MACHINE LEARNING

Lecture Notes

Melih Kandemir
University of Southern Denmark
kandemir@imada.sdu.dk

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOTATION

This table collects the symbols used throughout the book for quick reference. Chapter 1 states the underlying conventions (parentheses vs. brackets, uppercase vs. lowercase, etc.) in prose; this page is only a lookup table.

Sets and spaces

Symbol	Meaning
$\mathcal{X}$	Feature (input) space
$\mathcal{Y}$	Label (output) space
$\mathcal{H}$	Hypothesis class
$\mathcal{D}$	Data distribution over $\mathcal{X} \times \mathcal{Y}$
$\mathcal{D}_{\mathcal{X}}$	Marginal of $\mathcal{D}$ on $\mathcal{X}$
$[m]$	$\{1, \ldots, m\}$

Data and samples

Symbol	Meaning
S	A data set / sample
m	Size of S
d	Dimension of the feature space
C	Number of classes
x, x_i	An input (feature vector)
y, y_i	A true label
$\widehat{y}$	A predicted label
z, z_i	A score / pre-activation (also a training example in generic contexts)

Probability

Symbol	Meaning
$P(\cdot)$	Probability of an event or of a discrete random variable's value
$p(\cdot)$	Probability **density** of a continuous random variable

Symbol	Meaning
$\mathbb{E}[\cdot]$, $\mathrm{Var}[\cdot]$, $\mathrm{Cov}[\cdot,\cdot]$	Expectation, variance, covariance
$\mathbb{1}(\cdot)$	Indicator function
log	Natural logarithm
σ	Standard deviation (bandwidth of an RBF kernel in Chapter 9 only)
$\sigma_i \in \{-1,+1\}$	A Rademacher sign (from Chapter 5a onwards)

Losses and risks

Symbol	Meaning
$\ell(y,\widehat{y})$	Pointwise loss — true label first, prediction second
$R(h)$	True risk (generalization error) of h
$\widehat{R}_S(h)$	Empirical risk of h on S
$\mathcal{L}(\cdot)$	Training objective (empirical risk plus a regularizer)
$R^*_{\mathcal{D}}$	Bayes error of $\mathcal{D}$

Learning-theoretic quantities

Symbol	Meaning
h	A hypothesis
h_S	A hypothesis learned from S
f	The true labeling function
f^*	The Bayes predictor
A	A learning algorithm
ϵ	An accuracy parameter
δ	A failure probability
$d_{VC}(\mathcal{H})$	VC dimension of $\mathcal{H}$
$\tau_{\mathcal{H}}(m)$	Growth function of $\mathcal{H}$
$\mathfrak{R}, \widehat{\mathfrak{R}}_S$	Rademacher complexity (population, empirical)

Model and optimization symbols

Symbol	Meaning
w	A weight vector
λ	A regularization coefficient
α	A learning rate

Neural networks (Chapter 8)

Symbol	Meaning
$g(\cdot)$	Activation function
z_j^l	Pre-activation of neuron j in layer l
h_j^l	Activation (output) of neuron j in layer l
W_l, b^l	Weight matrix and bias vector of layer l
δ_j^l	Backpropagated error of neuron j in layer l

Kernel methods (Chapter 9)

Symbol	Meaning
$k(\cdot, \cdot)$	A kernel function
$\phi(\cdot)$	A feature map
$\mathcal{H}_k$	The RKHS of kernel k
K	A Gram matrix

CHAPTER 1

BASIC CONCEPTS

1.1 What is machine learning?

Notational conventions used throughout these notes.

Probability. $P(\cdot)$ denotes a probability — of an event, or of a discrete random variable taking a value, so that $P(X = x)$ is the probability mass function of a discrete X. Lowercase $p(\cdot)$ is reserved for the probability **density** of a continuous random variable, which is not itself a probability. Where a joint object mixes the two (a continuous feature vector with a discrete label, say) we write p for the joint and P for its discrete factors. Subscripts name the distribution being integrated against, e.g. $P_{S\sim\mathcal{D}^m}(\cdot)$. Probabilities always take parentheses; expectations, variances and covariances take brackets: $\mathbb{E}[\cdot]$, $\mathrm{Var}[\cdot]$, $\mathrm{Cov}[\cdot,\cdot]$, with subscripts naming the variable averaged over ($\mathbb{E}_S$, $\mathbb{E}_{x\sim\mathcal{D}}$). $\mathbb{1}(\cdot)$ is the indicator function and log always the **natural** logarithm.

Data. m is the size of a data set S, d the dimension of the feature space, C the number of classes, $[m] := \{1, \ldots, m\}$. Calligraphic letters denote spaces: $\mathcal{X}$ the feature space, $\mathcal{Y}$ the label space, $\mathcal{H}$ a hypothesis class, $\mathcal{D}$ a data distribution over $\mathcal{X} \times \mathcal{Y}$, and $\mathcal{D}_{\mathcal{X}}$ its marginal on $\mathcal{X}$.

Losses and risks. The pointwise loss is always written $\ell(y, \widehat{y})$ — **true label first, prediction second**. $R(h)$ is the true risk and $\widehat{R}_S(h)$ the empirical risk on S. A calligraphic $\mathcal{L}$ denotes a training objective (an empirical risk plus a regularizer).

Recurring symbols. h is a hypothesis, f the labeling function and f^* the Bayes predictor, A a learning algorithm, w a weight vector, ϵ an accuracy and δ a failure probability, α a learning rate, λ a regularization coefficient. From Chapter 5a onwards, $\sigma_i \in \{-1, +1\}$ are Rademacher signs; the activation function of a neural network is therefore written $g(\cdot)$, not $\sigma(\cdot)$, to keep the two apart. σ without a subscript is a standard deviation (and, in Chapter 9 only, the bandwidth of an RBF kernel, which is its universal name there).

Definition 1.1.1

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E". (Mitchell, 1997)
(Mitchell's P for "performance measure" is the only place in these notes where the letter P does not denote a probability; from the next section on we call it the **loss** ℓ and the **risk** R.)
Purpose: Designing algorithms to solve T with maximum P and minimum:

1. time complexity (same as in any application)
2. space complexity (same as in any application)
3. sample complexity (new!)

1.1.1 Formal definitions

Experience (E)

Our computer program observes the environment by means of a **data set**, a set of tuples called **data points**:

$$S := \{(x_1, y_1), \ldots, (x_m, y_m)\}.$$

Each tuple comprises an **input** observation $x_i \in \mathcal{X}$, also called a **feature vector**, and a corresponding output observation $y_i \in \mathcal{Y}$.

The set $\mathcal{X}$ is called the **feature space** and the set $\mathcal{Y}$ a **label space**. In some contexts, S is called a **sample**.

We assume that there exists a **labeling function** $f : \mathcal{X} \to \mathcal{Y}$ such that $y_i = f(x_i)$, which is **unknown** to us.

We further assume that each observation in the data set is an **independent and identically distributed (i.i.d.)** sample from an unknown data distribution $\mathcal{D}$. An i.i.d. data set is denoted as $S := \{(x_i, y_i) \overset{i.i.d.}{\sim} \mathcal{D} : i \in [m]\}$.

Imagine playing backgammon. Each rolling of the dice is a sample. These are independent samples if consecutive rolls do not influence each other. The samples are identically distributed if we use the same dice throughout the game. The i.i.d. assumption is at the heart of machine learning. The same process needs to be repeated for an agent to be able to learn it, just like us humans.

Task (T)

Our task is to predict the outputs y_i from input observations x_i as accurately as possible. In mathematical terms, we look for a **hypothesis** $h : \mathcal{X} \to \mathcal{Y}$ that would approximate the unknown labeling function, i.e. it will satisfy $f(x) \approx h(x), \forall x \in \mathcal{X}$.

- If $\mathcal{Y}$ is a set with discrete elements, then our task is called **classification**, because it boils down to assigning an input x_i to an appropriate class. Imagine x_i is a picture

expressed as an array of pixel intensity values, $\mathcal{Y}$ is then the type of the object in the foreground.
- If $\mathcal{Y}$ is a continuous set such as $\mathbb{R}$, then our task is called **regression**. Imagine x_i is the properties of a house (size, age, coordinates), $\mathcal{Y}$ can be its price.

We do not search over all conceivable maps $\mathcal{X} \to \mathcal{Y}$. We commit in advance to a **hypothesis class** $\mathcal{H} \subseteq \mathcal{Y}^{\mathcal{X}}$ and search inside it. This commitment is the **inductive bias** of the learner. Chapter 5 shows that learning is impossible without one.

Performance Measure (P)

We define what we mean by an **accurate prediction** via a **loss function**: $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ which maps a pair (y, y') comprising a prediction y' and a corresponding true label y to a score inversely proportional to the quality of the prediction. Commonsense design choices for supervised learning problems are:

1. the **zero-one loss** defined as $\ell(y, y') := \mathbb{1}(y \neq y')$ for classification, and
2. the **squared error** defined as $\ell(y, y') := (y - y')^2$ for regression.

Here $\mathbb{1}$ is the indicator function that returns 1 if the predicate in its argument holds (e.g. if y differs from y') and 0 otherwise. Put together, we are interested in a **learning algorithm** $A(\cdot)$ that takes a data set S as input and returns a hypothesis. Let us denote this hypothesis as h_S, where the subscript S is to highlight its dependence on the data set. Then we can express the learning process as $h_S \leftarrow A(S)$. We expect from this algorithm to minimize the **generalization error (true risk)**, which is defined for zero-one loss as:

$$R(h) := \mathbb{E}_{x \sim \mathcal{D}_{\mathcal{X}}} \big[\ell(f(x), h(x))\big] = P_{x \sim \mathcal{D}_{\mathcal{X}}} \big(f(x) \neq h(x)\big),$$

where $\mathcal{D}_{\mathcal{X}}$ is the marginal of $\mathcal{D}$ on the feature space and the second equality is specific to the zero-one loss. Here $P_{x \sim \mathcal{D}_{\mathcal{X}}}$ denotes the probability that the sample x will be observed. In words, this is the probability that our hypothesis will make a different prediction than the true labeling function. We will cover probability theory in the following chapters. For now it is sufficient to read it as the true chance of encountering x, relying on your intuitive understanding of the notion of chance.

1.1.2 The Empirical Risk Minimization (ERM) Paradigm

The goal of a learning algorithm A is to find the hypothesis that minimizes the generalization error:

$$h_* := \arg\min_{h \in \mathcal{H}} R(h).$$

We cannot solve this optimization problem because we do not know $\mathcal{D}$ and f. We only have an idea about these unknowns via the data set S. Let us then use the data set to curate a quantity that approximates the generalization error:

$$\widehat{R}_S(h) := \frac{1}{m}\sum_{i\in[m]} \ell(y_i, h(x_i)) = \frac{|\{i \in [m] : h(x_i) \neq y_i\}|}{m}$$

where $[m] := \{1, \ldots, m\}$ and $|\cdot|$ counts the number of elements a set contains, e.g. $|\{1, 2\}| = 2$. The quantity $\widehat{R}_S(h)$ is called the **training error** or **empirical risk**. We can then find a suitable hypothesis by minimizing the empirical risk:

$$h_S := \arg\min_{h\in\mathcal{H}} \widehat{R}_S(h).$$

This paradigm, called **Empirical Risk Minimization (ERM)**, is at the center of machine learning. Yet, it has certain weaknesses. For instance, consider the following solution:

$$h_S(x) := \begin{cases} y_i, & \text{if } \exists i \in [m] \text{ s.t. } x_i = x, \\ 0, & \text{otherwise.} \end{cases}$$

This solution will provide us zero $\widehat{R}_S(h_S)$. It may look like that the problem is solved, but actually this is the moment where all problems start.

1.1.3 Example: Polynomial curve fitting

We observe data like below for 50 input observations x and the corresponding outputs y. We aim to find a function $y = f(x)$ that maps inputs to outputs.

```
import matplotlib.pyplot as plt
import torch as th

th.manual_seed(6)  # fix the random draws so that the figures are reproducible

# This is the true data generating process.
# We do not have access to this information in a typical machine learning problem
inputs = th.linspace(0, 1, 50, dtype=th.float64)
outputs = th.sin(2 * th.pi * inputs)
labels = outputs + th.randn(inputs.shape[0], dtype=th.float64) * 0.25

plt.plot(inputs, labels, 'bo')
plt.xlabel("inputs (x)")
plt.ylabel("labels (y)")
plt.show()
```

Express our observations as a set of input-output tuples: $S = \{(x_1, y_1), \cdots, (x_{50}, y_{50})\}$. We would like to use them for two purposes:

- to **learn** the unknown function f,
- to **evaluate** the success of the learning process (i.e. how well we expect f to predict future observations)

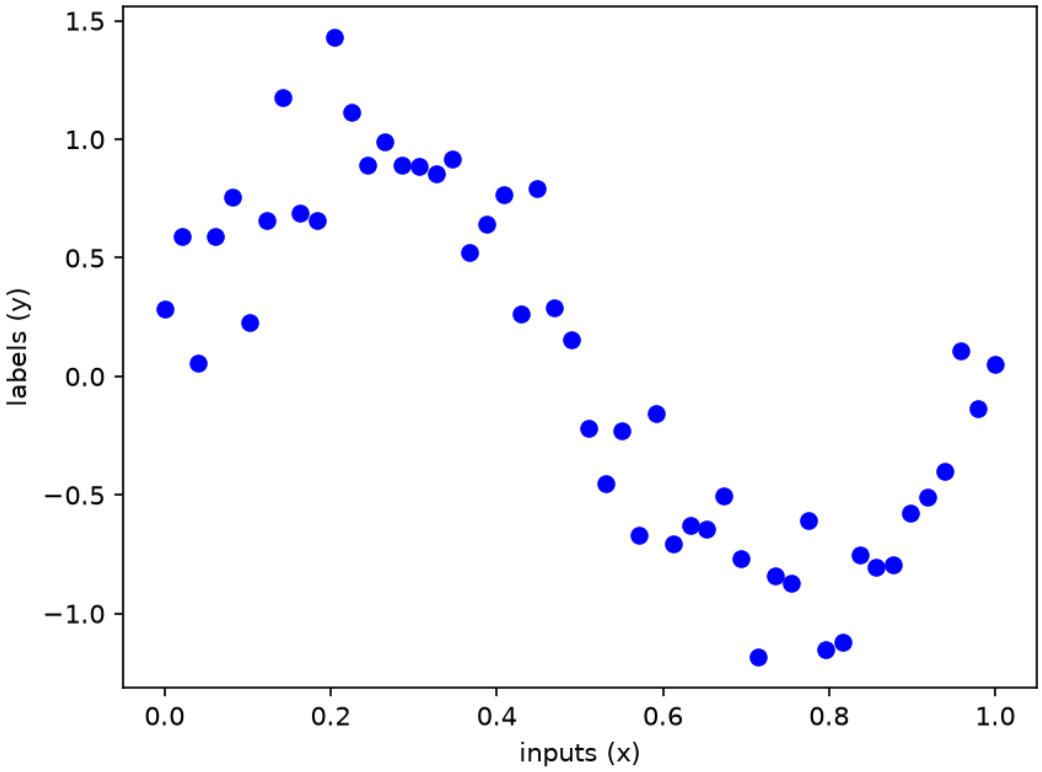


Figure 1.1: Synthetic noisy observations of a sinusoidal labeling function.

Let us then split our data into two equal partitions randomly, one per purpose: $S = S_{train} \cup S_{test}$.

S_{train} is called the **training set**. Our learning algorithm will use it to find a hypothesis that minimizes the empirical risk.

S_{test} is called the **test set**. We will use it to check how well the empirical risk represents the true risk. In formal terms, we are up to approximating $R(h_S)$.

```
num_samples = inputs.shape[0]
num_train_samples = num_samples // 2
# draw a random permutation of indices
idx = th.randperm(num_samples)
inputs_train = inputs[idx[:num_train_samples]]
labels_train = labels[idx[:num_train_samples]]
inputs_test = inputs[idx[num_train_samples:]]
labels_test = labels[idx[num_train_samples:]]
```

By visual inspection we find out that the input and the output have a nonlinear and periodic relationship. Let us then choose a hypothesis space suitable to this approximation, for instance a polynomial of degree M:

$$y(x) := \sum_{m=0}^{M} w_m x^m.$$

If $M = 0$, then our model learns to fit a constant: $y(x) = w_0$.
If $M = 1$, then it fits a line: $y(x) = w_0 + w_1 x$.
If $M = 2$, it fits a parabola: $y(x) = w_0 + w_1 x + w_2 x^2$, and so on.

```
class PolynomialModel:
    def __init__(self, num_polynomial_degrees=1, regularizer=0.0):
        self.M = num_polynomial_degrees
        self.regularization_factor = regularizer

    def extract_features(self, inputs):
        # Column m of the feature matrix Z holds x^m, for m = 0, ..., M
        powers = th.arange(self.M + 1, dtype=inputs.dtype)
        return inputs.unsqueeze(1) ** powers

    def learn(self, inputs, labels):
        Z = self.extract_features(inputs)
        ridge = self.regularization_factor * th.eye(Z.shape[1], dtype=Z.dtype)
        A = Z.T @ Z + ridge
        # w_S = (Z^T Z + lambda I)^{-1} Z^T y, obtained by solving the linear
        # system A w = Z^T y rather than by forming A^{-1} explicitly
        self.weights = th.linalg.solve(A, Z.T @ labels)

    def predict(self, inputs):
        return self.extract_features(inputs) @ self.weights
```

The **extract_features** function converts each one-dimensional observation to an $M-$dimensional **feature vector**. This operation is called **feature extraction**.

