercise 8.4.

(c) How do you recover b^*, $\mathbf{w}^*$ and $\boldsymbol{\xi}^*$ from $\mathbf{u}^*$?

(d) How do you determine which data points violate the margin, which data points are on the edge of the margin and which data points are correctly separated and outside the margin?

Similar to the hard-margin optimization problem (8.4), the soft-margin version (8.30) is a convex QP-problem. The corresponding dual problem can be derived using the same technique introduced in Section 8.2. We will just provide the very first step here, and let you finish the rest. For the soft-margin

SVM (8.30), the Lagrange function is

$$\begin{aligned} & \mathcal{L}(b, \mathbf{w}, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \\ = \; & \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + C\sum_{n=1}^{N}\xi_n + \sum_{n=1}^{N}\alpha_n(1 - \xi_n - y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)) - \sum_{n=1}^{N}\beta_n\xi_n, \end{aligned}$$

where $\alpha_n \geq 0$ are the Lagrange multipliers for $y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) \geq 1 - \xi_n$, and $\beta_n \geq 0$ are the Lagrange multipliers for $\xi_n \geq 0$. Then, the KKT condition tells us that at the optimal solution, $\partial\mathcal{L}/\partial\xi_n$ has to be 0. This means

$$C - \alpha_n - \beta_n = 0.$$

That is, α_n and β_n sum to C. We can then replace all β_n by $C - \alpha_n$ without loss of optimality. If we do so, the Lagrange dual problem simplifies to

$$\max_{\substack{\boldsymbol{\alpha}\geq\mathbf{0},\boldsymbol{\beta}\geq\mathbf{0}, \\ \alpha_n+\beta_n=C}} \; \min_{b,\mathbf{w},\boldsymbol{\xi}} \quad \mathcal{L}(b, \mathbf{w}, \boldsymbol{\xi}, \boldsymbol{\alpha}),$$

where

$$\begin{aligned} \mathcal{L} &= \tfrac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + C\sum_{n=1}^{N}\xi_n + \sum_{n=1}^{N}\alpha_n(1 - \xi_n - y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)) - \sum_{n=1}^{N}(C - \alpha_n)\xi_n \\ &= \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + \sum_{n=1}^{N}\alpha_n(1 - y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b)). \end{aligned}$$

Does the simplified objective function now look familiar? It is just the objective function that led us to the dual of the hard-margin SVM! ☺ We trust that you can then go through all the steps as in Section 8.2 to get the whole dual. When expressed in matrix form, the dual problem of (8.30) is

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^{\mathrm{T}}\mathrm{Q}_{\mathrm{D}}\boldsymbol{\alpha} - \mathbf{1}^{\mathrm{T}}\boldsymbol{\alpha} \\ \text{subject to} \quad & \mathbf{y}^{\mathrm{T}}\boldsymbol{\alpha} = 0; . \\ & \mathbf{0} \leq \boldsymbol{\alpha} \leq C \cdot \mathbf{1}. \end{aligned}$$

Interestingly, compared with (8.22), the only change of the dual problem is that each α_n is now upper-bounded by C, the penalty rate, instead of ∞. The formulation can again be solved by some general or specifically tailored quadratic programming packages.

For the soft-margin SVM, we get a similar expression for the final hypothesis in terms of the optimal dual solution, like that of the hard-margin SVM.

$$g(\mathbf{x}) = \operatorname{sign}\left(\sum_{\alpha_n^*>0} y_n\alpha_n^*\mathbf{x}_n^{\mathrm{T}}\mathbf{x} + b^*\right).$$

The kernel trick is still applicable as long as we can calculate the optimal b^* efficiently. Getting the optimal b^* from the optimal $\boldsymbol{\alpha}^*$, however, is slightly more complicated in the soft-margin case. The KKT conditions state that

$$\begin{aligned} \alpha_n^* \cdot \left(y_n(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n + b^*) - 1 + \xi_n^* \right) &= 0. \\ \beta_n^* \cdot \xi_n^* = (C - \alpha_n^*) \cdot \xi_n^* &= 0. \end{aligned}$$

If $\alpha_n^* > 0$, then $\left(y_n(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n + b^*) - 1 + \xi_n^* \right) = 0$ and hence

$$y_n(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n + b^*) = 1 - \xi_n^* \leq 1.$$

On the other hand, if $\alpha_n^* < C$, then $\xi_n^* = 0$ and hence

$$y_n(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n + b^*) \geq 1.$$

The two inequalities gives a range for the optimal b^*. When there is a support vector with $0 < \alpha_n^* < C$, we see that the inequalities can be combined to an equality $y_n(\mathbf{w}^{*\mathrm{T}}\mathbf{x}_n + b^*) = 1$, which can be used to pin down the optimal b^* as we did for the hard-margin SVM. In other cases, there are many choices of b^* and you can freely pick any.

The support vectors with $0 < \alpha_n^* < C$ are called the free support vectors, which are guaranteed to be on the boundary of the fat-hyperplane and hence also called margin support vectors. The support vectors with $\alpha_n^* = C$, on the other hand, are called the bounded support vectors (also called non-margin support vectors). They can be on the fat boundary, slightly violating the boundary but still correctly predicted, or seriously violating the boundary and erroneously predicted.

For separable data (in the Φ-transformed space), there exists some C' such that whenever $C \geq C'$, the soft-margin SVM produces exactly the same solution as the hard-margin SVM. Thus, the hard-margin SVM can be viewed as a special case of the soft-margin one, illustrated below.

Let's take a closer look at the parameter C in the soft-margin SVM formulation (8.30). If C is large, it means that we do want all violations (errors) ξ_n to be as small as possible with the possible trade-off being a smaller margin (higher complexity). If C is small, we will tolerate some amounts of errors, while possibly getting a less complicated hypothesis with large margin. Does the trade-off sound familiar? We have encountered such a trade-off in Chapter 4 when we studied regularization. Let

$$E_{\text{SVM}}(b, \mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \max(1 - y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b), 0).$$

The n-th term in $E_{\text{SVM}}(b, \mathbf{w})$ evaluates to 0 if there is no violation from the n-th example, so $y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) \geq 1$; otherwise, the nth term is the amount of violation for the corresponding data point. Therefore, the objective that we minimize in soft-margin SVM (8.30) can be re-written as the following optimization problem

$$\min_{b,\mathbf{w}} \lambda \mathbf{w}^{\mathrm{T}}\mathbf{w} + E_{\text{SVM}}(b, \mathbf{w}),$$

subject to the constraints, and where $\lambda = 1/2CN$. In other words, soft-margin SVM can be viewed as a special case of regularized classification with $E_{\text{SVM}}(b, \mathbf{w})$ as a surrogate for the in-sample error and $\frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$ (without b) as the regularizer. The $E_{\text{SVM}}(b, \mathbf{w})$ term is an upper bound on the classification in-sample error E_{in}, while the regularizer term comes from the large-margin concept and controls the effective model complexity.

Exercise 8.17

Show that $E_{\text{SVM}}(b, \mathbf{w})$ is an upper bound on the $E_{\text{in}}(b, \mathbf{w})$, where E_{in} is the classification 0/1 error.

In summary, soft-margin SVM can:

1. Deliver a large-margin hyperplane, and in so doing it can control the effective model complexity.
2. Deal with high- or infinite-dimensional transforms using the kernel trick,
3. Express the final hypothesis $g(\mathbf{x})$ using only a few support vectors, their corresponding Lagrange multipliers, and the kernel.
4. Control the sensitivity to outliers and regularize the solution through setting C appropriately.

When the regularization parameter C and the kernel are chosen properly, the soft-margin SVM is often observed to enjoy a low E_{out} with the useful properties above. These properties make the soft-margin SVM (*the* SVM for short) one of the most useful classification models and often the first choice in learning from data. It is a robust linear model with advanced nonlinear transform capability when used with a kernel.

8.5 Problems

Problem 8.1 Consider a data set with two data points $\mathbf{x}_\pm \in \mathbb{R}^d$ having class ± 1 respectively. Manually solve (8.4) by explicitly minimizing $\|\mathbf{w}\|^2$ subject to the two separation constraints.

Compute the optimal (maximum margin) hyperplane $(b^*, \mathbf{w}^*)$ and its margin. Compare with your solution to Exercise 8.1.

Problem 8.2 Consider a data set with three data points in $\mathbb{R}^2$:

$$\mathrm{X} = \begin{bmatrix} 0 & 0 \\ 0 & -1 \\ -2 & 0 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ +1 \end{bmatrix}$$

Manually solve (8.4) to get the optimal hyperplane $(b^*, \mathbf{w}^*)$ and its margin.

Problem 8.3 Manually solve the dual optimization from Example 8.8 to obtain the same $\boldsymbol{\alpha}^*$ that was obtained in the text using a QP-solver. Use the following steps.

(a) Show that the dual optimization problem is to minimize

$$\mathcal{L}(\boldsymbol{\alpha}) = 4\alpha_2^2 + 2\alpha_3^2 + \tfrac{9}{2}\alpha_4^2 - 4\alpha_2\alpha_3 - 6\alpha_2\alpha_4 + 6\alpha_3\alpha_4 - \alpha_1 - \alpha_2 - \alpha_3 - \alpha_4,$$

subject to the constraints

$$\begin{aligned} \alpha_1 + \alpha_2 &= \alpha_3 + \alpha_4; \\ \alpha_1, \alpha_2, \alpha_3, \alpha_4 &\geq 0. \end{aligned}$$

(b) Use the equality constraint to replace α_1 in $\mathcal{L}(\boldsymbol{\alpha})$ to get

$$\mathcal{L}(\boldsymbol{\alpha}) = 4\alpha_2^2 + 2\alpha_3^2 + \tfrac{9}{2}\alpha_4^2 - 4\alpha_2\alpha_3 - 6\alpha_2\alpha_4 + 6\alpha_3\alpha_4 - 2\alpha_3 - 2\alpha_4.$$

(c) Fix $\alpha_3, \alpha_4 \geq 0$ and minimize $\mathcal{L}(\boldsymbol{\alpha})$ in (b) with respect to α_2 to show that

$$\alpha_2 = \frac{\alpha_3}{2} + \frac{3\alpha_4}{4} \qquad \text{and} \qquad \alpha_1 = \alpha_3 + \alpha_4 - \alpha_2 = \frac{\alpha_3}{2} + \frac{\alpha_4}{4}.$$

Are these valid solutions for α_1, α_2?

(d) Use the expressions in (c) to reduce the problem to minimizing

$$\mathcal{L}(\boldsymbol{\alpha}) = \alpha_3^2 + \tfrac{9}{4}\alpha_4^2 + 3\alpha_3\alpha_4 - 2\alpha_3 - 2\alpha_4,$$

subject to $\alpha_3, \alpha_4 \geq 0$. Show that the minimum is attained when $\alpha_3 = 1$ and $\alpha_4 = 0$. What are α_1, α_2?

It's a relief to have QP-solvers for solving such problems in the general case!

Problem 8.4 Set up the dual problem for the toy data set in Exercise 8.2. Then, solve the dual problem and compute $\boldsymbol{\alpha}^*$, the optimal Lagrange multipliers.

Problem 8.5 [Bias and Variance of the Optimal Hyperplane] In this problem, you are to investigate the bias and variance of the optimal hyperplane in a simple setting. The input is $(x_1, x_2) \in [-1, 1]^2$ and the target function is $f(\mathbf{x}) = \text{sign}(x_2)$.

The hypothesis set $\mathcal{H}$ contains horizontal linear separators $h(\mathbf{x}) = \text{sign}(x_2 - a)$, where $-1 \leq a \leq 1$. Consider two algorithms:

Random: Pick a random separator from $\mathcal{H}$.

SVM: Pick the maximum margin separator from $\mathcal{H}$.

(a) Generate 3 data point uniformly in the upper half of the input-space and 3 data points in the lower half, and obtain g_{Random} and g_{SVM}.

(b) Create a plot of your data, and your two hypotheses.

(c) Repeat part (a) for a million data sets to obtain one million Random and SVM hypotheses.

(d) Give a histogram of the values of a_{Random} resulting from the random algorithm and another histogram of a_{SVM} resulting from the optimal separators. Compare the two histograms and explain the differences.

(e) Estimate the bias and var for the two algorithms. Explain your findings, in particular which algorithm is better for this toy problem.

Problem 8.6 Show that $\sum_{n=1}^{N} \|\mathbf{x}_n - \boldsymbol{\mu}\|^2$ is minimized at $\boldsymbol{\mu} = \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n$.

Problem 8.7 For any $\mathbf{x}_1, \ldots, \mathbf{x}_N$ with $\|\mathbf{x}_n\| \leq R$ and N even, show that there exists a balanced dichotomy $y_1, \ldots, y_n$ that satisfies

$$\sum_{n=1}^{N} y_n = 0, \text{ and } \qquad \left\| \sum_{n=1}^{N} y_n \mathbf{x}_n \right\| \leq \frac{NR}{\sqrt{N-1}}.$$

(This is the geometric lemma that is need to bound the VC-dimension of ρ-fat hyperplanes by $\lceil R^2/\rho^2 \rceil + 1$.) The following steps are a guide for the proof.

Suppose you randomly select $N/2$ of the labels $y_1, \ldots, y_N$ to be $+1$, the others being -1. By construction, $\sum_{n=1}^{N} y_n = 0$.

(a) Show $\left\| \sum_{n=1}^{N} y_n \mathbf{x}_n \right\|^2 = \sum_{n=1}^{N} \sum_{m=1}^{N} y_n y_m \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_m$.

(b) When $n = m$, what is $y_n y_m$? Show that $\mathbb{P}[y_n y_m = 1] = (\frac{N}{2} - 1)/(N-1)$ when $n \neq m$. Hence show that

$$\mathbb{E}[y_n y_m] = \begin{cases} 1 & m = n; \\ -\frac{1}{N-1} & m \neq n. \end{cases}$$

(c) Show that

$$\mathbb{E}\left[\left\|\sum_{n=1}^{N} y_n \mathbf{x}_n\right\|^2\right] = \frac{N}{N-1}\sum_{n=1}^{N} \|\mathbf{x}_n - \bar{\mathbf{x}}\|^2,$$

where the average vector $\bar{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x}_n$. *[Hint: Use linearity of expectation in (a), and consider the cases $m = n$ and $m \neq n$ separately.]*

(d) Show that $\sum_{n=1}^{N} \|\mathbf{x}_n - \bar{\mathbf{x}}\|^2 \leq \sum_{n=1}^{N} \|\mathbf{x}_n\|^2 \leq NR^2$ *[Hint: Problem 8.6.]*

(e) Conclude that

$$\mathbb{E}\left[\left\|\sum_{n=1}^{N} y_n \mathbf{x}_n\right\|^2\right] \leq \frac{N^2 R^2}{N-1},$$

and hence that

$$\mathbb{P}\left[\left\|\sum_{n=1}^{N} y_n \mathbf{x}_n\right\| \leq \frac{NR}{\sqrt{N-1}}\right] > 0.$$

This means for some choice of y_n, $\left\|\sum_{n=1}^{N} y_n \mathbf{x}_n\right\| \leq NR/\sqrt{N-1}$

This proof is called a probabilistic existence proof: if some random process can generate an object with *positive probability*, then that object must exist. Note that you prove existence of the required dichotomy without actually constructing it. In this case, the easiest way to construct a desired dichotomy is to randomly generate the balanced dichotomies until you have one that works.

Problem 8.8 We showed that if N points in the ball of radius R are shattered by hyperplanes with margin ρ, then $N \leq R^2/\rho^2 + 1$ when N is even. Now consider N odd, and $\mathbf{x}_1, \ldots, \mathbf{x}_N$ with $\|\mathbf{x}_n\| \leq R$ shattered by hyperplanes with margin ρ. Recall that $(\mathbf{w}, b)$ implements $y_1, \ldots, y_N$ with margin ρ if

$$\rho\|\mathbf{w}\| \leq y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b), \qquad \text{for } n = 1, \ldots, N. \tag{8.31}$$

Show that for $N = 2k + 1$ (odd), $N \leq R^2/\rho^2 + \frac{1}{N} + 1$ as follows:

Consider random labelings $y_1, \ldots, y_N$ of the N points in which k of the labels are $+1$ and $k+1$ are -1. Define $\ell_n = \frac{1}{k}$ if $y_n = +1$ and $\ell_n = \frac{1}{k+1}$ if $y_n = -1$.

(a) For any labeling with k labels being $+1$, show, by summing (8.31) and

using the Cauchy-Schwarz inequality, that

$$2\rho \leq \left\| \sum_{n=1}^{N} \ell_n y_n \mathbf{x}_n \right\|.$$

(b) Show that there exists a labeling, with k labels being $+1$, for which

$$\left\| \sum_{n=1}^{N} \ell_n y_n \mathbf{x}_n \right\| \leq \frac{2NR}{(N-1)\sqrt{N+1}}$$

.

(i) Show $\left\| \sum_{n=1}^{N} \ell_n y_n \mathbf{x}_n \right\|^2 = \sum_{n=1}^{N} \sum_{m=1}^{N} \ell_n \ell_m y_n y_m \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_m.$

(ii) For $m = n$, show $\mathbb{E}[\ell_n \ell_m y_n y_m] = \dfrac{1}{k(k+1)}$.

(iii) For $m \neq n$, show $\mathbb{E}[\ell_n \ell_m y_n y_m] = -\dfrac{1}{(N-1)k(k+1)}$.

[Hint: $\mathbb{P}[\ell_n \ell_m y_n y_m = 1/k^2] = k(k-1)/N(N-1)$.*]*

(iv) Show $\mathbb{E}\left[\left\| \sum_{n=1}^{N} \ell_n y_n \mathbf{x}_n \right\|^2 \right] = \dfrac{N}{(N-1)k(k+1)} \sum_{n=1}^{N} \|\mathbf{x}_n - \bar{\mathbf{x}}\|^2.$

(v) Use Problem 8.6 to conclude the proof as in Problem 8.7.

(c) Use (a) and (b) to show that $N \leq \dfrac{R^2}{\rho^2} + \dfrac{1}{N} + 1$.

Problem 8.9 Prove that for the separable case, if you remove a data point that is not a support vector, then the maximum margin classifier does not change. You may use the following steps are a guide. Let g be the maximum margin classifier for all the data, and g^- the maximum margin classifier after removal of a data point that is not a support vector.

(a) Show that g is a separator for $\mathcal{D}^-$, the data minus the non-support vector.

(b) Show that the maximum margin classifier is unique.

(c) Show that if g^- has larger margin than g on $\mathcal{D}^-$, then it also has larger margin on $\mathcal{D}$, a contradiction. Hence, conclude that g is the maximum margin separator for $\mathcal{D}^-$.

Problem 8.10 An essential support vector is one whose removal from the data set changes the maximum margin separator. For the separable case, show that there are at most $d+1$ essential support vectors. Hence, show that for the separable case,

$$E_{\mathrm{cv}} \leq \frac{d+1}{N}.$$

Problem 8.11 Consider the version of the PLA that uses the misclassified data point $\mathbf{x}_n$ with lowest index n for the weight update. Assume the data is separable and given in some fixed but arbitrary order, and when you remove a data point, you do not alter this order. In this problem, prove that

$$E_{\text{cv}}(\text{PLA}) \leq \frac{R^2}{N\rho^2},$$

where ρ is the margin (half the width) of the maximum margin separating hyperplane that would (for example) be returned by the SVM. The following steps are a guide for the proof.

(a) Use the result in Problem 1.3 to show that the number of updates T that the PLA makes is at most

$$T \leq \frac{R^2}{\rho^2}.$$

(b) Argue that this means that PLA only 'visits' at most R^2/ρ^2 different points during the course of its iterations.

(c) Argue that after leaving out any point $(\mathbf{x}_n, y_n)$ that is not 'visited', PLA will return the same classifier.

(d) What is the leave-one-out error e_n for these points that were not visited? Hence, prove the desired bound on $E_{\text{cv}}(\text{PLA})$.

Problem 8.12 Show that optimal solution for soft-margin optimal hyperplane (solving optimization problem (8.30)) with $C \to \infty$ will be the same solution that was developed using linear programming in Problem 3.6(c).

Problem 8.13 The data for Figure 8.6(b) are given below:

$y_n = +1$	$y_n = -1$
	$(0.491, 0.920)$
$(-0.494, 0.363)$	$(-0.892, -0.946)$
$(-0.311, -0.101)$	$(-0.721, -0.710)$
$(-0.0064, 0.374)$	$(0.519, -0.715)$
$(-0.0089, -0.173)$	$(-0.775, 0.551)$
$(0.0014, 0.138)$	$(-0.646, 0.773)$
$(-0.189, 0.718)$	$(-0.803, 0.878)$
$(0.085, 0.32208)$	$(0.944, 0.801)$
$(0.171, -0.302)$	$(0.724, -0.795)$
$(0.142, 0.568)$	$(-0.748, -0.853)$
	$(-0.635, -0.905)$

Use the data on the left with the 2nd and 3rd order polynomial transforms Φ_2, Φ_3 and the pseudo-inverse algorithm for linear regression from Chapter 3 to get weights $\tilde{\mathbf{w}}$ for your final final hypothesis in $\mathcal{Z}$-space. The final hypothesis in $\mathcal{X}$-space is:

$$g(\mathbf{x}) = \text{sign}(\tilde{\mathbf{w}}^{\mathrm{T}}\Phi(\mathbf{x}) + \tilde{b}).$$

(a) Plot the classification regions for your final hypothesis in $\mathcal{X}$-space. Your results should look something like:

(b) Which of fits in part (a) appears to have overfitted?

(c) Use the pseudo-inverse algorithm with regularization parameter $\lambda = 1$ to address the overfitting you identified in part (c). Give a plot of the resulting classifier.

Problem 8.14 The kernel trick can be used with any model as long as fitting the data and the final hypothesis only require the computation of dot-products in the $\mathcal{Z}$-space. Suppose you have a kernel K, so

$$\Phi(\mathbf{x})^{\mathrm{T}}\Phi(\mathbf{x}') = K(\mathbf{x}, \mathbf{x}').$$

Let Z be the data in the $\mathcal{Z}$-space. The pseudo-inverse algorithm for regularized regression computes optimal weights $\tilde{\mathbf{w}}^*$ (in the $\mathcal{Z}$-space) that minimize

$$E_{\text{aug}}(\tilde{\mathbf{w}}) = \|\mathrm{Z}\tilde{\mathbf{w}} - \mathbf{y}\|^2 + \lambda\tilde{\mathbf{w}}^{\mathrm{T}}\tilde{\mathbf{w}}.$$

The final hypothesis is $g(\mathbf{x}) = \text{sign}(\tilde{\mathbf{w}}^{\mathrm{T}}\Phi(\mathbf{x}))$. Using the representor theorem, the optimal solution can be written $\tilde{\mathbf{w}}^* = \sum_{n=1}^{N} \beta_n^* \mathbf{z}_n = \mathrm{Z}^{\mathrm{T}}\boldsymbol{\beta}^*$

(a) Show that $\boldsymbol{\beta}^*$ minimizes

$$E(\boldsymbol{\beta}) = \|\mathrm{K}\boldsymbol{\beta} - \mathbf{y}\|^2 + \lambda\boldsymbol{\beta}^{\mathrm{T}}\mathrm{K}\boldsymbol{\beta},$$

where K is the $N \times N$ Kernel-Gram matrix with entries $\mathrm{K}_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$.

(b) Show that K is symmetric.

(c) Show that the solution to the minimization problem in part (a) is:

$$\boldsymbol{\beta}^* = (\mathrm{K} + \lambda\mathrm{I})^{-1}\mathbf{y}.$$

Can $\boldsymbol{\beta}^*$ be computed without ever 'visiting' the $\mathcal{Z}$-space?

(d) Show that the final hypothesis is

$$g(\mathbf{x}) = \text{sign}\left(\sum_{n=1}^{N} \beta_n^* K(\mathbf{x}_n, \mathbf{x})\right).$$

Problem 8.15 **Structural Risk Minimization (SRM).** SRM is a useful framework for model selection. A *structure* is a nested sequence of hypothesis sets:

Suppose we use in-sample error minimization as our learning algorithm in each $\mathcal{H}_m$, so $g_m = \underset{h \in \mathcal{H}_m}{\text{argmin}}\, E_{\text{in}}(h)$, and select $g^* = \underset{g_m}{\text{argmin}}\, E_{\text{in}}(g_m) + \Omega(\mathcal{H}_m)$.

(a) Show that the in-sample error $E_{\text{in}}(g_m)$ is non-increasing in m. What about the penalty $\Omega(\mathcal{H}_m)$? How do you expect the VC-bound to behave with m.

(b) Assume that $g^* \in \mathcal{H}_m$ with *a priori* probability p_m. (In general, the p_m are not known.) Since $\mathcal{H}_m \subset \mathcal{H}_{m+1}$, $p_0 \leq p_1 \leq p_2 \leq \cdots \leq 1$. What components of the learning problem do the p_m's depend on.

(c) Suppose $g^* = g_m \in \mathcal{H}_m$. Show that

$$\mathbb{P}\left[|E_{\text{in}}(g_i) - E_{\text{out}}(g_i)| > \epsilon \mid g^* = g_m\right] \leq \frac{1}{p_i} \cdot 4m_{\mathcal{H}_i}(2N)e^{-\epsilon^2 N/8}.$$

Here, the conditioning is on selecting the function $g_m \in \mathcal{H}_m$. *[Hint: Bound the probability* $\mathbb{P}\left[\max_{g \in \mathcal{H}_i} |E_{\text{in}}(g) - E_{\text{out}}(g)| > \epsilon \mid g^* = g_m\right]$. *Use Bayes theorem to rewrite this as* $\frac{1}{p_i}\mathbb{P}\left[\max_{g \in \mathcal{H}_i} |E_{\text{in}}(g) - E_{\text{out}}(g)| > \epsilon \textbf{ and } g^* = g_m\right]$. *Use the fact that* $\mathbb{P}[A \textbf{ and } B] \leq \mathbb{P}[A]$. *and argue that you can apply the familiar VC-inequality to the resulting expression.]*

You may interpret this result as follows: if you use SRM and end up with g_m, then the generalization bound is a factor $\frac{1}{p_m}$ worse than the bound you would have gotten had you simply started with $\mathcal{H}_m$; that is the price you pay for allowing yourself the possibility to search more than $\mathcal{H}_m$. Typically simpler models occur earlier in the structure, and so the bound will be reasonable if the target function is simple (in which case p_m is large for small m). SRM works well in practice.

Problem 8.16 Which can be posed within the SRM framework: selection among different soft order constraints $\{\mathcal{H}_C\}_{C>0}$ or selecting among different regularization parameters $\{\mathcal{H}_\lambda\}_{\lambda>0}$ where the hypothesis set fixed at $\mathcal{H}$ and the learning algorithm is augmented error minimization with different regularization parameters λ.

Problem 8.17 Suppose we use "SRM" to select among an arbitrary set of models $\mathcal{H}_1, \ldots, \mathcal{H}_M$ with $d_{\text{vc}}(\mathcal{H}_{m+1}) > d_{\text{vc}}(\mathcal{H}_m)$ (as opposed to a structure in which the additional condition $\mathcal{H}_m \subset \mathcal{H}_{m+1}$ holds).

(a) Is it possible for $E_{\text{in}}(\mathcal{H}_m) < E_{\text{in}}(\mathcal{H}_{m+1})$?

(b) Let p_m be the probability that the process leads to a function $g_m \in \mathcal{H}_m$, with $\sum_m p_m = 1$. Give a bound for the generalization error in terms of $d_{\text{vc}}(\mathcal{H}_m)$.

Problem 8.18 Suppose that we can order the hypotheses in a model, $\mathcal{H} = \{h_1, h_2, \ldots\}$. Assume that $d_{\text{vc}}(\mathcal{H})$ is infinite. Define the hypothesis subsets $\mathcal{H}_m = \{h_1, h_2, \ldots, h_m\}$. Suppose you implement a learning algorithm for $\mathcal{H}$ as follows: start with h_1; if $E_{\text{in}}(h_1) \leq \nu$, stop and output h_1; if not try h_2; and so on ...

(a) Suppose that you output h_m, so you have effectively only searched the m hypotheses in $\mathcal{H}_m$. Can you use the VC-bound: (with high probability) $E_{\text{out}}(h_m) \leq \nu + \sqrt{\frac{\ln(2m/\delta)}{2N}}$? If yes, why? If no, why not?

(b) Formulate this process within the SRM framework. *[Hint: the $\mathcal{H}_m$'s form a structure.]*

(c) Can you make *any* generalization conclusion (remember, $d_{\text{vc}}(\mathcal{H}) = \infty$)? If yes, what is the bound on the generalization error, and when do you expect good generalization? If no, why?

Chapter 9

Learning Aides

Our quest has been for a low out-of-sample error. In the context of specific learning models, we have discussed techniques to fit the data in-sample, and ensure good generalization to out of sample. There are, however, additional issues that are likely to arise in any learning scenario, and a few simple enhancements can often yield a drastic improvement. For example: Did you appropriately preprocess the data to take into account arbitrary choices that might have been made during data collection? Have you removed any irrelevant dimensions in the data that are useless for approximating the target function, but can mislead the learning by adding stochastic noise? Are there properties that the target function is known to have, and if so, can these properties help the learning? Have we chosen the best model among those that are available to us? We wrap up our discussion of learning techniques with a few general tools that can help address these questions.

9.1 Input Preprocessing

Exercise 9.1

The Bank of Learning (BoL) gave Mr. Good and Mr. Bad credit cards based on their (Age, Income) input vector.

	Mr. Good	Mr. Bad
(Age in years, Income in thousands of \$)	(47,35)	(22,40)

Mr. Good paid off his credit card bill, but Mr. Bad defaulted. Mr. Unknown who has 'coordinates' (21yrs,\$36K) applies for credit. Should the BoL give him credit, according to the nearest neighbor algorithm? If income is measured in dollars instead of in "K" (thousands of dollars), what is your answer?

One could legitimately debate whether age (maturity) or income (financial resources) should be the determining factor in credit approval. It is, however,

decidedly not recommended for something like a credit approval to hinge on an apparently arbitrary choice made during data collection, such as the denomination by which income was measured. On the contrary, many standard design choices when learning from data intend each dimension to be treated equally (such as using the Euclidean distance metric in similarity methods or using the sum of squared weights as the regularization term in weight decay). Unless there is some explicit reason not to do so, the data should be presented to the learning algorithm with each dimension on an equal footing. To do so, we need to transform the data to a standardized setting so that we can be immune to arbitrary choices made during data collection.

Recall that the data matrix $\mathrm{X} \in \mathbb{R}^{n\times d}$ has, as its rows, the input data vectors $\mathbf{x}_1, \ldots, \mathbf{x}_\mathrm{N}$, where $\mathbf{x}_n \in \mathbb{R}^d$ (not augmented with a 1),

$$\mathrm{X} = \begin{bmatrix} \text{—} & \mathbf{x}_1^\mathrm{T} & \text{—} \\ \text{—} & \mathbf{x}_2^\mathrm{T} & \text{—} \\ & \vdots & \\ \text{—} & \mathbf{x}_\mathrm{N}^\mathrm{T} & \text{—} \end{bmatrix}.$$

The goal of input preprocessing is to transform the data $\mathbf{x}_n \mapsto \mathbf{z}_n$ to obtain the transformed data matrix Z which is standardized in some way. Let $\mathbf{z}_n = \Phi(\mathbf{x}_n)$ be the transformation. It is the data $(\mathbf{z}_n, y_n)$ that is fed into the learning algorithm to produce a learned hypothesis $\tilde{g}(\mathbf{z})$.[1] The final hypothesis g is

$$g(\mathbf{x}) = \tilde{g}(\Phi(\mathbf{x})).$$

Input Centering. Centering is a relatively benign transformation which removes any bias in the inputs by translating the origin. Let $\bar{\mathbf{x}}$ be the in-sample mean vector of the input data, $\bar{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x}_n$; in matrix notation, $\bar{\mathbf{x}} = \frac{1}{N}\mathrm{X}^\mathrm{T}\mathbf{1}$ ($\mathbf{1}$ is the column vector of N 1's). To obtain the transformed vector, simply subtract the mean from each data point,

$$\mathbf{z}_n = \mathbf{x}_n - \bar{\mathbf{x}}.$$

By direct calculation, one can verify that $\mathrm{Z} = \mathrm{X} - \mathbf{1}\bar{\mathbf{x}}^\mathrm{T}$. Hence,

$$\bar{\mathbf{z}} = \tfrac{1}{N}\mathrm{Z}^\mathrm{T}\mathbf{1} = \tfrac{1}{N}\mathrm{X}^\mathrm{T}\mathbf{1} - \tfrac{1}{N}\bar{\mathbf{x}}\mathbf{1}^\mathrm{T}\mathbf{1} = \bar{\mathbf{x}} - \tfrac{1}{N}\bar{\mathbf{x}} \cdot N = \mathbf{0},$$

where we used $\mathbf{1}^\mathrm{T}\mathbf{1} = N$ and the definition of $\bar{\mathbf{x}}$. Thus, the transformed vectors are 'centered' in that they have zero mean, as desired. It is clear that no information is lost by centering (as long as one has retained $\bar{\mathbf{x}}$, one can always recover $\mathbf{x}$ from $\mathbf{z}$). If the data is not centered, we can always center it, so from now on, for simplicity and without loss of generality, we will assume that the input data is centered, and so $\mathrm{X}^\mathrm{T}\mathbf{1} = \mathbf{0}$.

[1] Note that higher-level features or nonlinear transforms can also be used to transform the inputs before input processing.

Exercise 9.2

Define the matrix $\gamma = \mathrm{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^{\mathrm{T}}$. Show that $\mathrm{Z} = \gamma\mathrm{X}$. ($\gamma$ is called the centering operator (see Appendix B.3) which projects onto the space orthogonal to $\mathbf{1}$.)