The **learn** function takes a set of inputs and their corresponding labels in its arguments and fits the unknown model parameters $w_0, \cdots, w_M$ to data. We will cover later how exactly it does that.

The **predict** function takes only inputs in its arguments. It predicts their corresponding labels itself, hence the name. The source code of advanced machine learning models we will see later on follows the same template.

```
model = PolynomialModel(num_polynomial_degrees=3) # Create the model instance

# Training
model.learn(inputs_train, labels_train)

# Testing
predictions_test = model.predict(inputs_test)
root_mean_squared_error = ((labels_test - predictions_test) ** 2).mean().sqrt()

print(root_mean_squared_error.item())
```

```
0.25155540073010746
```

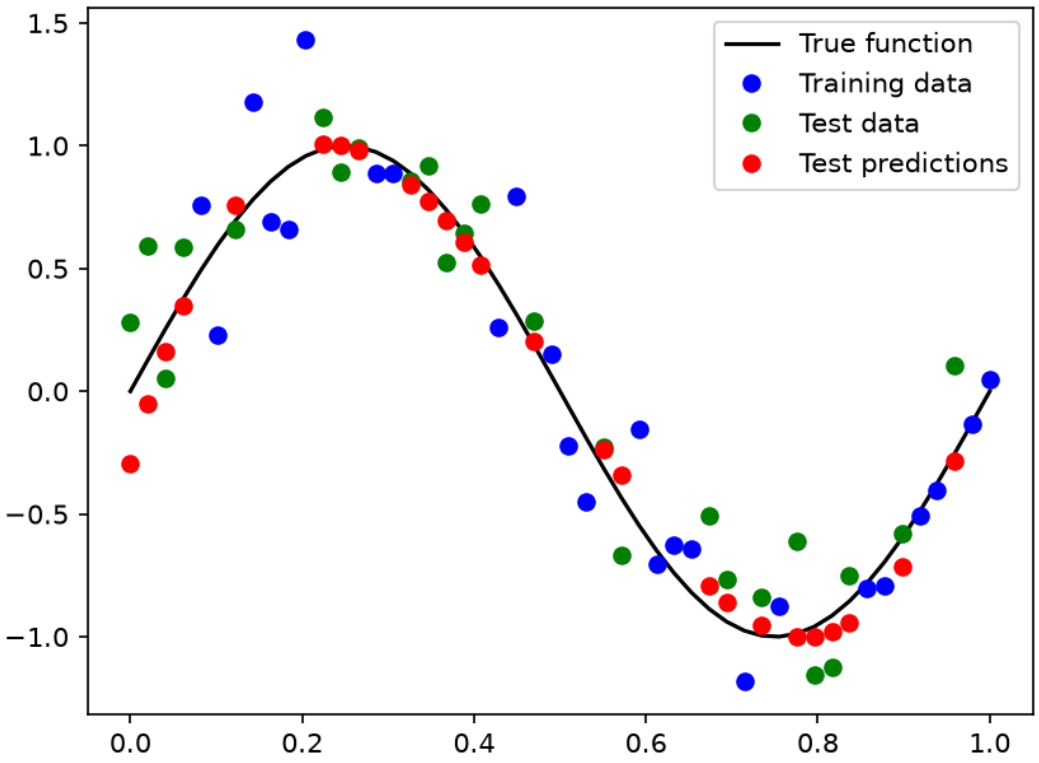


Figure 1.2: The true labeling function together with the training set, the test set, and the model's predictions on the test set.

Let us visualize all we have. In the plot below, the training set is shown as blue circles, test set as green circles, and the predictions of our machine learning model in red circles. The black curve is the labeling function we are trying to finding out. Note that we can observe only noisy evaluations of it, just as in the real life. Data collection processes are affected by unintended factors, which we call **noise**.

```
plt.plot(inputs,outputs,'k-', label="True function")
plt.plot(inputs_train,labels_train,'bo', label="Training data")
plt.plot(inputs_test, labels_test,'go', label="Test data")
plt.plot(inputs_test,predictions_test,'ro', label="Test predictions")
plt.legend(loc="upper right")
plt.show()
```

The properties of the model we have to choose before running the learning algorithm are called **hyperparameters**. For instance, the polynomial degree M is a hyperparameter. Let us simply try all options from 1 to 10 and see how it affects the model behavior.

Since we have a regression problem, we follow the common sense and choose the squared error loss function. We take its square-root to bring it to the same scale as the label space:

$$\mathrm{RMSE}_S(h) := \sqrt{\widehat{R}_S(h)} = \sqrt{\frac{1}{m}\sum_{i\in[m]}(h(x_i)-y_i)^2}.$$

This loss function is called the **Root Mean Squared Error (RMSE)**.

```
# The performance score
def root_mean_squared_error(labels_true, predictions):
    return ((labels_true - predictions) ** 2).mean().sqrt()

Mmax = 10
train_errors = th.zeros(Mmax, dtype=th.float64)
test_errors = th.zeros(Mmax, dtype=th.float64)

for M in range(Mmax):
    # Create the model instance
    model = PolynomialModel(num_polynomial_degrees=M)

    # Training: fit parameters to data on the training split.
    model.learn(inputs_train, labels_train)
    predictions_train = model.predict(inputs_train)
    train_err = root_mean_squared_error(labels_train, predictions_train)

    # Testing: predict on the test split and evaluate performance.
    predictions_test = model.predict(inputs_test)
    test_err = root_mean_squared_error(labels_test, predictions_test)

    train_errors[M] = train_err
    test_errors[M] = test_err

plt.plot(th.arange(Mmax), train_errors, "bo-", label="Training")
plt.plot(th.arange(Mmax), test_errors, "ro-", label="Test")
plt.xlabel("Polynomial degree")
plt.ylabel("Error")
plt.legend(loc="upper right")
plt.show()
```

Expected: As M increases, the training error $\widehat{R}_{S_{train}}(h_S)$ monotonically decreases.

Surprise: However, the test error $\widehat{R}_{S_{test}}(h_S)$ decreases up to $M = 6$, and then sharply increases.

What may have gone wrong? Let us take a closer look into individual cases.

```
model0 = PolynomialModel(num_polynomial_degrees=0)
model0.learn(inputs_train, labels_train)
pred_0 = model0.predict(inputs)

model1 = PolynomialModel(num_polynomial_degrees=1)
model1.learn(inputs_train, labels_train)
pred_1 = model1.predict(inputs)

model4 = PolynomialModel(num_polynomial_degrees=4)
```

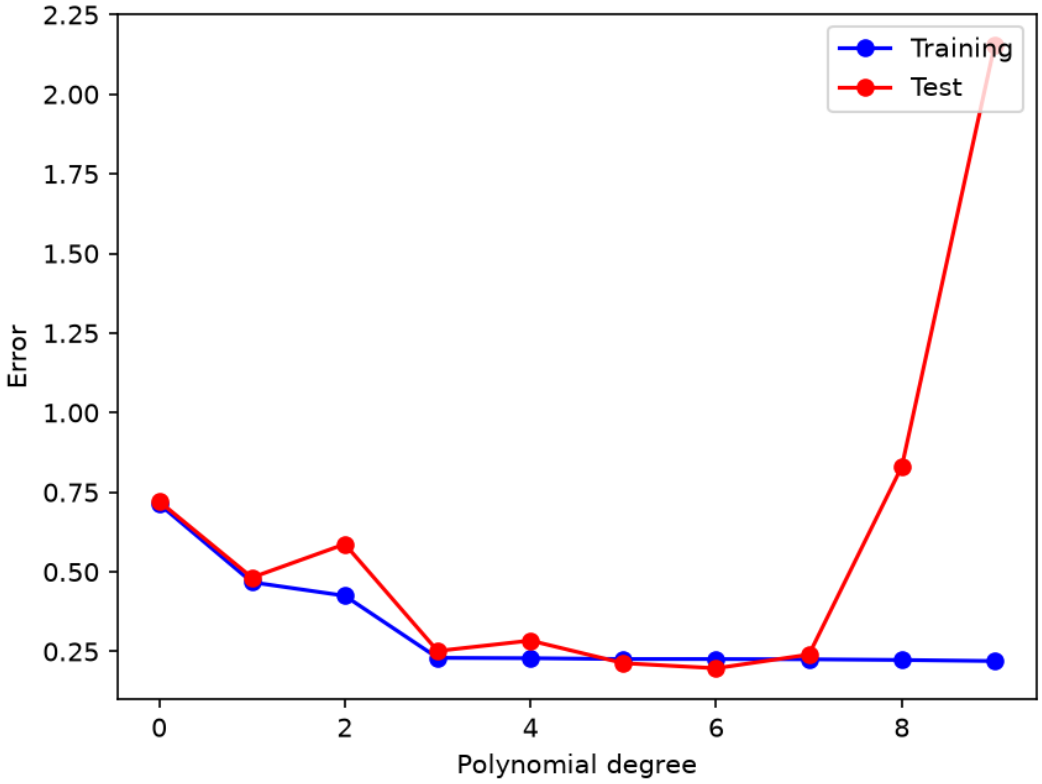


Figure 1.3: Training and test error as a function of the polynomial degree M.

```
model4.learn(inputs_train, labels_train)
pred_4 = model4.predict(inputs)

model9 = PolynomialModel(num_polynomial_degrees=9)
model9.learn(inputs_train, labels_train)
pred_9 = model9.predict(inputs)

fig, axs = plt.subplots(2, 2)

for ax in axs.flat:
    ax.plot(inputs, outputs,'k-', label="True f")
    ax.plot(inputs_train, labels_train,'ro', label="Tr data")

axs[0, 0].plot(inputs, pred_0,'b-', label="M=0")
axs[0, 1].plot(inputs, pred_1,'g-', label="M=1")
axs[1, 0].plot(inputs, pred_4,'c-', label="M=4")
axs[1, 1].plot(inputs, pred_9,'m-', label="M=9")

for ax in axs.flat:
    ax.legend(loc="upper right")
    ax.label_outer()
```

The model had poor performance because it could not express a periodic relationship with a constant $M = 0$ or a line $M = 1$. Poor model fit due to limited model capacity is called **underfitting**. An underfitted model performs poorly on both training and test data, i.e. it has high $\widehat{R}_{S_{train}}(h_S)$ and high $\widehat{R}_{S_{test}}(h_S)$.

What goes on for $M = 9$ is interesting. A polynomial with such high degree is flexible enough to make sharp turns. The model learned to use this flexibility to fit also to the left and right-most data points, but then it missed to capture the trend and diverged from the true labeling function. It is called **overfitting** when a model fits to training data extremely well, i.e. low $\widehat{R}_{S_{train}}(h_S)$, but delivers poor performance on test data, i.e. high $\widehat{R}_{S_{test}}(h_S)$.

REMARK: Designing highly-expressive models while keeping them immune to overfitting is one of the fundamental problems of machine learning.

To mitigate overfitting, we need to devise an algorithm that encourages minimum empirical risk and penalizes the increase of model capacity. Once choice could be as follows:

$$h_S := \arg\min_{h \in \mathcal{H}} \widehat{R}_S(h) + \lambda \sum_{m=0}^{M} w_m^2.$$

Here the term $\lambda \sum_{m=0}^{M} w_m^2$ is called a **regularizer** and λ a **regularization coefficient**. Let us see what happens then.

```
for M in range(Mmax):
    # Create the model instance
    model = PolynomialModel(num_polynomial_degrees=M, regularizer=0.01)
```

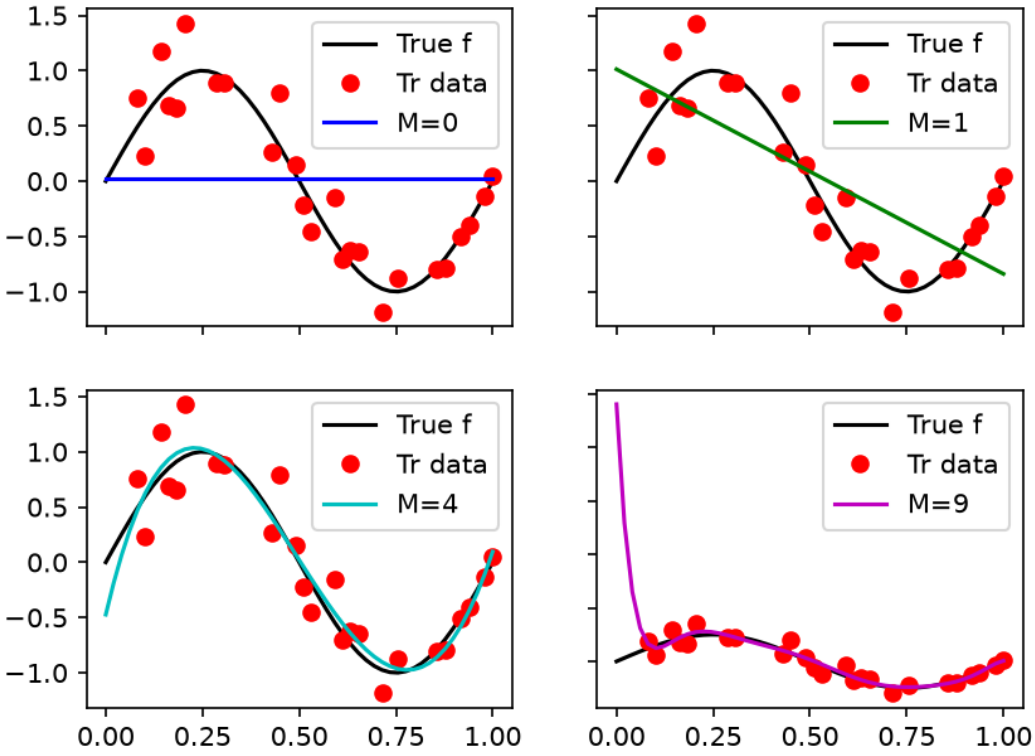


Figure 1.4: Polynomial fits of degree $M \in \{0, 1, 4, 9\}$ against the true function and training data, illustrating underfitting and overfitting.

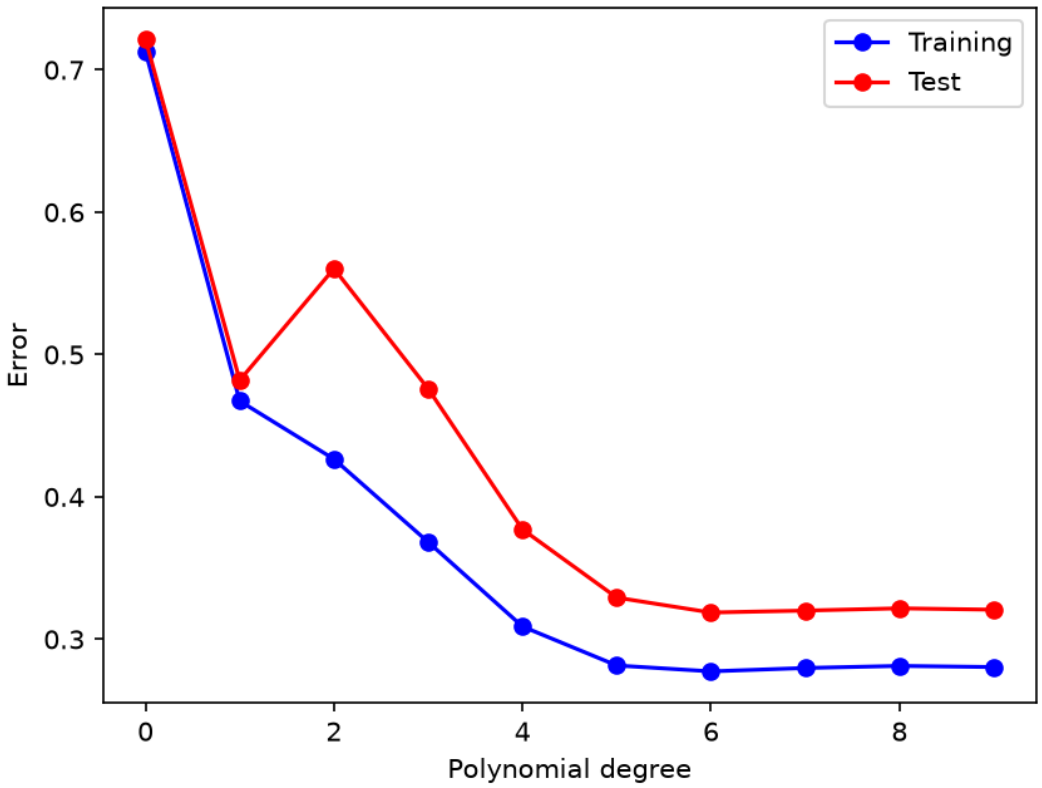


Figure 1.5: Training and test error versus polynomial degree with L_2 regularization, showing reduced overfitting compared to the unregularized fit.

```
    # Training: fit parameters to data on the training split.
    model.learn(inputs_train, labels_train)
    predictions_train = model.predict(inputs_train)
    train_err = root_mean_squared_error(labels_train, predictions_train)

    # Testing: predict on the test split and evaluate performance.
    predictions_test = model.predict(inputs_test)
    test_err = root_mean_squared_error(labels_test, predictions_test)

    train_errors[M] = train_err
    test_errors[M] = test_err

plt.plot(th.arange(Mmax), train_errors, "bo-", label="Training")
plt.plot(th.arange(Mmax), test_errors, "ro-", label="Test")
plt.xlabel("Polynomial degree")
plt.ylabel("Error")
plt.legend(loc="upper right")
plt.show()
```

As seen, having a regularizer term significantly mitigates overfitting when the model capacity, i.e. the polynomial degree, increases.