Input Normalization. Centering alone will not solve the problem that arose in Exercise 9.1. The issue there is one of scale, not bias. Inflating the scale of the income variable exaggerates differences in income when using the standard Euclidean distance, thereby affecting your decision. One solution is to ensure that all the input variables have the same scale. One measure of scale (or spread) is the standard deviation. Since the data is now centered, the in-sample standard deviation σ_i of input variable i is defined by

$$\sigma_i^2 = \frac{1}{N}\sum_{n=1}^{N} x_{ni}^2,$$

where x_{ni} is the ith component of the nth data point. Input normalization transforms the inputs so that each input variable has unit standard deviation (scale). Specifically, the transformation is

$$\mathbf{z}_n = \begin{bmatrix} z_{n1} \\ \vdots \\ z_{nd} \end{bmatrix} = \begin{bmatrix} x_{n1}/\sigma_1 \\ \vdots \\ x_{nd}/\sigma_d \end{bmatrix} = \mathrm{D}\mathbf{x}_n,$$

where D is a diagonal matrix with entries $D_{ii} = 1/\sigma_i$. The scale of all input variables is now 1, since $\tilde{\sigma}_i^2 = 1$ as the following derivation shows ($\tilde{\sigma}_i^2 = 1$ is the variance, or scale, of dimension i in the $\mathcal{Z}$ space).

$$\sigma_i^2(\mathbf{z}) = \frac{1}{N}\sum_{n=1}^{N} z_{ni}^2 = \frac{1}{N}\sum_{n=1}^{N} \frac{x_{ni}^2}{\sigma_i^2} = \frac{1}{\sigma_i^2}\cdot\underbrace{\left(\frac{1}{N}\sum_{n=1}^{N} x_{ni}^2\right)}_{\sigma_i^2} = 1.$$

Exercise 9.3

Consider the data matrix X and the transformed data matrix Z. Show that

$$\mathrm{Z} = \mathrm{XD} \qquad \text{and} \qquad \mathrm{Z}^{\mathrm{T}}\mathrm{Z} = \mathrm{D}\mathrm{X}^{\mathrm{T}}\mathrm{X}\mathrm{D}.$$

Input Whitening Centering deals with bias in the inputs. Normalization deals with scale. Our last concern is correlations. Strongly correlated input variables can have an unexpected impact on the outcome of learning. For example, with regularization, correlated input variables can render a friendly target function unlearnable. The next exercise shows that a simple function

may require excessively large weights to implement if the inputs are correlated. This means it is hard to regularize the learning, which in turn means you become susceptible to noise and overfitting.

Exercise 9.4

Let $\hat{x}_1$ and $\hat{x}_2$ be independent with zero mean and unit variance. You measure inputs $x_1 = \hat{x}_1$ and $x_2 = \sqrt{1-\epsilon^2}\hat{x}_1 + \epsilon\hat{x}_2$.

(a) What are variance(x_1), variance(x_2) and covariance(x_1, x_2)?

(b) Suppose $f(\hat{\mathbf{x}}) = \hat{w}_1\hat{x}_1 + \hat{w}_2\hat{x}_2$ (linear in the independent variables). Show that f is linear in the correlated inputs, $f(\mathbf{x}) = w_1x_1 + w_2x_2$. (Obtain w_1, w_2 as functions of $\hat{w}_1, \hat{w}_2$.)

(c) Consider the 'simple' target function $f(\hat{\mathbf{x}}) = \hat{x}_1 + \hat{x}_2$. If you perform regression with the correlated inputs $\mathbf{x}$ and regularization constraint $w_1^2 + w_2^2 \le C$, what is the maximum amount of regularization you can use (minimum value of C) and still be able to implement the target?

(d) What happens to the minimum C as the correlation increases ($\epsilon \to 0$).

(e) Assuming that there is significant noise in the data, discuss your results in the context of bias and var.

The previous exercise illustrates that if the inputs are correlated, then the weights cannot be independently penalized, as they are in the standard form of weight-decay regularization. If the measured input variables are correlated, then one should transform them to a set that are uncorrelated (at least in-sample). That is the goal of input whitening.[2]

Remember that the data is centered, so the in-sample covariance matrix is

$$\Sigma = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}} = \frac{1}{N}\mathrm{X}^{\mathrm{T}}\mathrm{X}. \tag{9.1}$$

$\Sigma_{ij} = \mathsf{cov}(x_i, x_j)$ is the in-sample covariance of inputs i and j; $\Sigma_{ii} = \sigma_i^2$ is the variance of input i. Assume that Σ has full rank and let its matrix square root be $\Sigma^{\frac{1}{2}}$, which satisfies $\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}} = \Sigma$ (see Problem 9.3 for the computation of $\Sigma^{\frac{1}{2}}$). Consider the whitening transformation

$$\mathbf{z}_n = \Sigma^{-\frac{1}{2}}\mathbf{x}_n,$$

where $\Sigma^{-\frac{1}{2}}$ is the inverse of $\Sigma^{\frac{1}{2}}$. In matrix form, $\mathrm{Z} = \mathrm{X}\Sigma^{-\frac{1}{2}}$. Z is whitened if $\frac{1}{N}\mathrm{Z}^{\mathrm{T}}\mathrm{Z} = \mathrm{I}$. We verify that Z is whitened as follows:

$$\frac{1}{N}\mathrm{Z}^{\mathrm{T}}\mathrm{Z} = \Sigma^{-\frac{1}{2}}\left(\frac{1}{N}\mathrm{X}^{\mathrm{T}}\mathrm{X}\right)\Sigma^{-\frac{1}{2}} = \Sigma^{-\frac{1}{2}}\Sigma\Sigma^{-\frac{1}{2}} = \left(\Sigma^{-\frac{1}{2}}\Sigma^{\frac{1}{2}}\right)\left(\Sigma^{\frac{1}{2}}\Sigma^{-\frac{1}{2}}\right) = \mathrm{I},$$

[2]The term whitening is inherited from signal processing where white noise refers to a signal whose frequency spectrum is uniform; this is indicative of a time series of independent noise realizations. The origin of the term white comes from white light which is a light signal whose amplitude distribution is uniform over all frequencies.

where we used $\Sigma = \Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}}$ and $\Sigma^{\frac{1}{2}}\Sigma^{-\frac{1}{2}} = \Sigma^{-\frac{1}{2}}\Sigma^{\frac{1}{2}} = \mathrm{I}$. Thus, for the transformed inputs, every dimension has scale 1 and the dimensions are pairwise uncorrelated. Centering, normalizing and whitening are illustrated on a toy data set in Figure 9.1.

Figure 9.1: Illustration of centering, normalization and whitening.

It is important to emphasize that input preprocessing does not throw away any information because the transformation is invertible: the original inputs can always be recovered from the transformed inputs using $\bar{\mathbf{x}}$ and Σ.

WARNING! Transforming the data to a more convenient format has a hidden trap which easily leads to data snooping.

> If you are using a test set to estimate your performance, make sure to determine any input transformation only using the training data. A simple rule: the test data should be kept locked away in its raw form until you are ready to test your final hypothesis. (See Example 5.3 for a concrete illustration of how data snooping can affect your estimate of the performance on your test set if input preprocessing in any way used the test set.) After you determine your transformation parameters from the training data, you should use these same parameters to transform your test data to evaluate your final hypothesis g.

9.2 Dimension Reduction and Feature Selection

The *curse of dimensionality* is a general observation that statistical tasks get exponentially harder as the dimensions increase. In learning from data, this manifests itself in many ways, the most immediate being computational. Simple algorithms, such as optimal (or near-optimal) k-means clustering, or determining the optimal linear separator, have a computational complexity which scales exponentially with dimensionality. Furthermore, a fixed number, N, of data points only sparsely populates a space whose volume is growing exponentially with d. So, simple rules like nearest neighbor get adversely affected because a test point's 'nearest' neighbor will likely be very far away and will not be a good representative point for predicting on the test point. The complexity of a hypothesis set, as could be measured by the VC dimension, will typically increase with d (recall that, for the simple linear perceptron, the VC dimension is $d+1$), affecting the generalization from in-sample to out-of-sample. The bottom line is that more data are needed to learn in higher-dimensional input spaces.

Exercise 9.5

Consider a data set with two examples,

$$(\mathbf{x}_1^{\mathrm{T}} = [-1, a_1, \ldots, a_d],\ y_1 = +1); \qquad (\mathbf{x}_2^{\mathrm{T}} = [1, b_1, \ldots, b_d],\ y_2 = -1),$$

where a_i, b_i are independent random ± 1 variables. Let $\mathbf{x}_{\mathsf{test}}^{\mathrm{T}} = [-1, -1, \ldots, -1]$. Assume that only the first component of $\mathbf{x}$ is relevant to f. However, the actual measured $\mathbf{x}$ has additional random components in the additional d dimensions. If the nearest neighbor rule is used, show, either mathematically or with an experiment, that the probability of classifying $\mathbf{x}_{\mathsf{test}}$ correctly is $\frac{1}{2} + O(\frac{1}{\sqrt{d}})$ (d is the number of irrelevant dimensions).

What happens if there is a third data point ($\mathbf{x}_3^{\mathrm{T}} = [1, c_1, \ldots, c_d], y_3 = -1$)?

The exercise illustrates that as you have more and more spurious (random) dimensions, the learned final hypothesis becomes useless because it is dominated by the random fluctuations in these spurious dimensions. Ideally, we should remove all such spurious dimensions before proceeding to the learning. Equivalently, we should retain only the few informative features. For the digits data from Chapter 3, we were able to obtain good performance by extracting just two features from the raw 16×16-pixel input image (size and symmetry), a dimension reduction from 256 raw features to 2 informative ones.

The features $\mathbf{z}$ are simply a transformation of the input $\mathbf{x}$,

$$\mathbf{z} = \Phi(\mathbf{x}),$$

where the number of features is the dimension of $\mathbf{z}$. If the dimension of $\mathbf{z}$ is less than the dimension of $\mathbf{x}$, then we have accomplished dimension reduction. The ideal feature is the target function itself, $z = f(\mathbf{x})$, since if we had this

feature, we are done. This suggests that quality feature selection may be as hard as the original learning problem of identifying f.

We have seen features and feature transforms many times before, for example, in the context of linear models and the non-linear feature transform in Chapter 3. In that context, the non-linear feature transform typically *increased* the dimension to handle the fact that the linear hypothesis was not expressive enough to fit the data. In that setting, increasing the dimension through the feature transform was attempting to improve E_{in}, and we did pay the price of poorer generalization. The reverse is also true. If we can *lower* the dimension without hurting E_{in} (as would be the case if we retained all the important information), then we will also improve generalization.

9.2.1 Principal Components Analysis (PCA)

Centering, scaling and whitening all attempt to correct for arbitrary choices that may have been made during data collection. Feature selection, such as PCA, is conceptually different. It attempts to get rid of redundancy or less informative dimensions to help, among other things, generalization. For example, the top right pixel in the digits data is almost always white, so it is a dimension that carries almost no information. Removing that dimension will not hurt the fitting, but will improve generalization.

PCA constructs a small number of *linear* features to summarize the input data. The idea is to rotate the axes (a linear transformation that defines a new coordinate system) so that the important dimensions in this new coordinate system become self evident and can be retained while the less important ones get discarded. Our toy data set can help crystallize the notion.

Once the data are rotated to the new 'natural' coordinate system, z_1 stands out as the important dimension of the transformed input. The second dimension, z_2, looks like a bunch of small fluctuations which we ought to ignore, in comparison to the apparently more informative and larger z_1. Ignoring the z_2 dimension amounts to setting it to zero (or just throwing away that coordinate), producing a 1-dimensional feature.

Exercise 9.6

Try to build some intuition for what the rotation is doing by using the illustrations in Figure 9.1 to qualitatively answer these questions.

(a) If there is a large offset (or bias) in both measured variables, how will this affect the 'natural axes', the ones to which the data will be rotated? Should you perform input centering before doing PCA?

[*Hint: Consider* *versus*]

(b) If one dimension (say x_1) is inflated disproportionately (e.g., income is measured in dollars instead of thousands of dollars). How will this affect the 'natural axes', the ones to which the data should be rotated? Should you perform input normalization before doing PCA?

(c) If you do input whitening, what will the 'natural axes' for the inputs be? Should you perform input whitening before doing PCA?

What if the small fluctuations in the z_2 direction were the actual important information on which f depends, and the large variability in the z_1 dimension are random fluctuations? Though possible, this rarely happens in practice, and if it does happen, then your input is corrupted by large random noise and you are in trouble anyway. So, let's focus on the case where we have a chance and discuss how to find this optimal rotation.

Intuitively, the direction $\mathbf{v}$ captures the largest fluctuations in the data, which could be measured by variance. If we project the input $\mathbf{x}_n$ onto $\mathbf{v}$ to get $z_n = \mathbf{v}^{\mathrm{T}}\mathbf{x}_n$, then the variance of z is $\frac{1}{N}\sum_{n=1}^{N} z_n^2$ (remember $\mathbf{x}_n$ and hence z_n have zero mean).

$$\begin{aligned}
\mathsf{var}[z] &= \frac{1}{N}\sum_{n=1}^{N} z_n^2 &&= \frac{1}{N}\sum_{n=1}^{N} \mathbf{v}^{\mathrm{T}}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}\mathbf{v} \\
&&&= \mathbf{v}^{\mathrm{T}}\left(\frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}\right)\mathbf{v} \\
&&&= \mathbf{v}^{\mathrm{T}}\Sigma\mathbf{v}.
\end{aligned}$$

To maximize $\mathsf{var}[z]$, we should pick $\mathbf{v}$ as the top eigenvector of Σ, the one with the largest eigenvalue. Before we get more formal, let us address an apparent conflict of interest. Whitening is a way to put your data into a spherically symmetric form so that all directions are 'equal'. This is recommended when you have no evidence to the contrary; you whiten the data because most learning algorithms treat every dimension equally (nearest neighbor, weight decay, etc.). PCA, on the other hand is highlighting specific directions which contain more variance. There is no use doing PCA after doing whitening, since every direction will be on an equal footing after whitening. You use

PCA precisely because the directions are not to be treated equally. PCA helps to identify and throw away the directions where the fluctuations are a result of small amounts of noise. After deciding which directions to throw away, you can now use whitening to put all the retained directions on an equal footing, if you wish.

Our visual intuition works well in 2-dimensions, but in higher dimension, when no single direction captures most of the fluctuation, we need a more principled approach, starting with a mathematical formulation of the task. We begin with the observation that a rotation of the data exactly corresponds to representing the data in a new (rotated) coordinate system.

Coordinate Systems A coordinate system is defined by an orthonormal basis, a set of mutually orthogonal unit vectors. The standard Euclidean coordinate system is defined by the Euclidean basis in d dimensions, $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_d$, where $\mathbf{u}_i$ is the ith standard basis vector which has a 1 in coordinate i and 0 for all other coordinates. The input vector $\mathbf{x}$ has the components $x_i = \mathbf{x}^{\mathrm{T}}\mathbf{u}_i$, and we can write

$$\mathbf{x} = \sum_{i=1}^{d} x_i \mathbf{u}_i = \sum_{i=1}^{d} (\mathbf{x}^{\mathrm{T}}\mathbf{u}_i)\mathbf{u}_i.$$

This can be done for any orthonormal basis $\mathbf{v}_1, \ldots, \mathbf{v}_d$,

$$\mathbf{x} = \sum_{i=1}^{d} z_i \mathbf{v}_i = \sum_{i=1}^{d} (\mathbf{x}^{\mathrm{T}}\mathbf{v}_i)\mathbf{v}_i,$$

where the *coordinates* in the basis $\mathbf{v}_1, \ldots, \mathbf{v}_d$ are $z_i = (\mathbf{x}^{\mathrm{T}}\mathbf{v}_i)$. The goal of PCA is to construct a more intuitive basis where some (hopefully the majority) of the coordinates are small and can be treated as small random fluctuations. These coordinates are going to be discarded, i.e., set to zero. The hope is that we have reduced the dimensionality of the problem while retaining most of the important information.

So, given the vector $\mathbf{x}$ (in the standard coordinate system) and some other coordinate system $\mathbf{v}_1, \ldots, \mathbf{v}_d$, we can define the transformed feature vector whose components are the coordinates $z_1, \ldots, z_d$ in this new coordinate system. Suppose that the first $k \leq d$ of these transformed coordinates are the informative ones, so we throw away the remaining coordinates to arrive at our *dimension-reduced* feature vector

$$\mathbf{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_k \end{bmatrix} = \begin{bmatrix} \mathbf{x}^{\mathrm{T}}\mathbf{v}_1 \\ \vdots \\ \mathbf{x}^{\mathrm{T}}\mathbf{v}_k \end{bmatrix} = \Phi(\mathbf{x}).$$

Exercise 9.7

(a) Show that $\mathbf{z}$ is a linear transformation of $\mathbf{x}$, $\mathbf{z} = \mathrm{V}^{\mathrm{T}}\mathbf{x}$. What are the dimensions of the matrix V and what are its columns?

(b) Show that the transformed data matrix is $\mathrm{Z} = \mathrm{XV}$.

(c) Show that $\sum_{i=1}^{d} z_i^2 = \sum_{i=1}^{d} x_i^2$ and hence that $\|\mathbf{z}\| \leq \|\mathbf{x}\|$.

If we kept all the components $z_1, \ldots, z_d$, then we can reconstruct $\mathbf{x}$ via

$$\mathbf{x} = \sum_{i=1}^{d} z_i \mathbf{v}_i.$$

Using only the first k components, the best reconstruction of $\mathbf{x}$ is

$$\hat{\mathbf{x}} = \sum_{i=1}^{k} z_i \mathbf{v}_i.$$

We have lost that part of $\mathbf{x}$ represented by the trailing coordinates of $\mathbf{z}$. The magnitude of the part we lost is captured by the reconstruction error

$$\|\mathbf{x} - \hat{\mathbf{x}}\|^2 = \left\| \sum_{i=k+1}^{d} z_i \mathbf{v}_i \right\|^2 = \sum_{i=k+1}^{d} z_i^2$$

(because $\mathbf{v}_1, \ldots, \mathbf{v}_d$ are *orthonormal*). The new coordinate system is good if the sum of the reconstruction errors over the data points is small. That is, if

$$\sum_{n=1}^{N} \|\mathbf{x}_n - \hat{\mathbf{x}}_n\|^2$$

is small. If the $\mathbf{x}_n$ are reconstructed with small error from $\mathbf{z}_n$ (i.e. $\hat{\mathbf{x}}_n \approx \mathbf{x}_n$), then not much information was lost. PCA finds a coordinate system that *minimizes* this total reconstruction error. The trailing dimensions will have the least possible information, and so even after throwing away those trailing dimensions, we can still almost reconstruct the original data. PCA is optimal, which means that no other *linear* method can produce coordinates with a smaller reconstruction error. The first k basis vectors, $\mathbf{v}_1, \ldots, \mathbf{v}_k$, of this optimal coordinate basis are called the top-k principal directions.

So, how do we find this optimal coordinate basis $\mathbf{v}_1, \ldots, \mathbf{v}_d$ (of which we only need $\mathbf{v}_1, \ldots, \mathbf{v}_k$ to compute our dimensionally reduced feature)? The solution to this problem has been known since 1936 when the remarkable *singular value decomposition* (SVD) was invented. The SVD is such a useful tool for learning from data that time spent mastering it will pay dividends (additional background on the SVD is given in Appendix B.2).

The Singular Value Decomposition (SVD). Any matrix, for example, our data matrix X, has a very special representation as the product of three matrices. Assume that $\mathrm{X} \in \mathbb{R}^{N \times d}$ with $N \geq d$ (a similar decomposition holds for X^{T} if $N < d$). Then,

$$\mathrm{X} = \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}$$

where $\mathrm{U} \in \mathbb{R}^{n \times d}$ has orthonormal columns, $\mathrm{V} \in \mathbb{R}^{d \times d}$ is an orthogonal matrix[3] and Γ is a non-negative diagonal matrix.[4] The diagonal elements $\gamma_i = \Gamma_{ii}$, where $\gamma_1 \geq \gamma_2 \geq \cdots \geq \gamma_d \geq 0$, are the *singular values* of X, ordered from largest to smallest. The number of non-zero singular values is the rank of X, which we will assume is d for simplicity. Pictorially,

The matrix U contains (as its columns) the *left singular vectors* of X, and similarly V contains (as its columns) the *right singular vectors* of X. Since U consists of orthonormal columns, $\mathrm{U}^{\mathrm{T}}\mathrm{U} = \mathrm{I}_d$. Similarly, $\mathrm{V}^{\mathrm{T}}\mathrm{V} = \mathrm{V}\mathrm{V}^{\mathrm{T}} = \mathrm{I}_d$. If X is square, then U will be square. Just as a square matrix maps an eigenvector to a multiple of itself, a more general non-square matrix maps a left (resp. right) singular vector to a multiple of the corresponding right (resp. left) singular vector, as verified by the following identities:

$$\mathrm{U}^{\mathrm{T}}\mathrm{X} = \Gamma\mathrm{V}^{\mathrm{T}}; \qquad \mathrm{X}\mathrm{V} = \mathrm{U}\Gamma.$$

It is convenient to have column representations of U and V in terms of the singular vectors, $\mathrm{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_d]$ and $\mathrm{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_d]$.

Exercise 9.8

Show $\mathrm{U}^{\mathrm{T}}\mathrm{X} = \Gamma\mathrm{V}^{\mathrm{T}}$ and $\mathrm{X}\mathrm{V} = \mathrm{U}\Gamma$, and hence $\mathrm{X}^{\mathrm{T}}\mathbf{u}_i = \gamma_i\mathbf{v}_i$ and $\mathrm{X}\mathbf{v}_i = \gamma_i\mathbf{u}_i$.

(The ith singular vectors and singular value $(\mathbf{u}_i, \mathbf{v}_i, \gamma_i)$ play a similar role to eigenvector-eigenvalue pairs.)

Computing the Principal Components via SVD. It is no coincidence that we used $\mathbf{v}_i$ for the right singular vectors of X and $\mathbf{v}_i$ in the mathematical formulation of the PCA task. The right singular vectors $\mathbf{v}_1, \ldots, \mathbf{v}_d$ are our optimal coordinate basis so that by ignoring the trailing components we incur the least reconstruction error.

[3] An orthogonal matrix is a square matrix with orthonormal columns, and therefore its inverse is the same as its transpose. So, $\mathrm{V}^{\mathrm{T}}\mathrm{V}=\mathrm{V}\mathrm{V}^{\mathrm{T}}=\mathrm{I}$.

[4] In the traditional linear algebra literature, Γ is typically denoted by Σ and its diagonal elements are the singular values $\sigma_1 \geq \cdots \geq \sigma_d$. We use Γ because we have reserved Σ for the covariance matrix of the input distribution and σ^2 for the noise variance.

Theorem 9.1 (Eckart and Young, 1936)**.** For any k, $\mathbf{v}_1, \ldots, \mathbf{v}_k$ (the top-k right singular vectors of the data matrix X) are a set of top-k principal component directions and the optimal reconstruction error is $\sum_{i=k+1}^{d} \gamma_i^2$.

The components of the dimensionally reduced feature vector are $z_i = \mathbf{x}^{\mathrm{T}}\mathbf{v}_i$ and the reconstructed vector is $\hat{\mathbf{x}} = \sum_{i=1}^{k} z_i \mathbf{v}_i$, which in matrix form is

$$\hat{\mathrm{X}} = \mathrm{X}\mathrm{V}_k\mathrm{V}_k^{\mathrm{T}}, \tag{9.2}$$

where $\mathrm{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$ is the matrix of top-k right singular vectors of X.

PCA Algorithm:
Inputs: The *centered* data matrix X and $k \geq 1$.

1: Compute the SVD of X: $[\mathrm{U}, \Gamma, \mathrm{V}] = \mathsf{svd}(\mathrm{X})$.
2: Let $\mathrm{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$ be the first k columns of V.
3: The PCA-feature matrix and the reconstructed data are

$$\mathrm{Z} = \mathrm{X}\mathrm{V}_k, \qquad \hat{\mathrm{X}} = \mathrm{X}\mathrm{V}_k\mathrm{V}_k^{\mathrm{T}}.$$

Note that PCA, along with the other input pre-processing tools (centering, rescaling, whitening) are all *unsupervised* - you do not need the y-values. The Eckart-Young theorem is quite remarkable and so fundamental in data analysis that it certainly warrants a proof.

Begin safe skip: You may skip the proof without compromising the logical sequence. A similar green box will tell you when to rejoin.

We will need some matrix algebra preliminaries which are useful general tools. Recall that the reconstructed data matrix $\hat{\mathrm{X}}$ is similar to the data matrix, having the reconstructed input vectors $\hat{\mathbf{x}}_n$ as its rows. The Frobenius norm $\|\mathrm{A}\|_F^2$ of a matrix $\mathrm{A} \in \mathbb{R}^{N\times d}$ is the analog of the Euclidean norm, but for matrices:

$$\|\mathrm{A}\|_F^2 \stackrel{\text{def}}{=} \sum_{n=1}^{N}\sum_{i=1}^{d} \mathrm{A}_{ij}^2 = \sum_{n=1}^{N} \|\mathrm{row}_n(\mathrm{A})\|^2 = \sum_{i=1}^{d} \|\mathrm{column}_i(\mathrm{A})\|^2.$$

The reconstruction error is exactly the Frobenius norm of the matrix difference between the original and reconstructed data matrices, $\|\mathrm{X} - \hat{\mathrm{X}}\|_F^2$.

Exercise 9.9

Consider an arbitrary matrix A, and any matrices U, V with orthonormal columns ($\mathrm{U}^{\mathrm{T}}\mathrm{U} = \mathrm{I}$ and $\mathrm{V}^{\mathrm{T}}\mathrm{V} = \mathrm{I}$).

(a) Show that $\|\mathrm{A}\|_F^2 = \mathrm{trace}(\mathrm{A}\mathrm{A}^{\mathrm{T}}) = \mathrm{trace}(\mathrm{A}^{\mathrm{T}}\mathrm{A})$.

(b) Show that $\|\mathrm{U}\mathrm{A}\mathrm{V}^{\mathrm{T}}\|_F^2 = \|\mathrm{A}\|_F^2$ (assume all matrix products exist). *[Hint: Use part (a).]*

Proof of the Eckart-Young Theorem. The preceding discussion was for general A. We now set A to be any orthonormal basis A, with columns $\mathbf{a}_1, \ldots, \mathbf{a}_d$,

$$\mathrm{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_d].$$

Since V is an orthonormal basis, we can write

$$\mathrm{A} = \mathrm{V}\boldsymbol{\Psi},$$

where $\boldsymbol{\Psi} = [\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_d]$. Since

$$\mathrm{I} = \mathrm{A}^{\mathrm{T}}\mathrm{A} = \boldsymbol{\Psi}^{\mathrm{T}}\mathrm{V}^{\mathrm{T}}\mathrm{V}\boldsymbol{\Psi} = \boldsymbol{\Psi}^{\mathrm{T}}\boldsymbol{\Psi},$$

we see that $\boldsymbol{\Psi}$ is orthogonal. Suppose (without loss of generality) that we will use the first k basis vectors of A for reconstruction. So, define

$$\mathrm{A}_k = [\mathbf{a}_1, \ldots, \mathbf{a}_k] = \mathrm{V}\boldsymbol{\Psi}_k,$$

where $\Psi_k = [\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_k]$. We use A_k to approximate X using the reconstruction $\hat{\mathrm{X}} = \mathrm{X}\mathrm{A}_k\mathrm{A}_k^{\mathrm{T}}$ from (9.2). Then,

$$\begin{aligned}
\|\mathrm{X} - \hat{\mathrm{X}}\|_F^2 &= \|\mathrm{X} - \mathrm{X}\mathrm{A}_k\mathrm{A}_k^{\mathrm{T}}\|_F^2 \\
&= \|\mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}} - \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}\mathrm{V}\boldsymbol{\Psi}_k\boldsymbol{\Psi}_k^{\mathrm{T}}\mathrm{V}^{\mathrm{T}}\|_F^2 \\
&= \|\mathrm{U}(\Gamma - \Gamma\boldsymbol{\Psi}_k\boldsymbol{\Psi}_k^{\mathrm{T}})\mathrm{V}^{\mathrm{T}}\|_F^2 \\
&= \|\Gamma(\mathrm{I} - \boldsymbol{\Psi}_k\boldsymbol{\Psi}_k^{\mathrm{T}})\|_F^2,
\end{aligned}$$

where we have used $\mathrm{V}^{\mathrm{T}}\mathrm{V} = \mathrm{I}_d$ and Exercise 9.9(b). By Exercise 9.9(a),

$$\|\Gamma(\mathrm{I} - \boldsymbol{\Psi}_k\boldsymbol{\Psi}_k^{\mathrm{T}})\|_F^2 = \mathrm{trace}(\Gamma(\mathrm{I} - \boldsymbol{\Psi}_k\boldsymbol{\Psi}_k^{\mathrm{T}})^2\Gamma).$$

Now, using the linearity and cyclic properties of the trace and the fact that $\mathrm{I} - \boldsymbol{\Psi}_k\boldsymbol{\Psi}_k^{\mathrm{T}}$ is a projection,

$$\begin{aligned}
\mathrm{trace}\left(\Gamma(\mathrm{I} - \boldsymbol{\Psi}_k\boldsymbol{\Psi}_k^{\mathrm{T}})^2\Gamma\right) &= \mathrm{trace}\left(\Gamma(\mathrm{I} - \boldsymbol{\Psi}_k\boldsymbol{\Psi}_k^{\mathrm{T}})\Gamma\right) \\
&= \mathrm{trace}(\Gamma^2) - \mathrm{trace}\left(\Gamma\boldsymbol{\Psi}_k\boldsymbol{\Psi}_k^{\mathrm{T}}\Gamma\right) \\
&= \mathrm{trace}(\Gamma^2) - \mathrm{trace}\left(\boldsymbol{\Psi}_k^{\mathrm{T}}\Gamma^2\boldsymbol{\Psi}_k\right).
\end{aligned}$$

The first term is independent of $\boldsymbol{\Psi}_k$, so we must maximize the second term. Since $\boldsymbol{\Psi}_k$ has k orthonormal columns, $\|\boldsymbol{\Psi}_k\|_F^2 = k$ (because the Frobenius norm is the sum of squared column norms). Let the rows of $\boldsymbol{\Psi}_k$ be $\mathbf{q}_1^{\mathrm{T}}, \ldots, \mathbf{q}_d^{\mathrm{T}}$. Then $0 \le \|\mathbf{q}_i\|^2 \le 1$ ($\boldsymbol{\Psi}$ has orthonormal rows and $\mathbf{q}_i$ are truncated rows of $\boldsymbol{\Psi}$), and $\sum_{i=1}^d \|\mathbf{q}_i\|^2 = k$ (because the Frobenius norm is the sum of squared row-norms). We also have that

$$\boldsymbol{\Psi}_k^{\mathrm{T}}\Gamma^2\boldsymbol{\Psi}_k = [\mathbf{q}_1, \ldots, \mathbf{q}_d]\begin{bmatrix}\gamma_1^2 & & \\ & \ddots & \\ & & \gamma_d^2\end{bmatrix}\begin{bmatrix}\mathbf{q}_1^{\mathrm{T}} \\ \vdots \\ \mathbf{q}_d\end{bmatrix} = \sum_{i=1}^d \gamma_i^2\mathbf{q}_i\mathbf{q}_i^{\mathrm{T}}.$$

Taking the trace and using linearity of the trace gives

$$\operatorname{trace}(\boldsymbol{\Psi}_k^{\mathrm{T}}\Gamma^2\boldsymbol{\Psi}_k) = \sum_{i=1}^{d} \gamma_i^2 \|\mathbf{q}_i\|^2.$$

To maximize $\operatorname{trace}(\boldsymbol{\Psi}_k^{\mathrm{T}}\Gamma^2\boldsymbol{\Psi}_k)$, the best possible way to spend our budget of k for $\sum_i \|\mathbf{q}_i\|^2$ is to put as much as possible into $\|\mathbf{q}_1\|$ and then $\|\mathbf{q}_2\|$ and so on, because $\gamma_1 \geq \gamma_2 \geq \cdots$. This will result in the $\|\mathbf{q}_1\|^2 = \cdots = \|\mathbf{q}_k\|^2 = 1$ and all the remaining $\mathbf{q}_i = \mathbf{0}$. Thus,

$$\operatorname{trace}(\boldsymbol{\Psi}_k^{\mathrm{T}}\Gamma^2\boldsymbol{\Psi}_k) \leq \sum_{i=1}^{k} \gamma_i^2.$$

We conclude that

$$\begin{aligned}
\|\mathrm{X} - \hat{\mathrm{X}}\|_F^2 &= \operatorname{trace}(\Gamma^2) - \operatorname{trace}\left(\boldsymbol{\Psi}_k^{\mathrm{T}}\Gamma^2\boldsymbol{\Psi}_k\right) \\
&= \sum_{i=1}^{d} \gamma_i^2 - \operatorname{trace}\left(\boldsymbol{\Psi}_k^{\mathrm{T}}\Gamma^2\boldsymbol{\Psi}_k\right) \\
&\geq \sum_{i=1}^{d} \gamma_i^2 - \sum_{i=1}^{k} \gamma_i^2 \\
&= \sum_{i=k+1}^{d} \gamma_i^2.
\end{aligned}$$

If $\mathrm{A} = \mathrm{V}$ so that $\boldsymbol{\Psi} = \mathrm{I}$, then indeed $\|\mathbf{q}_1\|^2 = \cdots = \|\mathbf{q}_k\|^2 = 1$ and we attain equality in the bound above, showing that the top-k right singular vectors of X do indeed give an optimal basis, which concludes the proof. ■

The proof of the theorem also shows that the optimal reconstruction error is the sum of squares of the trailing singular values of X, $\|\mathrm{X} - \hat{\mathrm{X}}\|_F^2 = \sum_{i=k+1}^{d} \gamma_i^2$.

End safe skip: Those who skipped the proof are now rejoining us for some examples of PCA in action.

Example 9.2. It is instructive to see PCA in action. The digits training data matrix has 7291 (16×16)-images ($d = 256$), so $\mathrm{X} \in \mathbb{R}^{7291\times 256}$. Let $\bar{\mathbf{x}}$ be the average image. First center the data, setting $\mathbf{x}_n \leftarrow \mathbf{x}_n - \bar{\mathbf{x}}$, to get a centered data matrix X.

Now, let's compute the SVD of this centered data matrix, $\mathrm{X} = \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}$. To do so we may use a built in SVD package that is pretty much standard on any numerical platform. It is also possible to only compute the top-k singular vectors in most SVD packages to obtain $\mathrm{U}_k, \Gamma_k, \mathrm{V}_k$, where U_k and V_k

(a) Reconstruction error (b) Top-2 PCA-features

Figure 9.2: PCA on the digits data. (a) Shows how the reconstruction error depends on the number of components k; about 150 features suffice to reconstruct the data almost perfectly. If all principal components are equally important, the reconstruction error would decrease linearly with k, which is not the case here. (b) Shows the two features obtained using the top two principal components. These features look as good as our hand-constructed features of symmetry and intensity from Example 3.5 in Chapter 3.

contain only the top-k singular vectors, and Γ_k the top-k singular values. The reconstructed matrix is

$$\hat{\mathrm{X}} = \mathrm{X}\mathrm{V}_k\mathrm{V}_k^{\mathrm{T}} = \mathrm{U}_k\Gamma_k\mathrm{V}_k^{\mathrm{T}}.$$

By the Eckart-Young theorem, $\hat{\mathrm{X}}$ is the best rank-k approximation to X. Let's look at how the reconstruction error depends on k, the number of features used. To do this, we plot the reconstruction error using k principle components as a percentage of the reconstruction error with zero components. With zero components, the reconstruction error is just $\|\mathrm{X}\|_F^2$. The result is shown in Figure 9.2(a). As can be seen, with just 50 components, the reconstruction error is about 10%. This is a rule of thumb to determine how many components to use: choose k to obtain a reconstruction error of less than 10%. This amounts to treating the bottom 10% of the fluctuations in the data as noise.

Let's reduce the dimensionality to 2 by projecting onto the top two principal components in V_2. The resulting features are shown in Figure 9.2(b). These features can be compared to the features obtained using intensity and symmetry in Example 3.5 on page 106. The features appear to be quite good, and suitable for solving this learning problem. Unlike the size and intensity features which we used in Example 3.5, the biggest advantage of these PCA features is that their construction is fully automated – you don't need to know anything about the digit recognition problem to obtain them. This is also their biggest disadvantage – the features are provably good at reconstructing the input data, but there is no guarantee that they will be useful for solving the

learning problem. In practice, a little thought about constructing features, together with such automated methods to then reduce the dimensionality, usually works best.

Finally, suppose you use the feature vector $\mathbf{z}$ in Figure 9.2(b) to build a classifier $\tilde{g}(\mathbf{z})$ to distinguish between the digit 1 and all the other digits. The final hypothesis to be applied to a test input $\mathbf{x}_{\text{test}}$ is

$$g(\mathbf{x}_{\text{test}}) = \tilde{g}(\mathrm{V}_2^{\mathrm{T}}(\mathbf{x}_{\text{test}} - \bar{\mathbf{x}})),$$

where $\tilde{g}$, V_2 and $\bar{\mathbf{x}}$ were constructed from the data: V_2 and $\bar{\mathbf{x}}$ from the data inputs, and $\tilde{g}$ from the targets and PCA-reduced inputs $(\mathrm{Z}, \mathbf{y})$ □

We end this section by justifying the "maximize the variance" intuition that began our discussion of PCA. The next exercise shows that the principal components do indeed represent the high variance directions in the data.

Exercise 9.10

Assume that the data matrix X is centered, and define the covariance matrix $\Sigma = \frac{1}{N}\mathrm{X}^{\mathrm{T}}\mathrm{X}$. Assume all the singular values of X are distinct. (What does this mean for the eigenvalues of Σ?) For a potential principal direction $\mathbf{v}$, we defined $z_n = \mathbf{x}_n^{\mathrm{T}}\mathbf{v}$ and showed that $\mathrm{var}(z_1, \ldots, z_N) = \mathbf{v}^{\mathrm{T}}\Sigma\mathbf{v}$.

(a) Show that the direction which results in the highest variance is $\mathbf{v}_1$, the top right singular vector of X.

(b) Show that we obtain the top-k principal directions, $\mathbf{v}_1, \ldots, \mathbf{v}_k$, by selecting k directions sequentially, each time obtaining the direction with highest variance that is orthogonal to all previously selected directions.

This shows that the top-k principal directions are the directions of highest variance.

(c) If you don't the data matrix X, but only know the covariance matrix Σ, can you obtain the principal directions? If so, how?

9.2.2 Nonlinear Dimension Reduction

Figure 9.3 illustrates one thing that can go wrong with PCA. You can see that the data in Figure 9.3(a) approximately lie on a one-dimensional surface (a curve). However, when we try to reconstruct the data using top-1 PCA, the result is a disaster, shown in Figure 9.3(b). This is because, even though the data lie on a curve, that curve is not linear. PCA can only construct linear features. If the data do not live on a lower dimensional 'linear manifold', then PCA will not work. If you are going to use PCA, here is a checklist that will help you determine whether PCA will work.

1. Do you expect the data to have *linear* structure, for example does the data lie in a linear subspace of lower dimension?

(a) Data in $\mathcal{X}$ space (b) Top-1 PCA reconstruction

Figure 9.3: (a) The data in $\mathcal{X}$ space does not 'live' in a lower dimensional *linear* manifold. (b) The reconstructed data using top-1 PCA data must lie on a line and therefore cannot accurately represent the original data.

2. Do the bottom principal components contain primarily small random fluctuations that correspond to noise and should be thrown away? The fact that they are small can be determined by looking at the reconstruction error. The fact that they are noise is not much more than a guess.
3. Does the target function f depend primarily on the top principal components, or are the small fluctuations in the bottom principal components key in determining the value of f? If the latter, then PCA will not help the machine learning task. In practice, it is difficult to determine whether this is true (without snooping ☺). A validation method can help determine whether to use PCA-dimension-reduction or not. Usually, throwing away the lowest principal components does not throw away significant information related to the target function, and what little it does throw away is made up for in the reduced generalization error bar because of the lower dimension.

Clearly, PCA will not work for our data in Figure 9.3. However, we are not dead yet. We have an ace up our sleeve, namely the all-powerful nonlinear transform. Looking at the data in Figure 9.3 suggests that the angular coordinate is important. So, lets consider a transform to the nonlinear feature space defined by polar coordinates.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow{\Phi} \begin{bmatrix} r \\ \theta \end{bmatrix} = \begin{bmatrix} \sqrt{x_1^2 + x_2^2} \\ \tan^{-1}(\frac{x_2}{x_1}) \end{bmatrix}$$

The data using polar-coordinates is shown in Figure 9.4(a). In this space, the data clearly lie on a linear subspace, appropriate for PCA. The top-1 PCA reconstructed data (in the nonlinear feature space) is shown in Figure 9.4(b).

(a) Transformed data in $\mathcal{Z}$ space (b) Reconstruction in $\mathcal{Z}$ space

Figure 9.4: PCA in a nonlinear feature space. (a) The transformed data in the $\mathcal{Z}$ space are approximately on a linear manifold. (b) Shows nonlinear reconstruction of the data in the nonlinear feature space.

We can obtain the reconstructed data in the original $\mathcal{X}$ space by transforming the red reconstructed points in Figure 9.4(b) back to $\mathcal{X}$ space, as shown below.

Exercise 9.11

Using the feature transform $\Phi : \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right] \mapsto \begin{bmatrix} x_1 \\ x_2 \\ x_1+x_2 \end{bmatrix}$, you have run top-1 PCA on your data $\mathbf{z}_1, \ldots, \mathbf{z}_n$ in $\mathcal{Z}$ space to obtain $\mathrm{V}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ and $\bar{\mathbf{z}} = \mathbf{0}$.

For the test point $\mathbf{x} = \left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right]$, compute $\mathbf{z}, \hat{\mathbf{z}}, \hat{\mathbf{x}}$.

($\mathbf{z}$ is the test point in $\mathcal{Z}$ space; $\hat{\mathbf{z}}$ is the reconstructed test point in $\mathcal{Z}$ space using top-1 PCA; $\hat{\mathbf{x}}$ is the reconstructed test point in $\mathcal{X}$ space.)

Exercise 9.11 illustrates that you may not always be able to obtain the reconstructed data in your original $\mathcal{X}$ space. For our spiral example, we can obtain

the reconstructed data in the $\mathcal{X}$ space because the polar feature transform is *invertible*; invertibility of Φ is essential to reconstruction in $\mathcal{X}$ space. In general, you may want to use a feature transform that is not invertible, for example the 2nd-order polynomial transform. In this case, you will not be able to reconstruct your data in the $\mathcal{X}$ space, but that need not prevent you from using PCA with the nonlinear feature transform, if your goal is prediction. You can transform to your $\mathcal{Z}$ space, run PCA in the $\mathcal{Z}$ space, discard the bottom principal components (dimension reduction), and run your learning algorithm using the top principal components. Finally, to classify a new test point, you first transform it, and then classify in the $\mathcal{Z}$ space. The entire work-flow is summarized in the following algorithm.

PCA with Nonlinear Feature Transform:
Inputs: The data $\mathrm{X}, \mathbf{y}$; $k \geq 1$; and, transform Φ.

1: Transform the data: $\mathbf{z}_n = \Phi(\mathbf{x}_n)$.
2: Center the data: $\mathbf{z}_n \leftarrow \mathbf{z}_n - \bar{\mathbf{z}}$ where $\bar{\mathbf{z}} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{z}_n$.
3: Obtain the centered data matrix Z whose rows are $\mathbf{z}_n$.
4: Compute the SVD of Z: $[\mathrm{U}, \Gamma, \mathrm{V}] = \mathsf{svd}(\mathrm{Z})$.
5: Let $\mathrm{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$ be the first k columns of V.
6: Construct the top-k PCA-feature matrix $\mathrm{Z}_k = \mathrm{Z}\mathrm{V}_k$.
7: Use the data $(\mathrm{Z}_k, \mathbf{y})$ to learn a final hypothesis $\tilde{g}$.
8: The final hypothesis is

$$g(\mathbf{x}) = \tilde{g}(\mathrm{V}_k^{\mathrm{T}}(\Phi(\mathbf{x}) - \bar{\mathbf{z}}))$$

There are other approaches to nonlinear dimension reduction. Kernel-PCA is a way to combine PCA with the nonlinear feature transform without visiting the $\mathcal{Z}$ space. Kernel-PCA uses a kernel in the $\mathcal{X}$ space in much the same way that kernels were used to combine the maximum-margin linear separator with the nonlinear transform to get the Kernel-Support Vector Machine. Two other popular approaches are the Neural-Network auto-encoder (which we discussed in the context of Deep Learning with Neural Networks in Section 7.6) and nonlinear principal curves and surfaces which are nonlinear analogues to the PCA-generated linear principal surfaces. With respect to reconstructing the data onto lower dimensional manifolds, there are also nonparametric techniques like the Laplacian Eigenmap or the Locally Linear Embedding (LLE). The problems explore some of these techniques in greater depth.

9.3 Hints And Invariances

Hints are tidbits of information about the target function that you know ahead of time (before you look at any data). You know these tidbits because you know something about the learning problem. Why do we need hints? If we had

an unlimited supply of data (input-output examples), we wouldn't need hints because the data contains all the information we need to learn.[5] In reality, though, the data is finite, and so not all properties about f may be represented in the data. Common hints are invariances, monotonicities and symmetries. For example, rotational invariance applies in most vision problems.

Exercise 9.12

Consider the following three hypothesis sets,

$$\begin{aligned}\mathcal{H}_+ &= \{h|h(x) = \text{sign}(wx + w_0);\ w \geq 0\},\\ \mathcal{H}_- &= \{h|h(x) = \text{sign}(wx + w_0);\ w < 0\},\end{aligned}$$

and $\mathcal{H} = \mathcal{H}_+ \cup \mathcal{H}_-$. The task is to perform credit approval based on the income x. The weights w and w_0 must be learned from data.

(a) What are the VC dimensions of $\mathcal{H}$ and $\mathcal{H}_\pm$?

(b) Which model will you pick (before looking at data)? Explain why.

The exercise illustrates *several* important points about learning from data, and we are going to beat these points into the ground. First, the choice between $\mathcal{H}$ and $\mathcal{H}_\pm$ brings the issue of generalization to light. Do you have enough data to use the more complex model, or do you have to use a simpler model to ensure that $E_{\text{in}} \approx E_{\text{out}}$? When learning from data, this is the first order of business. If you cannot expect $E_{\text{in}} \approx E_{\text{out}}$, you are doomed from step 1. Suppose you need to choose one of the simpler models to get good generalization. $\mathcal{H}_+$ and $\mathcal{H}_-$ are of equal 'simplicity'. As far as generalization is concerned, both models are equivalent. Let's play through both choices.

Suppose you choose $\mathcal{H}_-$. The nature of credit approval is that if you have more money, you are less likely to default. So, when you look at the data, it will look something like this (blue circles are $+1$)

No matter how you try to fit this data using hypotheses in $\mathcal{H}_-$, the in-sample error will be approximately $\frac{1}{2}$. Nevertheless, you will output one such final hypothesis g_-. You will have succeeded in that $E_{\text{in}}(g_-) \approx E_{\text{out}}(g_-)$. But you will have failed in that you will tell your client to expect about 50% out-of-sample error, and he will laugh at you.

Suppose, instead, you choose $\mathcal{H}_+$. Now you will be able to select a hypothesis g_+ with $E_{\text{in}}(g_+) \approx 0$; and, because of good generalization, you will be able to tell the client that you expect near perfect out-of-sample accuracy.

As in this credit example, you often have auxiliary knowledge about the problem. In our example, we have reason to pick $\mathcal{H}_+$ over $\mathcal{H}_-$, and by do-

[5] This is true from the informational point of view. However, from the computational point of view, a hint can still help. For example, knowing that the target is linear will considerably speed up your search by focusing it on linear models.

ing so, we are using the *hint* that one's credit status, by its very nature, is monotonically increasing in income.

Moral. Don't throw any old $\mathcal{H}$ at the problem just because your $\mathcal{H}$ is small and will give good generalization. More often than not, you will learn that you failed. Throw the *right* $\mathcal{H}$ at the problem, and if, *unfortunately*, you fail you will know it.

Exercise 9.13

For the problem in Exercise 9.12, why not try a hybrid procedure:

Start with $\mathcal{H}_+$. If $\mathcal{H}_+$ gives low E_{in}, stop; if not, use $\mathcal{H}_-$.

If your hybrid strategy stopped at $\mathcal{H}_+$, can you use the VC bound for $\mathcal{H}_+$? Can you use the VC-bound for $\mathcal{H}$?

(Problem 9.20 develops a VC-type analysis of such hybrid strategies within a framework called Structural Risk Minimization (SRM).)

Hints are pieces of prior information about the learning problem. By prior, we mean prior to looking at the data. Here are some common examples of hints.

Symmetry or Anti-symmetry hints: $f(\mathbf{x}) = f(-\mathbf{x})$ or $f(\mathbf{x}) = -f(-\mathbf{x})$. For example, in a financial application, if a historical pattern $\mathbf{x}$ means you should buy the stock, then the reverse pattern $-\mathbf{x}$ often means you should sell.

Rotational invariance: $f(\mathbf{x})$ depends only on $\|\mathbf{x}\|$.

General Invariance hints: For some transformation $\mathcal{T}$, $f(\mathbf{x}) = f(\mathcal{T}\mathbf{x})$. Invariance to scale, shift and rotation of an image are common in vision applications.

Monotonicity hints: $f(\mathbf{x} + \Delta\mathbf{x}) \geq f(\mathbf{x})$ if $\Delta\mathbf{x} \geq \mathbf{0}$. Sometimes you may only have monotonicity in some of the variables. For example, credit approval should be a monotonic function of income, but perhaps not monotonic in age.

Convexity hint: $f(\eta\mathbf{x} + (1-\eta)\mathbf{x}') \leq \eta f(\mathbf{x}) + (1-\eta)f(\mathbf{x}')$ for $0 \leq \eta \leq 1$.

Perturbation hint: f is close to a known function f', so $f = f' + \delta f$, where δf is small (sometimes called a catalyst hint).

One way to use a hint is to constrain the hypothesis set so that all hypotheses satisfy the hint: start with a hypothesis set $\tilde{\mathcal{H}}$ and throw out all hypotheses which do not satisfy the constraint to obtain $\mathcal{H}$. This will directly lower the size of the hypothesis set, improving your generalization ability. The next exercise illustrates the mechanics of this approach.

Exercise 9.14

Consider the linear model with the quadratic transform in two dimensions. So, the hypothesis set contains functions of the form

$$h(\mathbf{x}) = \text{sign}(w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2).$$

Determine constraints on $\mathbf{w}$ so that:

(a) $h(\mathbf{x})$ is monotonically increasing in $\mathbf{x}$.

(b) $h(\mathbf{x})$ is invariant under an arbitrary rotation of $\mathbf{x}$.

(c) The positive set $\{\mathbf{x} : h(\mathbf{x}) = +1\}$ is convex.

Since the hint is a known property of the target, throwing out hypotheses that do not match the hint should not diminish your ability to fit the target function (or the data). Unfortunately, this need not be the case if the target function was not in your hypothesis set to start with. The simple example below explains why. The figure shows a hypothesis set with three hypotheses (blue) and the target function (red) which is *known* to be monotonically increasing.

$$\mathcal{H} = \{h^*, h_1, h_2\}$$

Though f is monotonic, the best approximation to f in $\mathcal{H}$ is h^*, and h^* is not monotonic. So, if you remove all non-monotonic hypotheses from $\mathcal{H}$, all the remaining hypotheses are bad and you will underfit heavily.

The best approximation to f in $\mathcal{H}$ may not satisfy your hint.

When you outright enforce the hint, you certainly ensure that the final hypothesis has the property that f is known to have. But you also may exaggerate deficiencies that were in the hypothesis set to start with. Strictly enforcing a hint is like a hard constraint, and we have seen an example of this before when we studied regularization and the hard-order constraint in Chapter 4: we 'softened' the constraint, thereby giving the algorithm some flexibility to fit the data while still pointing the learning toward simpler hypotheses. A similar situation holds for hints. It is often better to inform the learning about the hint but allow the flexibility to violate the hint a little if that is warranted for fitting the data - that is, to softly enforce the hint. A popular and general way to softly enforce a hint is using *virtual examples*.

9.3.1 Virtual Examples

Hints are useful because they convey additional information that may not be represented in the finite data set. Virtual examples augment the data set so that the hint becomes represented in the data. The learning can then just focus on this augmented data without needing any new machinery to deal explicitly with the hint. The learning algorithm, by trying to fit all the data (the real and virtual examples) can trade-off satisfying the known hint about f (by attempting to fit the virtual examples) with approximating f (by attempting to fit the real examples). In contrast, when you explicitly constrain the model, you cannot take advantage of this trade-off; you simply do the best you can to fit the data while obeying the hint. If by slightly violating the hint you can get a hypothesis that is a supreme fit to the data, it's just too bad, because you don't have access to such a hypothesis. Let's set aside the ideological goal that known properties of f must be preserved and focus on the error (both in and out-of-sample); that is all that matters.

The general approach to constructing virtual examples is to imagine numerically testing if a hypothesis h satisfies the known hint. This will suggest how to build an error function that would equal zero if the hint is satisfied and non-zero values will quantify the degree to which the hint is not satisfied. Let's see how to do this by concrete example.

Invariance Hint. Let's begin with a simple example, the symmetry hint which says that the target function is symmetric: $f(\mathbf{x}) = f(-\mathbf{x})$. How would we test that a particular h has symmetry? Generate a set of arbitrary virtual pairs $\{\mathbf{v}_m, -\mathbf{v}_m\}$, $m = 1, \ldots, M$ ($\mathbf{v}$ for virtual), and test if $h(\mathbf{v}_m) = h(\mathbf{v}'_m)$ for every virtual pair. Thus, we can define the hint error

$$E_{\text{hint}}(h) = \frac{1}{M} \sum_{m=1}^{M} (h(\mathbf{v}_m) - h(-\mathbf{v}_m))^2.$$

Certainly, for the target function, $E_{\text{hint}}(f) = 0$. By allowing for non-zero E_{hint}, we can allow the possibility of slightly violating symmetry, provided that it gives us a much better fit of the data. So, define an augmented error

$$E_{\text{aug}}(h) = E_{\text{in}}(h) + \lambda E_{\text{hint}}(h) \tag{9.3}$$

to quantify the trade-off between enforcing the hint and fitting the data. By minimizing E_{aug}, we are not explicitly enforcing the hint, but encouraging it. The parameter λ controls how much we emphasize the hint over the fit. This looks suspiciously like regularization, and we will say more on this later.

How should one choose λ? The hint conveys information to help the learning. However, overemphasizing the hint by picking too large a value for λ can lead to underfitting just as over-regularizing can. If λ is too large, it amounts to strictly enforcing the hint, and we already saw how that can hurt. The answer: use validation to select λ. (See Chapter 4 for details on validation.)

How should one select virtual examples $\{\mathbf{v}_m, -\mathbf{v}_m\}$ to compute the hint error? As a start, use the real examples themselves. A virtual example for the symmetry hint would be $\{\mathbf{x}_n, -\mathbf{x}_n\}$, for $n = 1, \ldots, N$.

Exercise 9.15

Why might it be a good idea to use the data points to compute the hint error? When might it be better to use more hint examples?

[Hint ☺: Do you care if a hypothesis violates the hint in a part of the input space that has very low probability?]

A general invariance partitions the input space into disjoint regions,

$$\mathcal{X} = \cup_\alpha \mathcal{X}_\alpha.$$

Within any partition, the target function is constant. That is, if $\mathbf{x}, \mathbf{x}' \in \mathcal{X}_\alpha$ then $f(\mathbf{x}) = f(\mathbf{x}')$. A set of transformations $\mathcal{T}$ can be used to implicitly define the regions in the partition: $\mathbf{x}$ and $\mathbf{x}'$ are in the same partition $\mathcal{X}_\alpha$ if and only if $\mathbf{x}'$ can be obtained from $\mathbf{x}$ by applying a composition of transformations from $\mathcal{T}$. As an example, suppose $f(\mathbf{x})$ is invariant under arbitrary rotation of the input $\mathbf{x}$ in the plane. Then the invariant set $\mathcal{X}_r$ is the circle of radius r centered on the origin. For any two points on the same circle, the target function evaluates to the same value. To generate the virtual examples, take each data point $\mathbf{x}_n$ and determine (for example randomly) a virtual example $\mathbf{x}'_n$ that is in the same invariant set as $\mathbf{x}_n$. The hint error would then be

$$E_{\text{hint}}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n) - h(\mathbf{x}'_n))^2.$$

Notice that we can always compute the hint error, even though we do not know the target function on the virtual examples.

Monotonicity Hint. To test if h is monotonic, for each data point $\mathbf{x}_n$ we construct $\mathbf{x}'_n = \mathbf{x}_n + \Delta\mathbf{x}_n$ with $\Delta\mathbf{x}_n \geq \mathbf{0}$. Monotonicity implies that $h(\mathbf{x}'_n) \geq h(\mathbf{x}_n)$. We can thus construct a hint error of the form

$$E_{\text{hint}}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n) - h(\mathbf{x}'_n))^2[[h(\mathbf{x}'_n) < h(\mathbf{x}_n)]].$$

Each term in the hint error penalizes violation of the hint, and is set to be zero if the monotonicity is not violated at $\mathbf{x}_n$, regardless of the value of $(h(\mathbf{x}_n) - h(\mathbf{x}'_n))^2$.

Exercise 9.16

Give hint errors for rotational invariance, convexity and perturbation hints.

9.3.2 Hints Versus Regularization

Regularization (Chapter 4) boiled down to minimizing an augmented error

$$E_{\text{aug}}(h) = E_{\text{in}}(h) + \frac{\lambda}{N}\Omega(h),$$

where the regularizer $\Omega(h)$ penalized the 'complexity' of h. Implementing a hint by minimizing an augmented error as in (9.3) looks like regularization with regularizer $E_{\text{hint}}(h)$. Indeed the two are similar, but we would like to highlight the difference. Regularization directly combats the effects of noise (stochastic and deterministic) by encouraging a hypothesis to be simpler: the primary goal is to reduce the variance, and the price is usually a small increase in bias. A hint helps us choose a small hypothesis set that is likely to contain the target function (see Exercise 9.12): the primary motivation is to reduce the bias. By implementing the hint using a penalty term $E_{\text{hint}}(h)$, we will be reducing the variance but there will usually be no price to pay in terms of higher bias. Indirectly, the effect is to choose the right smaller hypothesis set.

> Regularization fights noise by pointing the learning toward simpler hypotheses; this applies to any target function. Hints fight bias by helping us choose wisely from among small hypothesis sets; a hint can hurt if it presents an incorrect bit of information about the target.

As you might have guessed, you can use both tools and minimize

$$E_{\text{aug}}(h) = E_{\text{in}}(h) + \frac{\lambda_1}{N}\Omega(h) + \lambda_2 E_{\text{hint}}(h),$$

thereby combating noise while at the same time incorporating information about the target function. It is possible for the two tools to send the learning in different directions, but this is rare (for example a hint that the target is a 10th order polynomial may get overridden by the regularizer which tells you there is not enough data to learn a 10th order polynomial). It generally pays huge dividends to incorporate properties of the target function, and regularization is a must because there is always noise.

9.4 Data Cleaning

Although having more data is often a good idea, sometimes less is more. Data cleaning attempts to identify and remove noisy or hard examples. A noisy 'outlier' example can seriously lead learning astray (stochastic noise). A complex data point can be just as bad (deterministic noise). To teach a two year old mathematics, you may start with induction or counting $1, 2, 3$. Both are correct 'data points', but the complex concepts of induction will only confuse the child (a simple learner).

Why remove the noisy data as opposed to simply incurring in-sample error on them? You don't just incur in-sample error on the noisy data. You incur

additional, unwanted in-sample error on the non-noisy data when the learning is led astray. When you remove the noisy data, you have fewer data points to learn from, but you will gain because these fewer data points have the useful information. The following regression example helps to illustrate.

(a) Fit led astray by outlier (b) Fit without outlier

When we retain the outlier point (red), the linear fit is distorted (even the sign of the slope changes). The principle is not hard to buy into: throw away detrimental data points. The challenge is to identify these detrimental data points. We discuss two approaches that are simple and effective.

9.4.1 Using a Simpler Model to Identify Noisy Data

The learning algorithm sees everything through the lens of the hypothesis set. If your hypothesis set is complex, then you will be able to fit a complex data set, so no data point will look like noise to you. On the other hand, if your hypothesis set is simple, many of the data points could look like noise. We can identify the hard examples by viewing the data through a slightly simpler model than the one you will finally use. One typical choice of the simpler model is a linear model. Data that the simpler model cannot classify are the hard examples, to be disregarded in learning with the complex model. It is a bit of an art to decide on a model that is simple enough to identify the noisy data, but not too simple that you throw away good data which are useful for learning with the more complex final model.

Rather than relying solely on a single simpler hypothesis to identify the noisy data points, it is generally better to have several instances of the simpler hypothesis. If a data point is misclassified by a majority of the simpler hypotheses, then there is more evidence that the data point is hard. One easy way to generate several such simpler hypotheses is to use the same hypothesis set (e.g. linear models) trained on different data sets generated by sampling data points from the original data with replacement (this is called Bootstrap-sampling from the original data set). Example 9.3 illustrates this technique with the digits data from Example 3.1.

Example 9.3. We randomly selected 500 examples of digits data $\mathcal{D}$, and generated more than 10,000 data sets of size 500. To generate a data set, we sampled data points from $\mathcal{D}$ with replacement (Bootstrap sampling).Each bootstrapped data set is used to construct a 'simple' hypothesis using the k-Nearest Neighbor rule for a large k. Each simple hypothesis misclassifies some of the data that was used to obtain that hypothesis. If a particular data point is misclassified by more than half the hypotheses which that data point influenced, we identify that data point as bad (stochastic or deterministic noise). Figure 9.5 shows the bad data for two different choices of the 'simple' model: (a) 5-NN; and (b) 101-NN. The simpler model (101-NN) identifies more con-

(a) 5-Nearest Neighbor (b) 101-Nearest Neighbor

Figure 9.5: Identifying noisy data using the 'simple' nearest neighbor rule. The data in black boxes are identified as noisy (a) using 5-Nearest Neighbor and (b) using 101-Nearest Neighbor. 101-Nearest Neighbor, the simpler of the two models, identifies more noisy data as expected.

fusing data points, as expected. The results depend on our choice of k in the k-NN algorithm. If k is large (generating a constant final hypothesis), then all the '+1'-points will be identified as confusing, because the '−1'-points are in the majority. In Figure 9.5, the points identified as confusing are intuitively so – solitary examples of one class inside a bastion of the other class.

If we choose, for our final classifier, the 'complex' 1-Nearest Neighbor rule, Figure 9.6(a) shows the classifier using all the data (a reproduction of Figure 6.2(a) in Chapter 6). A quick look at the decision boundary for the same 1-Nearest Neighbor rule after removing the bad data (see Figure 9.6(b)) visually confirms that the overfitting is reduced. What matters is the test error, which dropped from 1.7% to 1.1%, a one-third reduction in test error rate.

If computational resources permit, a refinement of this simple and effective method is to remove the noisy points sequentially: first remove the most confusing data point; now rerun the whole process to remove the next data

(a) All data, $E_{\text{test}} = 1.7\%$ (b) Cleaned data, $E_{\text{test}} = 1.1\%$

Figure 9.6: The 1-Nearest Neighbor classifier (a) using all the data and (b) after removing the bad examples as determined using 101-Nearest Neighbor. Removing the bad data gives a more believable decision boundary.

point and so on. The advantage of this sequential process is that a slightly confusing example may be redeemed after the true culprit is removed.

In our example, 101-Nearest Neighbor was the 'simple' model, and an example was bad if half the simple hypotheses misclassified that example. The simpler the model, the more outliers it will find. A lower threshold on how often an example is misclassified also results in more outliers. These are implementation choices, and we're in the land of heuristics here. In practice, a slightly simpler model than your final hypothesis and a threshold of 50% for the number of times an example is misclassified are reasonable choices. □

9.4.2 Computing a Validation Leverage Score

If, after removing an example from your data set, you can improve your test error, then by all means remove the example. Easier said than done – how do you know whether the test error will improve when you remove a particular data point? The answer is to use validation to estimate the adverse impact (or leverage) that a data point has on the final hypothesis g. If an example has large leverage with negative impact, then it is a very 'risky' example and should be disregarded during learning.

Let's first define leverage and then see how to estimate it. Recall that using data set $\mathcal{D}$, the learning produces final hypothesis g. When we discussed validation in Chapter 4, we introduced the data set $\mathcal{D}_n$ which contained all the data in $\mathcal{D}$ except $(\mathbf{x}_n, y_n)$. We denoted by g_n^- the final hypothesis that you get from from $\mathcal{D}_n$. We denote the *leverage score* of data point $(\mathbf{x}_n, y_n)$ by

$$\ell_n = E_{\text{out}}(g) - E_{\text{out}}(g_n^-).$$

The leverage score measures how valuable $(\mathbf{x}_n, y_n)$ is. If ℓ_n is large and positive, $(\mathbf{x}_n, y_n)$ is detrimental and should be discarded. To estimate ℓ_n, we need a validation method to estimate E_{out}. Any method for validation can be used. In Chapter 4 we introduced E_{cv}, the cross-validation estimate of $E_{\text{out}}(g)$; E_{cv} is an unbiased estimate of the out-of-sample performance when learning from $N-1$ data points. The algorithm to compute E_{cv} takes as input a data set $\mathcal{D}$ and outputs $E_{\text{cv}}(\mathcal{D})$, an estimate of the out-of-sample performance for the final hypothesis learned from $\mathcal{D}$. Since we get g_n^- from $\mathcal{D}_n$, if we run the cross validation algorithm on $\mathcal{D}_n$, we will get an estimate of $E_{\text{out}}(g_n^-)$. Thus,

$$\ell_n \approx E_{\text{cv}}(\mathcal{D}) - E_{\text{cv}}(\mathcal{D}_n). \tag{9.4}$$

You can replace the E_{cv} algorithm above with any validation algorithm, provided you run it once on $\mathcal{D}$ and once on $\mathcal{D}_n$. The computation of $E_{\text{cv}}(\mathcal{D})$ only needs to be performed once. Meanwhile, we are asking whether each data point in turn is helping. If we are using cross-validation to determine whether a data point is helping, $E_{\text{cv}}(\mathcal{D}_n)$ needs to be computed for $n = 1, \ldots, N$, requiring you to learn $N(N-1)$ times on data sets of size $N-2$. That is a formidable feat. For linear regression, this can be done more efficiently (see Problem 9.21). Example 9.4 illustrates the use of validation to compute the leverage for our toy regression example that appeared earlier.

Example 9.4. We use a linear model to fit the data shown in Figure 9.7(a). There are seven data points, so we need to compute

$$E_{\text{cv}}(\mathcal{D}) \qquad \text{and} \qquad E_{\text{cv}}(\mathcal{D}_1), \ldots, E_{\text{cv}}(\mathcal{D}_7).$$

We show the leverage $\ell_n \approx E_{\text{cv}}(\mathcal{D}) - E_{\text{cv}}(\mathcal{D}_n)$ for $n = 1, \ldots, 7$ in Figure 9.7(b).

(a) Data for linear regression (b) Leverage estimates

Figure 9.7: Estimating the leverage of data points using cross validation. (a) The data points with one outlier (b) The estimated leverage. The outlier has a large positive leverage indicative of a very noisy point.

Only the outlier has huge positive leverage, and it should be discarded. □

9.5 More on Validation

We introduced validation in Chapter 4 as a means to select a good regularization parameter or a good model. We saw in the previous section how validation can be used to identify noisy data points that are detrimental to learning. Validation gives an in-sample estimate for the out-of-sample performance. It lets us certify that the final hypothesis is good, and is such an important concept that many techniques have been developed. Time spent augmenting our tools for validation is well worth the effort, so we end this chapter with a few advanced techniques for validation. Don't fear. The methods are advanced, but the algorithms are simple.

9.5.1 Rademacher Penalties

The Rademacher penalty estimates the optimistic bias of the in-sample error.

$$E_{\text{out}}(g) = E_{\text{in}}(g) + \underbrace{\text{overfit penalty}}_{\text{Rademacher penalty estimates this}}.$$

One way to view the overfit penalty is that it represents the optimism in the in-sample error. You reduced the error all the way to $E_{\text{in}}(g)$. Part of that was real, and the rest was just you fooling yourself. The Rademacher overfit penalty attempts to estimate this optimism.

The intuitive leap is to realize that the optimism in E_{in} is a result of the fitting capability of your hypothesis set $\mathcal{H}$, not because, by accident, your data set was easy to fit. This suggests that if we can compute the optimism penalty for some data set, then we can apply it to other data sets, for example, to the data set $\mathcal{D} = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{\text{N}}, y_{\text{N}})$ that we are given. So let's try to find a data set *for which we can compute the optimism.* And indeed there is one such data set, namely a random one. Consider the data set

$$\mathcal{D}' = (\mathbf{x}_1, r_1), \ldots, (\mathbf{x}_{\text{N}}, r_{\text{N}}),$$

where $r_1, \ldots, r_n$ are generated independently by a random target function, $\mathbb{P}[r_n = +1] = \frac{1}{2}$. The r_n are called Rademacher variables. The inputs are the same as the original data set, so in that sense $\mathcal{D}'$ mimics $\mathcal{D}$. After learning on $\mathcal{D}'$ you produce hypothesis $g_{\mathbf{r}}$ with in-sample error $E'_{\text{in}}(g_{\mathbf{r}})$ (we use the prime $(\cdot)'$ to denote quantities with respect to the random problem). Clearly, $E'_{\text{out}}(g_{\mathbf{r}}) = \frac{1}{2}$, because the target function is random, so the optimism is $\frac{1}{2} - E'_{\text{in}}(g_{\mathbf{r}})$. Using this measure of optimism for our actual problem, we get the Rademacher estimate

$$\hat{E}_{\text{out}}(g) \quad = \quad E_{\text{in}}(g) + \left(\tfrac{1}{2} - E'_{\text{in}}(g_{\mathbf{r}})\right).$$

The Rademacher penalty is easy to compute: generate random targets for your data; perform an in-sample minimization to see how far below $\frac{1}{2}$ you can get the error, and use that as a penalty on your actual in-sample error.