CHAPTER 2

LINEAR PREDICTORS

Assume we observe data that look as below.

```
import matplotlib.pyplot as plt
import torch as th

th.manual_seed(0)  # fix the random draws so that the figures are reproducible

# This is the true data generating process.
# We do not have access to this information in a typical machine learning problem
inputs = th.linspace(-5, 5, 50, dtype=th.float64)
outputs = inputs * 0.5 + 3
labels = outputs + th.randn(inputs.shape[0], dtype=th.float64)

grid = th.linspace(-5, 5, 500, dtype=th.float64)
plt.plot(inputs, labels, 'bo')
plt.plot(grid, grid * 0.5 + 3, 'k-')
plt.xlabel("inputs (x)")
plt.ylabel("labels (y)")
plt.show()
```

Our visual inspection tells us that there is strong linear correlation between inputs and the outputs. Then let us choose our hypothesis set accordingly:

$$\mathcal{H} := \{h : h(x) = w_0 + w_1 x,\ (w_0, w_1) \in \mathbb{R}^2\}.$$

Denoting $w := (w_0, w_1)$, we can re-express the hypothesis as $h(x) := w^\top x$, where $\top$ indicates a matrix transpose.

Assume we are given a training set $S = \{(x_i, y_i) : i \in [m]\}$. Its mean squared-error empirical risk for our chosen hypothesis h is:

$$\widehat{R}_S(w) := \frac{1}{m} \sum_{i \in [m]} (w^\top x_i - y_i)^2.$$

Let us perform empirical risk minimization:

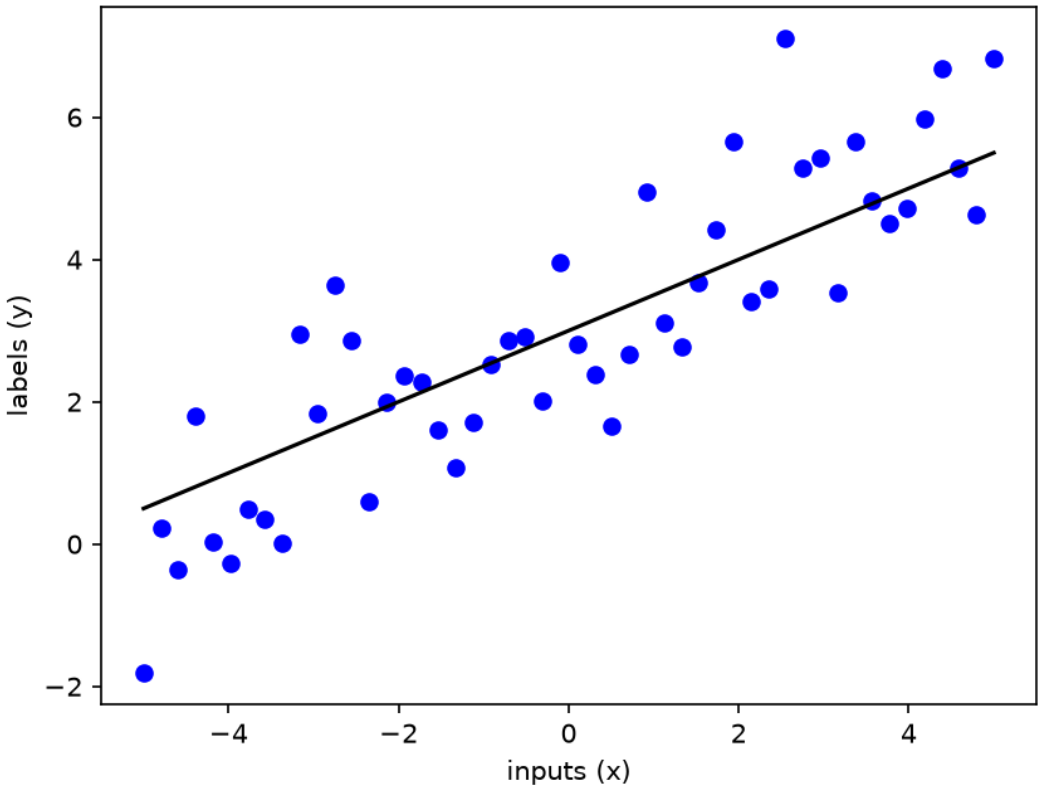


Figure 2.1: Synthetic data with a strong linear relationship between input and label, together with the underlying linear generating function.

$$w_S := \arg\min_w \widehat{R}_S(w).$$

The squared error loss gets its minimum value at the point where its derivative is zero. Hence, we can find the solution at w_S that satisfies $\nabla_w \widehat{R}_S(w)|_{w:=w_S} = 0$, that is

$$\begin{aligned}
\widehat{R}_S(w) &= \frac{1}{m}\sum_{i\in[m]}\left(w^\top x_i x_i^\top w - 2w^\top x_i y_i + y_i^2\right) \\
\Rightarrow \nabla_w \widehat{R}_S(w) &= \frac{1}{m}\sum_{i\in[m]}\left(2x_i x_i^\top w - 2x_i y_i\right) \triangleq 0 \\
&\Rightarrow \sum_{i\in[m]} x_i x_i^\top w_S = \sum_{i\in[m]} x_i y_i \\
&\Rightarrow w_S = \left(\sum_{i\in[m]} x_i x_i^\top\right)^{-1} \sum_{i\in[m]} x_i y_i
\end{aligned}$$

Fitting a linear model on data is often referred to as **least-squares regression** and w_S the **least-squares solution**.

Denote each data point by a column vector $z_i := (x_i, 1)$. Expressing the linear model as $w^\top z_i$ and using the definition of the dot product we have

$$w^\top z_i = x_i w_1 + w_0$$

where the term w_0 is referred to as the **bias**. The bias gives the model the opportunity to shift the predictor to an appropriate position before fitting the line. As we will see later, this flexibility sometimes brings substantial improvement in model fit.

Let us collect all input samples into a $\mathbb{R}^{m\times 2}$ matrix that contains data points on its rows:

$$Z = \begin{bmatrix} z_1^\top \\ \vdots \\ z_m^\top \end{bmatrix}$$

Let us also collect the corresponding labels in a vector $y := (y_1, \ldots, y_m)$. We can re-express the least-squares solution in vector notation as follows

$$w_S := (Z^\top Z)^{-1} Z^\top y.$$

Let us implement it and solve the task.

```
num_samples = inputs.shape[0]
num_train_samples = num_samples
idx = th.randperm(num_samples)
inputs_train = inputs[idx[:num_train_samples]]
labels_train = labels[idx[:num_train_samples]]

class LinearRegression:
```

```
    def extract_features(self, inputs):
        # Append a constant 1 to every input, so that w^T z = w_1 x + w_0
        ones = th.ones(inputs.shape[0], 1, dtype=inputs.dtype)
        return th.cat((inputs.unsqueeze(1), ones), dim=1)

    def learn(self, inputs, labels):
        Z = self.extract_features(inputs)
        # This is where the least-squares solution is implemented!
        # w_S = (Z^T Z)^{-1} Z^T y, obtained by solving the linear system
        # (Z^T Z) w = Z^T y, which is numerically safer than inverting Z^T Z
        self.weights = th.linalg.solve(Z.T @ Z, Z.T @ labels)

    def predict(self, inputs):
        return self.extract_features(inputs) @ self.weights

model = LinearRegression()

model.learn(inputs_train, labels_train)
predictions = model.predict(grid)

plt.plot(inputs, labels, 'bo')
plt.plot(grid, grid * 0.5 + 3, 'k-', label='true labeling function')
plt.plot(grid, predictions, 'r-', label='learned hypothesis')
plt.legend(loc="upper left")
plt.xlabel("inputs (x)")
plt.ylabel("labels (y)")
plt.show()
```

2.1 Metric spaces

We would like the vector spaces (such as feature spaces, parameters spaces etc.) used in machine learning to have some plausible properties. We can guarantee the existence of these properties by accompanying a continuous domain $\mathcal{X}$ with a distance function $dist : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$ that is

1. **positive:** $\forall a, a' \in \mathcal{X},\ a \neq a' \Rightarrow \text{dist}(a, a') > 0$,
2. **zero on the diagonal:** $\forall a \in \mathcal{X},\ \text{dist}(a, a) = 0$,
3. **symmetric:** $\forall a, a' \in \mathcal{X},\ \text{dist}(a, a') = \text{dist}(a', a)$.

We get these properties by assuming these vector spaces to be **metric spaces**, that is vector space that satisfies

- $\forall a, b, c \in \mathcal{X},\ \text{dist}(a, c) \leq \text{dist}(a, b) + \text{dist}(b, c)$, which is called the **triangle inequality**.

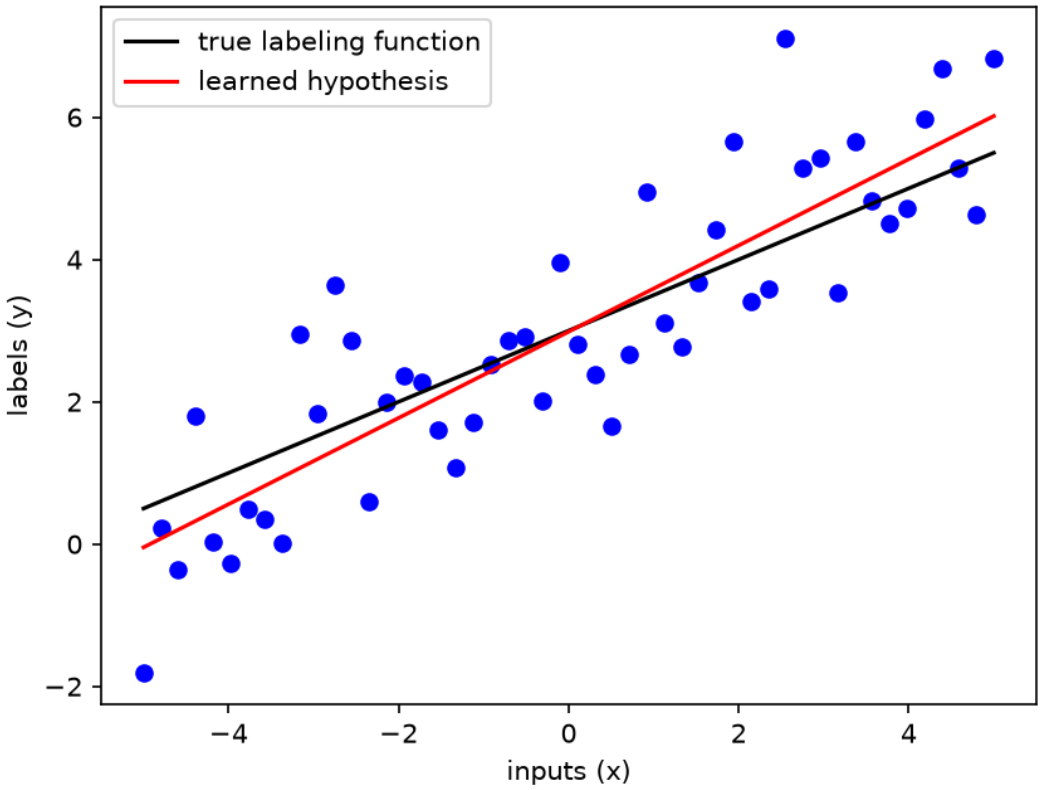


Figure 2.2: The learned least-squares line together with the true labeling function and the training data.

For $p \geq 1$, the L_p-norm of a vector $u \in \mathbb{R}^d$ is defined as follows

$$||u||_p := \Big(\sum_{j=1}^{d} |u_j|^p \Big)^{1/p}.$$

If $p = 2$, we get the well-known **Euclidean norm**.
If $p = 1$, we get the **Manhattan norm**.
As $p \to \infty$, we tend to get the **Maximum norm** (L_∞), i.e. $||u||_\infty := \max\{|u_1|, \ldots, |u_d|\}$.
We can use this norm to derive many nice metric spaces

$$\text{dist}(a, b) := ||a - b||_p.$$

For $0 < p < 1$ the same expression is still well defined and still traces the iso-contours plotted below, but it is no longer a norm: it violates the triangle inequality.

Let us plot the behavior of these distances as a function of p. The red lines below are iso-contours of $||u||_p = 1$ for $u = (a_1, a_2)$ and different choices of p. The unit ball shrinks as p decreases: a small p charges a vector heavily for spreading its mass over many coordinates, so it favours **sparse** vectors.

```
p_values = [0.5, 1, 2, 3, 7, float('inf')]
a1, a2 = th.meshgrid(th.linspace(-1.2, 1.2, 101),
                     th.linspace(-1.2, 1.2, 101), indexing='xy')
fig, axes = plt.subplots(ncols=(len(p_values) + 1) // 2,
                         nrows=2, figsize=(14, 7))
for p, ax in zip(p_values, axes.flat):
    if p == float('inf'):
        zz = th.maximum(a1.abs(), a2.abs())
    else:
        zz = (a1.abs()**p + a2.abs()**p)**(1./p)
    ax.contour(a1, a2, zz, [1], colors='red', linewidths=2)
    ax.set_title("p= {0}".format(p))

plt.show()
```

2.2 Regularized least squares

We have seen in Chapter 1 that $\widehat{R}_S(h_S) = 0$ can be achieved by memorizing the training set, resulting in overfitting. Let us then write the hypothesis space as a nested chain of subclasses of growing capacity, $\mathcal{H}_1 \subset \mathcal{H}_2 \subset \cdots$ with $\mathcal{H} = \bigcup_i \mathcal{H}_i$, and devise an objective that pursues the dual goals of fitting the data as well as possible and constraining model complexity concurrently:

$$h_S := \arg\min_{h \in \mathcal{H}} \widehat{R}_S(h) + \lambda \cdot \text{pen}(i(h)),$$
$$i(h) := \min\{i : h \in \mathcal{H}_i\}.$$

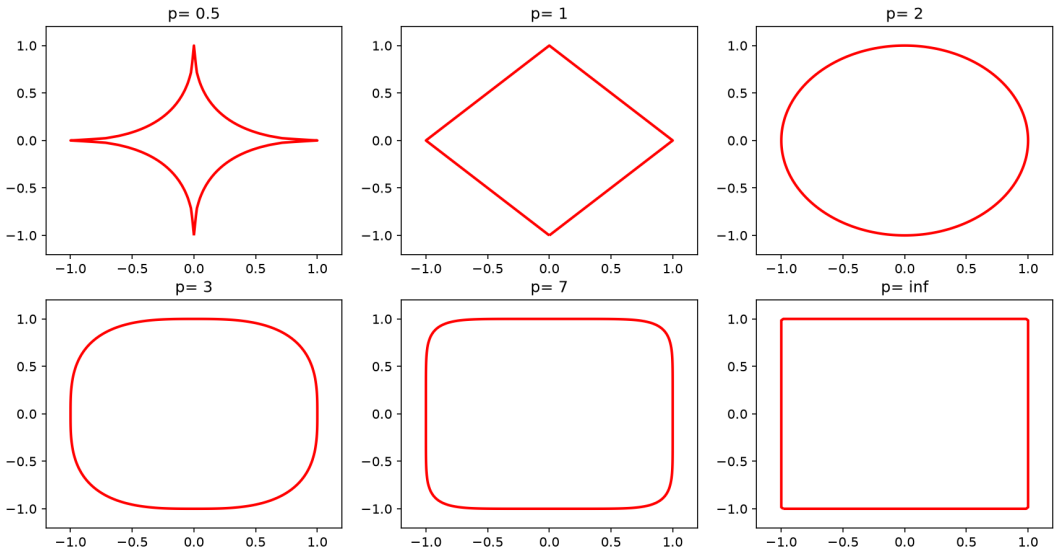


Figure 2.3: Unit-ball contours of the ℓ_p norm for $p \in \{0.5, 1, 2, 3, 7, \infty\}$, shrinking towards the axes as p decreases.

Here λ is called a **regularization coefficient** and $\text{pen}(\cdot)$, an increasing function of the subclass index, a **regularizer**. As we will see in more detail in Chapter 5, this paradigm is called **Structural Risk Minimization (SRM)**, one of the biggest achievements of machine learning research in the pre-deep-learning era.