Exercise 9.17

After you perform in-sample minimization to get $g_{\mathbf{r}}$, show that the Rademacher penalty for $E_{\text{in}}(g)$ is given by

$$\tfrac{1}{2} - E'_{\text{in}}(g_{\mathbf{r}}) = \max_{h \in \mathcal{H}} \left\{ \frac{1}{2N} \sum_{n=1}^{N} r_n h(\mathbf{x}_n) \right\},$$

which is proportional to the maximum correlation you can obtain with random signs using hypotheses in the data set.

A better way to estimate the optimism penalty is to compute the expectation over different realizations of the Rademacher variables: $\mathbb{E}_{\mathbf{r}}[\frac{1}{2} - E'_{\text{in}}(g_{\mathbf{r}})]$. In practice, one generates several random Rademacher data sets and takes the average optimism penalty. This means you have to do an in-sample error minimization for each Rademacher data set. The good news is that, with high probability, just a single Rademacher data set suffices.

It is possible to get a theoretical bound for $E_{\text{out}}(g)$ using the Rademacher overfit penalty (just like we did with the VC penalty). The bound is universal (because it holds for any learning scenario – target function and input probability distribution); but, unlike the VC penalty, it is *data dependent*, hence it tends to be tighter in practice. (Remember that the VC bound was independent of the data set because it computed $m_{\mathcal{H}}(N)$ on the worst possible data set of size N.) It is easy to use the Rademacher penalty, since you just need to do a single in-sample error optimization. In this sense the approach is similar to cross-validation, except the Rademacher penalty is computationally cheaper (recall that cross-validation required N in-sample optimizations).

The Rademacher penalty requires in-sample error *minimization* - you compute the worst-case optimism. However, it can also be used with regularization and penalized error minimization, because to a good approximation, penalized error minimization is equivalent to in-sample error minimization using a constrained hypothesis set (see Chapter 4). Thus, when we say "in-sample error minimization," read it as "run the learning algorithm."

9.5.2 The Permutation and Bootstrap Penalties

There are other random data sets for which we can compute the optimism. The permutation and Bootstrap estimates are based off these, and are just as easy to compute using an in-sample error minimization.

Permutation Optimism Penalty. Again, we estimate the optimism by considering how much we can overfit random data, but now we choose the y-values to be a random permutation of the actual y-values in the data, as opposed to random signs. Thus, the estimate easily generalizes to regression as well. The learning problem generated by a random permutation mimics the

actual learning problem more closely since it takes both the $\mathbf{x}$ and y values from the data. Consider a randomly permuted data set,

$$\mathcal{D}_{\boldsymbol{\pi}} = (\mathbf{x}_1, y_{\boldsymbol{\pi}_1}), \ldots, (\mathbf{x}_N, y_{\boldsymbol{\pi}_N}),$$

where $\boldsymbol{\pi}$ is a random permutation of $(1, \ldots, N)$. After in-sample error minimization on this randomly permuted data, you produce $g_{\boldsymbol{\pi}}$ with in-sample error on $\mathcal{D}_{\boldsymbol{\pi}}$ equal to $E_{\text{in}}^{\boldsymbol{\pi}}(g_{\boldsymbol{\pi}})$. Unlike with the Rademacher random learning problem, the out-of-sample error for this random learning problem is not simply $\frac{1}{2}$, because the target values are not random signs. We need to be a little more careful and obtain the expected error for the joint $(\mathbf{x}, y)$ target distribution that describes this random learning problem (see Problem 9.12 for the details). However, the final result is the same, namely that the overfit penalty (optimism on the random problem) is proportional to the correlation between the learned function $g_{\boldsymbol{\pi}}(\mathbf{x}_n)$ and the randomly permuted target values $y_{\boldsymbol{\pi}_n}$.[6] The permutation estimate of the out-of-sample error is

$$\hat{E}_{\text{out}}(g) = E_{\text{in}}(g) + \frac{1}{2N} \sum_{n=1}^{N} (y_{\boldsymbol{\pi}_n} - \bar{y}) g_{\boldsymbol{\pi}}(\mathbf{x}_n), \tag{9.5}$$

where $\bar{y}$ is the mean target value in the data. The second term is the permutation optimism penalty obtained from a single randomly permuted data set. Ideally, you should average the optimism penalty over several random permutations, even though, as with the Rademacher estimate, one can show that a single permuted data set suffices to give a good estimate with high probability. As already mentioned, the permutation estimate can be applied to regression, in which case it is more traditional not to rescale the sum of squared errors by $\frac{1}{4}$ (which is appropriate for classification). In this case, the $\frac{1}{2N}$ becomes $\frac{2}{N}$.

Bootstrap Optimism Penalty. The Rademacher and permutation data sets occupy two extremes. For the Rademacher data set, we choose the y-values as random signs, independent of the actual values in the data. For the permutation data set, we take the y-values directly from the data and randomly permute them - every target value is represented once, and can be viewed as sampling the target values from the data *without* replacement. The Bootstrap optimism penalty generates a random set of targets, one for each input, by sampling y-values in the data, but *with* replacement. Thus, the y-values represent the observed distribution, but not every y-value may be picked. The functional form of the estimate (9.5) is unchanged; one simply replaces all the terms in the overfit penalty with the corresponding terms for the Bootstrapped data set.

The permutation and Rademacher estimates may be used for model selection just as any other method which constructs a proxy for the out-of-sample

[6] Any such correlation is spurious, as no input-output relationship exists.

error. In some cases, as with leave-one-out cross validation, it is possible to compute the average (over the random data sets) of the overfit penalty analytically, which means that one can use the permutation estimate without having to perform any explicit in-sample error optimizations.

Example 9.5 (Permutation Estimate for Linear Regression with Weight Decay)**.** For linear models, we can compute the permutation estimate without explicitly learning on the randomly permuted data (see Problem 9.18 for details). The estimate for E_{out} is

$$\begin{aligned}\hat{E}_{\text{out}}(g) &= E_{\text{in}}(g) + \mathbb{E}_{\boldsymbol{\pi}}\left[\frac{2}{N}\sum_{n=1}^{N}(y_{\boldsymbol{\pi}_n} - \bar{y})g_{\boldsymbol{\pi}}(\mathbf{x}_n)\right] \\ &= E_{\text{in}}(g) + \frac{2\hat{\sigma}_y^2}{N}\left(\text{trace}(\text{H}) - \frac{\mathbf{1}^{\text{T}}\text{H}\mathbf{1}}{N}\right), \qquad (9.6)\end{aligned}$$

where $\hat{\sigma}_y^2 = \frac{N}{N-1}\mathsf{var}(y)$ is the unbiased estimate for the variance of the target values and $\text{H} = \text{X}(\text{X}^{\text{T}}\text{X} + \frac{\lambda}{N}\text{I})^{-1}\text{X}^{\text{T}}$ is the hat-matrix that depends only on the input data. When there is no regularization, $\lambda = 0$, so H is a projection matrix (projecting onto the columns of X), and $\text{trace}(\text{H}) = d + 1$. If the first column of X is a column of ones (the constant term in the regression), then $\text{H}\mathbf{1} = \mathbf{1}$ and the permutation estimate becomes

$$\hat{E}_{\text{out}}(g) = E_{\text{in}}(g) + \frac{2\hat{\sigma}_y^2 d}{N}.$$

This estimate is similar to the AIC estimate from information theory. It is also reminiscent of the test error in Exercise 3.4 which resulted in

$$E_{\text{out}}(g) = E_{\text{in}}(g) + \frac{2\sigma^2(d+1)}{N},$$

where σ^2 was the noise variance. The permutation estimate uses $\hat{\sigma}_y^2$ in place of σ^2. Observe that $\text{trace}(\text{H}) - \frac{1}{N}\mathbf{1}^{\text{T}}\text{H}\mathbf{1}$ plays the role of an effective dimension, d_{eff}. Problem 4.13 in Chapter 4 suggests some alternative choices for an effective dimension. □

9.5.3 Rademacher Generalization Bound

The theoretical justification for the Rademacher optimism penalty is that, as with the VC bound, it can be used to bound E_{out}. A similar though mathematically more complex justification can also be shown for the permutation estimate. The mathematical proof of this is interesting in that it presents new tools for analyzing generalization error. An energetic reader may wish to master these techniques by working through Problem 9.16. We present the result for a hypothesis set that is closed under negation ($h \in \mathcal{H}$ implies $-h \in \mathcal{H}$).

Theorem 9.6 (Rademacher Bound)**.** With probability at least $1-\delta$,

$$E_{\text{out}}(g) \le E_{\text{in}}(g) + \max_{h\in\mathcal{H}}\left\{\frac{1}{N}\sum_{n=1}^{N} r_n h(\mathbf{x}_n)\right\} + O\left(\sqrt{\frac{1}{N}\log\frac{1}{\delta}}\right).$$

The probability is with respect to the data set and a single realization of the Rademacher variables.

The overfit penalty in the bound is twice the Rademacher optimism penalty (we can ignore the third term which does not depend on the hypothesis set and is small). The bound is similar to the VC bound in that it holds for any target function and input distribution. Unlike the VC bound, it depends on the data set and can be significantly tighter than the VC bound.

9.5.4 Out-of-Sample Error Estimates for Regression

We close this chapter with a discussion of validation methods for regression, and a model selection experiment in the context of regression, to illustrate some of the issues to think about when using validation for model selection. Cross-validation and the permutation optimism penalty are general in that they can be applied to regression and only require in-sample minimization. They also require no assumptions on the data distribution to work. We call these types of approaches sampling approaches because they work using samples from the data.

The VC bound only applies to classification. There is an analog of the VC bound for regression. The statistics community has also developed a number of estimates. The estimates are multiplicative in form, so $E_{\text{out}} \approx (1+\Omega(p))E_{\text{in}}$, where $p = \frac{N}{d_{\text{eff}}}$ is the number of data points per effective degree of freedom (see Example 9.5 for one estimate of the effective degrees of freedom in a linear regression setting).

VC penalty factor	$E_{\text{out}} \le \dfrac{\sqrt{p}}{\sqrt{p} - \sqrt{1+\ln p + \frac{\ln N}{2d_{\text{eff}}}}} E_{\text{in}}$
Akaike's FPE	$E_{\text{out}} \approx \dfrac{p+1}{p-1} E_{\text{in}}$
Schwartz criterion	$E_{\text{out}} \approx \left(1 + \dfrac{\ln N}{p-1}\right) E_{\text{in}}$
Craven & Wahba's GCV	$E_{\text{out}} \approx \dfrac{p^2}{(p-1)^2} E_{\text{in}}$

For large p, FPE (final prediction error) and GCV (generalized cross validation) are similar: $E_{\text{out}} = (1 + 2p^{-1} + O(p^{-2}))E_{\text{in}}$. This suggests that the simpler estimator $E_{\text{out}} = (1 + \frac{2d_{\text{eff}}}{N})E_{\text{in}}$ might be good enough, and, from the practical point of view, it is good enough.

The statistical estimates (FPE, Schwartz and GCV) make model assumptions and are good asymptotic estimates modulo those assumptions. The VC penalty is more generally valid. However, it tends to be very conservative. In practice, the VC penalty works quite well, as we will see in the experiments.

Validation Experiments with Regression. The next exercise sets up an experimental design for studying validation and model selection. We consider two model selection problems. The first is to determine the right order of the polynomial to fit the data. In this case, we have models $\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2, \ldots$, corresponding to the polynomials of degree $0, 1, 2, \ldots$. The model selection task is to select one of these models. The second is to determine the right value of the regularization parameter λ in the weight decay regularizer. In this case, we fix a model (say) $\mathcal{H}_Q$ and minimize the augmented error $E_{\text{aug}}(\mathbf{w}) = E_{\text{in}}(\mathbf{w}) + \frac{\lambda}{N}\mathbf{w}^{\text{T}}\mathbf{w}$; we have models $\mathcal{H}_Q(\lambda_1), \mathcal{H}_Q(\lambda_2), \mathcal{H}_Q(\lambda_3), \ldots$, corresponding to the choices $\lambda_1, \lambda_2, \lambda_3, \ldots$ for the regularization parameter. The model selection task is to select one of these models, which corresponds to selecting the appropriate amount of regularization.

Exercise 9.18 [Experimental Design for Model Selection]

Use the learning set up in Exercise 4.2.

(a) **Order Selection.** For a single experiment, generate $\sigma^2 \in [0, 1]$ and the target function degree in $\{0, \ldots, 30\}$. Set $N = 100$ and generate a data set $\mathcal{D}$. Consider the models up to order 20: $\mathcal{H}_0, \ldots, \mathcal{H}_{20}$. For these models, obtain the final hypotheses $g_0, \ldots, g_{20}$ and their test errors $E_{\text{out}}^{(1)}, \ldots, E_{\text{out}}^{(20)}$. Now obtain estimates of the out-of-sample errors, $\hat{E}_{\text{out}}^{(1)}, \ldots, \hat{E}_{\text{out}}^{(20)}$, using the following validation estimates:

VC penalty; LOO-CV; Permutation estimate; FPE.

Plot the out-of-sample error and the validation estimates versus the model order, averaged over many experiments.

(b) λ **Selection.** Repeat the previous exercise to select the regularization parameter. Fix the order of the model to $Q = 5$. Randomly generate $\sigma^2 \in [0, 1]$ and the order of the target from $\{0, \ldots, 10\}$, and set $N = 15$. Use different models with $\lambda \in [0, 300]$.

(c) For model selection, you do not need a perfect error estimate. What is a weaker, but sufficient, requirement for a validation estimate to give good model selection?

(d) To quantify the quality of an error estimate, we may use it to select a model and compute the actual out-of-sample error of the selected model, versus the model with minimum out-of-sample error (after training with the same data). The regret is the relative increase in error incurred by using the selected model versus the optimum model. Compare the regret of different validation estimates, taking the average over many runs.

(a) E_{out} estimate vs. model order (b) E_{out} estimate vs. λ

Figure 9.8: Validation estimates of E_{out}. (a) polynomial models of different order Q; (b) models with different regularization parameter λ. The estimates and E_{out} are averaged over many runs; the 3 validation methods shown are E_{CV} (leave-one-out cross validation estimate), E_{perm} (permutation estimate), and E_{VC} (VC penalty bound). The VC penalty bound is *far* above the plot because it is so loose (indicated by the arrow); we have shifted it down to show its shape. The VC estimate is not useful for estimating E_{out} but can be useful for model selection.

The average performance of validation estimates of E_{out} from our implementation of Exercise 9.18 are shown in Figure 9.8. As can be seen from the figure, when it comes to estimating the out-of-sample error, the cross validation estimate is hands down favorite (on average). The cross validation estimate has a systematic bias because it is actually estimating the out-of-sample error when learning with $N-1$ examples, and the actual out-of-sample error with N examples will be lower. The out-of-sample error is asymmetric about its minimum – the price paid for overfitting increases more steeply than the price paid for underfitting. As can be seen by comparing where the minimum of these curves lies, the VC estimate is the most conservative. The VC overfit penalty strongly penalizes complex models, so that the optimum model (according to the VC estimate) is much simpler.

We now compare the different validation estimates for model selection (Exercise 9.18(d)). We look at the *regret*, the average percentage increase in E_{out} from using an E_{out}-estimate to pick a model versus picking the optimal model. These results are summarized in the table of Figure 9.9. The surprising thing is that the permutation estimate and the VC estimate seem to dominate, even though cross validation gives the best estimate of E_{out} on average. This has to do with the asymmetry of E_{out} around the best model. It pays to be conservative, and err on the side of the simpler model (underfitting) than on the side of the more complex model (overfitting), because the price paid for overfitting is far steeper than the price paid for underfitting. The permutation estimate tends to underfit, and the VC estimate even more so; these estimates are paying the 'small' price for underfitting most of the time. On the other hand,

E_{out} Estimate	Order Selection		λ Selection	
	Regret	Avg. Order	Regret	Avg. λ
E_{out}	0	10.0	0	7.93
E_{in}	43268	20.0	117	0
E_{CV}	540	9.29	18.8	23.1
E_{perm}	**185**	**7.21**	5.96	9.57
E_{FPE}	9560	11.42	51.3	18.1
E_{VC}	508	5.56	**3.50**	**125**

Figure 9.9: Using estimates of E_{out} model selection (Exercise 9.18(d)). The best validation estimate is highlighted in bold. For order selection, the order $Q \in \{0, \ldots, 20\}$, and for regularization $\lambda \in [0, 300]$. All results are averaged over many thousands of experiments. (Regret $= \frac{E_{\text{out}}(g) - E_{\text{out}}(\text{optimal})}{E_{\text{out}}(\text{optimal})}$.)

cross validation picks the correct model most of the time, but occasionally goes for too complex a model and pays a very steep price. In the end, the more conservative strategy wins. Using E_{in} is always the worst, no surprise.

You always need regularization, so $\lambda = 0$ is bad. Suppose we remove that case from our set of available models. Now repeat the experiment for selecting the λ in the range $[0.1, 300]$, as opposed to the range $[0, 300]$, a very minor difference in the available choices for λ. The results become

$\lambda \in [0.1, 300]$	E_{out}	E_{in}	E_{CV}	E_{perm}	E_{FPE}	E_{VC}
Regret	0	1.81	0.44	**0.39**	0.87	0.42

The permutation estimate is best. Cross validation is now much better than before. The in-sample error is still by far the worst. Why did all methods get much better? Because we curtailed their ability to err in favor of too complex a model. What we were experiencing is 'overfitting' during the process of model selection, and by removing the choice $\lambda = 0$, we 'regularized' the model selection process. The cross validation estimate is the most accurate, but also the most sensitive, and has a high potential to be led astray, thus it benefited most from the regularization of the model selection. The FPE estimate is a statistical estimate. Such statistical estimates typically make model assumptions and are most useful in the large N limit, so their applicability in practice may be limited. The statistical estimate fares okay, but, for this experiment, it is not really competitive with the permutation approach, cross validation, or the VC penalty.

Our recommendation. Do not arbitrarily pick models to select among. One can 'overfit' during model selection. Carefully decide on a set of models and use a robust validation method to select one. There is no harm in using multiple methods for model selection, and we recommend both the permutation estimate (robust, easy to compute and generally applies) and cross validation (general, easy to use and usually the most accurate for estimating E_{out}).

9.6 Problems

Problem 9.1 Consider data $(0, 0, +1), (0, 1, +1), (5, 5, -1)$ (in 2-D), where the first two entries are x_1, x_2 and the third is y.

(a) Implement the nearest neighbor method on the raw data and show the decision regions of the final hypothesis.

(b) Transform to whitened coordinates and run the nearest neighbor rule, showing the final hypothesis in the original space.

(c) Use principal components analysis and reduce the data to 1 dimension for your nearest neighbor classifier. Again, show the decision regions of the final hypothesis in the original space.

Problem 9.2 We considered three forms of input preprocessing: centering, normalization and whitening. The goal of input processing is to make the learning robust to unintentional choices made during data collection.

Suppose the data are $\mathbf{x}_n$ and the transformed vectors $\mathbf{z}_n$. Suppose that during data collection, $\mathbf{x}'_n$, a mutated version of $\mathbf{x}_n$, were measured, and input preprocessing on $\mathbf{x}'_n$ produces $\mathbf{z}'_n$. We would like to study when the $\mathbf{z}'_n$ would be the same as the $\mathbf{z}_n$. In other words, what kinds of mutations can the data vectors be subjected to without changing the result of your input processing. The learning will be robust to such mutations.

Which input processing methods are robust to the following mutations:

(a) Bias: $\mathbf{x}'_n = \mathbf{x}_n + \mathbf{b}$ where $\mathbf{b}$ is a constant vector.

(b) Uniform scaling: $\mathbf{x}'_n = \alpha\mathbf{x}_n$, where $\alpha > 0$ is a constant.

(c) Scaling: $\mathbf{x}'_n = \mathrm{A}\mathbf{x}_n$ where A is a diagonal non-singular matrix.

(d) Linear transformation: $\mathbf{x}'_n = \mathrm{A}\mathbf{x}_n$ where A is a non-singular matrix.

Problem 9.3 Let Σ be a symmetric positive definite matrix with eigen-decomposition $\Sigma = \mathrm{U}\Gamma\mathrm{U}^{\mathrm{T}}$ (see the Appendix), where U is orthogonal and Γ is positive diagonal. Show that

$$\Sigma^{\frac{1}{2}} = \mathrm{U}\Gamma^{\frac{1}{2}}\mathrm{U}^{\mathrm{T}} \qquad \text{and} \qquad \Sigma^{-\frac{1}{2}} = \mathrm{U}\Gamma^{-\frac{1}{2}}\mathrm{U}^{\mathrm{T}}.$$

What are $\Gamma^{\frac{1}{2}}$ and $\Gamma^{-\frac{1}{2}}$?

Problem 9.4 If A and V are orthonormal bases, and $\mathrm{A} = \mathrm{V}\psi$, show that $\psi = \mathrm{V}^{\mathrm{T}}\mathrm{A}$ and hence that ψ is an orthogonal matrix.

Problem 9.5 Give the SVD of matrix A defined in each case below.

(a) A is a diagonal matrix.

(b) A is a matrix with pairwise orthogonal rows.

(c) A is a matrix with pairwise orthogonal columns.

(d) Let A have SVD $\mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}$ and $\mathrm{Q}^{\mathrm{T}}\mathrm{Q} = \mathrm{I}$. What is the SVD of QA.

(e) A has blocks A_i along the diagonal, where A_i has SVD $\mathrm{U}_i\Gamma_i\mathrm{V}_i^{\mathrm{T}}$,

$$\mathrm{A} = \begin{bmatrix} \boxed{\mathrm{A}_1} & & & \mathbf{0} \\ & \boxed{\mathrm{A}_2} & & \\ & & \ddots & \\ \mathbf{0} & & & \boxed{\mathrm{A}_m} \end{bmatrix}.$$

Problem 9.6 For the digits data, suppose that the data were not centered first in Example 9.2. Perform PCA and obtain a 2-dimensional feature vector (give a plot). Are the transformed data whitened?

Problem 9.7 Assume that the input matrix X is centered, and construct the reduced feature vector $\mathbf{z}_n = \Gamma_k^{-1}\mathrm{V}_k^{\mathrm{T}}\mathbf{x}_n$ where V_k is the matrix of top-k right singular vectors of X. Show that the feature vectors $\mathbf{z}_n$ are whitened.

Problem 9.8 One critique of PCA is that it selects features without regard to target values. Thus, the features may not be so useful in solving the learning problem, even though they represent the input data well.

One heuristic to address this is to do a PCA separately for each class, and use the top principal direction for each class to construct the features. Use the digits data. Construct a two dimensional feature as follows.

1: Center all the data.
2: Perform PCA on the +1 data. Let $\mathbf{v}_1$ be the top principal direction.
3: Perform PCA on the -1 data. Let $\mathbf{v}_1$ be the top principal direction.
4: Use $\mathbf{v}_1$ and $\mathbf{v}_2$ to obtain the features

$$z_1 = \mathbf{v}_1^{\mathrm{T}}\mathbf{x} \qquad \text{and} \qquad z_2 = \mathbf{v}_2^{\mathrm{T}}\mathbf{x}.$$

We applied the algorithm to the digits data, giving the features shown above.

(a) Give a scatter plot of your resulting two features.

(b) Are the directions $\mathbf{v}_1$ and $\mathbf{v}_2$ necessarily orthogonal?

(c) Let $\mathrm{Z} = \mathrm{X}\hat{\mathrm{V}}$ be the feature matrix, where $\hat{\mathrm{V}} = [\mathbf{v}_1, \mathbf{v}_2]$. Show that the best reconstruction of X from Z is

$$\hat{\mathrm{X}} = \mathrm{Z}\hat{\mathrm{V}}^\dagger = \mathrm{X}\hat{\mathrm{V}}\hat{\mathrm{V}}^\dagger,$$

where $\hat{\mathrm{V}}^\dagger$ is the pseudo-inverse of$\hat{\mathrm{V}}$.

(d) Is this method of constructing features supervised or unsupervised?

Problem 9.9 Let $\mathrm{X} = \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}$ be the SVD of X. Give an alternative proof of the Eckart-Young theorem by showing the following steps.

(a) Let $\hat{\mathrm{X}}$ be a reconstruction of X whose rows are the data points in X projected onto k basis vectors. What is the rank($\hat{\mathrm{X}}$)?

(b) Show that $\|\mathrm{X} - \hat{\mathrm{X}}\|_F^2 \geq \|\mathrm{U}^{\mathrm{T}}(\mathrm{X} - \hat{\mathrm{X}})\mathrm{V}\|_F^2 = \|\Gamma - \mathrm{U}^{\mathrm{T}}\hat{\mathrm{X}}\mathrm{V}\|_F^2$.

(c) Let $\hat{\Gamma} = \mathrm{U}^{\mathrm{T}}\hat{\mathrm{X}}\mathrm{V}$. Show that rank($\hat{\Gamma}$) $\leq k$.

(d) Argue that the optimal choice for $\hat{\Gamma}$ must have all off-diagonal elements zero. *[Hint: What is the definition of the Frobenius norm?]*

(e) How many non-zero diagonals can $\hat{\Gamma}$ have.

(f) What is the best possible choice for $\hat{\Gamma}$.

(g) Show that $\hat{\mathrm{X}} = \mathrm{X}\mathrm{V}\mathrm{V}^{\mathrm{T}}$ results in such an optimal choice for $\hat{\Gamma}$.

Problem 9.10 **Data Snooping with Input Preprocessing.** You are going to run PCA on your data X and select the top-k PCA features. Then, use linear regression with these k features to produce your final hypothesis. You want to estimate E_{out} using cross validation. Here are two possible algorithms for getting a cross-validation error.

Algorithm 1

1: SVD: $[\mathrm{U}, \Gamma, \mathrm{V}] = \text{svd}(\mathrm{X})$.
2: Get features $\mathrm{Z} = \mathrm{X}\mathrm{V}_k$
3: **for** $n = 1 : N$ **do**
4: Leave out $(\mathbf{z}_n, y_n)$ to obtain data $\mathrm{Z}_n, \mathbf{y}_n$.
5: Learn $\mathbf{w}_n^-$ from $\mathrm{Z}_n, \mathbf{y}_n$
6: Error $\mathrm{e}_n = (\mathbf{x}_n^{\mathrm{T}}\mathbf{w}_n^- - y_n)^2$.
7: $E_1 = \text{average}(\mathrm{e}_1, \ldots, \mathrm{e}_N)$.

Algorithm 2

1: **for** $n = 1 : N$ **do**
2: Leave out $(\mathbf{x}_n, y_n)$ to obtain data $\mathrm{X}_n, \mathbf{y}_n$.
3: SVD: $[\mathrm{U}^-, \Gamma^-, \mathrm{V}^-] = \text{svd}(\mathrm{X}_n)$.
4: Get features $\mathrm{Z}_n = \mathrm{X}_n\mathrm{V}_k^-$.
5: Learn $\mathbf{w}_n^-$ from $\mathrm{Z}_n, \mathbf{y}_n$.
6: Error $\mathrm{e}_n = (\mathbf{x}_n^{\mathrm{T}}\mathrm{V}_2^-\mathbf{w}_n^- - y_n)^2$
7: $E_2 = \text{average}(\mathrm{e}_1, \ldots, \mathrm{e}_N)$.

In both cases, your final hypothesis is $g(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathrm{V}_k\mathbf{w}$ where $\mathbf{w}$ is learned from $\mathrm{Z} = \mathrm{X}\mathrm{V}_k$ form Algorithm 1 and $\mathbf{y}$. (Recall V_k is the matrix of top-k singular singular vectors.) We want to estimate $E_{\text{out}}(g)$ using either E_1 or E_2.

(a) What is the difference between Algorithm 1 and Algorithm 2?

(b) Run an experiment to determine which is a better estimate of $E_{\text{out}}(g)$.

(i) Set the dimension $d = 5$, the number of data points $N = 40$ and $k = 3$. Generate random target function weights $\mathbf{w}_f$, normally distributed. Generate N random normally distributed d-dimensional inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$. Generate targets $y_n = \mathbf{w}_f^{\mathrm{T}}\mathbf{x}_n + \epsilon_n$ where ϵ_n is independent Gaussian noise with variance 0.5.

(ii) Use Algorithm 1 and 2 to compute estimates of $E_{\text{out}}(g)$.

(iii) Compute $E_{\text{out}}(g)$.

(iv) Report $E_1, E_2, E_{\text{out}}(g)$.

(v) Repeat parts (i)–(iv) 10^6 times and report averages $\bar{E}_1, \bar{E}_2, \bar{E}_{\text{out}}$.

(c) Explain your results from (b) part (v).

(d) Which is the correct estimate for $E_{\text{out}}(g)$.

Problem 9.11 From Figure 9.8(a), performance degrades rapidly as the order of the model increases. What if the target function has a high order? One way to allow for higher order, but still get good generalization is to fix the effective dimension d_{eff}, and try a variety of orders. Given X, d_{eff} depends on λ. Fixing d_{eff} (to say 7) requires changing λ as the order increases.

(a) For fixed d_{eff}, when the order increases, will λ increase or decrease?

(b) Implement a numerical method for finding $\lambda(d_{\text{eff}})$ using one of the measures for the effective number of parameters (e.g., $d_{\text{eff}} = \mathrm{trace}(\mathrm{H}^2(\lambda))$, where $\mathrm{H}(\lambda) = \mathrm{X}(\mathrm{X}^{\mathrm{T}}\mathrm{X}+\lambda\mathrm{I})^{-1}\mathrm{X}^{\mathrm{T}}$ is the hat-matrix from Chapter 4). The inputs are d_{eff}, and X and the output should be $\lambda(d_{\text{eff}}, \mathrm{X})$.

(c) Use the experimental design in Exercise 9.18 to evaluate this approach. Vary the model order in $[0, 30]$ and set d_{eff} to 7. Plot the expected out-of-sample error versus the order.

(d) Comment on your results. How should your plot behave if d_{eff} alone controlled the out-of-sample error? What is the best model order using this approach?

Problem 9.12 In this problem you will derive the permutation optimism penalty in Equation (9.5). We compute the optimism for a particular data distribution and use that to penalize E_{in} as in (9.5). The data is $\mathcal{D} = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, and the permuted data set is

$$\mathcal{D}_{\boldsymbol{\pi}} = (\mathbf{x}_1, y_{\boldsymbol{\pi}_1}), \ldots, (\mathbf{x}_N, y_{\boldsymbol{\pi}_N}),$$

Define the 'permutation input distribution' to be uniform over $\mathbf{x}_1, \ldots, \mathbf{x}_N$. Define a 'permutation target function' $f_{\boldsymbol{\pi}}$ as follows: to compute target values $f_{\boldsymbol{\pi}}(\mathbf{x}_n)$, generate a random permutation $\boldsymbol{\pi}$ and set $f_{\boldsymbol{\pi}}(\mathbf{x}_n) = y_{\boldsymbol{\pi}_n}$. So,

$$\mathbb{P}[f_{\boldsymbol{\pi}}(\mathbf{x}_n) = y_m] = \frac{1}{N},$$

for $m = 1, \ldots, N$, independent of the particular $\mathbf{x}_n$. We define $E_{\text{out}}^{\boldsymbol{\pi}}(h)$ with respect to this target function on the inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$.

(a) Define $E_{\text{out}}^{\boldsymbol{\pi}}(h) = \frac{1}{4}\,\mathbb{E}_{\mathbf{x},\boldsymbol{\pi}}\left[(h(\mathbf{x}) - f_{\boldsymbol{\pi}}(\mathbf{x}))^2\right]$, where expectation is with respect to the permutation input and target joint distribution. Show that

$$E_{\text{out}}^{\boldsymbol{\pi}}(h) = \frac{1}{4N}\sum_{n=1}^{N}\mathbb{E}_{\boldsymbol{\pi}}[(h(\mathbf{x}_n) - f_{\boldsymbol{\pi}}(\mathbf{x}_n))^2].$$

(b) Show that

$$E_{\text{out}}^{\boldsymbol{\pi}}(h) = \frac{s_y^2}{4} + \frac{1}{4N}\sum_{n=1}^{N}(h(\mathbf{x}_n) - \bar{y})^2,$$

where $\bar{y} = \frac{1}{N}\sum_n y_n$ and $s_y^2 = \frac{1}{N}\sum_n (y_n - \bar{y})^2$ are the mean and variance of the target values.

(c) In-sample error minimization on $\mathcal{D}_{\boldsymbol{\pi}}$ yields $g_{\boldsymbol{\pi}}$. What is $E_{\text{in}}^{\boldsymbol{\pi}}(g_{\boldsymbol{\pi}})$?

(d) The permutation optimism penalty is $E_{\text{out}}^{\boldsymbol{\pi}}(g_{\boldsymbol{\pi}}) - E_{\text{in}}^{\boldsymbol{\pi}}(g_{\boldsymbol{\pi}})$. Show:

$$\text{permutation optimism penalty} = \frac{1}{2N}\sum_{n=1}^{N}(y_{\boldsymbol{\pi}_n} - \bar{y})g_{(\boldsymbol{\pi})}(\mathbf{x}_n). \tag{9.7}$$

(e) Show that the permutation optimism penalty is proportional to the correlation between the randomly permuted targets $y_{\boldsymbol{\pi}_n}$ and the learned function $g_{\boldsymbol{\pi}}(\mathbf{x}_n)$.

Problem 9.13 Repeat Problem 9.12, but , instead of defining the target distribution for the random problem using a random permutation (sampling the targets without replacement), use sampling of the targets with replacement (the Bootstrap distribution).
[Hint: Almost everything stays the same.]

Problem 9.14 In this problem you will investigate the Rademacher optimism penalty for the perceptron in one dimension, $h(x) = \text{sign}(x - w_0)$.

(a) Write a program to estimate the Rademacher optimism penalty:
  (i) Generate inputs $x_1, \ldots, x_N$ and random targets $r_1, \ldots, r_N$.
  (ii) Find the perceptron $g_{\mathbf{r}}$ with minimum in-sample error $E'_{\text{in}}(g_{\mathbf{r}})$.
  (iii) Compute the optimism penalty as $\frac{1}{2} - E'_{\text{in}}(g_{\mathbf{r}})$.
  Run your program for $N = 1, 2, \ldots, 10^3$ and plot the penalty versus N.