Let us next apply the idea of constraining the hypothesis space to the least squares problem:

$$w_S := \arg\min_w \frac{1}{m} \sum_{i \in [m]} (w^\top x_i - y_i)^2$$

$$\text{s.t.} \quad ||w||_p^p \leq \eta$$

where the abbreviation s.t. stands for **subject to**, meaning that we search for a solution within a feasible set of parameters $\{w : ||w||_p \leq \eta\}$. This constrained optimization problem can be expressed equivalently as:

$$w_S := \arg\min_w \max_{\lambda \geq 0} \frac{1}{m} \sum_{i \in [m]} (w^\top x_i - y_i)^2 + \lambda(||w||_p^p - \eta)$$

where $\lambda \geq 0$ is the Lagrange multiplier of the constraint. For every budget η there is a λ whose unconstrained problem has the same solution. Fixing λ instead of η and dropping the constant $\lambda\eta$ gives the **regularized least squares** objective:

$$w_S := \arg\min_w \underbrace{\frac{1}{m} \sum_{i \in [m]} (w^\top x_i - y_i)^2}_{\widehat{R}_S(w)} + \lambda ||w||_p^p.$$

The special case of $p = 2$ deserves detailed investigation:

$$w_S := \arg\min_w \frac{1}{m} \sum_{i \in [m]} (w^\top x_i - y_i)^2 + \lambda ||w||_2^2.$$

We can rewrite the loss of this optimization in vector form as:

$$\mathcal{L}_S(w) := \frac{1}{m} \|Zw - y\|_2^2 + \lambda w^\top w.$$

Let us find the optimal weights that minimize the loss by setting its gradient to zero once again:

$$\begin{aligned}
\mathcal{L}_S(w) &= \frac{1}{m} w^\top Z^\top Z w - \frac{2}{m} w^\top Z^\top y + \frac{1}{m} y^\top y + \lambda w^\top w \\
&= w^\top \left(\frac{1}{m} Z^\top Z + \lambda I \right) w - \frac{2}{m} w^\top Z^\top y + \frac{1}{m} y^\top y \\
\Rightarrow \nabla_w \mathcal{L}_S(w) &= 2 \left(\frac{1}{m} Z^\top Z + \lambda I \right) w - \frac{2}{m} Z^\top y \triangleq 0 \\
\Rightarrow w_S &= \left(\frac{1}{m} Z^\top Z + \lambda I \right)^{-1} \frac{1}{m} Z^\top y = \left(Z^\top Z + m\lambda I \right)^{-1} Z^\top y.
\end{aligned}$$

The result, known as **ridge regression** (Hoerl and Kennard, 1970), differs from the least-squares solution $w_S = (Z^\top Z)^{-1} Z^\top y$ by only the added term $m\lambda I$. This term makes the matrix invertible even when $Z^\top Z$ is singular, and it pulls every coordinate of w_S towards zero. Methods that pull parameters towards zero instead of fitting them freely are known in statistics as **parameter shrinkage** methods. Note that ridge shrinks but does not sparsify: it drives coefficients close to zero, never exactly to zero. We will see next that $p = 1$ does sparsify. The regularizer of ridge regression is referred to as **weight decay**, a name still in active use by deep learning libraries.

Let us next see ridge regression in action.

```
class RidgeRegression:
    def extract_features(self, inputs):
        ones = th.ones(inputs.shape[0], 1, dtype=inputs.dtype)
        return th.cat((inputs, ones), dim=1)

    def learn(self, inputs, labels, lambda_coef=0.1):
        Z = self.extract_features(inputs)
        # w_S = (Z^T Z + lambda I)^{-1} Z^T y: the least-squares system above
        # with lambda I added to the matrix being inverted
        ridge = lambda_coef * th.eye(Z.shape[1], dtype=Z.dtype)
        A = Z.T @ Z + ridge
        self.weights = th.linalg.solve(A, Z.T @ labels)

    def predict(self, inputs):
        return self.extract_features(inputs) @ self.weights
```

This time we work on the Diabetes data set that consists of 442 data points. Each data point represents a patient with the following 10 features:

1. age
2. sex
3. body-mass index
4. average blood pressure
5. total serum cholesterol
6. low-density lipoprotein level
7. high-density lipoprotein level
8. total cholesterol level
9. possibly log of serum triglycerides level
10. blood sugar level

The goal is to predict a quantitative progression of disease progression, the higher the more severe.

```
from sklearn.datasets import load_diabetes

# The only library-provided piece is the raw data set itself; the split,
# the normalization, and both estimators are implemented below.
d = load_diabetes()
X = th.tensor(d.data, dtype=th.float64)
y = th.tensor(d.target, dtype=th.float64)

# hold out 20% of the data as the test split
perm = th.randperm(X.shape[0])
n_test = int(0.20 * X.shape[0])
X_test, y_test = X[perm[:n_test]], y[perm[:n_test]]
X_train, y_train = X[perm[n_test:]], y[perm[n_test:]]

print("Means:")
print(X_train.mean(dim=0))
print("Variances:")
print(X_train.var(dim=0, unbiased=False))
```

```
Means:
tensor([-2.3438e-03,  5.7266e-05,  6.4150e-04, -1.2388e-03,
        -2.0432e-03, -2.1458e-03, -1.7280e-04, -1.1079e-03,
        -6.1292e-04, -1.4638e-03],
       dtype=torch.float64)
Variances:
tensor([0.0023, 0.0023, 0.0024, 0.0023, 0.0022,
        0.0021, 0.0022, 0.0022, 0.0023, 0.0023],
       dtype=torch.float64)
```

Notably, different features have different scales. This may cause artifacts in model fitting, such as prioritization of features according to their scales instead of their relevance to the predicted quantity of interest. We can mitigate these artifacts by **normalizing** the data. One commonplace approach is **z-score normalization**, which assumes that each feature is normal distributed with some mean μ and variance σ^2. That is, each coordinate x_j of a d-dimensional observation x comes from a sampling process as below:

$$\epsilon \sim \mathcal{N}(0, 1),$$
$$x_j = \mu_j + \sigma_j \epsilon.$$

This operation makes the assumption that all the observed differences across individual results (called factors of variation) stem from the first step, i.e. a sample from a standard normal distribution. The second step scales and shifts this sample. Z-score normalization reverses the second operation to bring all features to the first step:

$$x'_j := \frac{x_j - \mu_j}{\sigma_j},$$
$$\forall j \in [d].$$

For μ_j and σ_j, we use the sample mean and standard deviation of the corresponding feature, computed on the **training split only** so that no information leaks from the test split. This preprocessing step is also referred to as **standardization**. We will revisit its probability-theoretic justification later.

```
# z-score normalization, with mu and sigma taken from the training split only
mu = X_train.mean(dim=0)
sd = X_train.std(dim=0, unbiased=False)
X_train = (X_train - mu) / sd
X_test = (X_test - mu) / sd

print("Means:")
print(X_train.mean(dim=0))
print("Variances:")
print(X_train.var(dim=0, unbiased=False))
```

```
Means:
tensor([ 5.0180e-18,  3.6380e-17,  1.5054e-17,  3.7635e-18,
         2.8226e-17,  1.7563e-17, -4.0144e-17,  4.5162e-17,
        -5.0180e-18,  1.7563e-17],
       dtype=torch.float64)
Variances:
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
        1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
       dtype=torch.float64)
```

Now we are ready to train and test ridge regression on the Diabetes data set.

```
model_ridge = RidgeRegression()

model_ridge.learn(X_train, y_train, lambda_coef=1)

predictions = model_ridge.predict(X_train)
train_error = ((predictions - y_train)**2).mean().sqrt()
print("Train RMSE: {:.2f}".format(train_error))

predictions = model_ridge.predict(X_test)
test_error = ((predictions - y_test)**2).mean().sqrt()
print("Test RMSE: {:.2f}".format(test_error))
```

```
Train RMSE: 54.35
Test RMSE: 50.92
```

We have a decent result. But can we do even better? Can we actually sparsify our solution? Inspired by the shape of the L_p balls illustrated above, we can next try out $p = 1$. For d-dimensional input vectors x_i, the resulting objective reads:

$$\mathcal{L}_S(w) := \frac{1}{m} \sum_{i \in [m]} (w^\top x_i - y_i)^2 + \lambda \sum_{j=1}^{d} |w_j|.$$

This approach is known as **Least Absolute Shrinkage Selection Operator (LASSO) Regression** (Tibshirani, 1996).

Unlike ridge regression and least squares, the Lasso objective does not have an analytical solution. The L_1 term is not differentiable at $w_j = 0$, which is exactly where its solutions like to sit. No closed-form formula exists for a w that would satisfy

$$\nabla_w \Big(\frac{1}{m} \sum_{i \in [m]} (w^\top x_i - y_i)^2 + \lambda \sum_{j=1}^{d} |w_j| \Big) \triangleq 0.$$

If we cannot go straight to the value that minimizes the loss, we can instead find its direction and take a step towards there. Remember that the derivative of a function points to the direction towards which a function grows. Then negating it should point us towards a neighboring position with lower loss than our current position. It is reasonable to hope that repetitively taking small steps towards a neighboring point with lower loss will bring us to the point where the loss is minimum. Consider a series $w_0, w_1, \ldots$ created by the following succession rule and some arbitrary initialization w_0:

$$w_{t+1} := w_t - \alpha \nabla_w \mathcal{L}_S(w)|_{w:=w_t}$$

This approach is called **gradient descent**. It is in use with nearly all modern machine learning approaches. The coefficient $\alpha > 0$ is called a **learning rate**. It determines how fast the model parameters will change in a single iteration. Too large a learning rate may result in missing the optimal solution due to large oscillations around it. Too small a learning rate may result in infeasibly long training time.

Gradient descent requires repetitive evaluation of the gradient of the loss with respect to the parameters at every iteration: $\nabla_w \mathcal{L}_S(w)|_{w:=w_t}$. Hence its implementation on the computer requires an analytical calculation of this gradient. This may be time consuming for complex loss functions. Deep learning libraries such as PyTorch and TensorFlow allow us to automate this process, a mechanism called **automatic differentiation**: flagging a tensor with `requires_grad_()` makes the library record every operation it takes part in, and a later call to `backward()` replays that record in reverse, accumulating $\nabla_w \mathcal{L}_S(w)$ into the tensor's `.grad` field. Note that only the *gradient computation* is automated — the update rule $w_{t+1} := w_t - \alpha \nabla_w \mathcal{L}_S(w)$ is still ours to write, and we write it out explicitly below rather than delegating it to a library optimizer, so that nothing about the algorithm is hidden.

```
class LassoRegression:
    def __init__(self, n_dims, lambda_coef=1.0):
        self.lambda_coef = lambda_coef
        # The parameters are plain tensors flagged for gradient tracking:
        # requires_grad_() tells torch to record every operation they take
        # part in, so that backward() can later replay the record in reverse.
        self.w = th.randn(n_dims, 1).requires_grad_()
        self.b = th.randn(1).requires_grad_()

    def predict(self, inputs):
        return inputs @ self.w + self.b

    def learn(self, inputs, labels, alpha=0.01, num_steps=1):
        for _ in range(num_steps):
            # Forward pass: evaluate the objective at the current parameters
            loss = ((self.predict(inputs) - labels)**2).mean() \
                   + self.lambda_coef * self.w.abs().sum()
            # Backward pass: automatic differentiation fills w.grad and b.grad
            loss.backward()
            # The gradient descent step itself, written out explicitly. The
            # no_grad() block switches the recording off, since the update is
            # not part of the function we are differentiating.
            with th.no_grad():
                for param in (self.w, self.b):
                    param -= alpha * param.grad
                    param.grad.zero_()   # clear it for the next iteration


# Single precision is enough for an iterative solver, and it is what the
# deep learning libraries use by default
X_train = X_train.float()
X_test = X_test.float()
y_train = y_train.float().reshape(-1, 1)
y_test = y_test.float().reshape(-1, 1)

# Train our model
```

```
model_lasso = LassoRegression(n_dims=X_train.shape[1], lambda_coef=1)
# Number of gradient descent iterations
num_iterations = 250

# Collect the train and test errors here.
train_errors = th.zeros(num_iterations)
test_errors = th.zeros(num_iterations)

for ii in range(num_iterations):
    model_lasso.learn(X_train, y_train)

    with th.no_grad():   # evaluation needs no gradient record
        predictions = model_lasso.predict(X_train)
        train_errors[ii] = ((predictions - y_train)**2).mean().sqrt()

        # Test our model
        predictions = model_lasso.predict(X_test)
        test_errors[ii] = ((predictions - y_test)**2).mean().sqrt()

# Plot the learning curve
plt.plot(th.arange(num_iterations), train_errors, 'b-', label="Train RMSE")
plt.plot(th.arange(num_iterations), test_errors, 'r-', label="Test RMSE")
plt.xlabel("Iteration")
plt.ylabel("RMSE")
plt.legend(loc="upper right")
plt.show()
```

The figure above is called the **learning curve**. It depicts the evolution of model performance throughout the learning process.

REMARK: Learning curves give plenty of information about the model behavior. Hence, plotting and visually inspecting them facilitates debugging.

Let us next see how much shrinkage we gained from the Lasso and ridge regression regularizers. As visible below, Lasso sparsifies the parameters more than ridge.

```
fig, axes = plt.subplots(ncols=2, nrows=1, figsize=(14, 7))

axes[0].bar(th.arange(X_train.shape[1]),
            model_lasso.w.view(-1).detach())
axes[0].set_ylim(-40,40)
axes[0].set_title("Lasso weights")
axes[1].bar(th.arange(X_train.shape[1]),
            model_ridge.weights[:10])
axes[1].set_ylim(-40,40)
axes[1].set_title("Ridge weights")
plt.show()
```

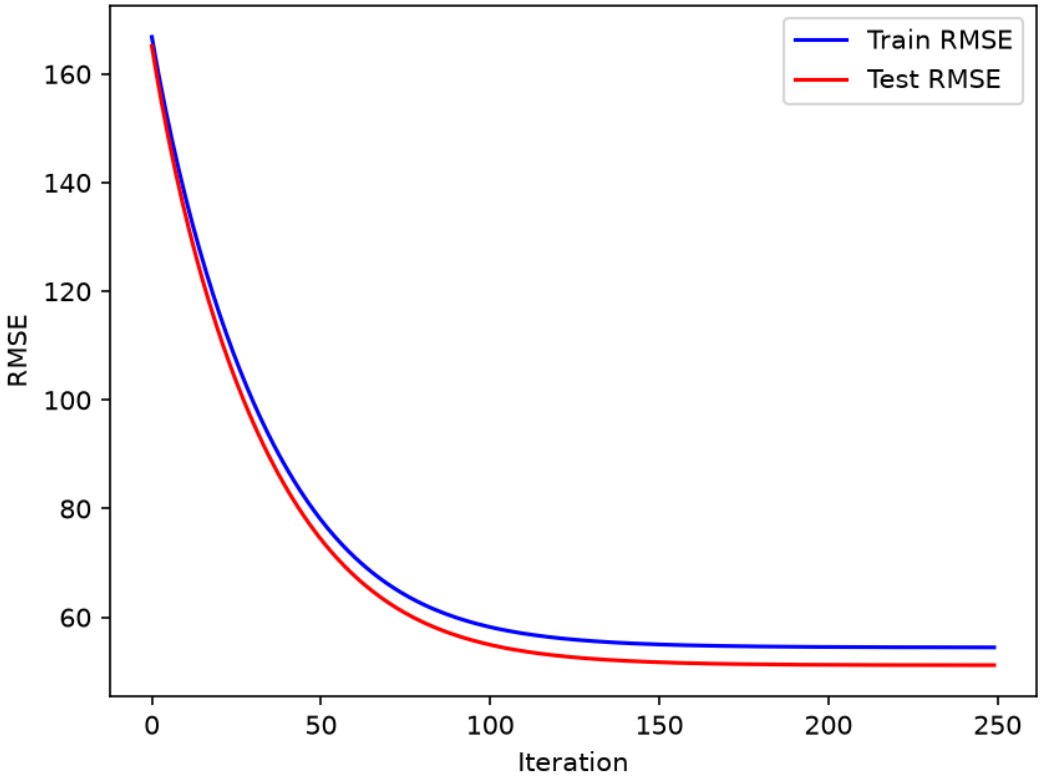


Figure 2.4: Learning curve: training and test RMSE of Lasso regression across gradient descent iterations.

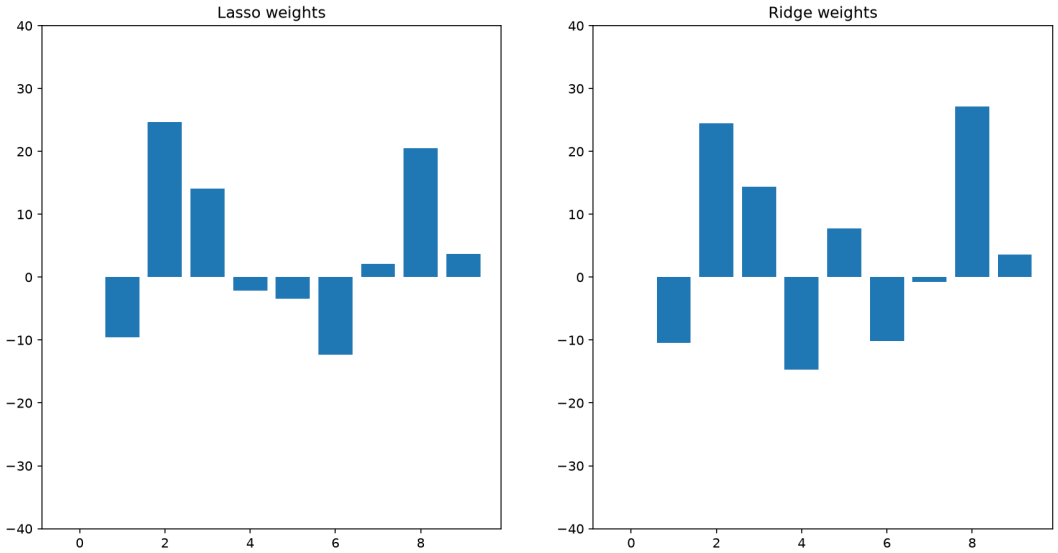


Figure 2.5: Learned weight coefficients under Lasso versus ridge regularization, showing that Lasso drives more coefficients to zero.

CHAPTER 3

CLASSIFICATION

3.1 Logistic Regression

Assume we have a binary classification problem. We aim to fit a h to a data set $S = \{(x_i, y_i) | i \in [m]\}$ that consists of (x, y) pairs with $y \in \{0, 1\}$.

As in the previous lecture, we choose the hypothesis to be a linear function of the input:

$$\mathcal{H} := \{h : h(x) = w^\top x,\ w \in \mathbb{R}^d\}.$$

For notational convenience, let us ignore the bias term as one can always append a constant one to the input vector x to account for the bias.

The difference of the current setting is that the output space is discrete. Let us interpret the output of our hypothesis, the **score** $z_i := w^\top x_i$, as follows:

- Its sign indicates the class label. If $z_i > 0$, then the hypothesis predicts that x_i belongs to class 1. If $z_i < 0$, then the hypothesis predicts that x_i belongs to class 0.
- Its magnitude $|z_i|$ indicates the confidence of the hypothesis about the class assignment.

When interpreted this way, z_i is called the **discriminant function** (in Chapter 8 the same quantity reappears, under the same symbol, as the pre-activation of a neuron). While readily usable in its current shape, such a model is not easy to train. We desire a loss function that is differentiable with respect to the parameters w. The sign function is not differentiable. We can achieve differentiability by converting the discriminant function to a probability. Let $P_i := P(y_i{=}1 \mid x_i)$ denote the probability that x_i belongs to class 1, whose logarithm we take to be proportional to the discriminant function. That is:

$$\log P_i \propto w^\top x_i.$$

Considering also that the probability that x_i is a member of class 0 is $1 - P_i$, we arrive at the following equation:

$$\log \frac{P_i}{1 - P_i} = w^\top x_i.$$

Let us solve for P_i stepwise:

$$
\begin{aligned}
&\frac{P_i}{1-P_i} = e^{w^\top x_i} \\
&\Rightarrow P_i = e^{w^\top x_i} - P_i e^{w^\top x_i} \\
&\Rightarrow P_i + P_i e^{w^\top x_i} = e^{w^\top x_i} \\
&\Rightarrow P_i(1 + e^{w^\top x_i}) = e^{w^\top x_i} \\
&\Rightarrow P_i = \frac{e^{w^\top x_i}}{1 + e^{w^\top x_i}} \\
&\Rightarrow P_i = \frac{e^{w^\top x_i} e^{-w^\top x_i}}{(1 + e^{w^\top x_i}) e^{-w^\top x_i}} \\
&\Rightarrow P_i = \frac{1}{1 + e^{-w^\top x_i}}.
\end{aligned}
$$

The result $g(u) = \frac{1}{1+e^{-u}}$ has the form of a function known as the **sigmoid function**. The expression $\log(P_i/(1-P_i))$ is sometimes called the **log odds** or the **logit function**. It is the inverse of the sigmoid function: $g^{-1}(P) = \log(P/(1-P))$ for some probability P. When $u := w^\top x_i$, the model is called **logistic regression** (Cox, 1958).

We would like our hypothesis to maximize the true class probability of all data points in the data set. Since we assume our data points to be independent, we can maximize the probability of each data point separately. Hence, we can maximize the probability of the data set by maximizing the product of the probabilities of each data point

$$\prod_{i=1}^{m} P_i^{y_i}(1-P_i)^{1-y_i}.$$

This is equivalent to maximizing the sum of the logarithms of the probabilities of each data point. Hence, we can define the **binary cross-entropy loss** as its negation

$$\mathcal{L}_{CE}(w) = -\sum_{i=1}^{m} \log g(w^\top x_i)^{y_i}(1-g(w^\top x_i))^{1-y_i} = -\sum_{i=1}^{m} \Big[y_i \log g(w^\top x_i) + (1-y_i)\log(1-g(w^\top x_i))\Big].$$

3.1.1 Extension to multi-class classification

Let us quickly extend the above formulation to multi-class classification. Assume we have C classes, that is we aim to fit a h to a data set $S = \{(x_i, y_i)|i \in [m]\}$ that consists of (x, y) pairs with $y \in \{1, \ldots, C\}$. We will then need to model the class probabilities of C different classes, $P_1, \ldots, P_C$, from C per-class logits $u_c := w_c^\top x_i$:

$$\log P_c \propto u_c = w_c^\top x_i.$$

Solving for P_c and assuring that the class probabilities sum up to one yields:

$$\text{softmax}(u)_c = \frac{e^{u_c}}{\sum_{c'=1}^{C} e^{u_{c'}}},$$

which is called the **softmax** function. The related loss function is then:

$$\mathcal{L}_{CE}(W) = -\sum_{i=1}^{m} \log \text{softmax}(u_i)_{y_i} = \sum_{i=1}^{m} \Big\{ -w_{y_i}^\top x_i + \log\Big(\sum_{c=1}^{C} e^{w_c^\top x_i}\Big)\Big\},$$

where $u_i := (w_1^\top x_i, \ldots, w_C^\top x_i)$ and $W = [w_1 \ldots w_C]$ is a matrix of the weight vectors for each class. This loss function is called the **cross-entropy loss**. We will revisit it and understand better why it is given this particular name.

Let us take the well-known **Iris** data set as an example. Our task is to classify the iris flowers into three species. The data set is available in the sklearn library. The data set consists of 150 data points. Each data point has four features: sepal length, sepal width, petal length, and petal width. The data points are labeled as one of the three species: i) setosa, ii) versicolor, and iii) virginica. The data set is balanced: There are 50 data points for each species.

```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import torch as th

th.manual_seed(0)

# The only library-provided piece is the raw data set itself; the split and
# every step of the classifier are implemented below.
X_all, y_all = load_iris(return_X_y=True)
X_all = th.tensor(X_all).float()
y_all = th.tensor(y_all).long()

perm = th.randperm(X_all.shape[0])
n_train = X_all.shape[0] // 2
X_train, y_train = X_all[perm[:n_train]], y_all[perm[:n_train]]
X_test, y_test = X_all[perm[n_train:]], y_all[perm[n_train:]]
```

Let us put together what we learned thus far to make up an as general algorithm as possible. Strictly speaking, let us use the cross-entropy loss for empirical risk minimization and use an L_p regularizer with tunable p to control the complexity of the model. The resulting optimization problem is:

$$\mathcal{L}(W) := \mathcal{L}_{CE}(W) + \lambda \sum_{c=1}^{C} ||w_c||_p^p.$$

We minimize it by gradient descent, exactly as for the Lasso in Chapter 2: the softmax cross-entropy is written out in its log-sum-exp form derived above, automatic differentiation supplies $\nabla_W \mathcal{L}(W)$, and the update rule is coded by hand.

```
class GeneralizedLinearClassifier:
    def __init__(self, n_dims, n_classes=3, lambda_coef=1.0, p=2):
```

```
        self.lambda_coef, self.p = lambda_coef, p
        self.W = th.randn(n_dims, n_classes).requires_grad_()
        self.b = th.randn(n_classes).requires_grad_()

    def predict(self, inputs):
        return inputs @ self.W + self.b        # the C per-class logits u_c

    def cross_entropy(self, logits, labels):
        # -log softmax(u)_y = -u_y + log sum_c exp(u_c), exactly the loss
        # derived above. The row maximum is subtracted before exponentiating
        # so that exp() cannot overflow; it cancels out of the difference.
        shift = logits.max(dim=1, keepdim=True).values
        log_norm = shift.squeeze(1) + (logits - shift).exp().sum(dim=1).log()
        return (log_norm - logits.gather(1, labels.view(-1, 1)).squeeze(1)).mean()

    def learn(self, inputs, labels, alpha=0.1, num_steps=1):
        for _ in range(num_steps):
            # Forward pass: cross-entropy risk plus the L_p regularizer
            loss = self.cross_entropy(self.predict(inputs), labels) \
                   + self.lambda_coef * (self.W.abs()**self.p).sum()
            # Backward pass: automatic differentiation fills W.grad and b.grad
            loss.backward()
            # The gradient descent step, written out explicitly
            with th.no_grad():
                for param in (self.W, self.b):
                    param -= alpha * param.grad
                    param.grad.zero_()
```

Let us train our model next and plot its learning curve, i.e. how its error changes across iterations.

```
# z-score normalization, with mu and sigma taken from the training split only
mu = X_train.mean(dim=0)
sd = X_train.std(dim=0, unbiased=False)
X_train = (X_train - mu) / sd
X_test = (X_test - mu) / sd

model_glc = GeneralizedLinearClassifier(n_dims=X_train.shape[1],
                                        lambda_coef=0.01, p=2)
num_iterations = 1000
train_errors = th.zeros(num_iterations)
test_errors = th.zeros(num_iterations)

for ii in range(num_iterations):
    model_glc.learn(X_train, y_train)
    with th.no_grad():
        pred_class = model_glc.predict(X_train).argmax(dim=1)
```

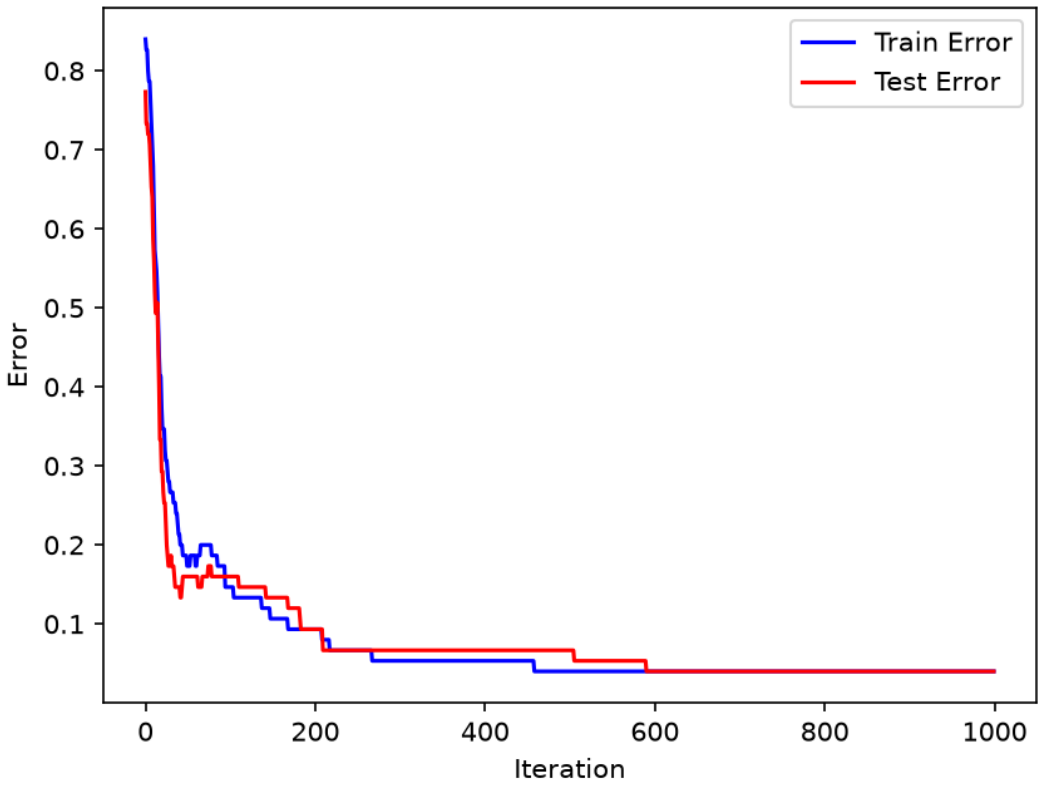


Figure 3.1: Training and test error of the generative linear classifier across gradient descent iterations.

```
        train_errors[ii] = (pred_class != y_train).float().mean()
        pred_class = model_glc.predict(X_test).argmax(dim=1)
        test_errors[ii] = (pred_class != y_test).float().mean()

plt.plot(th.arange(num_iterations), train_errors, 'b-', label="Train Error")
plt.plot(th.arange(num_iterations), test_errors, 'r-', label="Test Error")
plt.xlabel("Iteration"); plt.ylabel("Error"); plt.legend(loc="upper right")
plt.show()
```

3.2 Performance Metrics for Classification

The goal of classification is to recognize a pattern, e.g. an object. From the viewpoint of a single object type, there can be four different outcomes of a classification task:

- True Positive (TP): The object is correctly classified as positive.
- False Positive (FP): The object is incorrectly classified as positive.
- True Negative (TN): The object is correctly classified as negative.
- False Negative (FN): The object is incorrectly classified as negative.

Let us make a matrix of these possible outcomes as a latex table:

	Predicted Positive	**Predicted Negative**
Actual Positive	TP	FN
Actual Negative	FP	TN

The version of this table where its entries are filled with the number of data points that fall into each category is called a **confusion matrix**. We can compute many performance metrics from the confusion matrix:

1. **Accuracy**: (TP + TN) / (TP + FP + TN + FN), the ratio of the number of correctly classified objects to the total number of objects.
2. **Precision**: TP / (TP + FP), the ratio of the number of correctly classified objects to the number of objects classified as the object type.
3. **Recall** (a.k.a. **True Positive Rate**, **Sensitivity**): TP / (TP + FN), the ratio of the number of correctly classified objects to the number of objects that are actually of the object type.
4. **F1 score**: 2 × Precision × Recall / (Precision + Recall), the harmonic mean of the precision and the recall.
5. **False Positive Rate**: FP / (FP + TN), the ratio of the number of objects that are not of the object type but classified as the object type to the total number of objects that are not of the object type.

The rationale behind the F1 score is that precision and recall are rates. Harmonic mean is a more sensible score to take the average of multiple rates. Consider a car that travels a distance d with speed x and returns with speed y. The average speed of the whole travel is

$$\frac{2d}{\frac{d}{x}+\frac{d}{y}} = \frac{2}{\frac{1}{x}+\frac{1}{y}}$$

which is the harmonic mean of x and y. In words, harmonic mean is the reciprocal of the average of the reciprocals of a set of quantities.

One can calculate the confusion matrix also for more than two classes. Let us see how it looks like for the Iris data set.

```
with th.no_grad():
    pred_class = model_glc.predict(X_test).argmax(dim=1)

C = 3
# cm[a, p] counts the test points with actual label a and prediction p
cm = th.bincount(y_test * C + pred_class, minlength=C * C).reshape(C, C)
print(cm)

names = ['setosa', 'versicolor', 'virginica']
fig, ax = plt.subplots()
ax.imshow(cm, cmap='Blues')
```

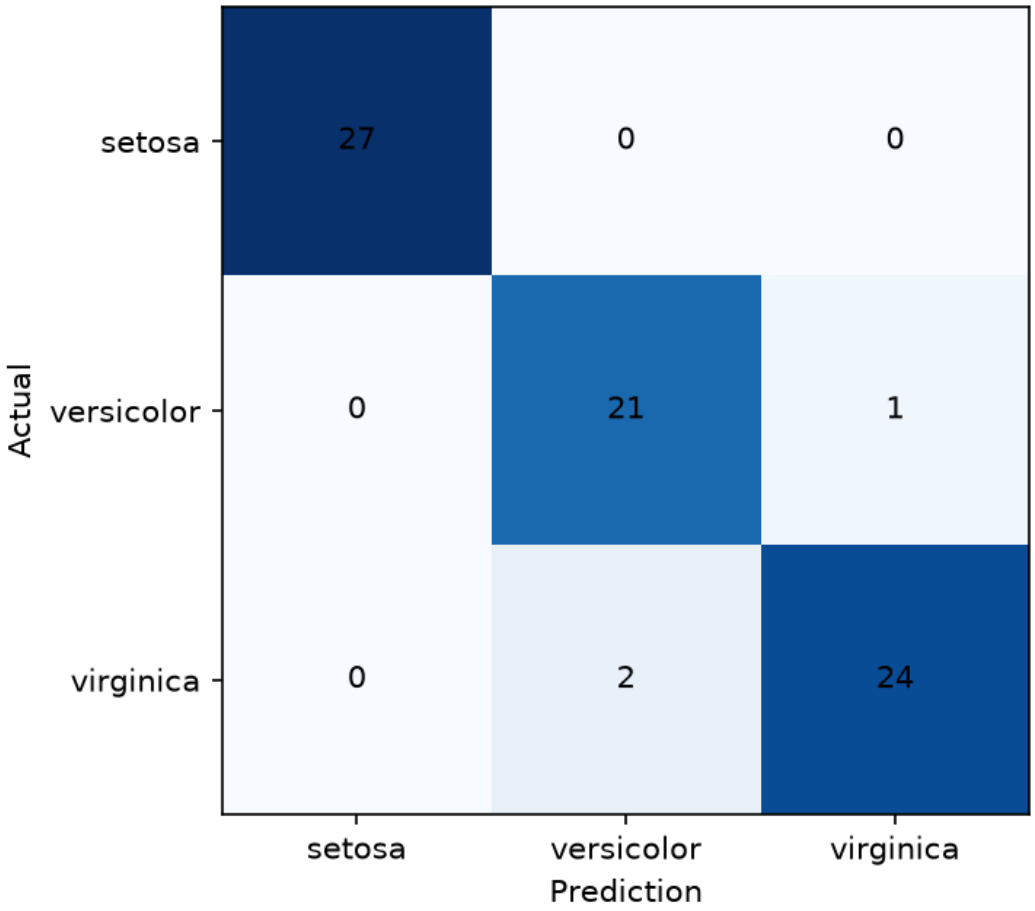


Figure 3.2: Confusion matrix of the classifier's predictions on the Iris test set.

```
for a in range(C):
    for p in range(C):
        ax.text(p, a, cm[a, p].item(), ha='center', va='center')
ax.set_xticks(range(C), names); ax.set_yticks(range(C), names)
ax.set_xlabel("Prediction"); ax.set_ylabel("Actual")
plt.show()
```

```
tensor([[27,  0,  0],
        [ 0, 21,  1],
        [ 0,  2, 24]])
```

Except for accuracy, the other four performance metrics need to be calculated separately for each class. For instance, let us calculate them for versicolor, the class the model finds hardest.

```
c = 1                                          # versicolor
precision = cm[c, c] / cm[:, c].sum()          # sum towards the column
```

```
recall = cm[c, c] / cm[c, :].sum()              # sum towards the row
print("Precision: ", precision.item())
print("Recall: ", recall.item())
print("F1: ", (2 * precision * recall / (precision + recall)).item())
```

```
Precision:  0.9130434989929199
Recall:  0.9545454382896423
F1:  0.9333332777023315
```

Another important performance metric is **Receiver Operating Characteristics (ROC)**. It is a plot of the true positive rate (TPR) against the false positive rate (FPR) for all possible thresholds applicable on the discriminant function. The **Area Under the ROC curve (AUC)** is a measure of the performance of the classifier. The higher the AUC, the better the classifier. A perfect classifier has an AUC of 1. A classifier that performs no better than random guessing has an AUC of 0.5.