(b) Repeat part (a) 10,000 times to compute the average Rademacher penalty and give a plot of the penalty versus N.

(c) On the same plot show the function $1/\sqrt{N}$; how does the Rademacher penalty compare to $1/\sqrt{N}$? What is the VC-penalty for this learning model and how does it compare with the Rademacher penalty?

Problem 9.15 The Rademacher optimism penalty is

$$E'_{\text{out}}(g_{\mathbf{r}}) - E'_{\text{in}}(g_{\mathbf{r}}) = \frac{1}{2} - E'_{\text{in}}(g_{\mathbf{r}}).$$

Let $\mathcal{H}$ have growth function $m_{\mathcal{H}}(N)$. Show that, with probability at least $1-\delta$,

$$\text{Rademacher optimism penalty} \leq \sqrt{\frac{1}{2N} \ln \frac{2m_{\mathcal{H}}(N)}{\delta}}.$$

[Hint: For a single hypothesis gives $\mathbb{P}[|E_{\text{out}}(h) - E_{\text{in}}(h)| > \epsilon] \leq 2e^{-2N\epsilon^2}$*.]*

Problem 9.16 *[Hard]* **Rademacher Generalization Bound.** The Rademacher optimism penalty bounds the test error for binary classification (as does the VC-bound). This problem guides you through the proof.

We will need McDiarmid's inequality (a generalization of the Hoeffding bound):

Lemma 9.1 (McDiarmid, 1989)**.** *Let $X_i \in A_i$ be independent random variables, and Q a function, $Q : \prod_i A_i \mapsto \mathbb{R}$, satisfying*

$$\sup_{\substack{x \in \prod_i A_i \\ z \in A_j}} |Q(\mathbf{x}) - q(x_1, \ldots, x_{j-1}, z, x_{j+1}, \ldots, x_n)| \leq c_j,$$

for $j = 1, \ldots, n$. Then, $t > 0$,

$$\mathbb{P}[Q(X_1, \ldots, X_n) - \mathbb{E}[Q(X_1, \ldots, X_n)] \geq t] \leq \exp\left(-\frac{2t^2}{\sum_{i=1}^n c_i^2}\right).$$

Corollary 9.2. *With probability at least $1-\delta$,*

$$\Big|Q(X_1, \ldots, X_n) - \mathbb{E}[Q(X_1, \ldots, X_n)]\Big| \leq \sqrt{\frac{1}{2}\sum_{i=1}^n c_i^2 \ln \frac{2}{\delta}}.$$

(a) Assume that the hypothesis set is symmetric ($h \in \mathcal{H}$ implies $-h \in \mathcal{H}$). Prove that with probability at least $1-\delta$,

$$E_{\text{out}}(g) \leq E_{\text{in}}(g) + \mathbb{E}_{r,\mathcal{D}}\left[\max_{h \in \mathcal{H}} \frac{1}{N}\sum_{n=1}^N r_n h(\mathbf{x}_n)\right] + \sqrt{\frac{1}{2N}\log\frac{2}{\delta}}. \quad (9.8)$$

To do this, show the following steps.

(i) $E_{\text{out}}(g) \leq E_{\text{in}}(g) + \max_{h \in \mathcal{H}}\{E_{\text{out}}(h) - E_{\text{in}}(h)\}$.

(ii) $\max_{h \in \mathcal{H}}\{E_{\text{out}}(h) - E_{\text{in}}(h)\} = \frac{1}{2}\max_{h \in \mathcal{H}}\left\{\frac{1}{N}\sum_{n=1}^N y_n h(\mathbf{x}_n) - \mathbb{E}_{\mathbf{x}}[f(\mathbf{x})h(\mathbf{x})]\right\}$.

(iii) Show that $\frac{1}{2}\max_{h \in \mathcal{H}}\left\{\frac{1}{N}\sum_{n=1}^N y_n h(\mathbf{x}_n) - \mathbb{E}_{\mathbf{x}}[f(\mathbf{x})h(\mathbf{x})]\right\}$ is upper bounded with probability at least $1-\delta$ by

$$\tfrac{1}{2}\,\mathbb{E}_{\mathcal{D}}\left[\max_{h \in \mathcal{H}}\left\{\tfrac{1}{N}\sum_{n=1}^N y_n h(\mathbf{x}_n) - \mathbb{E}_{\mathbf{x}}[f(\mathbf{x})h(\mathbf{x})]\right\}\right] + \sqrt{\tfrac{1}{2N}\log\tfrac{2}{\delta}}$$

[Hint: Use McDairmid's inequality with

$$Q(\mathbf{x}_1,\ldots,\mathbf{x}_N,y_1,\ldots,y_N)=\max_{h\in\mathcal{H}}\left\{\tfrac{1}{N}\sum_{n=1}^N y_n h(\mathbf{x}_n)-\mathbb{E}_{\mathbf{x}}[f(\mathbf{x})h(\mathbf{x})]\right\}.$$

Show that perturbing data point $(\mathbf{x}_n,y_n)$ changes Q by at most $\frac{2}{N}$.]

(iv) Let $\mathcal{D}'=(\mathbf{x}'_1,y'_1),\ldots,(\mathbf{x}'_N,y'_N)$ be an independent (ghost) data set. Show that

$$\mathbb{E}_{\mathbf{x}}[f(\mathbf{x})h(\mathbf{x})]=\tfrac{1}{N}\sum_{n=1}^N\mathbb{E}_{\mathcal{D}'}[y'_n h(\mathbf{x}'_n)].$$

Hence, show that $\max_{h\in\mathcal{H}}\left\{\frac{1}{N}\sum_{n=1}^N y_n h(\mathbf{x}_n)-\mathbb{E}_{\mathbf{x}}[f(\mathbf{x})h(\mathbf{x})]\right\}$ equals

$$\max_{h\in\mathcal{H}}\left\{\mathbb{E}_{\mathcal{D}'}\left[\tfrac{1}{N}\sum_{n=1}^N(y_n h(\mathbf{x}_n)-y'_n h(\mathbf{x}'_n))\right]\right\}.$$

By convexity, $\max_h\{\mathbb{E}[\cdot]\}\le\mathbb{E}[\max_h\{\cdot\}]$, hence show that

$$\begin{aligned}&\max_{h\in\mathcal{H}}\left\{\tfrac{1}{N}\sum_{n=1}^N y_n h(\mathbf{x}_n)-\mathbb{E}_{\mathbf{x}}[f(\mathbf{x})h(\mathbf{x})]\right\}\\ \le\;&\mathbb{E}_{\mathcal{D}'}\left[\max_{h\in\mathcal{H}}\tfrac{1}{N}\sum_{n=1}^N(y_n h(\mathbf{x}_n)-y'_n h(\mathbf{x}'_n))\right].\end{aligned}$$

Conclude that

$$E_{\text{out}}(g)\le E_{\text{in}}(g)+\tfrac{1}{2}\mathbb{E}_{\mathcal{D},\mathcal{D}'}\left[\max_{h\in\mathcal{H}}\tfrac{1}{N}\sum_{n=1}^N(y_n h(\mathbf{x}_n)-y'_n h(\mathbf{x}'_n))\right]+\sqrt{\tfrac{1}{2N}\log\tfrac{2}{\delta}}.$$

The remainder of the proof is to bound the second term on the RHS.

(v) Let $r_1,\ldots,r_N$ be *arbitrary* ±1 Rademacher variables. Show that

$$\begin{aligned}&\mathbb{E}_{\mathcal{D},\mathcal{D}'}\left[\max_{h\in\mathcal{H}}\left\{\tfrac{1}{N}\sum_{n=1}^N(y_n h(\mathbf{x}_n)-y'_n h(\mathbf{x}'_n))\right\}\right]\\ =\;&\mathbb{E}_{\mathcal{D},\mathcal{D}'}\left[\max_{h\in\mathcal{H}}\left\{\tfrac{1}{N}\sum_{n=1}^N r_n(y_n h(\mathbf{x}_n)-y'_n h(\mathbf{x}'_n))\right\}\right]\\ \le\;&2\,\mathbb{E}_{\mathcal{D}}\left[\max_{h\in\mathcal{H}}\left\{\tfrac{1}{N}\sum_{n=1}^N r_n y_n h(\mathbf{x}_n)\right\}\right].\end{aligned}$$

[Hint: Argue that $r_n=-1$ effectively switches $\mathbf{x}_n$ with $\mathbf{x}'_n$ which is just a relabeling of variables in the expectation over $\mathbf{x}_n,\mathbf{x}'_n$. For the second step, use $\max\{A-B\}\le\max|A|+\max|B|$ and the fact that $\mathcal{H}$ is symmetric.]

(vi) Since the bound in (v) holds for any $\mathbf{r}$, we can take the expectation over independent r_n with $\mathbb{P}[r_n=+1]=\frac{1}{2}$. Hence, show that

$$\begin{aligned}&\mathbb{E}_{\mathcal{D},\mathcal{D}'}\left[\max_{h\in\mathcal{H}}\left\{\tfrac{1}{N}\sum_{n=1}^N(y_n h(\mathbf{x}_n)-y'_n h(\mathbf{x}'_n))\right\}\right]\\ \le\;&2\,\mathbb{E}_{\mathbf{r},\mathcal{D}}\left[\max_{h\in\mathcal{H}}\left\{\tfrac{1}{N}\sum_{n=1}^N r_n h(\mathbf{x}_n)\right\}\right],\end{aligned}$$

and obtain (9.8). *[Hint: what is the distribution of $r_n y_n$?]*

(b) In part (a) we obtained a generalization bound in terms of twice the *expected* Rademacher optimism penalty. To prove Theorem 9.6, show that this expectation can be well approximated by a single realization.

(i) Let $Q(r_1, \ldots, r_N, \mathbf{x}_1, \ldots, \mathbf{x}_n) = \max_{h \in \mathcal{H}} \left\{ \frac{1}{N} \sum_{n=1}^N r_n h(\mathbf{x}_n) \right\}$. Show that if you change an input of Q, its value changes by at most $\frac{2}{N}$.

(ii) Show that with probability at least $1 - \delta$,

$$\mathbb{E}_{\mathbf{r},\mathcal{D}} \left[\max_{h \in \mathcal{H}} \left\{ \frac{1}{N} \sum_{n=1}^N r_n h(\mathbf{x}_n) \right\} \right] \leq \max_{h \in \mathcal{H}} \left\{ \frac{1}{N} \sum_{n=1}^N r_n h(\mathbf{x}_n) \right\} + \sqrt{\frac{2}{N} \ln \frac{2}{\delta}}.$$

(iii) Apply the union bound to show that with probability at least $1 - \delta$,

$$E_{\text{out}}(g) \leq E_{\text{in}}(g) + \max_{h \in \mathcal{H}} \left\{ \frac{1}{N} \sum_{n=1}^N r_n h(\mathbf{x}_n) \right\} + \sqrt{\frac{9}{2N} \ln \frac{4}{\delta}}.$$

Problem 9.17 *[Hard]* **Permutation Generalization Bound.** Prove that with probability at least $1 - \delta$,

$$E_{\text{out}}(g) \leq E_{\text{in}}(g) + \mathbb{E}_{\boldsymbol{\pi}} \left[\max_{h \in \mathcal{H}} \frac{1}{N} \sum_{n=1}^N y_n^{(\boldsymbol{\pi})} h(\mathbf{x}_n) \right] + O(\sqrt{\frac{1}{N} \log \frac{1}{\delta}}).$$

The second term is similar to the permutation optimism penalty, differing by $\bar{y}\, \mathbb{E}_{\boldsymbol{\pi}} [\bar{g}_{\boldsymbol{\pi}}]$, which is zero for balanced data.

[Hint: Up to introducing the r_n, you can follow the proof in Problem 9.16; now pick a distribution for $\boldsymbol{r}$ to mimic permutations. For some helpful tips, see "A Permutation Approach to Validation," Magdon-Ismail, Mertsalov, 2010.]

Problem 9.18 **Permutation Penalty for Linear Models.** For linear models, the predictions on the data are $\hat{\mathbf{y}}^{(\boldsymbol{\pi})} = \mathrm{H}\mathbf{y}^{(\boldsymbol{\pi})}$, H is the hat matrix, $\mathrm{H}(\lambda) = \mathrm{X}(\mathrm{X}^{\mathrm{T}}\mathrm{X} + \lambda \mathrm{I})^{-1}\mathrm{X}^{\mathrm{T}}$, which is independent of $\boldsymbol{\pi}$. For regression, the permutation optimism penalty from (9.7) is $\frac{2}{N} \sum_{n=1}^N (y_{\boldsymbol{\pi}_n} - \bar{y}) g_{(\boldsymbol{\pi})}(\mathbf{x}_n)$. (we do not divide the squared error by 4 for regression).

(a) Show that for a single permutation, permutation penalty is:

$$\frac{2}{N} \sum_{m,n=1}^N \mathrm{H}_{mn} (y_{\boldsymbol{\pi}_m} y_{\boldsymbol{\pi}_n} - \bar{y} y_{\boldsymbol{\pi}_n}).$$

(b) Show that: $\mathbb{E}_{\boldsymbol{\pi}}[y_{\boldsymbol{\pi}_n}] = \bar{y}$, and $\mathbb{E}_{\boldsymbol{\pi}}[y_{\boldsymbol{\pi}_m} y_{\boldsymbol{\pi}_n}] = \begin{cases} \bar{y}^2 + s_y^2 & m = n, \\ \bar{y}^2 - \frac{1}{N-1} s_y^2 & m \neq n. \end{cases}$

($\bar{y}$ and s_y^2 are defined in Problem 9.12(b).)

(c) Take the expectation of the penalty in (a) and prove Equation (9.6):

$$\text{permutation optimism penalty} = \frac{2\hat{\sigma}_y^2}{N} \left(\mathrm{trace}(\mathrm{S}) - \frac{\mathbf{1}^{\mathrm{T}} \mathrm{S} \mathbf{1}}{N} \right),$$

where $\hat{\sigma}_y^2 = \frac{N}{N-1} s_y^2$ is the unbiased estimate of the target variance.

Problem 9.19 Repeat Problem 9.18 for the Bootstrap optimism penalty and show that

$$\text{Bootstrap optimism penalty} = \frac{2s_y^2}{N}\text{trace}(\mathrm{H}).$$

Compare this to the permutation optimism penalty in Problem 9.18.

[Hint: How does Problem 9.18(b) change for the Bootstrap setting?]

Problem 9.20 This is a repeat of Problem 5.2. Structural Risk Minimization (SRM) is a useful framework for model selection that is related to Occam's Razor. Define a *structure* – a nested sequence of hypothesis sets:

The SRM framework picks a hypothesis from each $\mathcal{H}_i$ by minimizing E_{in}. That is, $g_i = \underset{h\in\mathcal{H}_i}{\text{argmin}}\, E_{\text{in}}(h)$. Then, the framework selects the final hypothesis by minimizing E_{in} *and* the model complexity penalty Ω. That is, $g^* = \underset{i=1,2,\cdots}{\text{argmin}}(E_{\text{in}}(g_i)+\Omega(\mathcal{H}_i))$. Note that $\Omega(\mathcal{H}_i)$ should be non-decreasing in i because of the nested structure.

(a) Show that the in-sample error $E_{\text{in}}(g_i)$ is non-increasing in i.

(b) Assume that the framework finds $g^* \in \mathcal{H}_i$ with probability p_i. How does p_i relate to the complexity of the target function?

(c) Argue that the p_i's are unknown but $p_0 \le p_1 \le p_2 \le \cdots \le 1$.

(d) Suppose $g^* = g_i$. Show that

$$\mathbb{P}\left[|E_{\text{in}}(g_i) - E_{\text{out}}(g_i)| > \epsilon \mid g^* = g_i\right] \le \frac{1}{p_i}\cdot 4m_{\mathcal{H}_i}(2N)e^{-\epsilon^2 N/8}.$$

Here, the conditioning is on selecting g_i as the final hypothesis by SRM. *[Hint: Use the Bayes theorem to decompose the probability and then apply the VC bound on one of the terms]*

You may interpret this result as follows: if you use SRM and end up with g_i, then the generalization bound is a factor $\frac{1}{p_i}$ worse than the bound you would have gotten had you simply started with $\mathcal{H}_i$.

Problem 9.21 **Cross-Validation Leverage for Linear Regression.]** In this problem, compute an expression for the leverage defined in Equation (9.4) for linear regression with weight decay regularization. We will use the same notation from Problem 4.26, so you may want to review that problem and some of the tools developed there.

To simulate leaving out the data point $(\mathbf{z}_m, y_m)$, set the mth row of Z and the mth entry of $\mathbf{y}$ to zero, to get data matrix $\mathrm{Z}^{(m)}$ and target vector $\mathbf{y}^{(m)}$, so $\mathcal{D}_m = (\mathrm{Z}^{(m)}, \mathbf{y}^{(m)})$. We need to compute the cross validation error for this data set $E_{\mathrm{cv}}(\mathcal{D}_m)$. Let $\hat{\mathrm{H}}^{(m)}$ be the hat matrix you get from doing linear regression with the data $\mathrm{Z}^{(m)}, \mathbf{y}^{(m)}$.

(a) Show that $E_{\mathrm{cv}}(\mathcal{D}_m) = \dfrac{1}{N-1} \displaystyle\sum_{n \neq m} \left(\frac{\hat{y}_n^{(m)} - y_n}{1 - \mathrm{H}_{nn}^{(m)}} \right)^2$.

(b) Use the techniques from Problem 4.26 to show that

$$\mathrm{H}_{nk}^{(m)} = \mathrm{H}_{nk} + \frac{\mathrm{H}_{mn}\mathrm{H}_{mk}}{1 - \mathrm{H}_{mm}}.$$

(c) Similarly, show that

$$\hat{y}_n^{(m)} = \hat{y}_n + \left(\frac{\hat{y}_m - y_m}{1 - \mathrm{H}_{mm}} \right) \mathrm{H}_{mn}.$$

[Hint: Use part (c) of Problem 4.26.]

(d) Show that $E_{\mathrm{cv}}(\mathcal{D}_m)$ is given by

$$\frac{1}{N-1} \sum_{n=1}^{N} \left(\frac{\hat{y}_n - y_n + \left(\frac{\hat{y}_m - y_m}{1 - \mathrm{H}_{mm}} \right) \mathrm{H}_{mn}}{1 - \mathrm{H}_{nn} - \frac{\mathrm{H}_{mn}^2}{1 - \mathrm{H}_{mm}}} \right)^2 + \frac{1}{N-1} \left(\frac{\hat{y}_m - y_m}{1 - 2\mathrm{H}_{mm}} \right)^2.$$

(e) Give an expression for the leverage ℓ_m. What is the running time to compute the leverages $\ell_1, \ldots, \ell_N$.

Problem 9.22 The data set for Example 9.3 is

$$\mathrm{X} = \begin{bmatrix} 1 & 0.51291 \\ 1 & 0.46048 \\ 1 & 0.3504 \\ 1 & 0.095046 \\ 1 & 0.43367 \\ 1 & 0.70924 \\ 1 & 0.11597 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} 0.36542 \\ 0.22156 \\ 0.15263 \\ 0.10355 \\ 0.10015 \\ 0.26713 \\ 2.3095 \end{bmatrix}.$$

Implement the algorithm from Problem 9.21 to compute the leverages for all the data points as you vary the regularization parameter λ.

Give a plot of the leverage of the last data point as a function of λ. Explain the dependence you find in your plot.

Problem 9.23 For a sample of 500 digits data (classifying "1" versus "not 1"), use linear regression to compute the leverage of the data points (even though the problem is a classification problem). You can use the algorithm in Problem 9.21.

Give a similar plot to Figure 9.5 where you highlight in a black box all the data points with the top-10 largest (positive) leverages.

Give some intuition for which points are highlighted.

Problem 9.24 **The Jackknife Estimate.** The jackknife is a general statistical technique used to reduce the bias of an estimator, based on the assumption that the estimator is asymptotically (as N increases) unbiased. Let $\mathcal{Z} = \mathbf{z}_1, \ldots, \mathbf{z}_N$ be a sample set. In the case of learning, the sample set is the data, so $\mathbf{z}_n = (\mathbf{x}_n, y_n)$. We wish to estimate a quantity t using an estimator $\hat{t}(\mathcal{Z})$ (some function of the sample $\mathcal{Z}$). Assume that $\hat{t}(\mathcal{Z})$ is asymptotically unbiased, satisfying

$$\mathbb{E}_{\mathcal{Z}}[\hat{t}(\mathcal{Z})] = t + \frac{a_1}{N} + \frac{a_2}{N^2} + \cdots .$$

The bias is $O\left(\frac{1}{N}\right)$. Let $\mathcal{Z}_n = \mathbf{z}_1, \ldots, \mathbf{z}_{n-1}, \mathbf{z}_n, \mathbf{z}_{n+1}, \ldots, \mathbf{z}_N$ be the leave one out sample sets (similar to cross validation), and consider the estimates using the leave one out samples $\hat{t}_n = \hat{t}(\mathcal{Z}_n)$.

(a) Argue that $\mathbb{E}_{\mathcal{Z}}[\hat{t}_n] = t + \dfrac{a_1}{N-1} + \dfrac{a_2}{(N-1)^2} + \cdots$.

(b) Define $\hat{\tau}_n = N\hat{t}(\mathcal{Z}) - (N-1)\hat{t}_n$. Show that $\mathbb{E}_{\mathcal{Z}}[\hat{\tau}_n] = t - \frac{a_2}{N(N-1)} + O(\frac{1}{N^2})$. ($\hat{\tau}_n$ has an asymptotically asymptotically smaller bias than $t(\mathcal{Z})$.) The $\hat{\tau}_n$ are called pseudo-values because they have the "correct" expectation. A natural improvement is to take the average of the pseudo-values, and this is the jackknife estimate: $\hat{t}_J(\mathcal{Z}) = \frac{1}{N}\sum_{n=1}^{N}\hat{\tau}_n = N\hat{t}(\mathcal{Z}) - \frac{N-1}{N}\sum_{n=1}^{N}\hat{t}(\mathcal{Z}_n)$.

(c) *Applying the Jackknife to variance estimation.* Suppose that the sample is a bunch of independent random values $x_1, \ldots, x_N$ from a distribution whose variance σ^2 we wish to estimate. We suspect that the sample variance, $s^2 = \frac{1}{N}\sum_{n=1}^{N} x_n^2 - \frac{1}{N^2}\left(\sum_{n=1}^{N} x_n\right)^2$, should be a good estimator, i.e., it has the assumed form for the bias (it does). Let s_n^2 be the sample variances on the leave one out samples. Show that

$$s_n^2 = \frac{1}{N-1}\left(\sum_{m=1}^{N} x_m^2 - x_n^2\right) - \frac{1}{(N-1)^2}\left(\sum_{m=1}^{N} x_m - x_n\right)^2 .$$

Hence show that jackknife estimate $Ns^2 - \frac{N-1}{N}\sum_n s_n^2$ is $s_J^2 = \frac{N}{N-1}s^2$, which is the well known unbiased estimator of the variance. In this particular case, the jackknife has completely removed the bias (automatically).

(d) What happens to the jackknife if $t(\mathcal{Z})$ has an asymptotic bias?

(e) If the leading order term in the bias was $\frac{1}{N^2}$, does the jackknife estimate have a better or worse bias (in magnitude)?

Problem 9.25 The Jackknife for Validation. (see also the previous problem) If E_{in} is an asymptotically unbiased estimate of E_{out}, we may use the jackknife to reduce this bias and get a better estimate. The sample is the data. We want to estimate the expected out-of-sample error when learning from N examples, $\mathcal{E}_{\text{out}}(N)$. We estimate $\mathcal{E}_{\text{out}}(N)$ by the in-sample error $E_{\text{in}}(g^{(\mathcal{D})})$.

(a) What kind of bias (+ve or -ve) will E_{in} have. When do you expect it to be asymptotically unbiased?

(b) Assume that the bias has the required form: $\mathbb{E}_{\mathcal{D}}[E_{\text{in}}(g^{(\mathcal{D})})] = \mathcal{E}_{\text{out}}(N) + \frac{a_1}{N} + \frac{a_2}{N^2} + \cdots$. Now consider one of the leave one out data sets $\mathcal{D}_n$, which would produce the estimates $E_{\text{in}}(g^{(\mathcal{D}_n)})$. Show that:

i. the pseudo-values are $NE_{\text{in}}(g^{(\mathcal{D})}) - (N-1)E_{\text{in}}(g^{(\mathcal{D}_n)})$;

ii. the jackknife estimate is: $E_J = NE_{\text{in}}(g^{(\mathcal{D})}) - \frac{N-1}{N}\sum_{n=1}^{N} E_{\text{in}}(g^{(\mathcal{D}_n)})$.

(c) Argue that $\mathbb{E}_{\mathcal{D}}[E_{\text{in}}(g^{(\mathcal{D}_n)})] = \mathcal{E}_{\text{out}}(N-1) + \frac{a_1}{N-1} + \frac{a_2}{(N-1)^2} + \cdots$, and hence show that the expectation of the jackknife estimate is given by

$$\mathbb{E}_{\mathcal{D}}[E_J] = \mathcal{E}_{\text{out}}(N) + (N-1)(\mathcal{E}_{\text{out}}(N) - \mathcal{E}_{\text{out}}(N-1)) + O\left(\frac{1}{N^2}\right).$$

(d) If the learning curve converges, having a form $\mathcal{E}_{\text{out}}(N) = E + \frac{b_1}{N} + \frac{b_2}{N^2} + \cdots$, then show that $\mathbb{E}_{\mathcal{D}}[E_J] = \mathcal{E}_{\text{out}}(N) + \frac{b_1}{N} + O\left(\frac{1}{N^2}\right)$.

The jackknife replaced the term $\frac{a_1}{N}$ in the bias by $\frac{b_1}{N}$. (In a similar vein to cross validation, the jackknife replaces the bias of the in-sample estimate with the bias in the learning curve.) When will the jackknife be helpful?

Problem 9.26 The Jackknife Estimate for Linear Models. The jackknife validation estimate can be computed analytically for linear models. Define the matrix $\mathrm{H}^{(\delta)} = \mathrm{H}^2 - \mathrm{H}$, and let $\mathbf{y}^{(\delta)} = \mathrm{H}^{(\delta)}\mathbf{y}$.

(a) Show that the jackknife estimate is given by

$$E_J = \frac{1}{N}\sum_{n=1}^{N}\left(\frac{(\hat{y}_n - y_n)^2}{1-\mathrm{H}_{nn}} - \frac{2y_n^{(\delta)}(\hat{y}_n - y_n)}{1-\mathrm{H}_{nn}} - \frac{\mathrm{H}_{nn}^{(\delta)}(\hat{y}_n - y_n)^2}{(1-\mathrm{H}_{nn})^2}\right). \tag{9.9}$$

[Hint: you may find some of the formulas used in deriving the cross validation estimate for linear models useful from Problem 4.26.]

(b) When $\lambda = 0$, what is E_J? Compare to E_{cv} in (4.13). Show that $E_{\text{in}} \le E_J \le E_{\text{cv}}$. *[Hint:* H *is a projection matrix, so* $\mathrm{H}^2 = \mathrm{H}$ *when* $\lambda = 0$*; also,* $\mathrm{H}_{nn} = \mathbf{x}_n^{\mathrm{T}}(\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathbf{x}_n$*, so show that* $0 \le \mathrm{H}_{nn} \le 1$ *(show that for any positive definite (invertible) matrix* A*,* $\mathbf{x}_n^{\mathrm{T}}(\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}} + A)^{-1}\mathbf{x}_n \le 1$*).]*

Problem 9.27 **Feature Selection.** Let $x \in \mathbb{R}^d$ and let $\mathcal{H}$ be the perceptron model $\text{sign}(\mathbf{w}^{\text{T}}\mathbf{x})$. The *zero norm*, $\|\mathbf{w}\|_0$, is the number of non-zero elements in $\mathbf{w}$. Let $\mathcal{H}_k = \{\mathbf{w} : \|\mathbf{w}\|_0 \leq k\}$. The hypothesis set $\mathcal{H}_k$ contains the classifiers which use only k of the dimensions (or *features*) to classify the input. Model selection among the $\mathcal{H}_k$ corresponds to selecting the optimal number of features. Picking a particular hypothesis in $\mathcal{H}_k$ corresponds to picking the best set of k features.

(a) Show that order selection is a special case of feature selection.

(b) Show that $d_{\text{vc}}(\mathcal{H}_k) \leq \max(d+1, (2k+1)\log_2 d) = O(k \log d)$, which for $k = o(d)$ is an improvement over the trivial bound of $d+1$.
[Hint: Show that $d_{\text{vc}} \leq M$ for any M which satisfies $2^M > M^{k+1}\binom{d}{k}$; to show this, remember that $\mathcal{H}_k$ is the union of smaller models (how many? what is their VC-dimension?), and use the relationship between the VC-dimension of a model and the maximum number of dichotomies it can implement on any M points. Now use the fact that $\binom{d}{k} \leq \left(\frac{ed}{k}\right)^k$]

Problem 9.28 **Forward and Backward Subset Selection (FSS and BSS)** For selecting k features, we fix $\mathcal{H}_k$, and ask for the hypothesis which minimizes E_{in} (typically $k \ll d$). There is no known efficient algorithm to solve this problem (it is a NP-hard problem to even get close to the minimum E_{in}). FSS and BSS are greedy heuristics to solve the problem.

FSS: Start with $k = 1$ and find the best single feature dimension i_1 yielding minimum E_{in}. Now fix this feature dimension and find the next feature dimension i_2 which when coupled with the feature dimension i_1 yields minimum E_{in}. Continue in this way, adding one feature at a time, until you have k features.

BSS: Start with all d features and now remove one feature at a time until you are down to k features. Each time you remove the feature which results in minimum increase in E_{in}.

(a) Which of FSS and BSS do you expect to be more efficient. Suppose that to optimize with i features and N data points takes $O(iN)$ time. What is the asymptotic run time of FSS and BSS. For what values of k would you use FSS over BSS and vice versa. *[Hint: show that FSS takes $N\sum_{i=1}^{k} i(d+1-i)$ time and BSS takes $N\sum_{i=1}^{d-k}(d-i)(d+1-i)$.]*

(b) Show that $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_d$ are a structure. (See Problem 9.20.)

(c) Implement FSS. Use the bound on the VC-dimension given in the Problem 9.27 to perform SRM using the digits data to classify 1 (class $+1$) from $\{2,3,4,5,6,7,8,9,0\}$ (all in class -1).

(d) What are the first few most useful features? What is the optimal number of features (according to SRM)?

Epilogue

This book set the stage for a deeper exploration into Learning From Data by developing the foundations. It *is* possible to learn from data, and you have all the basic tools to do so. The linear model coupled with the right features and an appropriate nonlinear transform, together with the right amount of regularization, pretty much puts you into the thick of the game, and you will be in good stead as long as you keep in mind the three basic principles: simple is better (Occam's razor), avoid data snooping and beware of sampling bias.

Where to go from here? There are two main directions. One is to learn more sophisticated learning techniques, and the other is to explore different learning paradigms. Let us preview these two directions to give the reader a better understanding of the 'map' of learning from data.

The linear model can be used as a building block for other popular techniques. A cascade of linear models, mostly with soft thresholds, creates a neural network. A robust algorithm for linear models, based on quadratic programming, creates support vector machines. An efficient approach to nonlinear transformation in support vector machines creates kernel methods. A combination of different models in a principled way creates boosting and ensemble learning. There are other successful models and techniques, and more to come for sure.

In terms of other paradigms, we have briefly mentioned unsupervised learning and reinforcement learning. There is a wealth of techniques for these learning paradigms, including methods that mix labeled and unlabeled data. Active learning and online learning, which we also mentioned briefly, have their own techniques and theories. In addition, there is a school of thought that treats learning as a completely probabilistic paradigm using a Bayesian approach, and there are useful probabilistic techniques such as Gaussian processes. Last but not least, there is a school that treats learning as a branch of the theory of computational complexity, with emphasis on asymptotic results.

Of course, the ultimate test of any engineering discipline is its impact in real life. There is no shortage of successful applications of learning from data. Some of the application domains have specialized techniques that are worth exploring, e.g., computational finance and recommender systems.

Learning from data is a very dynamic field. Some of the hot techniques and theories at times become just fads, and others gain traction and become

part of the field. What we have emphasized in this book are the necessary fundamentals that give any student of learning from data a solid foundation, and enable him or her to venture out and explore further techniques and theories, or perhaps to contribute their own.

Further Reading

Learning From Data **book forum** (at AMLBook.com).

Y. S. Abu-Mostafa. The Vapnik-Chervonenkis dimension: Information versus complexity in learning. *Neural Computation*, 1(3):312 317, 1989.

Y. S. Abu-Mostafa, X. Song, A. Nicholson, and M. Magdon-Ismail. The bin model. Technical Report CaltechCSTR:2004.002, California Institute of Technology, 2004.

R. Ariew. *Ockham's Razor: A Historical and Philosophical Analysis of Ockham's Principle of Parsimony.* University of Illinois Press, 1976.

R. Bell, J. Bennett, Y. Koren, and C. Volinsky. The million dollar programming prize. *IEEE Spectrum*, 46(5):29 33, 2009.

A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information Processing Letters*, 24(6):377 380, 1987.

A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the Association for Computing Machinery*, 36(4):929 965, 1989.

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

P. Burman. A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods. *Biometrika*, 76(3): 503 514, 1989.

T. M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, 14(3):326 334, 1965.

M. H. DeGroot and M. J. Schervish. *Probability and Statistics.* Addison Wesley, fourth edition, 2011.

V. Fabian. Stochastic approximation methods. *Czechoslovak Mathematical Journal*, 10(1):123 159, 1960.

W. Feller. *An Introduction to Probability Theory and Its Applications.* Wiley, third edition, 1968.

A. Frank and A. Asuncion. UCI machine learning repository, 2010. URL `http://archive.ics.uci.edu/ml`.

J. H. Friedman. On bias, variance, 0/1 loss, and the curse-of-dimensionality. *Data Mining and Knowledge Discovery*, 1(1):55 77, 1997.

S. I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, 1990.

Z. Ghahramani. Unsupervised learning. In *Advanced Lectures in Machine Learning (MLSS '03)*, pages 72 112, 2004.

G. H. Golub and C. F. van Loan. *Matrix computations.* Johns Hopkins University Press, 1996.

D. C. Hoaglin and R. E. Welsch. The hat matrix in regression and ANOVA. *American Statistician*, 32:17–22, 1978.

W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13 30, 1963.

R. C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63 91, 1993.

R. A. Horn and C. R. Johnson. *Matrix Analysis.* Cambridge University Press, 1990.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237 285, 1996.

A. I. Khuri. *Advanced calculus with applications in statistics.* Wiley-Interscience, 2003.

R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial intelligence (IJCAI '95)*, volume 2, pages 1137 1143, 1995.