Let us plot the ROC curve for the Iris data set.

```
with th.no_grad():
    logits = model_glc.predict(X_test)
    # The class posteriors P_c = softmax(u)_c of the model
    probs = (logits - logits.max(dim=1, keepdim=True).values).exp()
    probs = probs / probs.sum(dim=1, keepdim=True)

# One-vs-rest score for versicolor (class 1) and the matching binary labels
scores = probs[:, 1]
positives = (y_test == 1)

# Sweep the threshold down through the observed scores, in decreasing order,
# and read off TPR and FPR at every position.
order = th.argsort(scores, descending=True)
hits = positives[order].float()
tp = th.cat([th.zeros(1), hits.cumsum(0)])
fp = th.cat([th.zeros(1), (1 - hits).cumsum(0)])
tpr, fpr = tp / tp[-1], fp / fp[-1]
auc = th.trapz(tpr, fpr)                # area under the curve, by trapezoids

plt.figure()
plt.plot(fpr, tpr, 'b-', label="versicolor (AUC = {:.2f})".format(auc))
plt.plot([0, 1], [0, 1], 'k--', label="random guessing (AUC = 0.50)")
plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
plt.legend(loc="lower right")
plt.show()
```

REMARK: Accuracy is a meaningful performance metric only when the classes are evenly distributed. The cases when the classes are not evenly distributed are called **imbalanced classification problems**. In such cases, we need to use other metrics such as precision, recall, and F1 score.

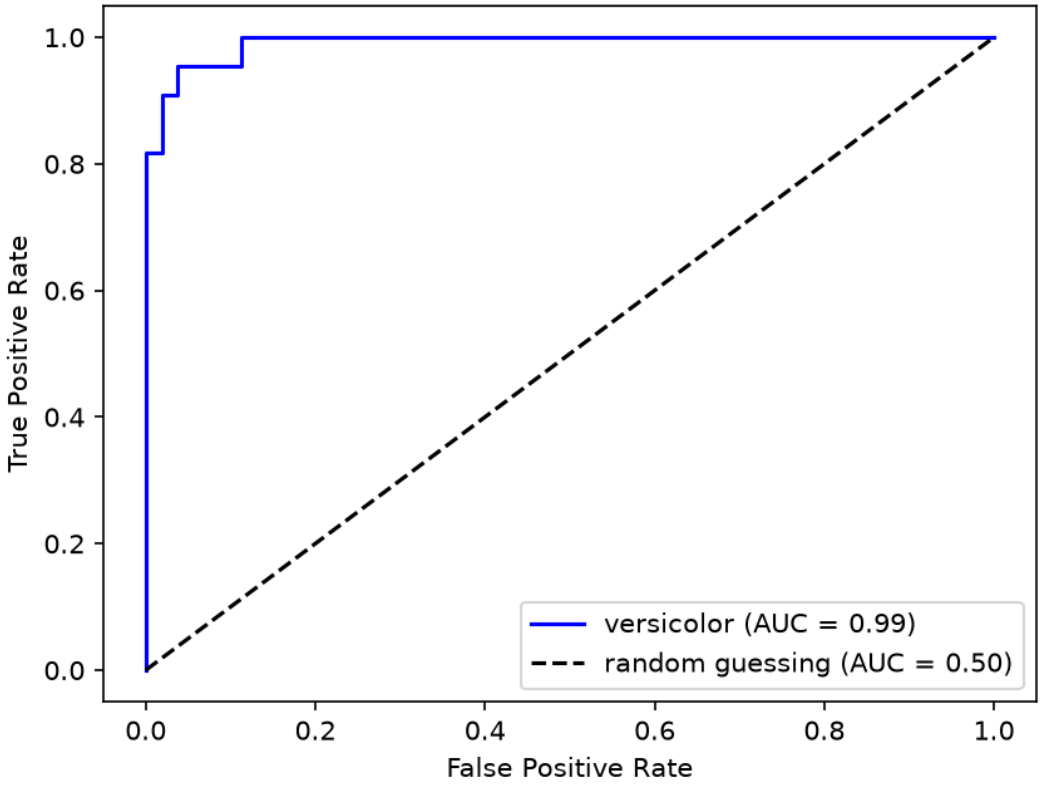


Figure 3.3: ROC curve for the versicolor-vs-rest classifier, with the corresponding AUC compared against random guessing.

3.3 K-Fold Cross Validation

As noticeable in the above example, the observed performance may depend greatly on the particular train-test split. To mitigate this problem, we can use **K-fold cross validation**. The idea is to split the data set into K folds. Then, we train the model on $K-1$ folds and test it on the remaining fold. We repeat this process K times, each time using a different fold as the test set. The final performance metric is the average of the performance metrics obtained in each iteration.

3.4 K-Nearest Neighbors (kNN) Classifier

The **k-nearest neighbors** (Fix and Hodges, 1951) classifier assigns a query point to the class of the majority of its k nearest neighbors. It is a **non-parametric classifier**, that is it does not represent the model with a fixed number of parameters. Instead it memorizes the whole training data and uses it to make predictions. The number of neighbors k is a hyperparameter of the model. While being a strong classifier, its prediction-time complexity is unacceptable: $O(m)$, where m is the number of training data points.

One can implement a kNN in various ways. Given a query point x, one can find its k nearest neighbors $Ne(x;k) = \{(x_i, y_i) : [k]\}$ in the training data set and assign x to the class of the majority of its k nearest neighbors:

$$\widehat{y} = \arg\max_{c\in[C]} \sum_{i=1}^{k} \mathbb{1}(y_i = c).$$

One can also assign a weight to each neighbor inversely proportional to its distance to the query point and assign x to the class of the majority of its k nearest neighbors weighted by the inverse of their distances to x.

$$\widehat{y} = \arg\max_{c\in[C]} \sum_{i=1}^{k} \mathbb{1}(y_i = c)\frac{1}{\text{dist}(x, x_i)}$$

for some distance function $\text{dist}(\cdot,\cdot)$ on $\mathcal{X}$, as defined in Chapter 2.

The **Voronoi cell** of a point x is the set of points whose nearest neighbor is x. The Voronoi cells of the training data points form a partition of the input space. The decision boundary of a 1-nearest neighbor classifier is the set of points that are equidistant to two or more training data points. For general k, the decision boundary instead sits where the *majority label* among the k nearest neighbors changes — a subset of the points equidistant between two training points, but not all of them, since moving across such a point need not flip the majority vote. See an illustration from the Iris data set below.

```
from matplotlib.colors import ListedColormap

# A tiny subsample, so that the individual Voronoi cells stay visible
sub = th.randperm(X_all.shape[0])[:10]
X_knn, y_knn = X_all[sub][:, :2], y_all[sub]
```

```
def knn_predict(queries, X, y, k=1, weights="uniform", n_classes=3):
    """Assign each query to the class with the largest (weighted) vote among
    its k nearest training points -- the two rules stated above."""
    d = th.cdist(queries, X)                          # pairwise distances
    dist, idx = d.topk(k, dim=1, largest=False)       # the k nearest neighbours
    w = th.ones_like(dist) if weights == "uniform" else 1.0 / (dist + 1e-12)
    votes = th.zeros(queries.shape[0], n_classes)
    votes.scatter_add_(1, y[idx], w)                  # accumulate 1(y_i = c) * w
    return votes.argmax(dim=1)

cmap_light = ListedColormap(["orange", "cyan", "cornflowerblue"])
cmap_bold = ["darkorange", "c", "darkblue"]
names = ['setosa', 'versicolor', 'virginica']

# The mesh on which the decision regions are painted
pad = 0.5
g0 = th.linspace(X_knn[:, 0].min() - pad, X_knn[:, 0].max() + pad, 200)
g1 = th.linspace(X_knn[:, 1].min() - pad, X_knn[:, 1].max() + pad, 200)
gg0, gg1 = th.meshgrid(g0, g1, indexing='xy')
mesh = th.stack([gg0.reshape(-1), gg1.reshape(-1)], dim=1)

for weights in ["uniform", "distance"]:
    # weights=uniform: All points in each neighborhood are weighted equally
    # weights=distance: weight points by the inverse of their distance
    zz = knn_predict(mesh, X_knn, y_knn, k=1, weights=weights).reshape(gg0.shape)

    fig, ax = plt.subplots()
    ax.pcolormesh(gg0, gg1, zz, cmap=cmap_light, shading="auto")
    for c in range(3):
        pts = X_knn[y_knn == c]
        ax.scatter(pts[:, 0], pts[:, 1], c=cmap_bold[c],
                   edgecolor="black", label=names[c])
    ax.set_xlabel("sepal length (cm)"); ax.set_ylabel("sepal width (cm)")
    ax.legend()
    ax.set_title("3-Class classification (k = 1, weights = '%s')" % weights)

plt.show()
```

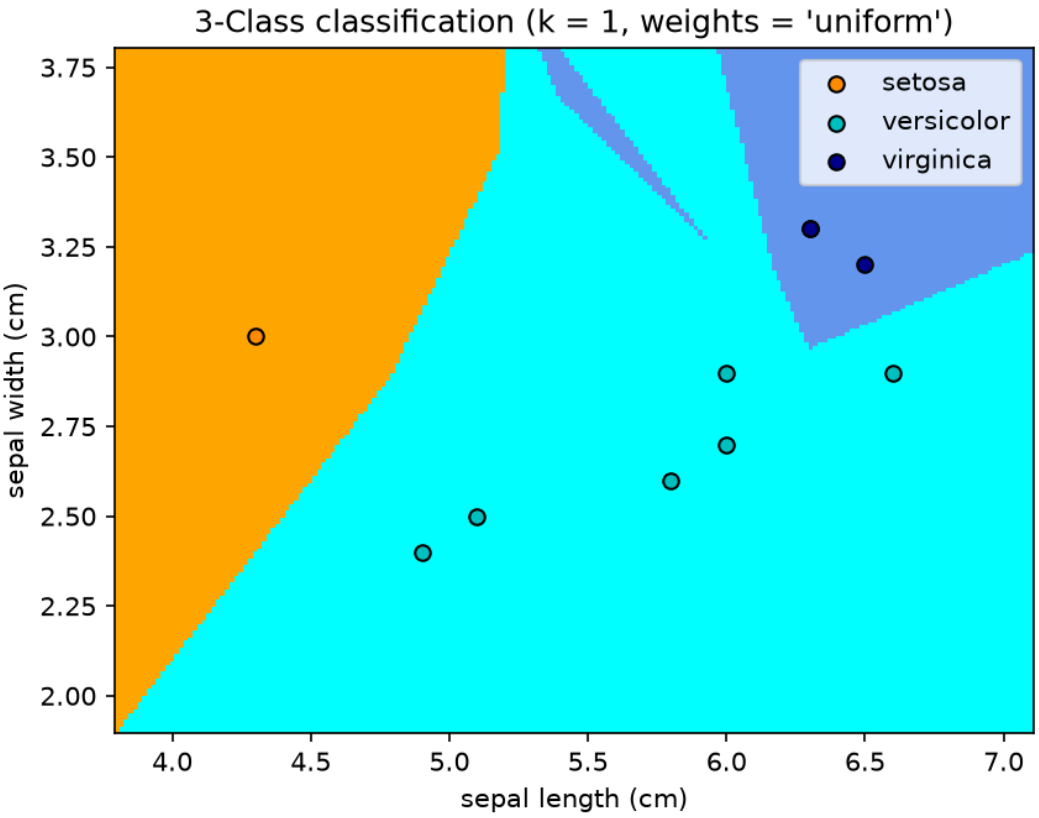


Figure 3.4: 1-nearest-neighbor decision regions on the Iris data with uniform neighbor weighting.

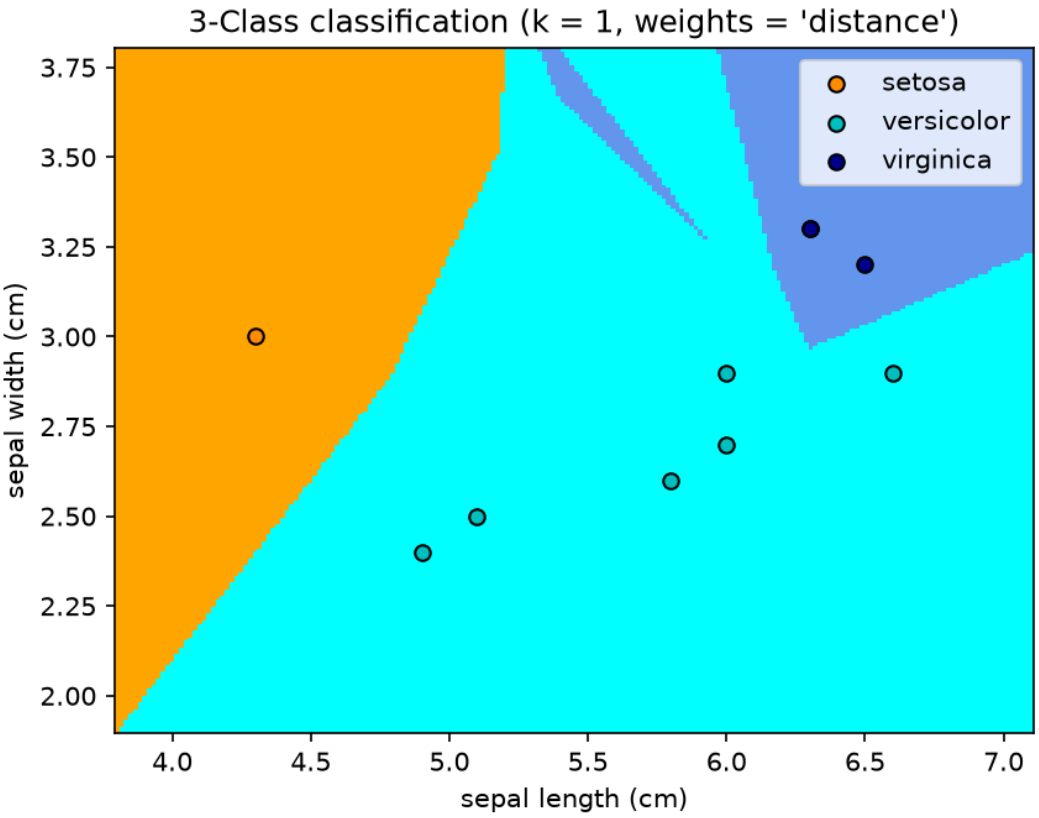


Figure 3.5: 1-nearest-neighbor decision regions on the Iris data with distance-based neighbor weighting.

CHAPTER 4

PROBABILITY THEORY

This chapter builds the probabilistic machinery the rest of the course runs on. The notational conventions it uses were fixed in Chapter 1 and are assumed throughout.

4.1 Measure-Theoretic Foundations

Sample Space (Ω): The set of all possible outcomes of a random process.

Event (A): A subset of the sample space, $A \subset \Omega$.

2^Ω is the **powerset** of Ω. Let $\mathcal{F} \subseteq 2^\Omega$ be the collection of events we are willing to assign a probability to. Then $\mathcal{F}$ is called the **event space**. Technically speaking, it is a **σ-algebra**: a collection of subsets of Ω that contains Ω itself and is closed under complementation and countable unions. That is, $\Omega \in \mathcal{F}$, and if $A_1, A_2, \cdots \in \mathcal{F}$, then:

1. $A_1^c \in \mathcal{F}$,
2. $\bigcup_{i=1}^{\infty} A_i \in \mathcal{F}$.

Closure under countable intersections, $\bigcap_{i=1}^{\infty} A_i \in \mathcal{F}$, follows from these two axioms via De Morgan's law: $\bigcap_i A_i = \left(\bigcup_i A_i^c\right)^c$.

Why we need this at all. For discrete sample spaces, such as the outcome of the roll of a pair of dice, it is safe to take $\mathcal{F} = 2^\Omega$: every subset is an event, and we can assign a probability to it. For continuous sample spaces, such as $\Omega = \mathbb{R}$, the powerset $2^{\mathbb{R}}$ is too large — it contains pathological sets to which no consistent notion of "length" (and hence probability) can be assigned. The fix is to only ever ask for the probability of sets built from intervals. The **Borel σ-algebra** $\mathcal{B}(\mathbb{R})$ is the smallest σ-algebra containing all open intervals $(a, b) \subset \mathbb{R}$; it contains every set we will ever need in practice (all open sets, closed sets, countable unions and intersections thereof) while excluding the pathological ones. When $\Omega = \mathbb{R}^d$, we use $\mathcal{F} = \mathcal{B}(\mathbb{R}^d)$, the σ-algebra generated analogously by open rectangles.

A **probability measure** is a function $P : \mathcal{F} \to [0, 1]$ such that:

1. $P(\Omega) = 1$,
2. $P(A) \geq 0$ for all $A \in \mathcal{F}$,
3. **(σ-additivity)** if $A_1, A_2, \ldots$ are pairwise disjoint events (i.e. $A_i \cap A_j = \emptyset$ for all $i \neq j$), then $P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i)$.

Probability Space: The triple $(\Omega, \mathcal{F}, P)$. This is the complete mathematical object a random experiment is modeled by. Everything else in this chapter — random variables, expectations, concentration inequalities — is built on top of it.

Intuitive meaning of probability. While P is defined axiomatically above, the frequentist intuition behind it is the long-run frequency of an event across repeated, independent trials:

$$P(A) = \lim_{n\to\infty} \frac{n_A}{n},$$

where n_A is the number of times event A occurs in n trials. This intuition is useful for building understanding, but it is not the definition; the axioms above are.

4.2 Basic Rules

The rules below follow directly from the axioms and are standard fare; we state them compactly since they will be used freely throughout the rest of the notes without re-derivation.

- **Union bound.** From the inclusion-exclusion identity $P(A \cup B) = P(A) + P(B) - P(A \cap B)$ and $P(A \cap B) \geq 0$, we get $P(A \cup B) \leq P(A) + P(B)$. By induction, for any (not necessarily disjoint) events $A_1, \ldots, A_n$,

$$P\Big(\bigcup_{i=1}^{n} A_i\Big) \leq \sum_{i=1}^{n} P(A_i).$$

  This inequality — despite its triviality — is the single most used tool in this course: it lets us bound the probability that *any one* of many bad events happens by summing the probabilities of each bad event individually.

- **Conditional probability.** $P(A \mid B) := \dfrac{P(A \cap B)}{P(B)}$ for $P(B) > 0$, read as "the probability of A given that B has occurred." Rearranged, this gives the **product rule**: $P(A \cap B) = P(A \mid B)P(B)$.

- **Independence.** Events A, B are **independent** if $P(A \mid B) = P(A)$, equivalently $P(A \cap B) = P(A)P(B)$.

- **Law of total probability.** If $B_1, \ldots, B_n$ partition Ω (pairwise disjoint, $\bigcup_i B_i = \Omega$), then $P(A) = \sum_{i=1}^{n} P(A \mid B_i)P(B_i)$.

- **Bayes' rule** (Bayes and Price, 1763). From the product rule applied both ways, $P(A \mid B)P(B) = P(A \cap B) = P(B \mid A)P(A)$, hence

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}.$$

While each of these facts is elementary on its own, Bayes' rule has outsized implications for statistical inference and machine learning. Assume $H_1, \ldots, H_n$ are mutually exclusive

hypotheses partitioning a hypothesis space $\mathcal{H}$, and $\mathcal{D}$ is observed data. Combining Bayes' rule with the law of total probability (using $H_1, \ldots, H_n$ as the partition) gives

$$P(H_i \mid \mathcal{D}) = \frac{P(\mathcal{D} \mid H_i)P(H_i)}{P(\mathcal{D})} = \frac{P(\mathcal{D} \mid H_i)P(H_i)}{\sum_{j=1}^{n} P(\mathcal{D} \mid H_j)P(H_j)}.$$

Here:

- $P(H_i)$ is our **prior belief** in hypothesis H_i,
- $P(H_i \mid \mathcal{D})$ is our **posterior belief** in H_i after observing the data $\mathcal{D}$,
- $P(\mathcal{D} \mid H_i)$ is the **likelihood** of H_i, and $P(\mathcal{D})$ is the **evidence**.

This gives a general recipe for statistical inference: start with a prior belief, collect observations, update the belief. Chapter 6 builds an entire class of learning algorithms on exactly this recipe.

4.3 Random Variables

The probability measure P maps *sets* to numbers. It is not always convenient to work directly with sets; we would rather describe outcomes numerically and delegate the set-theoretic bookkeeping to the machinery below. This is what a random variable does.

Definition 4.3.1: Random variable

Let $(\Omega, \mathcal{F}, P)$ be a probability space and $(\Lambda, \mathcal{G})$ a measurable space (e.g. $\Lambda = \mathbb{R}^k$ with its Borel σ-algebra $\mathcal{G} = \mathcal{B}(\mathbb{R}^k)$). A **random variable** is a function $X : \Omega \to \Lambda$ that is **measurable**, meaning

$$X^{-1}(A) := \{\omega \in \Omega : X(\omega) \in A\} \in \mathcal{F}, \quad \forall A \in \mathcal{G}.$$

Measurability is exactly the condition needed for the expression "$P(X \in A)$" to make sense: it guarantees that the set of outcomes ω mapping into A is itself an event, i.e. something P can be evaluated on. Given this, we define the **induced probability measure** (a.k.a. the **law** or **distribution**) of X as

$$P_X(A) := P(X^{-1}(A)) = P(\{\omega \in \Omega : X(\omega) \in A\}), \quad A \in \mathcal{G}.$$

$(\Lambda, \mathcal{G}, P_X)$ is itself a probability space. This is the sense in which a random variable "transports" a probability space we may not care about the internals of (e.g. the space of all coin-toss sequences) onto a space we do care about (e.g. $\{0, 1, \ldots, n\}$, the count of heads).

Discrete random variables. When Λ is countable, e.g. $\Lambda = \{0, 1, \ldots, C-1\}$, every subset of Λ is measurable and it suffices to specify P_X on singletons:

$$P(X = x) := P_X(\{x\}) = P(\{\omega \in \Omega : X(\omega) = x\}), \quad x \in \Lambda.$$

This is called the **probability mass function (pmf)** of X, and $P_X(A) = \sum_{x \in A} P(X = x)$ recovers the measure on any event A. Note that we write a pmf with a **capital** P, because it *is* the probability of the event $\{X = x\}$ — no new object is being introduced. Lowercase p is reserved, throughout these notes, for the densities of continuous random variables defined next, which are **not** probabilities of anything.

Continuous random variables. When $\Lambda = \mathbb{R}^k$, singletons typically carry zero probability ($P_X(\{x\}) = 0$), so a pmf is useless; instead we ask for P_X to be absolutely continuous with respect to the Lebesgue measure μ_{Leb} (ordinary k-dimensional volume). This means there exists a function $p_X \geq 0$, called the **probability density function (pdf)**, such that

$$P_X(A) = \int_A p_X(x)\, d\mu_{\text{Leb}}(x), \qquad \forall A \in \mathcal{B}(\mathbb{R}^k).$$

For $k = 1$ this is the familiar $P_X(A) = \int_A p_X(x)\, dx$. Not every random variable admits a density (some are discrete, some are neither), but every case relevant to this course is either discrete or admits one.

Example : three coin tosses

Toss a fair coin three times; write H for heads and T for tails. The sample space is $\Omega = \{HHH, HHT, HTH, HTT, THH, THT, TTH, TTT\}$, and since the coin is fair and the tosses are independent, $P(\omega) = \left(\frac{1}{2}\right)^3 = \frac{1}{8}$ for every $\omega \in \Omega$. Let $X : \Omega \to \{0, 1, 2, 3\}$ count the number of heads. Then X is a (measurable, since Ω is finite and $\mathcal{F} = 2^\Omega$) random variable with pmf

1. $P(X = 0) = P(\{TTT\}) = \frac{1}{8}$,
2. $P(X = 1) = P(\{TTH, THT, HTT\}) = \frac{3}{8}$,
3. $P(X = 2) = P(\{HTH, HHT, THH\}) = \frac{3}{8}$,
4. $P(X = 3) = P(\{HHH\}) = \frac{1}{8}$.

Random variables also let us query compound events easily, e.g.

$$P(X \geq 2) = P(\{HTH, HHT, THH, HHH\}) = \frac{1}{2}.$$

Joint distributions, marginals, independence. Given two random variables X, Y on the same probability space, their **joint distribution** is $P(X = x, Y = y) := P(\{\omega : X(\omega) = x, Y(\omega) = y\})$ (with the analogous density definition in the continuous case). We say X and Y are **independent** if $P(X = x, Y = y) = P(X = x)P(Y = y)$ for all x, y — the random-variable counterpart of event independence above.

When a variable in a joint distribution is not of direct interest, it is called a **nuisance variable**, and we can eliminate it by the law of total probability. This operation is called **marginalization**. For instance, if X is the outcome of a fair die roll and $Y \in \{\text{even}, \text{odd}\}$ its parity,

$$\begin{aligned} P(X = x) &= \sum_{y \in \{\text{even},\text{odd}\}} P(X = x, Y = y) \\ &= \sum_y P(X = x \mid Y = y) P(Y = y). \end{aligned}$$

Using $P(X = x)$ lets us query events concerning X alone, with Y's influence already averaged out proportionally to its own probability of occurrence. (As a sanity check on the definition of independence above: for a fair die, $X \le 4$ and $Y =$ even are independent — $P(X \le 4, Y{=}\text{even}) = \frac{2}{6}$ equals $P(X \le 4)P(Y{=}\text{even}) = \frac{4}{6} \cdot \frac{3}{6}$ — while $X \le 3$ and $Y =$ even are not, since low rolls are disproportionately odd.)

4.4 Expectation as an Integral

Definition 4.4.1: Expectation

The **expectation** of a random variable $X : \Omega \to \mathbb{R}$ is the Lebesgue integral of X against P:

$$\mathbb{E}[X] := \int_\Omega X(\omega)\, dP(\omega).$$

This single definition specializes to the two formulas you have likely seen before, depending on the law of X:

- if X is discrete with pmf $P(X = \cdot)$, the abstract integral reduces to a sum: $\mathbb{E}[X] = \sum_{x \in \Lambda} x\, P(X = x)$;
- if X is continuous with pdf p_X, it reduces to a Riemann-style integral: $\mathbb{E}[X] = \int_{\mathbb{R}} x\, p_X(x)\, dx$.

Framing expectation as a single integral against P, rather than as two unrelated formulas, is precisely the payoff of the measure-theoretic viewpoint: every fact we prove about $\mathbb{E}[\cdot]$ below (linearity, monotonicity) holds simultaneously for discrete and continuous random variables — and for mixtures of the two — because it is proven once, at the level of the integral, rather than twice.

In the three coin tosses example above, $\mathbb{E}[X] = 0 \cdot \frac{1}{8} + 1 \cdot \frac{3}{8} + 2 \cdot \frac{3}{8} + 3 \cdot \frac{1}{8} = 1.5$.

The expectation is also called the **mean** or **first moment** of X; it summarizes the **central tendency** of X with a single number, but says nothing about its **spread**. Spread is measured by the **variance**:

$$\mathrm{Var}[X] := \mathbb{E}\big[(X - \mathbb{E}[X])^2\big] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

In the example above, $\mathrm{Var}[X] = 0^2 \cdot \frac{1}{8} + 1^2 \cdot \frac{3}{8} + 2^2 \cdot \frac{3}{8} + 3^2 \cdot \frac{1}{8} - 1.5^2 = 0.75$. The square root of the variance is the **standard deviation**, commonly denoted σ; here $\sigma = \sqrt{0.75} \approx 0.866$.

Variance is also called the **second (central) moment** of X. In general, the **k-th central moment** is $\mathbb{E}[(X - \mathbb{E}[X])^k]$ for a positive integer k: the first is (trivially) zero and the second is the variance. The third and fourth become scale-free, and acquire their usual names, only after being standardized by the appropriate power of σ: the **skewness** $\mathbb{E}[(X - \mathbb{E}[X])^3]/\sigma^3$ measures the asymmetry of the distribution, and the **kurtosis** $\mathbb{E}[(X - \mathbb{E}[X])^4]/\sigma^4$ the heaviness of its tails.

For two jointly distributed random variables X, Y, the **covariance** quantifies how their deviations from their own means co-vary:

$$\mathrm{Cov}[X, Y] := \mathbb{E}\big[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])\big] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y].$$

Setting $Y = X$ recovers the variance, $\mathrm{Cov}[X, X] = \mathrm{Var}[X]$.

4.5 Estimators and Bias

Estimator. A function that maps a sample S to a value $\widehat{\theta}(S)$ is called an **estimator** of a parameter θ. We write the estimator and the value it produces with the same symbol $\widehat{\theta}$, suppressing the argument S when it is clear from context. The running example throughout this chapter is the **sample mean** of an i.i.d. sample $X_1, \dots, X_m \sim X$,

$$\widehat{\mu}_m := \frac{1}{m} \sum_{i=1}^{m} X_i,$$

which estimates the mean $\mu := \mathbb{E}[X]$.

Bias. The bias of an estimator is $\mathrm{Bias}(\widehat{\theta}) := \mathbb{E}[\widehat{\theta}(S)] - \theta$. An estimator is **unbiased** if $\mathrm{Bias}(\widehat{\theta}) = 0$. For example, $\widehat{\mu}_m$ is unbiased for μ:

$$\mathrm{Bias}(\widehat{\mu}_m) = \mathbb{E}[\widehat{\mu}_m] - \mu = \mathbb{E}\Big[\frac{1}{m}\sum_{i=1}^{m} X_i\Big] - \mu = \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}[X_i] - \mu = \frac{1}{m}\sum_{i=1}^{m} \mu - \mu = 0.$$

By contrast, the sample variance $\widehat{\sigma}^2_m := \frac{1}{m}\sum_{i=1}^{m}(X_i - \widehat{\mu}_m)^2$ is a *biased* estimator of σ^2; the **unbiased sample variance** $\frac{1}{m-1}\sum_{i=1}^{m}(X_i - \widehat{\mu}_m)^2$ corrects for this via **Bessel's correction**. (We reserve S for a data set throughout these notes, which is why the sample variance is not written S^2 as it often is in statistics texts.)

Consistency. An estimator is **consistent** if $\lim_{m\to\infty} P(|\widehat{\theta}(S) - \theta| \geq \epsilon) = 0$ for every $\epsilon > 0$: the probability of a large error vanishes as the sample grows. The next section builds the tools to establish (and quantify) consistency.

4.6 Concentration of Measure Inequalities

Theorem 4.6.1: Markov's inequality

Let X be a non-negative random variable. Then for any $\epsilon > 0$,

$$P(X \geq \epsilon) \leq \frac{\mathbb{E}[X]}{\epsilon}.$$

Proof. Since $X \geq 0$, the pointwise inequality $X \geq \epsilon\, \mathbb{1}(X \geq \epsilon)$ holds for every outcome: if $X < \epsilon$ the right side is $0 \leq X$, and if $X \geq \epsilon$ the right side is $\epsilon \leq X$. Taking expectations and using monotonicity and linearity of the integral,

$$\mathbb{E}[X] \geq \epsilon\, \mathbb{E}\big[\mathbb{1}(X \geq \epsilon)\big] = \epsilon\, P(X \geq \epsilon),$$

where the last equality is the defining property of the indicator: $\mathbb{E}[\mathbb{1}(A)] = P(A)$. Dividing by $\epsilon > 0$ gives the claim $\square$

Note how little the proof uses: only non-negativity and linearity. This is why Markov's inequality is the weakest of the three bounds in this section — and why every sharper bound below is obtained by first *transforming* X into a quantity to which Markov can be applied more profitably.

Example :
Consider ten Bernoulli random variables $X_1, \dots, X_{10}$ with $P(X_i = 1) = P(X_i = 0) = 0.5$. Since $\mathbb{E}[\sum_{i=1}^{10} X_i] = 5$, Markov's inequality only gives $P(\sum_{i=1}^{10} X_i \geq 3) \leq 5/3$, which exceeds 1 and is thus **vacuous**.

Theorem 4.6.2: Chebyshev's inequality

Let X have finite mean μ and variance σ^2. Then for any $\epsilon > 0$,

$$P(|X - \mu| \geq \epsilon) \leq \frac{\sigma^2}{\epsilon^2}.$$

Proof. The events $\{|X - \mu| \geq \epsilon\}$ and $\{(X - \mu)^2 \geq \epsilon^2\}$ are identical, since $u \mapsto u^2$ is increasing on $[0, \infty)$. The random variable $(X - \mu)^2$ is non-negative, so Theorem 4.1 applies to it with threshold ϵ^2:

$$P(|X - \mu| \geq \epsilon) = P\big((X - \mu)^2 \geq \epsilon^2\big) \leq \frac{\mathbb{E}[(X - \mu)^2]}{\epsilon^2} = \frac{\mathrm{Var}[X]}{\epsilon^2} = \frac{\sigma^2}{\epsilon^2}. \qquad \square$$

This is the transformation trick announced above, in its simplest form: squaring turns a two-sided deviation into a non-negative quantity, and buys a factor $1/\epsilon$ over Markov by paying with the assumption of a finite second moment. Applying Markov to e^{sX} instead of $(X - \mu)^2$ — the **Chernoff trick** — buys an *exponential* improvement, at the price of assuming a finite moment generating function; this is exactly the route Theorem 4.4 takes below.