J. Langford. Tutorial on practical prediction theory for classification. *Journal of Machine Learning Research*, 6:273 306, 2005.

L. Li and H.-T. Lin. Optimizing 0/1 loss for perceptrons by random coordinate descent. In *Proceedings of the 2007 International Joint Conference on Neural Networks (IJCNN '07)*, pages 749 754, 2007.

H.-T. Lin and L. Li. Support vector machinery for infinite ensemble learning. *Journal of Machine Learning Research*, 9(2):285–312, 2008.

M. Magdon-Ismail and K. Mertsalov. A permutation approach to validation. *Statistical Analysis and Data Mining*, 3(6):361–380, 2010.

M. Magdon-Ismail, A. Nicholson, and Y. S. Abu-Mostafa. Learning in the presence of noise. In S. Haykin and B. Kosko, editors, *Intelligent Signal Processing*. IEEE Press, 2001.

M. Markatou, H. Tian, S. Biswas, and G. Hripcsak. Analysis of variance of cross-validation estimators of the generalization error. *Journal of Machine Learning Research*, 6:1127 1168, 2005.

M. L. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, expanded edition, 1988.

T. Poggio and S. Smale. The mathematics of learning: Dealing with data. *Notices of the American Mathematical Society*, 50(5):537 544, 2003.

K. Popper. *The logic of scientific discovery*. Routledge, 2002.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386 408, 1958.

F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, 1962.

B. Settles. Active learning literature survey. Technical Report 1648, University of Wisconsin-Madison, 2010.

J. Shawe-Taylor, P. L. Bartlett, R. C. Williamson, and M. Anthony. A framework for structural risk minimisation. In *Learning Theory: 9th Annual Conference on Learning Theory (COLT '96)*, pages 68 76, 1996.

L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27 (11):1134 1142, 1984.

V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications*, 16:264 280, 1971.

V. N. Vapnik, E. Levin, and Y. L. Cun. Measuring the VC-dimension of a learning machine. *Neural Computation*, 6(5):851 876, 1994.

G.-X. Yuan, C.-H. Ho, and C.-J. Lin. Recent advances of large-scale linear classification. *Proceedings of IEEE*, 2012.

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

Appendix A

Proof of the VC Bound

In this Appendix, we present the formal proof of Theorem 2.5. It is a fairly elaborate proof, and you may skip it altogether and just take the theorem for granted, but you won't know what you are missing ☺ !

Theorem A.1 (Vapnik, Chervonenkis, 1971)**.**

$$\mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon\right] \leq 4m_{\mathcal{H}}(2N)e^{-\frac{1}{8}\epsilon^2 N}.$$

This inequality is called the VC Inequality, and it implies the VC bound of Theorem 2.5. The inequality is valid for any target function (deterministic or probabilistic) and any input distribution. The probability is over data sets of size N. Each data set is generated *iid* (independent and identically distributed), with each data point generated independently according to the joint distribution $P(\mathbf{x}, y)$. The event $\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon$ is equivalent to the union over all $h \in \mathcal{H}$ of the events $|E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon$; this union contains the event that involves g in Theorem 2.5. The use of the supremum (a technical version of the maximum) is necessary since $\mathcal{H}$ can have a continuum of hypotheses.

The main challenge to proving this theorem is that $E_{\text{out}}(h)$ is difficult to manipulate compared to $E_{\text{in}}(h)$, because $E_{\text{out}}(h)$ depends on the entire input space rather than just a finite set of points. The main insight needed to overcome this difficulty is the observation that we can get rid of $E_{\text{out}}(h)$ altogether because the deviations between E_{in} and E_{out} can be essentially captured by deviations between two in-sample errors: E_{in} (the original in-sample error) and the in-sample error on a *second* independent data set (Lemma A.2). We have seen this idea many times before when we use a test or validation set to estimate E_{out}. This insight results in two main simplifications:

1. The supremum of the deviations over infinitely many $h \in \mathcal{H}$ can be reduced to considering only the dichotomies implementable by $\mathcal{H}$ on the

two independent data sets. That is where the growth function $m_{\mathcal{H}}(2N)$ enters the picture (Lemma A.3).

2. The deviation between two *independent* in-sample errors is 'easy' to analyze compared to the deviation between E_{in} and E_{out} (Lemma A.4).

The combination of Lemmas A.2, A.3 and A.4 proves Theorem A.1.

A.1 Relating Generalization Error to In-Sample Deviations

Let's introduce a second data set $\mathcal{D}'$, which is independent of $\mathcal{D}$, but sampled according to the same distribution $P(\mathbf{x}, y)$. This second data set is called a *ghost* data set because it doesn't really exist; it is a just a tool used in the analysis. We hope to bound the term $\mathbb{P}[|E_{\text{in}} \quad E_{\text{out}}|$ is large] by another term $\mathbb{P}[|E_{\text{in}} \quad E'_{\text{in}}|$ is large], which is easier to analyze.

The intuition behind the formal proof is as follows. For any single hypothesis h, because $\mathcal{D}'$ is fresh, sampled independently from $P(\mathbf{x}, y)$, the Hoeffding Inequality guarantees that $E'_{\text{in}}(h) \approx E_{\text{out}}(h)$ with a high probability. That is, when $|E_{\text{in}}(h) \quad E_{\text{out}}(h)|$ is large, with a high probability $|E_{\text{in}}(h) \quad E'_{\text{in}}(h)|$ is also large. Therefore, $\mathbb{P}[|E_{\text{in}}(h) \quad E_{\text{out}}(h)|$ is large] can be approximately bounded by $\mathbb{P}[|E_{\text{in}}(h) \quad E'_{\text{in}}(h)|$ is large].

We are trying to bound the probability that E_{in} is far from E_{out}. Let $E'_{\text{in}}(h)$ be the 'in-sample' error for hypothesis h on $\mathcal{D}'$. Suppose that E_{in} is far from E_{out} with some probability (and similarly E'_{in} is far from E_{out}, with that same probability, since E_{in} and E'_{in} are identically distributed). When N is large, the probability is roughly Gaussian around E_{out}, as illustrated in the figure to the right. The red region represents the cases when E_{in} is far from E_{out}. In those cases, E'_{in} is far from E_{in} about half the time, as illustrated by the green region. That is, $\mathbb{P}[|E_{\text{in}} \quad E_{\text{out}}|$ is large] can be approximately bounded by $2\,\mathbb{P}\,[|E_{\text{in}} \quad E'_{\text{in}}|$ is large].

This argument provides some intuition that the deviations between E_{in} and E_{out} can be captured by the deviations between E_{in} and E'_{in}. The argument can be carefully extended to multiple hypotheses.

Lemma A.2.

$$\left(1 - 2e^{-\frac{1}{2}\epsilon^2 N}\right)\mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) - E_{\text{out}}(h)| > \epsilon\right] \leq \mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2}\right],$$

where the probability on the RHS is over $\mathcal{D}$ and $\mathcal{D}'$ jointly.

Proof. We can assume that $\mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon\right] > 0$, otherwise there is nothing to prove.

$$\begin{aligned}
& \mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \tfrac{\epsilon}{2}\right] \\
\geq\ & \mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \textbf{ and } \sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon\right] \quad \text{(A.1)} \\
=\ & \mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon\right] \times \\
& \mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, \sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon\right].
\end{aligned}$$

Inequality (A.1) follows because $\mathbb{P}[\mathcal{B}_1] \geq \mathbb{P}[\mathcal{B}_1 \textbf{ and } \mathcal{B}_2]$ for any two events $\mathcal{B}_1, \mathcal{B}_2$. Now, let's consider the last term:

$$\mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, \sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon\right].$$

The event on which we are conditioning is a set of data sets with non-zero probability. Fix a data set $\mathcal{D}$ in this event. Let h^* be any hypothesis for which $|E_{\text{in}}(h^*) \quad E_{\text{out}}(h^*)| > \epsilon$. One such hypothesis must exist given that $\mathcal{D}$ is in the event on which we are conditioning. The hypothesis h^* does not depend on $\mathcal{D}'$, but it does depend on $\mathcal{D}$.

$$\begin{aligned}
& \mathbb{P}\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, \sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon\right] \\
\geq\ & \mathbb{P}\left[|E_{\text{in}}(h^*) \quad E'_{\text{in}}(h^*)| > \tfrac{\epsilon}{2} \,\middle|\, \sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon\right] \quad \text{(A.2)} \\
\geq\ & \mathbb{P}\left[|E'_{\text{in}}(h^*) \quad E_{\text{out}}(h^*)| \leq \tfrac{\epsilon}{2} \,\middle|\, \sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon\right] \quad \text{(A.3)} \\
\geq\ & 1 - 2e^{\ \frac{1}{2}\epsilon^2 N}. \quad \text{(A.4)}
\end{aligned}$$

1. Inequality (A.2) follows because the event "$|E_{\text{in}}(h^*) \quad E'_{\text{in}}(h^*)| > \frac{\epsilon}{2}$" implies "$\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \frac{\epsilon}{2}$".

2. Inequality (A.3) follows because the events "$|E'_{\text{in}}(h^*) \quad E_{\text{out}}(h^*)| \leq \frac{\epsilon}{2}$" and "$|E_{\text{in}}(h^*) \quad E_{\text{out}}(h^*)| > \epsilon$" (which is given) imply "$|E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \frac{\epsilon}{2}$".

3. Inequality (A.4) follows because h^* is fixed with respect to $\mathcal{D}'$ and so we can apply the Hoeffding Inequality to $\mathbb{P}[|E'_{\text{in}}(h^*) \quad E_{\text{out}}(h^*)| \leq \frac{\epsilon}{2}]$.

Notice that the Hoeffding Inequality applies to $\mathbb{P}[|E'_{\text{in}}(h^*) \quad E_{\text{out}}(h^*)| \leq \frac{\epsilon}{2}]$ for any h^*, as long as h^* is fixed with respect to $\mathcal{D}'$. Therefore, it also applies

to any weighted average of $\mathbb{P}[|E'_{\text{in}}(h^*) \quad E_{\text{out}}(h^*)| \leq \frac{\epsilon}{2}]$ based on h^*. Finally, since h^* depends on a particular $\mathcal{D}$, we take the weighted average over all $\mathcal{D}$ in the event

$$\text{“} \sup_{h \in \mathcal{H}} |E_{\text{in}}(h) \quad E_{\text{out}}(h)| > \epsilon \text{”}$$

on which we are conditioning, where the weight comes from the probability of the particular $\mathcal{D}$. Since the bound holds for every $\mathcal{D}$ in this event, it holds for the weighted average. ■

Note that we can assume $e^{-\frac{1}{2}\epsilon^2 N} < \frac{1}{4}$, because otherwise the bound in Theorem A.1 is trivially true. In this case, $1 \quad 2e^{-\frac{1}{2}\epsilon^2 N} > \frac{1}{2}$, so the lemma implies

$$\mathbb{P}\left[\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) - E_{\text{out}}(h)| > \epsilon\right] \leq 2\,\mathbb{P}\left[\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2}\right].$$

A.2 Bounding Worst Case Deviation Using the Growth Function

Now that we have related the generalization error to the deviations between in-sample errors, we can actually work with $\mathcal{H}$ restricted to two data sets of size N each, rather than the infinite $\mathcal{H}$. Specifically, we want to bound

$$\mathbb{P}\left[\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2}\right],$$

where the probability is over the joint distribution of the data sets $\mathcal{D}$ and $\mathcal{D}'$. One equivalent way of sampling two data sets $\mathcal{D}$ and $\mathcal{D}'$ is to first sample a data set S of size $2N$, then randomly partition S into $\mathcal{D}$ and $\mathcal{D}'$. This amounts to randomly sampling, *without replacement*, N examples from S for $\mathcal{D}$, leaving the remaining for $\mathcal{D}'$. Given the joint data set S, let

$$\mathbb{P}\left[\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, S\right]$$

be the probability of deviation between the two in-sample errors, where the probability is taken over the random partitions of S into $\mathcal{D}$ and $\mathcal{D}'$. By the law of total probability (with $\sum$ denoting sum or integral as the case may be),

$$\begin{aligned}
&\mathbb{P}\left[\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2}\right] \\
= &\sum_{S} \mathbb{P}[S] \times \mathbb{P}\left[\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, S\right] \\
\leq &\sup_{S} \; \mathbb{P}\left[\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, S\right].
\end{aligned}$$

Let $\mathcal{H}(S)$ be the dichotomies that $\mathcal{H}$ can implement on the points in S. By definition of the growth function, $\mathcal{H}(S)$ cannot have more than $m_{\mathcal{H}}(2N)$ dichotomies. Suppose it has $M \leq m_{\mathcal{H}}(2N)$ dichotomies, realized by $h_1, \ldots, h_M$. Thus,

$$\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) - E'_{\text{in}}(h)| = \sup_{h \in \{h_1, \ldots, h_M\}} |E_{\text{in}}(h) - E'_{\text{in}}(h)| .$$

Then,

$$\begin{aligned}
& \mathbb{P}\left[\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, S\right] \\
= \; & \mathbb{P}\left[\sup_{h \in \{h_1, \ldots, h_M\}} |E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, S\right] \\
\leq \; & \sum_{m=1}^{M} \mathbb{P}\left[|E_{\text{in}}(h_m) - E'_{\text{in}}(h_m)| > \tfrac{\epsilon}{2} \,\middle|\, S\right] && \text{(A.5)} \\
\leq \; & M \times \sup_{h \in \mathcal{H}} \mathbb{P}\left[|E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, S\right], && \text{(A.6)}
\end{aligned}$$

where we use the union bound in (A.5), and overestimate each term by the supremum over all possible hypotheses to get (A.6). After using $M \leq m_{\mathcal{H}}(2N)$ and taking the sup operation over S, we have proved:

Lemma A.3.

$$\begin{aligned}
& \mathbb{P}\left[\sup_{h \in \mathcal{H}} |E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2}\right] \\
\leq \; & m_{\mathcal{H}}(2N) \times \sup_{S} \sup_{h \in \mathcal{H}} \mathbb{P}\left[|E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, S\right],
\end{aligned}$$

where the probability on the LHS is over $\mathcal{D}$ and $\mathcal{D}'$ jointly, and the probability on the RHS is over random partitions of S into two sets $\mathcal{D}$ and $\mathcal{D}'$.

The main achievement of Lemma A.3 is that we have pulled the supremum over $h \in \mathcal{H}$ outside the probability, at the expense of the extra factor of $m_{\mathcal{H}}(2N)$.

A.3 Bounding the Deviation between In-Sample Errors

We now address the purely combinatorial problem of bounding

$$\sup_{S} \sup_{h \in \mathcal{H}} \mathbb{P}\left[|E_{\text{in}}(h) \quad E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \,\middle|\, S\right],$$

which appears in Lemma A.3. We will prove the following lemma. Then, Theorem A.1 can be proved by combining Lemmas A.2, A.3 and A.4 taking $1 \quad 2e^{-\frac{1}{2}\epsilon^2 N} \geq \frac{1}{2}$ (the only case we need to consider).

Lemma A.4. For *any* h and *any* S,

$$\mathbb{P}\left[|E_{\text{in}}(h) - E'_{\text{in}}(h)| > \tfrac{\epsilon}{2} \mid S\right] \leq 2e^{-\frac{1}{8}\epsilon^2 N},$$

where the probability is over random partitions of S into two sets $\mathcal{D}$ and $\mathcal{D}'$.

Proof. To prove the result, we will use a result, which is also due to Hoeffding, for sampling *without replacement*:

Lemma A.5 (Hoeffding, 1963)**.** Let $\mathcal{A} = \{a_1, \ldots, a_{2N}\}$ be a set of values with $a_n \in [0,1]$, and let $\mu = \frac{1}{2N}\sum_{n=1}^{2N} a_n$ be their mean. Let $\mathcal{D} = \{z_1, \ldots, z_N\}$ be a sample of size N, sampled from $\mathcal{A}$ uniformly *without* replacement. Then

$$\mathbb{P}\left[\left|\frac{1}{N}\sum_{n=1}^{N} z_n - \mu\right| > \epsilon\right] \leq 2e^{-2\epsilon^2 N}.$$

We apply Lemma A.5 as follows. For the $2N$ examples in S, let $a_n = 1$ if $h(\mathbf{x}_n) \neq y_n$ and $a_n = 0$ otherwise. The $\{a_n\}$ are the errors made by h on S. Now randomly partition S into $\mathcal{D}$ and $\mathcal{D}'$, i.e., sample N examples from S without replacement to get $\mathcal{D}$, leaving the remaining N examples for $\mathcal{D}'$. This results in a sample of size N of the $\{a_n\}$ for $\mathcal{D}$, sampled uniformly without replacement. Note that

$$E_{\text{in}}(h) = \frac{1}{N}\sum_{a_n \in \mathcal{D}} a_n, \quad \text{and} \quad E'_{\text{in}}(h) = \frac{1}{N}\sum_{a'_n \in \mathcal{D}'} a'_n.$$

Since we are sampling without replacement, $S = \mathcal{D} \cup \mathcal{D}'$ and $\mathcal{D} \cap \mathcal{D}' = \emptyset$, and so

$$\mu \quad \frac{1}{2N}\sum_{n \quad 1}^{2N} a_n - \frac{E_{\text{in}}(h) + E'_{\text{in}}(h)}{2}.$$

It follows that $|E_{\text{in}} - \mu| > t \iff |E_{\text{in}} \quad E'_{\text{in}}| > 2t$. By Lemma A.5,

$$\mathbb{P}\left[|E_{\text{in}}(h) - E'_{\text{in}}(h)| > 2t\right] \leq 2e^{-2t^2 N}.$$

Substituting $t = \frac{\epsilon}{4}$ gives the result. ■

e-Appendix B

Linear Algebra

The basic storage structure for the data is a matrix X which has N rows, one for each data point; each data point is a row-vector.

$$\mathrm{X} = \begin{bmatrix} \text{—}\mathbf{x}_1^{\mathrm{T}}\text{—} \\ \text{—}\mathbf{x}_2^{\mathrm{T}}\text{—} \\ \vdots \\ \text{—}\mathbf{x}_N^{\mathrm{T}}\text{—} \end{bmatrix} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,d} \\ & \vdots & & \\ x_{N,1} & x_{N,2} & \cdots & x_{N,d} \end{bmatrix}$$

When we write $\mathrm{X} \in \mathbb{R}^{N \times d}$ we mean a matrix like the one above, which in this case is $N \times d$ (N rows and d columns). Linear algebra plays an important role in manipulating such matrices when learning from data, because the matrix X can be viewed as a linear operator that takes a set of weights $\mathbf{w}$ and outputs in-sample predictions:

$$\hat{\mathbf{y}} = \mathrm{X}\mathbf{w}.$$

B.1 Basic Properties of Vectors and Matrices

A (column) vector $\mathbf{v} \in \mathbb{R}^d$ has d components $v_1, \ldots, v_d$. The matrix X above could be viewed as a set of d column vectors or a set of N row vectors. A vector can be multiplied by a scalar in the usual way, by multiplying each component by the scalar, and two vectors can be added together by adding the respective components together.

The vectors $\mathbf{v}_1, \ldots, \mathbf{v}_m \in \mathbb{R}^d$ are linearly independent if no non-trivial linear combination of them can equal $\mathbf{0}$:

$$\sum_{i=1}^{m} \alpha_i \mathbf{v}_i = \mathbf{0} \implies \alpha_i = 0 \text{ for } i = 1, \ldots, m.$$

If the vectors are not linearly independent, then they are linearly dependent. Given two vectors $\mathbf{v}, \mathbf{u}$, the standard Euclidean inner product (dot product)

is $\mathbf{v}^{\mathrm{T}}\mathbf{u} = \sum_{i=1}^{d} v_i u_i$ and the Euclidean norm is $\|\mathbf{v}\|^2 = \mathbf{v}^{\mathrm{T}}\mathbf{v} = \sum_{i=1}^{d} v_i^2$. Let θ be the angle between $\mathbf{v}$ and $\mathbf{u}$ in the standard geometric sense. Then $\mathbf{v}^{\mathrm{T}}\mathbf{u} = \|\mathbf{v}\|\|\mathbf{u}\|\cos\theta \implies (\mathbf{v}^{\mathrm{T}}\mathbf{u})^2 \leq \|\mathbf{v}\|^2\|\mathbf{u}\|^2$, where the latter inequality is known as the Cauchy-Schwarz inequality. The two vectors $\mathbf{v}$ and $\mathbf{u}$ are orthogonal if $\mathbf{v}^{\mathrm{T}}\mathbf{u} = 0$; if in addition $\|\mathbf{v}\| = \|\mathbf{u}\| = 1$, then the two vectors are orthonormal (orthogonal and have unit norm).

A basis $\mathbf{v}_1, \ldots, \mathbf{v}_d$ has the property that any $\mathbf{u} \in \mathbb{R}^d$ can be written as a *unique* linear combination of the basis vectors, $\mathbf{u} = \sum_{i=1}^{d} \alpha_i \mathbf{v}_i$. It follows that the basis must be linearly independent. (We have implicitly assumed that the cardinality of any basis is d, which is indeed the case.) A basis $\mathbf{v}_1, \ldots, \mathbf{v}_d$ is an orthonormal basis if $\mathbf{v}_i^{\mathrm{T}}\mathbf{v}_j = \delta_{ij}$ (equal to 1 if $i = j$ and zero otherwise), that is the basis vectors are pairwise orthonormal.

Exercise B.1

This exercise introduces some fundamental properties of vectors and bases.

(a) Are the following sets of vectors dependent or independent?

$$\left\{\begin{bmatrix}1\\2\end{bmatrix}\right\} \quad \left\{\begin{bmatrix}1\\2\end{bmatrix}, \begin{bmatrix}2\\4\end{bmatrix}\right\} \quad \left\{\begin{bmatrix}1\\2\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right\} \quad \left\{\begin{bmatrix}1\\2\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}3\\1\end{bmatrix}\right\}$$

(b) Which of the sets in (a) span $\mathbb{R}^2$?

(c) Show that the expansion $\mathbf{u} = \sum_{i=1}^{d} \alpha_i \mathbf{v}_i$ holds for unique α_i if and only if $\mathbf{v}_1, \ldots, \mathbf{v}_d$ are independent.

(d) Which of the sets in (a) are a basis for $\mathbb{R}^2$?

(e) Show that any set of vectors containing the zero vector is dependent.

(f) Show: if $\mathbf{v}_1, \ldots, \mathbf{v}_m \in \mathbb{R}^d$ are independent then $m \leq d$ (the maximum cardinality of an independent set is the dimension d). *[Hint: Induction on d.]*

(g) Show: if $\mathbf{v}_1, \ldots, \mathbf{v}_m$ span $\mathbb{R}^d$, then $m \geq d$. *[Hint: The span does not change if you add a multiple of one of the vectors to another. Hence, transform the set to a more convenient one.]*

(h) Show: every basis has cardinality equal to the dimension d.

(i) Show: If $\mathbf{v}_1, \mathbf{v}_2$ are orthogonal, they are independent. If $\mathbf{v}_1, \mathbf{v}_2$ are independent, then $\mathbf{v}_1, \mathbf{v}_2 - \lambda\mathbf{v}_1$ have the same span and are orthogonal, where $\lambda = \mathbf{v}_1^{\mathrm{T}}\mathbf{v}_2 / \mathbf{v}_1^{\mathrm{T}}\mathbf{v}_1$. What if $\mathbf{v}_1, \mathbf{v}_2$ are dependent?

(j) Show that any set of independent vectors can be transformed to a set of pairwise orthogonal vectors with the same span.

(k) Given a basis $\mathbf{v}_1, \ldots, \mathbf{v}_d$ show how to construct an orthonormal basis $\hat{\mathbf{v}}_1, \ldots, \hat{\mathbf{v}}_d$. *[Hint: Start by making $\mathbf{v}_2, \ldots, \mathbf{v}_d$ orthogonal to $\mathbf{v}_1$.]*

(l) If $\mathbf{v}_1, \ldots, \mathbf{v}_d$ is an orthonormal basis, then show that any vector $\mathbf{u}$ has the (unique) expansion $\mathbf{u} = \sum_{i=1}^{d} (\mathbf{u}^{\mathrm{T}}\mathbf{v}_i)\mathbf{v}_i$.

(m) Show: Any set of d linearly independent vectors is a basis for $\mathbb{R}^d$.

If $\mathbf{v}_1, \ldots, \mathbf{v}_d$ is an orthonormal basis, then the coefficients $(\mathbf{u}^{\mathrm{T}}\mathbf{v}_i)$ in the expansion $\mathbf{u} = \sum_{i=1}^{d}(\mathbf{u}^{\mathrm{T}}\mathbf{v}_i)\mathbf{v}_i$ are the coordinates of $\mathbf{u}$ with respect to the (ordered) basis $\mathbf{v}_1, \ldots, \mathbf{v}_d$. These d coordinates form a vector in d dimensions. When we write $\mathbf{u}$ as a vector of its coordinates, as we have been doing, we are implicitly assuming that these coordinates are with respect to the standard orthonormal basis $\mathbf{e}_1, \ldots, \mathbf{e}_d$:

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \mathbf{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \ldots, \quad \mathbf{e}_d = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

Matrices. Let $\mathrm{A} \in \mathbb{R}^{N\times d}$ ($N \times d$ matrix) and $\mathrm{B}, \mathrm{C} \in \mathbb{R}^{d\times M}$ be arbitrary real valued matrices. For the (i, j)-th entry of a matrix, we may write A_{ij} or $[\mathrm{A}]_{ij}$ (the latter when we want to explicitly identify A as a matrix). The transpose $\mathrm{A}^{\mathrm{T}} \in \mathbb{R}^{d\times N}$ is a matrix whose entries are given by $[\mathrm{A}^{\mathrm{T}}]_{ij} = [\mathrm{A}]_{ji}$. The matrix-vector and matrix-matrix products are defined in the usual way,

$$\begin{aligned} [\mathrm{A}\mathbf{x}]_i &= \textstyle\sum_{j=1}^{d} A_{ij}x_j, && \text{where } \mathbf{x} \in \mathbb{R}^d,\ \mathrm{A} \in \mathbb{R}^{N\times d} \text{ and } \mathrm{A}\mathbf{x} \in \mathbb{R}^N; \\ [\mathrm{AB}]_{ij} &= \textstyle\sum_{k=1}^{d} \mathrm{A}_{ik}\mathrm{B}_{kj}, && \text{where } \mathrm{A} \in \mathbb{R}^{N\times d},\ B \in \mathbb{R}^{d\times M} \text{ and } \mathrm{AB} \in \mathbb{R}^{N\times M}. \end{aligned}$$

In general, when we refer to products of matrices below, assume that all the products exist. Note that $\mathrm{A}(\mathrm{B}+\mathrm{C}) = \mathrm{AB}+\mathrm{AC}$ and $(\mathrm{AB})^{\mathrm{T}} = \mathrm{B}^{\mathrm{T}}\mathrm{A}^{\mathrm{T}}$. The $d\times d$ identity matrix I_d is the matrix whose columns are the standard basis, having diagonal entries 1 and zeros elsewhere. If A is a square matrix ($d = N$), the inverse A^{-1} (if it exists) satisfies

$$\mathrm{AA}^{-1} = \mathrm{A}^{-1}\mathrm{A} = \mathrm{I}_d.$$

Note that

$$\begin{aligned} (\mathrm{A}^{\mathrm{T}})^{-1} &= (\mathrm{A}^{-1})^{\mathrm{T}}; \\ (\mathrm{AB})^{-1} &= (\mathrm{B})^{-1}\mathrm{A}^{-1}. \end{aligned}$$

A matrix is invertible if and only if its columns (also rows) are linearly independent (in which case the columns are a basis).

If the matrix A has orthonormal columns, then $\mathrm{A}^{\mathrm{T}}\mathrm{A} = \mathrm{I}_d$; if in addition A is square, then it is orthogonal (and invertible). It is often convenient to refer to the columns of a matrix, and we will write $\mathrm{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_d]$. Using this notation, one can write a matrix vector product as

$$\mathrm{A}\mathbf{x} = \sum_{i=1}^{d} x_i\mathbf{a}_i,$$

from which we see that a matrix can be viewed as an operator that takes the input $\mathbf{x}$ and transforms it into a linear combination of its columns. The range of a matrix A is the subspace spanned by its columns, $\text{range}(\mathrm{A}) = \text{span}(\{\mathbf{a}_1, \ldots, \mathbf{a}_d\})$. It is also useful to define the subspace spanned by the rows of A, which is the range of A^{T}. The dimension of the range of A is called the column-rank of A. Similarly, the dimension of the subspace spanned by the rows is the row-rank of A. It is a useful fact that the row and column ranks are equal, and so we can define the rank of a matrix, denoted ρ, as the dimension of its range. Note that

$$\rho(\mathrm{A}) = \text{rank}(\mathrm{A}) = \text{rank}(\mathrm{A}^{\mathrm{T}}\mathrm{A}) = \text{rank}(\mathrm{A}\mathrm{A}^{\mathrm{T}}).$$

The matrix $\mathrm{A} \in \mathbb{R}^{N \times d}$ has full column rank if $\text{rank}(\mathrm{A}) = d$; it has full row rank if $\text{rank}(\mathrm{A}) = N$. Note that $\text{rank}(\mathrm{A}) \leq \min(N, d)$, since the dimension of a space spanned by ℓ vectors is at most ℓ (see Exercise B.1(g)).

Exercise B.2

Let $\mathrm{A} = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$, $\mathrm{B} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 0 \end{bmatrix}$, $\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$.

(a) Compute (if the quantities exist): AB; $\mathrm{A}\mathbf{x}$; $\mathrm{B}\mathbf{x}$; BA; $\mathrm{B}^{\mathrm{T}}\mathrm{A}^{\mathrm{T}}$; $\mathbf{x}^{\mathrm{T}}\mathrm{A}\mathbf{x}$; $\mathrm{B}^{\mathrm{T}}\mathrm{AB}$; A^{-1}.

(b) Show that $\mathbf{x}^{\mathrm{T}}\mathrm{A}\mathbf{x} = \sum_{i=1}^{3}\sum_{j=1}^{3} x_i x_j \mathrm{A}_{ij}$.

(c) Find a $\mathbf{v}$ for which $\mathrm{A}\mathbf{v} = \lambda\mathbf{v}$. What are the possible choices of λ?

(d) How many linearly independent rows are there in B? How many linearly independent columns are there in B?

(e) Given any matrix $\mathrm{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_d]$, let $\mathbf{c}_1, \ldots, \mathbf{c}_r$ be a basis for the span of $\mathbf{a}_1, \ldots, \mathbf{a}_d$. Let C be the matrix whose columns are this basis, $C = [\mathbf{c}_1, \ldots, \mathbf{c}_r]$. Show that for some matrix R, one can write

$$\mathrm{A} = \mathrm{CR}.$$

(i) What is the column-rank of A?

(ii) What are the dimensions of R?

(iii) Show that every row in A is a linear combination of rows in R.

(iv) Hence, show that the dimension of the subspace spanned by the rows of A is at most r, the column-rank. That is,

$$\text{column-rank}(\mathrm{A}) \geq \text{row-rank}(\mathrm{A}).$$

(v) Show that $\text{column-rank}(\mathrm{A}) \leq \text{row-rank}(\mathrm{A})$, and hence that the column-rank equals the row-rank. *[Hint: Consider* A^{T}*.]*

B.2 SVD and Pseudo-Inverse

The singular value decomposition (SVD) of a matrix A is one of the most useful matrix decompositions. For the matrix A, assume $d \leq N$ and the rank $\rho \leq d$. The SVD of A factorizes A into the product of three special matrices:

$$\mathrm{A} = \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}.$$

The matrix $\mathrm{U} \in \mathbb{R}^{N\times\rho}$ has orthonormal columns that are called the left singular vectors of A; the matrix $\mathrm{V} \in \mathbb{R}^{d\times\rho}$ has orthonormal columns that are called right singular vectors of A. So,

$$\mathrm{U}^{\mathrm{T}}\mathrm{U} = \mathrm{V}^{\mathrm{T}}\mathrm{V} = \mathrm{I}_{\rho}.$$

The matrix $\Gamma \in \mathbb{R}^{\rho\times\rho}$ is diagonal and has as its diagonal entries the (positive) singular values of A, $\gamma_i = [\Gamma]_{ii}$. Typically, the singular values are ordered, so that $\gamma_1 \geq \cdots \geq \gamma_\rho$; in this case, the first column of U is called the top left singular vector, and similarly the first column of V is called the top right singular vector. The condition number of A is the ratio of the largest to smallest singular values: $\kappa = \gamma_1/\gamma_\rho$, which plays an important role in the stability of algorithms involving matrices, such as solving the linear regression problem or inverting the matrix. Algorithms exist to compute the SVD in $O(Nd\min(N,d))$ time. If only a few of the top singular vectors and singular values are needed, these can be obtained more efficiently using iterative subspace methods such as power iteration.

If A is not invertible (for example, not square), it is useful to define the Moore-Penrose pseudo-inverse $\mathrm{A}^\dagger$, which satisfies four properties:

$$(i)\ \mathrm{A}\mathrm{A}^\dagger\mathrm{A} = \mathrm{A};\ (ii)\ \mathrm{A}^\dagger\mathrm{A}\mathrm{A}^\dagger = \mathrm{A}^\dagger;\ (iii)\ (\mathrm{A}\mathrm{A}^\dagger)^{\mathrm{T}} = \mathrm{A}\mathrm{A}^\dagger;\ (iv)\ (\mathrm{A}^\dagger\mathrm{A})^{\mathrm{T}} = \mathrm{A}^\dagger\mathrm{A}.$$

The pseudo-inverse functions as an inverse and plays an important role in linear regression.

Exercise B.3

If the SVD of A is $\mathrm{A} = \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}$, by checking all four properties, verify that

$$\mathrm{A}^\dagger = \mathrm{V}\Gamma^{-1}\mathrm{U}^{\mathrm{T}}$$

is a pseudo-inverse of A. Also show that $(\mathrm{A}^{\mathrm{T}})^\dagger = (\mathrm{A}^\dagger)^{\mathrm{T}}$.

If A and B both have rank d or if one of $\mathrm{A}, \mathrm{B}^{\mathrm{T}}$ has orthonormal columns, then $(\mathrm{AB})^\dagger = (\mathrm{B})^\dagger\mathrm{A}^\dagger$. The matrix $\Pi_{\mathrm{A}} = \mathrm{A}\mathrm{A}^\dagger = \mathrm{U}\mathrm{U}^{\mathrm{T}}$ is the projection operator onto the range (column-space) of A; this projection operator can be used to decompose a vector $\mathbf{x}$ into two orthogonal parts, the part $\mathbf{x}_{\mathrm{A}}$ in the range of A and the part $\mathbf{x}_{\mathrm{A}_\perp}$ orthogonal to the range of A:

$$\mathbf{x} = \underbrace{\Pi_{\mathrm{A}}\mathbf{x}}_{\mathbf{x}_{\mathrm{A}}} + \underbrace{(\mathrm{I} - \Pi_{\mathrm{A}})\mathbf{x}}_{\mathbf{x}_{\mathrm{A}_\perp}}.$$

A projection operator satisfies $\Pi^2 = \Pi$. For example $\Pi_{\mathrm{A}}^2 = (\mathrm{A}\mathrm{A}^\dagger\mathrm{A})\mathrm{A}^\dagger = \mathrm{A}\mathrm{A}^\dagger$.

B.3 Symmetric Matrices

A square matrix is symmetric if $\mathrm{A} = \mathrm{A}^{\mathrm{T}}$ (anti-symmetric if $\mathrm{A} = -\mathrm{A}^{\mathrm{T}}$). Symmetric matrices play a special role because the covariance matrix of the data is symmetric. If X is the data matrix, then the centered data matrix $\mathrm{X}_{\mathrm{cen}}$ is obtained by subtracting from each data point the mean vector

$$\boldsymbol{\mu} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n = \frac{1}{N}\mathrm{X}^{\mathrm{T}}\mathbf{1},$$

where $\mathbf{1}$ is the N-dimensional vector of ones. In matrix form,

$$\begin{aligned}\mathrm{X}_{\mathrm{cen}} &= \mathrm{X} - \mathbf{1}\boldsymbol{\mu}^{\mathrm{T}} \\ &= \mathrm{X} - \frac{1}{N}\mathbf{1}\mathbf{1}^{\mathrm{T}}\mathrm{X} \\ &= \left(\mathrm{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^{\mathrm{T}}\right)\mathrm{X}.\end{aligned}$$

From this expression, it is easy to see why $\left(\mathrm{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^{\mathrm{T}}\right)$ is called the centering operator. The covariance matrix Σ is given by

$$\Sigma = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu})(\mathbf{x}_n - \boldsymbol{\mu})^{\mathrm{T}} = \frac{1}{N}\mathrm{X}_{\mathrm{cen}}^{\mathrm{T}}\mathrm{X}_{\mathrm{cen}}.$$

Via the decomposition $\mathrm{A} = \frac{1}{2}(\mathrm{A} + \mathrm{A}^{\mathrm{T}}) + \frac{1}{2}(\mathrm{A} - \mathrm{A}^{\mathrm{T}})$, every matrix can be decomposed into the sum of its symmetric part and its anti-symmetric part. Analogous to the SVD, the spectral theorem says that any symmetric matrix admits a spectral or eigen-decomposition of the form

$$\mathrm{A} = \mathrm{U}\Lambda\mathrm{U}^{\mathrm{T}},$$

where U has orthonormal columns that are the *eigenvectors* of A, so $\mathrm{U}^{\mathrm{T}}\mathrm{U} = \mathrm{I}_\rho$ and Λ is diagonal with entries $\lambda_i = [\Lambda]_{ii}$ that are the eigenvalues of A. Each column $\mathbf{u}_i$ of U for $i = 1, \ldots, \rho$ is an eigenvector of A with eigenvalue λ_i (the number of non-zero eigenvalues is equal to the rank). Via the identity $\mathrm{AU} = \mathrm{U}\Lambda$, one can verify that the eigenvalues and corresponding eigenvectors satisfy the relation

$$\mathrm{A}\mathbf{u}_i = \lambda_i\mathbf{u}_i.$$

All eigenvalues of a symmetric matrix are real. Eigenvalues and eigenvectors can be defined more generally using the above equation (even for non-symmetric matrices), but we will only be concerned with symmetric matrices. If the λ_i are ordered, with $\lambda_1 \geq \cdots \geq \lambda_\rho$, then λ_1 is the top eigenvalue, with corresponding eigenvector $\mathbf{u}_1$, and so on. Note that the eigen-decomposition is similar to the SVD ($\mathrm{A} = \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}$) with the flexibility to have negative entries along the diagonal in Λ. Since $\mathrm{AA}^{\mathrm{T}} = \mathrm{U}\Lambda^2\mathrm{U}^{\mathrm{T}} = \mathrm{U}\Gamma^2\mathrm{U}^{\mathrm{T}}$, one identifies that $\lambda_i^2 = \gamma_i^2$. That is, up to a sign, the eigenvalues and singular values of a symmetric matrix are the same.

B.3.1 Positive Semi-Definite Matrices

The matrix A is symmetric positive semi-definite (SPSD) if it is symmetric and for every non-zero $\mathbf{x} \in \mathbb{R}^n$,

$$\mathbf{x}^{\mathrm{T}} \mathrm{A} \mathbf{x} \geq 0.$$

One writes $\mathrm{A} \succeq 0$; if for every non-zero $\mathbf{x}$,

$$\mathbf{x}^{\mathrm{T}} \mathrm{A} \mathbf{x} > 0,$$

then A is positive definite (PD) and one writes $\mathrm{A} \succ 0$. If A is positive definite, then $\lambda_i > 0$ (all its eigenvalues are positive), in which case $\lambda_i = \gamma_i$ and the eigen-decomposition and the SVD are identical. We can write $\Lambda = \mathrm{S}^2$ and identify $\mathrm{A} = \mathrm{US}(\mathrm{US})^{\mathrm{T}}$ from which one deduces that every SPSD matrix has the form $\mathrm{A} = \mathrm{ZZ}^{\mathrm{T}}$. The covariance matrix is an example of an SPSD matrix.

B.4 Trace and Determinant

The trace and determinant are defined for a square matrix A (so $N = d$). The trace is the sum of the diagonal elements,

$$\mathrm{trace}(\mathrm{A}) = \sum_{i=1}^{N} \mathrm{A}_{ii}.$$

The trace operator is cyclic:

$$\mathrm{trace}(\mathrm{AB}) = \mathrm{trace}(\mathrm{BA})$$

(when both products are defined). From the cyclic property, if O has orthonormal columns (so $\mathrm{O}^{\mathrm{T}}\mathrm{O} = \mathrm{I}$), then $\mathrm{trace}(\mathrm{OAO}^{\mathrm{T}}) = \mathrm{trace}(\mathrm{A})$. Setting $\mathrm{O} = \mathrm{U}$ from the eigen-decomposition of A,

$$\mathrm{trace}(\mathrm{A}) = \mathrm{trace}(\Lambda) = \sum_{i=1}^{N} \lambda_i$$

(sum of eigenvalues). Note that $\mathrm{trace}(\mathrm{I}_k) = k$.

The determinant of A, written $|\mathrm{A}|$, is most easily defined using the totally anti-symmetric Levi-Civita symbol $\varepsilon_{i_1,\ldots,i_N}$ (if you swap any two indices then the value negates, and $\varepsilon_{1,2,\ldots,N} = 1$; if any index is repeated, the value is zero):

$$|\mathrm{A}| = \sum_{i_1,\ldots,i_N=1}^{N} \varepsilon_{i_1,\ldots,i_N} \mathrm{A}_{1i_1} \cdots \mathrm{A}_{Ni_N}.$$

Since every summand has exactly one term from each column, if you scale any column, the determinant is correspondingly scaled. The anti-symmetry

of $\varepsilon_{i_1,\ldots,i_N}$ implies that if you swap any two columns (or rows) of A, then the determinant changes sign. If any two columns are identical, by swapping them, the determinant must change sign, and hence the determinant must be zero. If you add a vector to a column, then the determinant becomes a sum,

$$|[\mathbf{a}_1,\ldots,\mathbf{a}_i+\mathbf{v},\ldots,\mathbf{a}_N]| = |[\mathbf{a}_1,\ldots,\mathbf{a}_i,\ldots,\mathbf{a}_N]| + |[\mathbf{a}_1,\ldots,\mathbf{v},\ldots,\mathbf{a}_N]|.$$

If $\mathbf{v}$ is a multiple of a column then the second term vanishes, and so adding a multiple of one column to another column does not change the determinant. By separating out the summation with respect to i_1 in the definition of the determinant (the first row of A), we get a useful recursive formula for the determinant known as its expansion by cofactors. We illustrate for a 3×3 matrix below:

$$\begin{aligned}
\begin{vmatrix} \mathrm{A}_{11} & \mathrm{A}_{12} & \mathrm{A}_{13} \\ \mathrm{A}_{21} & \mathrm{A}_{22} & \mathrm{A}_{23} \\ \mathrm{A}_{31} & \mathrm{A}_{32} & \mathrm{A}_{33} \end{vmatrix} &= \begin{matrix} \mathrm{A}_{11} & \mathrm{A}_{12} & \mathrm{A}_{13} \\ \mathrm{A}_{21} & \multirow{2}{*}{$\begin{vmatrix} \mathrm{A}_{22} & \mathrm{A}_{23} \\ \mathrm{A}_{32} & \mathrm{A}_{33} \end{vmatrix}$} & \\ \mathrm{A}_{31} & & \end{matrix} - \begin{vmatrix} \mathrm{A}_{11} & \mathrm{A}_{12} & \mathrm{A}_{13} \\ \mathrm{A}_{21} & \mathrm{A}_{22} & \mathrm{A}_{23} \\ \mathrm{A}_{31} & \mathrm{A}_{32} & \mathrm{A}_{33} \end{vmatrix} + \begin{matrix} \mathrm{A}_{11} & \mathrm{A}_{12} & \mathrm{A}_{13} \\ \multirow{2}{*}{$\begin{vmatrix} \mathrm{A}_{21} & \mathrm{A}_{22} \\ \mathrm{A}_{31} & \mathrm{A}_{32} \end{vmatrix}$} & & \mathrm{A}_{23} \\ & & \mathrm{A}_{33} \end{matrix} \\
&= \mathrm{A}_{11}(\mathrm{A}_{22}\mathrm{A}_{33}-\mathrm{A}_{23}\mathrm{A}_{32}) - \mathrm{A}_{12}(\mathrm{A}_{21}\mathrm{A}_{33}-\mathrm{A}_{23}\mathrm{A}_{31}) \\
&\quad + \mathrm{A}_{13}(\mathrm{A}_{21}\mathrm{A}_{32}-\mathrm{A}_{22}\mathrm{A}_{31}).
\end{aligned}$$

(Notice the alternating signs as we expand along the first row.) Geometrically, the determinant of A equals the volume enclosed by the parallelepiped whose sides are the column vectors $\mathbf{a}_1,\ldots,\mathbf{a}_N$. This geometric fact is important when changing variables in a multi-dimensional integral, because the infinitesimal volume element transforms according to the determinant of the Jacobian. A useful fact about the determinant of a product of square matrices is

$$|\mathrm{AB}| = |\mathrm{A}||\mathrm{B}|.$$

Exercise B.4

Show the following useful properties of determinants.

(a) $|\mathrm{I}_k| = 1$; if D is diagonal, then $|D| = \prod_{i=1}^N [D]_{ii}$.

(b) Show that $|\mathrm{A}| = \dfrac{1}{N!}\displaystyle\sum_{i_1,\ldots,i_N}\sum_{j_1,\ldots,j_N} \varepsilon_{i_1\ldots i_N}\varepsilon_{j_1\ldots j_N}\mathrm{A}_{i_1j_1}\cdots\mathrm{A}_{i_Nj_N}$.

(c) Using (b), argue that $|\mathrm{A}| = |\mathrm{A}^{\mathrm{T}}|$.

(d) If O is orthogonal (square with orthonormal columns, i.e. $\mathrm{O}^{\mathrm{T}}\mathrm{O} = \mathrm{I}_N$), then $|\mathrm{O}| = \pm 1$ and $|\mathrm{OAO}^{\mathrm{T}}| = |\mathrm{A}|$. *[Hint: $|\mathrm{O}^{\mathrm{T}}\mathrm{O}| = |\mathrm{O}||\mathrm{O}^{\mathrm{T}}|$.]*

(e) $|\mathrm{A}^{-1}| = 1/|\mathrm{A}|$. *[Hint: $|\mathrm{A}^{-1}\mathrm{A}| = |\mathrm{I}_n|$.]*

(f) For symmetric A,

$$|\mathrm{A}| = \prod_{i=1}^n \lambda_i \qquad \text{(product of eigenvalues of A).}$$

[Hint: $\mathrm{A} = \mathrm{U}\Lambda\mathrm{U}^{\mathrm{T}}$, where U is orthogonal.]

B.4.1 Inverse and Determinant Identities

The inverse of the covariance matrix plays a role in many learning algorithms. Hence, it is important to be able to update the inverse efficiently for small changes in a matrix. If A and B are square and invertible, then the following useful identities are known as the Sherman-Morrison-Woodbury inversion and determinant formulae:

$$\begin{aligned}(\mathrm{A}+\mathrm{XBY}^{\mathrm{T}})^{-1} &= \mathrm{A}^{-1}-\mathrm{A}^{-1}\mathrm{X}(\mathrm{B}^{-1}+\mathrm{Y}^{\mathrm{T}}\mathrm{A}^{-1}\mathrm{X})^{-1}\mathrm{Y}^{\mathrm{T}}\mathrm{A}^{-1};\\ |\mathrm{A}+\mathrm{XBY}^{\mathrm{T}}| &= |\mathrm{A}||\mathrm{B}||\mathrm{B}^{-1}+\mathrm{Y}^{\mathrm{T}}\mathrm{A}^{-1}\mathrm{X}|.\end{aligned}$$

An important special case is the inverse and determinant updates due to a rank 1 update of a matrix. Setting $\mathrm{X}=\mathbf{x}$, $\mathrm{B}=1$ and $\mathrm{Y}=\mathbf{y}$,

$$\begin{aligned}(\mathrm{A}+\mathbf{x}\mathbf{y}^{\mathrm{T}})^{-1} &= \mathrm{A}^{-1}-\frac{\mathrm{A}^{-1}\mathbf{x}\mathbf{y}^{\mathrm{T}}\mathrm{A}^{-1}}{1+\mathbf{y}^{\mathrm{T}}\mathrm{A}^{-1}\mathbf{x}};\\ |\mathrm{A}+\mathbf{x}\mathbf{y}^{\mathrm{T}}| &= |\mathrm{A}|(1+\mathbf{y}^{\mathrm{T}}\mathrm{A}^{-1}\mathbf{x}).\end{aligned}$$

Setting $\mathbf{x}=\pm\mathbf{y}$ gives the updates for $\mathrm{A}\pm\mathbf{x}\mathbf{x}^{\mathrm{T}}$. If A has a block representation, we can get similar updates to the inverse and determinant. Let

$$\mathrm{A}=\begin{bmatrix}\mathrm{A}_{11} & \mathrm{A}_{12}\\ \mathrm{A}_{21} & \mathrm{A}_{22}\end{bmatrix},$$

and define

$$\begin{aligned}\mathrm{F}_1 &= \mathrm{A}_{11}-\mathrm{A}_{12}\mathrm{A}_{22}^{-1}\mathrm{A}_{21}\\ \mathrm{F}_2 &= \mathrm{A}_{22}-\mathrm{A}_{21}\mathrm{A}_{11}^{-1}\mathrm{A}_{12}\end{aligned}$$

Then,

$$\begin{aligned}\mathrm{A}^{-1} &= \begin{bmatrix}\mathrm{F}_1^{-1} & -\mathrm{A}_{11}^{-1}\mathrm{A}_{12}\mathrm{F}_2^{-1}\\ -\mathrm{F}_2^{-1}\mathrm{A}_{21}\mathrm{A}_{11}^{-1} & \mathrm{F}_2^{-1}\end{bmatrix}, \qquad \text{and}\\ |\mathrm{A}| &= |A_{22}||\mathrm{F}_1|=|\mathrm{A}_{11}||\mathrm{F}_2|.\end{aligned}$$

Again, an important special case is when

$$\mathrm{A}=\begin{bmatrix}\mathrm{X} & \mathbf{b}\\ \mathbf{b}^{\mathrm{T}} & c\end{bmatrix}.$$

In this case,

$$\begin{aligned}\mathrm{A}^{-1} &= \begin{bmatrix}\mathrm{X}^{-1} & \mathbf{0}\\ \mathbf{0}^{\mathrm{T}} & 0\end{bmatrix}+\frac{1}{c-\mathbf{b}^{\mathrm{T}}\mathrm{X}^{-1}\mathbf{b}}\begin{bmatrix}\mathrm{X}^{-1}\mathbf{b}\mathbf{b}^{\mathrm{T}}\mathrm{X}^{-1} & -\mathrm{X}^{-1}\mathbf{b}\\ -\mathbf{b}^{\mathrm{T}}\mathrm{X}^{-1} & 1\end{bmatrix}, \qquad \text{and}\\ |\mathrm{A}| &= |\mathrm{X}|(c-\mathbf{b}^{\mathrm{T}}\mathrm{X}^{-1}\mathbf{b}).\end{aligned}$$

Exercise B.5

Use the formula for the determinant of a 2×2 block matrix to show Sylvester's determinant theorem for matrices $\mathrm{A} \in \mathbb{R}^{N \times d}$ and $\mathrm{B} \in \mathbb{R}^{d \times N}$:

$$|\mathrm{I}_n + \mathrm{AB}| = |\mathrm{I}_d + \mathrm{BA}|.$$

$\left[\textit{Hint: Consider} \begin{vmatrix} \mathrm{I}_n & \mathrm{A} \\ -\mathrm{B} & \mathrm{I}_d \end{vmatrix}\right]$

Use Sylvester's theorem to show the Sherman-Morrison-Woodbury determinant identity

$$|\mathrm{A} + \mathrm{XBY}^{\mathrm{T}}| = |\mathrm{A}||\mathrm{B}||\mathrm{B}^{-1} + \mathrm{Y}^{\mathrm{T}}\mathrm{A}^{-1}\mathrm{X}|.$$

[Hint: $\mathrm{A} + \mathrm{XBY}^{\mathrm{T}} = \mathrm{A}(\mathrm{I} + \mathrm{A}^{-1}\mathrm{XBY}^{\mathrm{T}})$*, and use Sylvester's theorem.]*

B.5 Inner Products, Matrix and Vector Norms

For vectors $\mathbf{x}$ and $\mathbf{z}$, the inner product is a symmetric, bilinear positive definite function usually denoted $\langle \mathbf{x}, \mathbf{z} \rangle$. The standard Euclidean inner product (dot product) is and example:

$$\langle \mathbf{x}, \mathbf{z} \rangle = \mathbf{x} \bullet \mathbf{z} = \mathbf{x}^{\mathrm{T}}\mathbf{z} = \sum_{i=1}^{d} x_i z_i.$$

The inner-product induced norm of a vector $\mathbf{x}$ (the Euclidean norm) is

$$\|\mathbf{x}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle = \mathbf{x}^{\mathrm{T}}\mathbf{x} = \sum_{i=1}^{d} x_i^2,$$

where the latter two equalities are for the Euclidean inner product. The Cauchy-Schwarz inequality bounds the inner-product by the induced norms,

$$\langle \mathbf{x}, \mathbf{z} \rangle^2 \leq \|\mathbf{x}\|^2 \|\mathbf{z}\|^2,$$

with equality if and only if $\mathbf{x}$ and $\mathbf{z}$ are linearly dependent. The special case of the Cauchy-Schwarz inequality for the Euclidean inner product was given earlier in the chapter, $(\mathbf{x} \bullet \mathbf{z}) = (\sum_{i=1}^{d} x_i z_i)^2 \leq \sum_{i=1}^{d} x_i^2 \sum_{j=1}^{d} z_j^2$. The Pythagorean theorem applies to the square of a sum of vectors, and can be obtained by expanding $(\mathbf{x} + \mathbf{y})^{\mathrm{T}}(\mathbf{x} + \mathbf{y})$ to obtain:

$$\|\mathbf{x} + \mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 + 2\mathbf{x}^{\mathrm{T}}\mathbf{y}.$$

If $\mathbf{x}$ is orthogonal to $\mathbf{y}$, then $\|\mathbf{x} + \mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2$, which is the familiar Pythagorean theorem from geometry where $\mathbf{x}+\mathbf{y}$ is the diagonal of the triangle with sides given by the vectors $\mathbf{x}$ and $\mathbf{y}$.

Associated with any vector norm is the spectral (or operator) matrix norm $\|\mathrm{A}\|_2$ which measures how large a vector transformed by A can get.

$$\|\mathrm{A}\|_2 = \max_{\|\mathbf{x}\|=1} \|\mathrm{A}\mathbf{x}\|;$$

If U has orthonormal columns then $\|\mathrm{U}\mathbf{x}\| = \|\mathbf{x}\|$ for any $\mathbf{x}$. Analogous to the Euclidean norm of a vector is the Frobenius matrix norm $\|\mathrm{A}\|_F$ which sums the squares of the entries:

$$\|\mathrm{A}\|_F^2 = \mathrm{trace}(\mathrm{A}\mathrm{A}^{\mathrm{T}}) = \mathrm{trace}(\mathrm{A}^{\mathrm{T}}\mathrm{A}) = \sum_{i=1}^{n}\sum_{j=1}^{d} \mathrm{A}_{ij}^2.$$

Exercise B.6

Using the SVD of A, $\mathrm{A} = \mathrm{U}\Gamma\mathrm{V}^{\mathrm{T}}$, show that

$$\begin{aligned} \|\mathrm{A}\|_2 &= \gamma_1 && \text{(top singular value);} \\ \|\mathrm{A}\|_F^2 &= \sum_{i=1}^{\rho} \gamma_i^2 && \text{(sum of squared singular values).} \end{aligned}$$

Note that

$$\|\mathrm{A}\|_{2,F} = \|\mathrm{A}^{\mathrm{T}}\|_{2,F};$$
$$\|\mathrm{A}\|_2 \le \|\mathrm{A}\|_F \le \sqrt{\rho}\|\mathrm{A}\|_2,$$

where $\rho = \mathrm{rank}(\mathrm{A})$. The matrix norms satisfy the triangle inequality and a property known as submultiplicativity, which bounds the norm of a product,

$$\begin{aligned} \|\mathrm{A}+\mathrm{B}\|_{2,F} &\le \|\mathrm{A}\|_{2,F} + \|\mathrm{B}\|_{2,F}; \\ \|\mathrm{A}\mathrm{B}\|_2 &\le \|\mathrm{A}\|_2\|\mathrm{B}\|_2; \\ \|\mathrm{A}\mathrm{B}\|_F &\le \|\mathrm{A}\|_2\|\mathrm{B}\|_F. \end{aligned}$$

The generalized Pythagorean theorem states that if $\mathrm{A}^{\mathrm{T}}\mathrm{B} = 0$ or $\mathrm{A}\mathrm{B}^{\mathrm{T}} = 0$ then

$$\|\mathrm{A}+\mathrm{B}\|_F^2 = \|\mathrm{A}\|_F^2 + \|\mathrm{B}\|_F^2$$
$$\max\{\|\mathrm{A}\|_2^2, \|\mathrm{B}\|_2^2\} \le \|\mathrm{A}+\mathrm{B}\|_2^2 \le \|\mathrm{A}\|_2^2 + \|\mathrm{B}\|_2^2.$$

B.5.1 Linear Hyperplanes and Quadratic Forms

A linear scalar function has the form

$$\ell(\mathbf{x}) = w_0 + \mathbf{w}^{\mathrm{T}}\mathbf{x},$$

where $\mathbf{x}, \mathbf{w} \in \mathbb{R}^d$. A hyperplane in d dimensions is defined by all points $\mathbf{x}$ satisfying $\ell(\mathbf{x}) = 0$. For a generic point $\mathbf{x}$, its geometric distance to the hyperplane is given by

$$\mathrm{distance}(\mathbf{x}, (w_0, \mathbf{w})) = \frac{|w_0 + \mathbf{w}^{\mathrm{T}}\mathbf{x}|}{\|\mathbf{w}\|}.$$

The vector $\mathbf{w}$ is normal to the hyperplane. A quadratic form $q(\mathbf{x})$ is a scalar function

$$q(\mathbf{x}) = w_0 + \mathbf{w}^{\mathrm{T}}\mathbf{x} + \tfrac{1}{2}\mathbf{x}^{\mathrm{T}}\mathrm{Q}\mathbf{x}.$$

When Q is positive definite, the manifold $q(\mathbf{x}) = 0$ defines an ellipsoid in d-dimensions (recall that $\mathbf{x} \in \mathbb{R}^d$).

B.6 Vector and Matrix Calculus

Let $E(\mathbf{x})$ be a scalar function of a vector $\mathbf{x} \in \mathbb{R}^d$. The gradient $\nabla E(\mathbf{x})$ is a d-dimensional vector function of $\mathbf{x}$ whose components are the partial derivatives; the Hessian $\mathrm{H}_E(\mathbf{x})$ is a $d \times d$ matrix function of $\mathbf{x}$ whose entries are the second order partial derivatives:

$$\begin{aligned}
[\nabla E(\mathbf{x})]_i &= \frac{\partial}{\partial x_i} E(\mathbf{x}); \\
[\mathrm{H}_E(\mathbf{x})]_{ij} &= \frac{\partial^2}{\partial x_i \partial x_j} E(\mathbf{x}).
\end{aligned}$$

The gradient and Hessian of the linear and quadratic forms are:

$$\begin{array}{rll}
\text{linear form:} & \nabla \ell(\mathbf{x}) = \mathbf{w}; & \mathrm{H}_\ell(\mathbf{x}) = 0 \\
\text{quadratic form:} & \nabla q(\mathbf{x}) = \mathbf{w} + \mathrm{Q}\mathbf{x}; & \mathrm{H}_q(\mathbf{x}) = \mathrm{Q}
\end{array}$$

For the general quadratic term $\mathbf{x}^{\mathrm{T}}\mathrm{A}\mathbf{x}$, the gradient is

$$\nabla(\mathbf{x}^{\mathrm{T}}\mathrm{A}\mathbf{x}) = (\mathrm{A} + \mathrm{A}^{\mathrm{T}})\mathbf{x}.$$

A necessary and sufficient condition for $\mathbf{x}^*$ to be a local minimum of $E(\mathbf{x})$ is that the gradient at $\mathbf{x}^*$ is zero and the Hessian at $\mathbf{x}^*$ is positive definite. The Taylor expansion of $E(\mathbf{x})$ around $\mathbf{x}_0$ up to second order terms is

$$E(\mathbf{x}) = E(\mathbf{x}_0) + \nabla E(\mathbf{x})^{\mathrm{T}}(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^{\mathrm{T}}\mathrm{H}_{\mathrm{E}}(\mathbf{x})(\mathbf{x} - \mathbf{x}_0) + \cdots .$$

If $\mathbf{x}_0$ is a local minimum, the gradient term vanishes and the Hessian term is positive.

B.6.1 Multidimensional Integration

If $\mathbf{z} = \mathbf{q}(\mathbf{x})$ is a vector function of $\mathbf{x}$, then the Jacobian matrix J contains the derivatives of the components of $\mathbf{z}$ with respect to the components of $\mathbf{x}$:

$$[\mathrm{J}]_{ij} = \frac{\partial z_j}{\partial x_i}.$$

The Jacobian is important when performing a change of variables from $\mathbf{x}$ to $\mathbf{z}$ in a multidimensional integral, because it relates the volume element in $\mathbf{x}$-space to the volume element in $\mathbf{z}$-space. Specifically, the volume elements are related by

$$d\mathbf{x} = \frac{1}{|\mathrm{J}|} d\mathbf{z}.$$

As an example of using the Jacobian to transform variables in an integral, consider the multidimensional Gaussian distribution

$$P(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})},$$

where $\boldsymbol{\mu} \in \mathbb{R}^d$ is the mean vector and $\Sigma \in \mathbb{R}^{d\times d}$ is the covariance matrix. We can evaluate the expectations

$$\begin{aligned} \mathbb{E}[\mathbf{x}] &= \int d\mathbf{x}\ \mathbf{x} \cdot P(\mathbf{x}) \qquad \text{and} \\ \mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}] &= \int d\mathbf{x}\ \mathbf{x}\mathbf{x}^{\mathrm{T}} \cdot P(\mathbf{x}) \end{aligned}$$

by making a change of variables. Specifically, let $\Sigma = \mathrm{U}\Lambda\mathrm{U}^{\mathrm{T}}$, and change variables to $\mathbf{z} = \Lambda^{-1/2}\mathrm{U}^{\mathrm{T}}(\mathbf{x}-\boldsymbol{\mu})$. The Jacobian is $\mathrm{J} = \Lambda^{-1/2}\mathrm{U}^{\mathrm{T}}$ with determinant $|\mathrm{J}| = |\Lambda|^{-1/2}$. The integrals for the expectations then transform to

$$\begin{aligned} \mathbb{E}[\mathbf{x}] &= \int d\mathbf{z}\ (\mathrm{U}\Lambda^{1/2}\mathbf{z} + \boldsymbol{\mu}) \cdot \frac{e^{-\frac{1}{2}\mathbf{z}^{\mathrm{T}}\mathbf{z}}}{(2\pi)^{d/2}} \\ &= \boldsymbol{\mu} \\ \mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}] &= \int d\mathbf{z}\ (\mathrm{U}\Lambda^{1/2}\mathbf{z}\mathbf{z}^{\mathrm{T}}\Lambda^{1/2}\mathrm{U}^{\mathrm{T}} + \mathrm{U}\Lambda^{1/2}\mathbf{z}\boldsymbol{\mu}^{\mathrm{T}} + \boldsymbol{\mu}\mathbf{z}^{\mathrm{T}}\Lambda^{1/2}\mathrm{U}^{\mathrm{T}} + \boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}}) \cdot \frac{e^{-\frac{1}{2}\mathbf{z}^{\mathrm{T}}\mathbf{z}}}{(2\pi)^{d/2}} \\ &= \Sigma + \boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}}. \end{aligned}$$

Exercise B.7

Show that the expressions for the expectations above do indeed transform to the integrals claimed in the formulae above. Carry through the integrations (filling in the necessary steps using standard techniques) to obtain the final results that are claimed above.

B.6.2 Matrix Derivatives

We now consider derivatives of functions of a matrix X. Let $q(\mathrm{X})$ be a scalar function of a matrix X. The derivative $\partial q(\mathrm{X})/\partial \mathrm{X}$ is a matrix of the same size as X with entries

$$\left[\frac{\partial}{\partial \mathrm{X}} q(\mathrm{X})\right]_{ij} = \frac{\partial}{\partial \mathrm{X}_{ij}} q(\mathrm{X}).$$

Similarly if a matrix X is a function of a scalar z then the the derivative $\partial \mathrm{X}(z)/\partial z$ is a matrix of the same size as X obtained by taking the derivative of each entry of X:

$$\left[\frac{\partial}{\partial z}\mathrm{X}(z)\right]_{ij} = \frac{\partial}{\partial z}\mathrm{X}_{ij}(z).$$

The matrix derivatives of several interesting functions can be expressed in a convenient form:

Function	$\frac{\partial}{\partial \mathrm{X}}$
$\mathrm{trace}(\mathrm{AXB})$	$\mathrm{A}^{\mathrm{T}}\mathrm{B}^{\mathrm{T}}$
$\mathrm{trace}(\mathrm{AXX}^{\mathrm{T}}\mathrm{B})$	$\mathrm{A}^{\mathrm{T}}\mathrm{B}^{\mathrm{T}}\mathrm{X} + \mathrm{BAX}$
$\mathrm{trace}(\mathrm{X}^{\mathrm{T}}\mathrm{AX})$	$(\mathrm{A} + \mathrm{A}^{\mathrm{T}})\mathrm{X}$
$\mathrm{trace}(\mathrm{X}^{-1}\mathrm{A})$	$-\mathrm{X}^{-1}\mathrm{A}^{\mathrm{T}}\mathrm{X}^{-1}$
$\mathrm{trace}(\theta(\mathrm{BX})\mathrm{A})$	$\mathrm{B}^{\mathrm{T}}\left(\theta'(\mathrm{BX}) \otimes \mathrm{A}^{\mathrm{T}}\right)$
$\mathrm{trace}(\mathrm{A}\theta(\mathrm{BX})^{\mathrm{T}}\theta(\mathrm{BX}))$	$\mathrm{B}^{\mathrm{T}}\left(\theta'(\mathrm{BX}) \otimes [\theta(\mathrm{BX})(\mathrm{A} + \mathrm{A}^{\mathrm{T}})]\right)$
$\mathbf{a}^{\mathrm{T}}\mathrm{X}\mathbf{b}$	$\mathbf{a}\mathbf{b}^{\mathrm{T}}$
$\lvert\mathrm{AXB}\rvert$	$\lvert\mathrm{AXB}\rvert(\mathrm{X}^{\mathrm{T}})^{-1}$
$\ln\lvert\mathrm{X}\rvert$	$(\mathrm{X}^{\mathrm{T}})^{-1}$

(In all cases, assume the matrix products are well defined and the argument to the trace is a square matrix. When A and B have the same size $\mathrm{A} \otimes \mathrm{B}$ denotes component-wise multiplication, $[\mathrm{A} \otimes \mathrm{B}]_{ij} = A_{ij}B_{ij}$. A function applied to a matrix denotes the application of the function to each element, $[\theta(\mathrm{A})]_{ij} = \theta(A_{ij})$; θ' is the function which is the functional derivative of θ.) We can also get the derivatives of matrix functions of a parameter z:

Function	$\frac{\partial}{\partial z}$
X^{-1}	$-\mathrm{X}^{-1}\frac{\partial \mathrm{X}}{\partial z}\mathrm{X}^{-1}$
AXB	$\mathrm{A}\frac{\partial \mathrm{X}}{\partial z}\mathrm{B}$
XY	$\mathrm{X}\frac{\partial \mathrm{Y}}{\partial z} + \frac{\partial \mathrm{X}}{\partial z}\mathrm{Y}$
$\ln\lvert\mathrm{X}\rvert$	$\mathrm{trace}\left(\mathrm{X}^{-1}\frac{\partial \mathrm{X}}{\partial z}\right)$

$$\mathrm{X} = \mathrm{X}(z);\ \mathrm{Y} = \mathrm{Y}(z)$$

e-Appendix C

The E-M Algorithm

The E-M algorithm is a convenient heuristic for likelihood maximization. The E-M algorithm will never decrease the likelihood. Our discussion will focus on mixture models, the GMM being a special case, even though the E-M algorithm applies to more general settings.

Let $P_k(\mathbf{x};\theta_k)$ be a density for $k = 1,\ldots,K$, where θ_k are the parameters specifying P_k. We will refer to each P_k as a bump. In the GMM setting, all the P_k are Gaussians, and $\theta_k = \{\boldsymbol{\mu}_k, \Sigma_k\}$ (the mean vector and covariance matrix for each bump). A mixture model is a weighted sum of these K bumps,

$$P(\mathbf{x};\Theta) = \sum_{k=1}^{K} w_k P_k(\mathbf{x};\theta_k),$$

where the weights satisfy $w_k \geq 0$ and $\sum_{k=1}^{K} w_k = 1$ and we have collected all the parameters into a single grand parameter, $\Theta = \{w_1,\ldots,w_K;\theta_1,\ldots,\theta_K\}$. Intuitively, to generate a random point $\mathbf{x}$, you first pick a bump according to the probabilities $w_1,\ldots,w_K$. Suppose you pick bump k. You then generate a random point from the bump density P_k.

Given data $X = \mathbf{x}_1,\ldots,\mathbf{x}_N$ generated independently, we wish to estimate the parameters of the mixture which maximize the log-likelihood,

$$\begin{aligned}
\ln P(X|\Theta) &= \ln \prod_{n=1}^{N} P(\mathbf{x}_n|\Theta) \\
&= \ln \prod_{n=1}^{N} \left(\sum_{k=1}^{K} w_k P_k(\mathbf{x}_n;\theta_k) \right) \\
&= \sum_{n=1}^{N} \ln \left(\sum_{k=1}^{K} w_k P(\mathbf{x}_n|\theta_k) \right). \qquad \text{(C.1)}
\end{aligned}$$

In the first step above, $P(X|\Theta)$ is a product because the data are independent. Note that X is known and fixed. What is not known is which particular bump

was used to generate data point $\mathbf{x}_n$. Denote by $j_n \in \{1, \ldots, K\}$ the bump that generated $\mathbf{x}_n$ (we say $\mathbf{x}_n$ is a 'member' of bump j_n). Collect all bump memberships into a set $J = \{j_1, \ldots, j_N\}$. If we knew which data belonged to which bump, we can estimate each bump density's parameters separately, using only the data belonging to that bump. We call (X, J) the *complete data*. If we know the complete data, we can easily optimize the log-likelihood. We call X the *incomplete data*. Though X is all we can measure, it is still called the 'incomplete' data because it does not contain enough information to easily determine the optimal parameters Θ^* which minimize $E_{\text{in}}(\Theta)$. Let's see mathematically how knowing the complete data helps us.

To get the likelihood of the complete data, we need the joint probability $\mathbb{P}[\mathbf{x}_n, j_n|\Theta]$. Using Bayes' theorem,

$$\begin{aligned}\mathbb{P}[\mathbf{x}_n, j_n|\Theta] &= \mathbb{P}[j_n|\Theta]\,\mathbb{P}\left[\mathbf{x}_n|j_n, \Theta\right] \\ &= w_{j_n} P_{j_n}(\mathbf{x}_n; \theta_{j_n}).\end{aligned}$$

Since the data are independent,

$$\begin{aligned}P(X, J|\Theta) &= \prod_{n=1}^{N} \mathbb{P}[\mathbf{x}_n, j_n|\Theta] \\ &= \prod_{n=1}^{N} w_{j_n} P_{j_n}(\mathbf{x}_n; \theta_{j_n}).\end{aligned}$$

Let N_k be the number of occurrences of bump k in J, and let X_k be those data points corresponding to the bump k, so $X_k = \{\mathbf{x}_n \in X : j_n = k\}$. We compute the log-likelihood for the complete data as follows:

$$\begin{aligned}\ln P(X, J|\Theta) &= \sum_{n=1}^{N} \ln w_{j_n} + \sum_{n=1}^{N} \ln P_{j_n}(\mathbf{x}_n; \theta_{j_n}) \\ &= \sum_{k=1}^{K} N_k \ln w_k + \sum_{k=1}^{K} \underbrace{\sum_{\mathbf{x}_n \in X_k} \ln P_k(\mathbf{x}_n; \theta_k)}_{L_k(X_k, \theta_k)} \\ &= \sum_{k=1}^{K} N_k \ln w_k + \sum_{k=1}^{K} L_k(X_k; \theta_k). \qquad \text{(C.2)}\end{aligned}$$

There are two simplifications which occur in (C.2) from knowing the complete data (X, J). The w_k (in the first term) are separated from the θ_k (in the second term); and, the second term is the sum of K non-interacting log-likelihoods $L_k(X_k, \theta_k)$ corresponding to the data belonging to X_k and only involving bump k's parameters θ_k. Each log-likelihood L_k can be optimized independently of the others. For many choices of P_k, $L_k(X_k; \theta_k)$ can be optimized analytically, even though the log-likelihood for the incomplete data in (C.1) is intractable. The next exercise asks you to analytically maximize (C.2) for the GMM.

Exercise C.1

(a) Maximize the first term in (C.2) subject to $\sum_k w_k = 1$, and show that the optimal weights are $w_k^* = N_k/N$. *[Hint: Lagrange multipliers.]*

(b) For the GMM,

$$P_k(\mathbf{x}; \boldsymbol{\mu}_k, \Sigma_k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^{\mathrm{T}}\Sigma_k^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right).$$

Maximize $L_k(X_k; \boldsymbol{\mu}_k, \Sigma_k)$ to obtain the optimal parameters:

$$\begin{aligned}\boldsymbol{\mu}_k^* &= \frac{1}{N_k}\sum_{\mathbf{x}_n \in X_k} \mathbf{x}_n;\\ \Sigma_k^* &= \frac{1}{N_k}\sum_{\mathbf{x}_n \in X_k} (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}.\end{aligned}$$

These are exactly the parameters you would expect. $\boldsymbol{\mu}_k^*$ is the in-sample mean for the data belonging to bump k; similarly, Σ_k^* is the in-sample covariance matrix.
[Hint: Set $\mathrm{S_k} = \Sigma_k^{-1}$ and optimize with respect to $\mathrm{S_k}$. Also, from the Linear Algebra e-appendix, you may find these derivatives useful:

$$\frac{\partial}{\partial \mathrm{S}}(\mathbf{z}^{\mathrm{T}}\mathrm{S}\mathbf{z}) = \mathbf{z}\mathbf{z}^{\mathrm{T}}; \quad \textit{and} \quad \frac{\partial}{\partial \mathrm{S}}\ln|\mathrm{S}| = \mathrm{S}^{-1}.]$$

In reality, we do not have access to J, and hence it is called a 'hidden variable'. So what can we do now? We need a heuristic to maximize the likelihood in Equation (C.1). One approach is to guess J and maximize the resulting complete likelihood in Equation (C.2). This almost works. Instead of maximizing the complete likelihood for a *single* guess, we consider an average of the complete likelihood over all possible guesses. Specifically, we treat J as an unknown random variable and maximize the expected value (with respect to J) of the complete log-likelihood in Equation (C.2). This expected value is as easy to minimize as the complete likelihood. The mathematical implementation of this idea will lead us to the E-M Algorithm, which stands for Expectation-Maximization Algorithm. Let's start with a simpler example.

Example C.1. You have two opaque bags. Bag 1 has red and green balls, with μ_1 being the fraction of red balls. Bag 2 has red and blue balls with μ_2 being the fraction of red. You pick four balls in independent trials as follows. First pick one of the bags at random, each with probability $\frac{1}{2}$; then, pick a ball at random from the bag. Here is the sample of four balls you got: 🟢🔴🔵🔵, one green, one red and two blue. The task is to estimate μ_1 and μ_2. It would be much easier if we knew which bag each ball came from.

Here is one way to reason. Half the balls will come from Bag 1 and the other half from Bag 2. The blue balls come from Bag 2 (that's already Bag 2's budget of balls), so the other two should come from Bag 1: 🟢🔴 | 🔵🔵. Using in-sample estimates, $\hat{\mu}_1 = \frac{1}{2}$ and $\hat{\mu}_2 = 0$. We can get these same estimates

using a maximum likelihood argument. The log-likelihood of the data is

$$\ln(1-\mu_1) + 2\ln(1-\mu_2) + \ln(\mu_1+\mu_2) - 4\ln 2. \tag{C.3}$$

The reader should explicitly maximize the above expression with respect to $\mu_1, \mu_2 \in [0,1]$ to obtain the estimates $\hat{\mu}_1 = \frac{1}{2}, \hat{\mu}_2 = 0$. In our data set of four balls, there is a red one, so it seems a little counter-intuitive that we would estimate $\hat{\mu}_2 = 0$, for isn't there a *positive* probability that the red ball came from Bag 2? Nevertheless, $\hat{\mu}_2 = 0$ *is* the estimate that *maximizes* the probability of generating the data. You are uneasy with this, and rightly so, because we put all our eggs into this single 'point' estimate; a very unnatural thing given that any point estimate has infinitesimal probability of being correct. Nevertheless, that is the maximum likelihood method, and we are following it.

Here is another way to reason. 'Half' of each red ball came from Bag 1 and the other 'half' from Bag 2. So, $\hat{\mu}_1 = \frac{1}{2}/(1+\frac{1}{2}) = \frac{1}{3}$ because $\frac{1}{2}$ a red ball came from Bag 1 out of a total of $1+\frac{1}{2}$ balls. Similarly, $\hat{\mu}_2 = \frac{1}{2}/(2+\frac{1}{2}) = \frac{1}{5}$. This reasoning is wrong because it does not correctly use the knowledge that the ball is red. For example, as we just reasoned, $\hat{\mu}_1 = \frac{1}{3}$ and $\hat{\mu}_2 = \frac{1}{5}$. But, if these estimates are correct, and indeed $\hat{\mu}_1 > \hat{\mu}_2$, then a red ball is more likely to come from Bag 1, so more than half if it should come from Bag 1. This contradicts the original assumption that led us to these estimates.

Now, let's see how expectation-maximization solves this problem. The reasoning is similar to our false start above; it just adds iteration till consistency. We begin by considering the two cases for the red ball. Either it is from Bag 1 or Bag 2. We can compute the log-likelihood for each of these two cases:

$$\begin{aligned} &\ln(1-\mu_1) + \ln(\mu_1) + 2\ln(1-\mu_2) - 4\ln 2 && \text{(Bag 1);} \\ &\ln(1-\mu_1) + \ln(\mu_2) + 2\ln(1-\mu_2) - 4\ln 2 && \text{(Bag 2).} \end{aligned}$$

Suppose we have estimates $\hat{\mu}_1$ and $\hat{\mu}_2$. Using Bayes theorem, we can compute $p_1 = \mathbb{P}[\text{Bag 1} \mid \hat{\mu}_1, \hat{\mu}_2]$ and $p_2 = \mathbb{P}[\text{Bag 2} \mid \hat{\mu}_1, \hat{\mu}_2]$. The reader can verify that

$$p_1 = \frac{\hat{\mu}_1}{\hat{\mu}_1 + \hat{\mu}_2}, \qquad p_2 = \frac{\hat{\mu}_2}{\hat{\mu}_1 + \hat{\mu}_2}.$$

Now comes the *expectation* step. Compute the expected log-likelihood using p_1 and p_2:

$$\mathbb{E}[\text{log-likelihood}] = \ln(1-\mu_1) + p_1\ln(\mu_1) + p_2\ln(\mu_2) + 2\ln(1-\mu_2) - 4\ln 2. \tag{C.4}$$

Next comes the *maximization* step. Treating p_1, p_2 as constants, maximize the expected log-likelihood with respect to μ_1, μ_2 and update $\hat{\mu}_1, \hat{\mu}_2$ to these optimal values. Notice that the log-likelihood in (C.3) has an interaction term $\ln(\mu_1+\mu_2)$ which complicates the maximization. In the expected log-likelihood (C.4), μ_1 and μ_2 are decoupled, and so the maximization can be implemented *separately* for each variable. The reader can verify that maximizing the expected log-likelihood gives the updates:

$$\hat{\mu}_1 \leftarrow \frac{p_1}{1+p_1} = \frac{\hat{\mu}_1}{2\hat{\mu}_1 + \hat{\mu}_2} \qquad \text{and} \qquad \hat{\mu}_2 \leftarrow \frac{p_2}{2+p_2} = \frac{\hat{\mu}_2}{2\hat{\mu}_1 + 3\hat{\mu}_2}.$$

The full algorithm just iterates this update process with the new estimates. Let's see what happens if we start (arbitrarily) with estimates $\hat{\mu}_1 = \hat{\mu}_2 = \frac{1}{2}$:

	Iteration number									
	0	1	2	3	4	5	6	7	...	1000
$\hat{\mu}_1$	$\frac{1}{2}$	$\frac{1}{3}$	0.38	0.41	0.43	0.45	0.45	0.46	...	0.49975
$\hat{\mu}_2$	$\frac{1}{2}$	$\frac{1}{5}$	0.16	0.13	0.10	0.09	0.07	0.07	...	0.0005

We have highlighted in blue the result of the first iteration, which is exactly the estimate from our earlier faulty reasoning. When $\mu_1 = \mu_2$ our faulty reasoning matches the E-M step. If we continued this table, it is not hard to see what will happen: $\hat{\mu}_1 \to \frac{1}{2}$ and $\hat{\mu}_2 \to 0$.

Exercise C.2

When the E-M algorithm converges, it must be that

$$\hat{\mu}_1 = \frac{\hat{\mu}_1}{2\hat{\mu}_1 + \hat{\mu}_2} \qquad \text{and} \qquad \hat{\mu}_2 = \frac{\hat{\mu}_2}{2\hat{\mu}_1 + 3\hat{\mu}_2}.$$

Solve these consistency conditions, and report your estimates for $\hat{\mu}_1, \hat{\mu}_2$?

It's miraculous that by maximizing an expected log-likelihood using a *guess* for the parameters, we end up converging to the true maximum likelihood solution. Why is this useful? Because the maximizations for μ_1 and μ_2 are decoupled. We trade a maximization of a complicated likelihood of the incomplete data for a bunch of simpler maximizations that we iterate. □

C.1 Derivation of the E-M Algorithm

We now derive the E-M strategy and show that it will always improve the likelihood of the incomplete data.

> *Maximizing an expected log-likelihood of the complete data increases the likelihood of the incomplete data.*

What is surprising is that to compute the expected log-likelihood, we use a guess, since we don't know the best model. So it is really a guess for the expected log-likelihood that one maximizes and this increases the likelihood.

Let Θ' be any set of parameters, and define $P(J|X, \Theta')$ as the conditional probability distribution for J given the data and *assuming* that Θ' is the actual probability model for the data. The probability $P(J|X, \Theta')$ is well defined, even if Θ' is not the probability model which generated the data. We will see how to compute $P(J|X, \Theta')$ soon, but for the moment assume that it is always positive (which means that every possible assignment of $\mathbf{x}_1, \ldots, \mathbf{x}_N$ to bumps $j_1, \ldots, j_n$ has non-zero probability under Θ'). This will always be the case

unless some of the P_k have bounded support or Θ' is some degenerate mixture. The following derivation establishes a connection between the log-likelihood for the incomplete data and the expectation (over J) of the log-likelihood for the complete data.

$$\begin{aligned}
L(\Theta) &= \ln P(X|\Theta) \\
&\overset{(a)}{=} \ln \sum_J P(X, J|\Theta) \\
&\overset{(b)}{=} \ln \sum_J \frac{P(X, J|\Theta)}{P(J|X, \Theta')} P(J|X, \Theta') \\
&\overset{(c)}{=} \ln \sum_J \frac{P(X, J|\Theta) P(X|\Theta')}{P(X, J|, \Theta')} P(J|X, \Theta') \\
&\overset{(d)}{\geq} \sum_J \ln \left(\frac{P(X, J|\Theta) P(X|\Theta')}{P(X, J|, \Theta')} \right) P(J|X, \Theta') \\
&\overset{(e)}{=} L(\Theta') + \mathbb{E}_{J|X,\Theta'}[\ln P(X, J|\Theta)] - \mathbb{E}_{J|X,\Theta'}[\ln P(X, J|\Theta')] \\
&\overset{(f)}{=} L(\Theta') + Q(\Theta|X, \Theta') - Q(\Theta'|X, \Theta') \qquad \text{(C.5)}
\end{aligned}$$

(a) follows by the law of total probability; (b) is justified because $P(J|X, \Theta')$ is positive; (c) follows from Bayes' theorem; (d) follows because the summation $\sum_J (\cdot) P(J \mid X, \Theta')$ is an expectation using the probabilities $P(J \mid X, \Theta')$ and by Jensen's inequality $\ln \mathbb{E}[\cdot] \geq \mathbb{E}[\ln(\cdot)]$ because $\ln(x)$ is concave; (e) follows because $\ln P(X|\Theta')$ is independent of J and so

$$\sum_J P(J|X, \Theta') \ln P(X|\Theta') = \ln P(X|\Theta') \sum_J P(J|X, \Theta') = \ln P(X|\Theta') \cdot 1;$$

finally, in (f), we have defined the function

$$Q(\Theta|X, \Theta') = \mathbb{E}_{J|X,\Theta'}[\ln P(X, J|\Theta)].$$

The function Q is a function of Θ, though its definition depends the distribution of J which in turn depends on the incomplete data X and the model Θ'. We have proved the following result.

Theorem C.2. If $Q(\Theta|X, \Theta') > Q(\Theta'|X, \Theta')$, then $L(\Theta) > L(\Theta')$.

In words, fix Θ' and compute the 'posterior' distribution of J conditioned on the data X with parameters Θ'. Now for the parameters Θ, compute the *expected log-likelihood* of the complete data (X, J) where the expectation is taken with respect to this posterior distribution for J that we just obtained. This distribution for J is fixed, depending on X, Θ', but *it does not depend on* Θ. Find Θ^* that maximizes this expected log-likelihood, and you are guaranteed to improve the actual likelihood. This theorem leads naturally to the E-M algorithm that follows.

E-M Algorithm

1: Initialize Θ_0 at $t = 0$.
2: At step t let the parameter estimate be Θ_t.
3: **[Expectation]** For X, Θ_t, compute the function of $Q_t(\Theta)$:

$$Q_t(\Theta) = \mathbb{E}_{J|X,\Theta_t}[\ln P(X, J|\Theta)],$$

which is the expected log-likelihood for the complete data.
4: **[Maximization]** Update Θ to maximize $Q_t(\Theta)$:

$$\Theta_{t+1} = \operatorname*{argmax}_{\Theta} Q_t(\Theta).$$

5: Increment $t \to t+1$ and repeat steps 3,4 till convergence.

In the algorithm, we need to compute $Q_t(\Theta)$, which amounts to computing an expectation with respect to $P(J|X, \Theta_t)$. We illustrate this process with our mixture model.

Recall that J is the vector of bump memberships. Since the data are independent, to compute $P(J|X, \Theta_t)$ we can compute this 'posterior' for each data point and then take the product. We need $\gamma_{nk} = P(j_n = k|\mathbf{x}_n, \Theta_t)$, the probability that data point $\mathbf{x}_n$ came from bump k. By Bayes' theorem,

$$\begin{aligned}\gamma_{nk} = P(j_n = k|\mathbf{x}_n, \Theta_t) &= \frac{P(\mathbf{x}_n, k|\Theta_t)}{P(\mathbf{x}_n|\Theta_t)} \\ &= \frac{\hat{w}_k P_k(\mathbf{x}_n|\hat{\theta}_k)}{\sum_{\ell=1}^{K} \hat{w}_\ell P_\ell(\mathbf{x}_n|\hat{\theta}_\ell)},\end{aligned}$$

where $\Theta_t = \{\hat{w}_1, \ldots, \hat{w}_K; \hat{\theta}_1, \ldots, \hat{\theta}_K\}$ and $P(J|X, \Theta') = \prod_{n=1}^{N} \gamma_{nj_n}$. We can now compute $Q_t(\Theta)$,

$$Q_t(\Theta) = \mathbb{E}_J[\ln P(X, J|\Theta)],$$

where the expectation is with respect to the ('fictitious') probabilities γ_{nk} that determine the distribution of the random variable J. These probabilities depend on X and Θ_t. Let $\hat{N}_k$ (a random variable) be the number of occurrences of bump k in the random variable J; similarly, let $\hat{X}_k$ be the random set containing the data points of bump k. From Equation (C.2),

$$\begin{aligned}Q_t(\Theta) &= \sum_{k=1}^{K} \mathbb{E}_J[\hat{N}_k] \ln w_k + \sum_{k=1}^{K} \mathbb{E}_J \left[\sum_{\mathbf{x}_n \in \hat{X}_k} \ln P(\mathbf{x}_n; \theta_k) \right] \\ &\overset{(a)}{=} \sum_{k=1}^{K} \mathbb{E}_J \left[\sum_{n=1}^{N} z_{nk} \right] \ln w_k + \sum_{k=1}^{K} \mathbb{E}_J \left[\sum_{n=1}^{N} z_{nk} \ln P(\mathbf{x}_n; \theta_k) \right] \\ &\overset{(b)}{=} \sum_{k=1}^{K} N_k \ln w_k + \sum_{k=1}^{K} \sum_{n=1}^{N} \gamma_{nk} \ln P(\mathbf{x}_n; \theta_k),\end{aligned}$$

where $\Theta = \{w_1, \ldots, w_K; \theta_1, \ldots, \theta_K\}$. In (a), we introduced an indicator random variable $z_{nk} = [\![\mathbf{x}_n \in \hat{X}_k]\!]$ which is 1 if $\mathbf{x}_n$ is from bump k and 0 otherwise; in (b), we defined

$$N_k = \mathbb{E}_J[\hat{N}_k] = \mathbb{E}_J\left[\sum_{n=1}^{N} z_{nk}\right] = \sum_{n=1}^{N} \gamma_{nk},$$

where we used $\mathbb{E}[z_{nk}] = \gamma_{nk}$. Now that we have an explicit functional form for $Q_t(\Theta)$, we can perform the maximization step. Observe that the bump-parameters $\theta_k \in \Theta$ are occurring in independent terms, and so can be optimized separately. As for the first term, observe that

$$\begin{aligned}\frac{1}{N}\sum_{k=1}^{K} N_k \ln w_k &= \sum_{k=1}^{K} \frac{N_k}{N} \ln \frac{w_k}{N_k/N} + \sum_{k=1}^{K} \frac{N_k}{N} \ln \frac{N_k}{N} \\ &\leq \sum_{k=1}^{K} \frac{N_k}{N} \ln \frac{N_k}{N},\end{aligned}$$

where the last inequality follows from Jensen's inequality and the concavity of the logarithm, which implies:

$$\sum_{k=1}^{K} \frac{N_k}{N} \ln \frac{w_k}{N_k/N} \leq \ln\left(\sum_{k=1}^{K} \frac{N_k}{N} \cdot \frac{w_k}{N_k/N}\right) = \ln\left(\sum_{k=1}^{K} w_k\right) = \ln(1) = 0.$$

Equality holds when $w_k = N_k/N$. Maximizing $Q_t(w_1, \ldots, w_K, \theta_1, \ldots, \theta_K)$, therefore, gives the following updates:

$$w_k^* = \frac{N_k}{N} = \frac{\sum_{n=1}^{N} \gamma_{nk}}{N}; \tag{C.6}$$

$$\theta_k^* = \operatorname*{argmax}_{\theta_k} \sum_{n=1}^{N} \gamma_{nk} \ln P(\mathbf{x}_n; \theta_k). \tag{C.7}$$

The update to get w_k^* can be viewed as follows. A fraction γ_{nk} of data point $\mathbf{x}_n$ belongs to bump k. Thus, $N_k = \sum_n \gamma_{nk}$ is the total number of data points belonging to bump k. Similarly, the update to get θ_k^* can be viewed as follows. The 'likelihood' for the parameter θ_k of bump k is just the weighted sum of the likelihoods of each point, weighted by the fraction of the point that belongs to bump k. This intuitive interpretation of the update is another reason for the popularity of the E-M algorithm.

The E-M algorithm for mixture density estimation is remarkably simple once we get past the machinery used to set it up: we have an analytic update for the weights w_k and K *separate* optimizations for each bump parameter θ_k. The miracle is that these simple updates are *guaranteed* to improve the log-likelihood (from Theorem C.2). There are other ways to maximize the likelihood, for example using gradient and Hessian based iterative optimization techniques. However, the E-M algorithm is simpler and works well in practice.

Example Let's derive the E-M update for the GMM with current parameter estimate is $\Theta_t = \{\hat{w}_1, \ldots, \hat{w}_K; \hat{\boldsymbol{\mu}}_1, \ldots, \hat{\boldsymbol{\mu}}_K; \hat{\Sigma}_1, \ldots, \hat{\Sigma}_K\}$. Let $\hat{\mathrm{S}}_k = (\hat{\Sigma}_k)^{-1}$. The posterior bump probabilities γ_{nk} for the parameters Θ_t are:

$$\gamma_{nk} = \frac{\hat{w}_k P(\mathbf{x}_n | \hat{\boldsymbol{\mu}}_k, \hat{\mathrm{S}}_k)}{\sum_{\ell=1}^{K} \hat{w}_\ell P(\mathbf{x}_n | \hat{\boldsymbol{\mu}}_\ell, \hat{\mathrm{S}}_\ell)},$$

where $P(\mathbf{x}|\boldsymbol{\mu}, \mathrm{S}) = (2\pi)^{-d/2} |\mathrm{S}|^{1/2} \exp(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \mathrm{S}(\mathbf{x} - \boldsymbol{\mu}))$. The w_k update is immediate from (C.6), which matches Equation (6.9). Since

$$\ln P(\mathbf{x}|\boldsymbol{\mu}, \mathrm{S}) = -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \mathrm{S}(\mathbf{x} - \boldsymbol{\mu}) + \frac{1}{2} \ln |\mathrm{S}| - \frac{d}{2} \ln(2\pi),$$

to get the $\boldsymbol{\mu}_k$ and S_k updates using (C.7), we need to minimize

$$\sum_{n=1}^{N} \gamma_{nk} \left((\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}} \mathrm{S}_k (\mathbf{x}_n - \boldsymbol{\mu}_k) - \ln |\mathrm{S}_k| \right).$$

Setting the derivative with respect to $\boldsymbol{\mu}_k$ to $\mathbf{0}$, gives

$$2\mathrm{S_k} \sum_{n=1}^{N} \gamma_{nk} (\mathbf{x}_n - \boldsymbol{\mu}_k) = \mathbf{0}.$$

Since S_k is invertible, $\boldsymbol{\mu}_k \sum_{n=1}^{N} \gamma_{nk} = \sum_{n=1}^{N} \gamma_{nk} \mathbf{x}_n$, and since $N_k = \sum_{n=1}^{N} \gamma_{nk}$, we obtain the update in (6.9),

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} \mathbf{x}_n.$$

To take the derivative with respect to S_k, we use the identities in the hint of Exercise C.1(b). Setting the derivative with respect to S_k to zero gives

$$\sum_{n=1}^{N} \gamma_{nk} (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}} - \mathrm{S}_k^{-1} \sum_{n=1}^{N} \gamma_{nk} = 0,$$

or

$$\mathrm{S}_k^{-1} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}.$$

Since $\mathrm{S}_k^{-1} = \Sigma_k$, we recover the update in (6.9).

Commentary The E-M algorithm is a remarkable example of a recurring theme in learning. We want to learn a model Θ that explains the data. We start with a guess $\hat{\Theta}$ that is *wrong*. We use this wrong model to estimate some other quantities of the world (the bump memberships in our example). We now learn a new model which is better than the old model at explaining the combined data plus the inaccurate estimates of the other quantities. Miraculously, by doing this, we bootstrap ourselves up to a better model, one that is better at explaining the data. This theme reappears in Reinforcement Learning as well. If we didn't know better, it seems like a free lunch.

C.2 Problems

Problem C.1 Consider the general case of Example C.1. The sample has N balls, with N_g green, N_b blue and N_r red ($N_g + N_b + N_r = N$). Show that the log-likelihood of the incomplete data is

$$N_g \ln(1-\mu_1) + N_b \ln(1-\mu_2) + N_r \ln(\mu_1+\mu_2) - N \ln 2. \tag{C.8}$$

What are the maximum likelihood estimates for μ_1, μ_2. (Be careful with the cases $N_g > N/2$ and $N_b > N/2$.

Problem C.2 Consider the general case of Example C.1 as in Problem C.1, with N_g green, N_b blue and N_r red balls.

(a) Suppose your starting estimates are $\hat{\mu}_1, \hat{\mu}_2$. For a red ball, what are $p_1 = \mathbb{P}[\text{Bag 1}|\hat{\mu}_1, \hat{\mu}_2]$ and $p_2 = \mathbb{P}[\text{Bag 2}|\hat{\mu}_1, \hat{\mu}_2]$

(b) Let N_{r_1} and N_{r_2} (with $N_r = N_{r_1} + N_{r_2}$) be the number of red balls from Bag 1 and 2 respectively. Show that the log-likelihood of the complete data is

$$N_g \ln(1-\mu_1) + N_b \ln(1-\mu_2) + N_{r_1} \ln(\mu_1) + N_{r_2} \ln(\mu_2) - N \ln 2.$$

(c) Compute the function $Q_t(\mu_1, \mu_2)$ by taking the expectation of the log-likelihood of the complete data. Show that

$$Q_t(\mu_1, \mu_2) = N_g \ln(1-\mu_1) + N_b \ln(1-\mu_2) + p_1 N_r \ln(\mu_1) + p_2 N_r \ln(\mu_2) - N \ln 2.$$

(d) Maximize $Q_t(\mu_1, \mu_2)$ to obtain the E-M update.

(e) Show that repeated E-M iteration will ultimately converge to

$$\hat{\mu}_1 = \frac{N - 2N_g}{N} \qquad \hat{\mu}_2 = \frac{N - 2N_b}{N}.$$

Problem C.3 A sequence of N balls, $X = x_1, \ldots, x_N$ is drawn iid as follows. There are 2 bags. Bag 1 contains only red balls and bag 2 contains red and blue balls. A fraction π in this second bag are red. A bag is picked randomly with probability $\frac{1}{2}$ and one of the balls is picked randomly from that bag; $x_n = 1$ if ball n is red and 0 if it is blue. You are given N and the number of red balls $N_r = \sum_{n=1}^{N} x_n$.

(a) (i) Show that the likelihood $P[X|\pi, N]$ is

$$P[X|\pi, N] = \prod_{n=1}^{N} \left(\frac{1+\pi}{2}\right)^{x_n} \left(\frac{1-\pi}{2}\right)^{1-x_n}.$$

Maximize to obtain an estimate for π (be careful with $N_r < N/2$).

(ii) For $N_r = 600$ and $N = 1000$, what is your estimate of π.

(b) Maximizing the likelihood is tractable for this simple problem. Now develop an E-M iterative approach.

(i) What is an appropriate hidden/unmeasured variable $J = j_1, \ldots, j_N$.

(ii) Give a formula for the likelihood for the full data, $\mathbb{P}[X, J|\pi, N]$.

(iii) If at step t your estimate is π_t, for the expectation step, compute $Q_t(\pi) = \mathbb{E}_{J|X,\pi_t}[-\ln \mathbb{P}[X, J|\pi, N]]$ and show that

$$Q_t(\pi) = \frac{\pi_t}{1+\pi_t} N_r \ln \pi + (N - N_r) \ln(1-\pi),$$

and hence show that the E-M update is given by

$$\pi_{t+1} = \frac{\pi_t N_r}{\pi_t N_r + (1+\pi_t)(N - N_r)}.$$

What are the limit points when $N_r \geq N/2$ and $N_r < N/2$?

(iv) Plot π_t versus t, starting from $\pi_0 = 0.9$ and $\pi_0 = 0.2$, with $N_r = 600, N = 1000$.

(v) The values of the hidden variables can often be useful. After convergence of the E-M, how could you get estimates of the hidden variables?

Problem C.4 [E-M for Supervised Learning] We wish to learn a function $f(x)$ which predicts the temperature as a function of the time x of the day, $x \in [0, 1]$. We believe that the temperature has a linear dependence on time, so we model f with the linear hypotheses, $h(x) = w_0 + w_1 x$.

We have N data points $y_1, \ldots, y_N$, the temperature measurements on different (independent) days, where $y_n = f(x_n) + \epsilon_n$ and $\epsilon_n \sim N(0, 1)$ is iid zero mean Gaussian noise with unit variance. The problem is that we do not know the time x_n at which these measurements were made. Assume that each temperature measurement was taken at some random time in the day, chosen uniformly on $[0, 1]$.

(a) Show that the log-likelihood for weights $\mathbf{w}$ is

$$\ln \mathbb{P}[\mathbf{y}|\mathbf{w}] = \sum_{n=1}^{N} \ln \left(\int_0^1 dx \; \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(y_n - w_0 - w_1 x)^2} \right).$$

(b) What is the natural hidden variable $J = j_1, \ldots, j_N$.

(c) Compute the log-likelihood for the complete data $\ln \mathbb{P}[\mathbf{y}, J|\mathbf{w}]$.

(d) Let $\gamma_n(x|\mathbf{w}) = P(x_n = x|y_n, \mathbf{w})$. Show that, for $x \in [0, 1]$,

$$\gamma_n(x|\mathbf{w}) = \frac{\exp\left(-\frac{1}{2}(y_n - w_0 - w_1 x)^2\right)}{\int_0^1 dx \; \exp\left(-\frac{1}{2}(y_n - w_0 - w_1 x)^2\right)}.$$

Hence, compute $Q_t(\mathbf{w})$.

(e) Let $\alpha_n = \mathbb{E}_{\gamma_n(x|\mathbf{w}_t)}[x]$ and $\beta_n = \mathbb{E}_{\gamma_n(x|\mathbf{w}_t)}[x^2]$ (expectations taken with respect to the distribution $\gamma_n(x|\mathbf{w}_t)$). Show that the EM-updates are

$$\begin{aligned} w_1(t+1) &= \frac{\frac{1}{N}\sum_{i=1}^N (y_i - \overline{y})(\alpha_i - \overline{\alpha})}{\overline{\beta} - \overline{\alpha}^2}; \\ w_0(t+1) &= \overline{y} - w_1(t+1)\overline{\alpha}; \end{aligned}$$

where, $\overline{(\cdot)}$ denotes averaging (eg. $\overline{\alpha} = \frac{1}{N}\sum_{n=1}^N \alpha_n$) and $\mathbf{w}(t)$ are the weights at iteration t.

(f) What happens if the temperature measurement is not at a uniformly random time, but at a time distributed according to an unknown $P(x)$? You have to maintain an estimate $P_t(x)$. Now, show that

$$\gamma_n(x|\mathbf{w}) = \frac{P_t(x)\exp\left(-\frac{1}{2}(y_n - w_0 - w_1 x)^2\right)}{\int_0^1 dx\ P_t(x)\exp\left(-\frac{1}{2}(y_n - w_0 - w_1 x)^2\right)},$$

and that the updates in (e) are unchanged, except that they use this new $\gamma_n(x|\mathbf{w}_t)$. Show that the update to P_t is given by

$$P_{t+1}(x) = \frac{1}{N}\sum_{n=1}^N \gamma_n(x|\mathbf{w}_t).$$

What happens if you tried to maximize the log-likelihood for the incomplete data, instead of using the E-M approach?