Consider $S = \{X_1, \dots, X_m\}$ i.i.d. Bernoulli with sample average $\widehat{\mu}_m$ as defined above. How much can $\widehat{\mu}_m$ deviate from μ? Markov's inequality alone gives us

$$P(\widehat{\mu}_m - \mathbb{E}[\widehat{\mu}_m] \geq \epsilon) = P(\widehat{\mu}_m \geq \mu + \epsilon) \leq \frac{\mathbb{E}[\widehat{\mu}_m]}{\mu + \epsilon} = \frac{\mu}{\mu + \epsilon},$$

a bound that does not shrink no matter how large m is — useless for arguing that more data helps. Chebyshev's inequality does better:

$$\begin{aligned} P(|\widehat{\mu}_m - \mathbb{E}[\widehat{\mu}_m]| \geq \epsilon) &\leq \frac{\mathrm{Var}[\widehat{\mu}_m]}{\epsilon^2} = \frac{\mathrm{Var}\big[\frac{1}{m}\sum_{i=1}^{m} X_i\big]}{\epsilon^2} = \frac{1}{m^2}\frac{\sum_{i=1}^{m}\mathrm{Var}[X_i]}{\epsilon^2} \\ &= \frac{1}{m^2}\frac{m\,\mathrm{Var}[X_i]}{\epsilon^2} = \frac{\sigma^2}{m\epsilon^2}, \end{aligned}$$

using $\mathrm{Var}[\sum_i X_i] = \sum_i \mathrm{Var}[X_i]$ for independent X_i (second step) and $\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$ for a constant a (third step). This bound shrinks as $\Theta(1/m)$, giving the **weak law of large numbers**: $\square$

Theorem 4.6.3: Weak law of large numbers

Let $X_1, \dots, X_m$ be i.i.d. with finite mean μ and variance σ^2. Then for any $\epsilon > 0$, $\lim_{m\to\infty} P(|\widehat{\mu}_m - \mu| \geq \epsilon) = 0$, where $\widehat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} X_i$.

Proof. $\widehat{\mu}_m$ is unbiased, $\mathbb{E}[\widehat{\mu}_m] = \mu$, and the display preceding this theorem computed $\mathrm{Var}[\widehat{\mu}_m] = \sigma^2/m$. Chebyshev's inequality (Theorem 4.2) applied to $\widehat{\mu}_m$ therefore gives

$$0 \leq P\big(|\widehat{\mu}_m - \mu| \geq \epsilon\big) \leq \frac{\sigma^2}{m\epsilon^2} \xrightarrow[m\to\infty]{} 0$$

for every fixed $\epsilon > 0$, and the claim follows by squeezing □

Hence $\widehat{\mu}_m$ is a consistent estimator of μ. Note what the proof needed and what it did not: finiteness of σ^2 was used, but no boundedness and no distributional shape. (Dropping the finite-variance assumption still leaves the theorem true, by a truncation argument we do not need here.)

4.6.1 Hoeffding's inequality

Chebyshev's inequality gets tighter at the polynomial rate $1/m$. Boundedness lets us do much better — exponentially in m, with a bound due to (Hoeffding, 1963) that we derive below.

The device is the **Chernoff trick**: instead of applying Markov's inequality to X or to $(X-\mu)^2$, apply it to e^{sX} for a free parameter $s > 0$, then optimize over s. What makes this profitable is that the exponential turns a sum of independent variables into a *product* of expectations, and that each factor can be bounded using boundedness alone. The latter is the content of the following lemma.

Lemma 4.6.4: Hoeffding's lemma

Let X be a random variable with $\mathbb{E}[X] = 0$ and $P(a \leq X \leq b) = 1$. Then for every $s \in \mathbb{R}$,

$$\mathbb{E}\big[e^{sX}\big] \leq \exp\Big(\frac{s^2(b-a)^2}{8}\Big).$$

Proof. Note $a \leq 0 \leq b$, since $\mathbb{E}[X] = 0$. Because $u \mapsto e^{su}$ is convex, every $x \in [a,b]$, written as the convex combination $x = \lambda b + (1-\lambda)a$ with $\lambda = \frac{x-a}{b-a}$, satisfies

$$e^{sx} \leq \frac{x-a}{b-a}e^{sb} + \frac{b-x}{b-a}e^{sa}.$$

Taking expectations and using $\mathbb{E}[X] = 0$ to kill the x-dependent parts,

$$\mathbb{E}\big[e^{sX}\big] \leq \frac{-a}{b-a}e^{sb} + \frac{b}{b-a}e^{sa} = p\,e^{sb} + (1-p)e^{sa},$$
$$p := \frac{-a}{b-a} \in [0,1].$$

Substitute $u := s(b-a)$, so that $sa = -pu$, and factor out e^{sa}:

$$p\,e^{sb} + (1-p)e^{sa} = e^{-pu}\big(1 - p + p\,e^{u}\big) = e^{\varphi(u)},$$
$$\varphi(u) := -pu + \log\big(1 - p + pe^{u}\big).$$

It remains to show $\varphi(u) \le u^2/8$. Differentiating,

$$\varphi'(u) = -p + \frac{pe^{u}}{1-p+pe^{u}},$$
$$\varphi''(u) = q(u)\big(1-q(u)\big) \quad \text{with} \quad q(u) := \frac{pe^{u}}{1-p+pe^{u}} \in [0,1].$$

Hence $\varphi(0) = 0$, $\varphi'(0) = 0$, and $\varphi''(u) \le \frac{1}{4}$ everywhere, since $q(1-q) \le \frac{1}{4}$ for $q \in [0,1]$. Taylor's theorem with Lagrange remainder around 0 therefore gives $\varphi(u) = \frac{1}{2}\varphi''(\xi)u^2 \le u^2/8$ for some ξ between 0 and u. Substituting back $u = s(b-a)$ completes the proof □

Theorem 4.6.5: Hoeffding's inequality

Let $X_1, \dots, X_m$ be independent random variables such that $P(a_i \le X_i \le b_i) = 1$ for all i. Then for any $\epsilon > 0$,

$$P\Big(\frac{1}{m}\sum_{i=1}^{m} X_i - \mathbb{E}\Big[\frac{1}{m}\sum_{i=1}^{m} X_i\Big] \ge \epsilon\Big) \le \exp\Big(\frac{-2m^2\epsilon^2}{\sum_{i=1}^{m}(b_i-a_i)^2}\Big).$$

Proof. Center the variables, $Y_i := X_i - \mathbb{E}[X_i]$, so that $\mathbb{E}[Y_i] = 0$ and Y_i lies in an interval of the same width $b_i - a_i$. For any $s > 0$, the map $u \mapsto e^{su}$ is increasing, so Markov's inequality (Theorem 4.1) applied to the non-negative variable $e^{s\sum_i Y_i}$ gives

$$\begin{aligned} P\Big(\sum_{i=1}^{m} Y_i \ge m\epsilon\Big) = P\Big(e^{s\sum_i Y_i} \ge e^{sm\epsilon}\Big) &\le e^{-sm\epsilon}\,\mathbb{E}\Big[e^{s\sum_i Y_i}\Big] \\ &= e^{-sm\epsilon}\prod_{i=1}^{m}\mathbb{E}\big[e^{sY_i}\big] \\ &\le e^{-sm\epsilon}\exp\Big(\frac{s^2}{8}\sum_{i=1}^{m}(b_i-a_i)^2\Big), \end{aligned}$$

where the equality on the second line uses independence (the expectation of a product of independent variables factorizes) and the last step applies Lemma 4.1 to each Y_i separately. The bound holds for every $s > 0$, so we may take the best one. Writing $V := \sum_i (b_i - a_i)^2$, the exponent $-sm\epsilon + s^2V/8$ is a convex quadratic in s, minimized at $s^* = 4m\epsilon/V > 0$, where its value is

$$-\frac{4m^2\epsilon^2}{V} + \frac{2m^2\epsilon^2}{V} = -\frac{2m^2\epsilon^2}{V}.$$

Substituting and dividing the event by m gives the claim □

Applying Theorem 4.4 to the variables $-X_1, \dots, -X_m$ (which lie in $[-b_i, -a_i]$, of the same width) bounds the opposite tail by the same quantity:

$$P\Big(\mathbb{E}\Big[\frac{1}{m}\sum_{i=1}^{m} X_i\Big] - \frac{1}{m}\sum_{i=1}^{m} X_i \geq \epsilon\Big) \leq \exp\Big(\frac{-2m^2\epsilon^2}{\sum_{i=1}^{m}(b_i - a_i)^2}\Big),$$

and adding the two gives the two-sided version with an extra factor of 2.

We will use these inequalities to bound the generalization performance of learning algorithms. Let X_i be the loss of the i-th training example; then $\widehat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} X_i$ is the empirical risk and $\mu = \mathbb{E}[\widehat{\mu}_m]$ the expected risk. It is the lower tail that we need — the event that the true risk exceeds what training led us to believe. If $X_i \in [0, 1]$ for all i (so $b_i - a_i = 1$), it simplifies to

$$P(\mu \geq \widehat{\mu}_m + \epsilon) \leq e^{-2m\epsilon^2}.$$

Setting $\delta := e^{-2m\epsilon^2}$ and solving for ϵ gives $\epsilon = \sqrt{\log(1/\delta)/(2m)}$, hence

$$P\Big(\mu \geq \widehat{\mu}_m + \sqrt{\tfrac{\log(1/\delta)}{2m}}\Big) \leq \delta,$$
$$\text{equivalently} \quad P\Big(\mu \leq \widehat{\mu}_m + \sqrt{\tfrac{\log(1/\delta)}{2m}}\Big) \geq 1 - \delta.$$

Since we will later allocate μ to generalization performance and $\widehat{\mu}_m$ to training performance, it is this second, complementary form that we will reuse directly. Concentration inequalities of this form are called **generalization bounds**, and are the basis of statistical learning theory (Chapter 5).

4.6.2 Uniform Hoeffding bounds over several estimators

A single Hoeffding bound controls the deviation of *one* empirical average from its mean. In machine learning we rarely evaluate a single quantity: we compare many hypotheses' training errors, or track several estimators built from ranges (loss scales) that differ from one another. We now extend Theorem 4.4 to this setting, using nothing more than the union bound.

Setup. Suppose we have K estimators $\widehat{\mu}_m^{(1)}, \dots, \widehat{\mu}_m^{(K)}$, where $\widehat{\mu}_m^{(k)} := \frac{1}{m}\sum_{i=1}^{m} X_i^{(k)}$ is built from m independent observations $X_1^{(k)}, \dots, X_m^{(k)}$ (independent across i; the K estimators themselves need not be independent of each other) with each $X_i^{(k)} \in [a_k, b_k]$, i.e. the k-th estimator's underlying random variable lives in its *own* interval, of its own width $b_k - a_k$. Write $\mu^{(k)} := \mathbb{E}[\widehat{\mu}_m^{(k)}]$.

Theorem 4.6.6: Uniform Hoeffding bound

For any $\epsilon_1, \ldots, \epsilon_K > 0$,

$$P\Big(\exists k \in [K] : \mu^{(k)} - \widehat{\mu}_m^{(k)} \geq \epsilon_k\Big) \leq \sum_{k=1}^{K} \exp\Big(\frac{-2m\epsilon_k^2}{(b_k - a_k)^2}\Big).$$

In the special case $\epsilon_k \equiv \epsilon$ and $b_k - a_k \equiv (b-a)$ for all k (a common range for every estimator), this simplifies to

$$P\Big(\exists k \in [K] : \mu^{(k)} - \widehat{\mu}_m^{(k)} \geq \epsilon\Big) \leq K \exp\Big(\frac{-2m\epsilon^2}{(b-a)^2}\Big).$$

Proof. Let B_k denote the "bad" event $\{\mu^{(k)} - \widehat{\mu}_m^{(k)} \geq \epsilon_k\}$. Then

$$\begin{aligned} P\Big(\exists k \in [K] : \mu^{(k)} - \widehat{\mu}_m^{(k)} \geq \epsilon_k\Big) &= P\Big(\bigcup_{k=1}^{K} B_k\Big) \\ &\leq \sum_{k=1}^{K} P(B_k) \\ &\leq \sum_{k=1}^{K} \exp\Big(\frac{-2m\epsilon_k^2}{(b_k - a_k)^2}\Big), \end{aligned}$$

where the first inequality is the union bound and the second applies Theorem 4.4 (in its lower-tail form) to each B_k separately, using only the independence of $X_1^{(k)}, \ldots, X_m^{(k)}$ *within* estimator k — no relationship between different estimators $k \neq k'$ is required. Note that the m variables of estimator k all lie in the same interval, so $\sum_{i=1}^{m}(b_k - a_k)^2 = m(b_k - a_k)^2$, which is what turns the m^2 in Theorem 4.4 into the m above. □

What this buys us. Setting the right-hand side to a target confidence δ and solving for a common ϵ (in the equal-range case) gives $\epsilon = (b-a)\sqrt{\log(K/\delta)/(2m)}$, so that simultaneously, for every one of the K estimators,

$$P\Big(\mu^{(k)} \leq \widehat{\mu}_m^{(k)} + (b-a)\sqrt{\tfrac{\log(K/\delta)}{2m}},\ \forall k \in [K]\Big) \geq 1 - \delta.$$

Two consequences matter directly for what follows in Chapter 5:

- **Effect of the hypothesis class.** If each estimator k corresponds to the training loss of a distinct hypothesis h_k in a finite hypothesis class $\mathcal{H} = \{h_1, \ldots, h_K\}$, the bound degrades only logarithmically in $K = |\mathcal{H}|$ — a much larger hypothesis class costs comparatively little in confidence, but every additional hypothesis we simultaneously monitor must be paid for somewhere, and it is paid for here, inside the square root. This is the seed of the finite-hypothesis-class generalization bound proved in Chapter 5.

- **Effect of loss scale.** If different hypotheses (or different loss functions) produce losses on different scales $[a_k, b_k]$, the *per-estimator* range $(b_k - a_k)$ enters the bound directly and linearly. A hypothesis whose loss can swing across a wide range is intrinsically harder to certify than one confined to a narrow range, even before we ask anything about the hypothesis class's size or structure — boundedness of the *loss*, not just of the sample, is doing real work.

CHAPTER 5

STATISTICAL LEARNING THEORY

Solving hard real-world problems with machine learning demands advanced algorithmic designs. These designs should be based on a solid understanding of the principles that determine the behavior of learning algorithms. Statistical learning theory provides mathematical tools to develop such an understanding. It seeks answers to fundamental questions about the nature of learning from data such as

- Under which conditions can a learning algorithm generalize from the training data to new data?
- How can we measure the complexity of a learning task?
- How can we control the complexity of a learning algorithm?
- How can we estimate the generalization performance of a learning algorithm?

Every generalization bound in this course, no matter how it is derived, has the same shape: with high probability, the gap $R(h) - \widehat{R}_S(h)$ between true and empirical risk, uniformly over $h \in \mathcal{H}$, is controlled by a single number that measures **how complex $\mathcal{H}$ is**. What changes between different theories of generalization is only *how that number is defined* — as a raw count ($|\mathcal{H}|$, this chapter), a combinatorial shattering capacity ($d_{VC}(\mathcal{H})$, this chapter), an average best-case correlation with random noise (**Rademacher complexity**), a description length in a fixed metric (**covering numbers**), or a KL-divergence budget between a data-dependent posterior and a fixed prior over hypotheses (**PAC-Bayes bounds**) — the last three being the subject of Chapter 5a. This chapter builds the foundational machinery — the finite-hypothesis-class bound, PAC learnability, the No Free Lunch theorem, VC dimension, and the Fundamental Theorem of Statistical Learning — using the coarsest and most concrete of these complexity measures, $|\mathcal{H}|$ and $d_{VC}(\mathcal{H})$. Chapter 5a then shows that these are two special cases of one abstract "complexity of a hypothesis class" idea, of which Rademacher complexity, covering numbers, and PAC-Bayes bounds are three more.

Based on what we covered in the probability theory chapter, let us start from investigating the last of the four questions above.

5.1 Generalization Bounds

Remember the basic definitions below.

Definition 5.1.1: Generalization error

Given a hypothesis $h \in \mathcal{H}$, a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to [0, 1]$, and a data distribution $(x, y) \sim \mathcal{D}$, the generalization error of h is defined as

$$R(h) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(y, h(x))].$$

Definition 5.1.2: Empirical error

Given a hypothesis $h \in \mathcal{H}$, a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to [0, 1]$, and a data set $S = \{(x_1, y_1), ..., (x_m, y_m)\}$, the empirical error of h is defined as

$$\widehat{R}_S(h) = \frac{1}{m}\sum_{i=1}^{m} \ell(y_i, h(x_i)).$$

Intuitively, $R(h)$ is the error we actually care about but can never observe directly, since it averages over the whole (unknown) data distribution $\mathcal{D}$; $\widehat{R}_S(h)$ is the only thing we can compute, since it averages over the finite sample S we happen to have. The rest of this chapter is about how well the second quantity stands in for the first.

Using the definitions above and the concentration inequalities we covered in the probability theory chapter, we can derive the following theorem.

Theorem 5.1.3: Generalization bound

Let $\mathcal{H}$ be a finite hypothesis set. Then for any $\delta \in (0, 1)$, with probability at least $1-\delta$ over the choice of $S \sim \mathcal{D}^m$, for all $h \in \mathcal{H}$

$$R(h) \leq \widehat{R}_S(h) + \sqrt{\frac{\log|\mathcal{H}| + \log\frac{1}{\delta}}{2m}}.$$

Proof. Let $\mathcal{H} = \{h_1, ..., h_{|\mathcal{H}|}\}$. Then we have

$$\begin{aligned} P_{S\sim\mathcal{D}^m}\left(\exists h_i \in \mathcal{H} : R(h_i) - \widehat{R}_S(h_i) > \epsilon\right) &= P_{S\sim\mathcal{D}^m}\left(\bigcup_{i=1}^{|\mathcal{H}|}\left\{R(h_i) - \widehat{R}_S(h_i) > \epsilon\right\}\right) \\ &\leq \sum_{i=1}^{|\mathcal{H}|} P_{S\sim\mathcal{D}^m}\left(R(h_i) - \widehat{R}_S(h_i) > \epsilon\right) \\ &\leq |\mathcal{H}|\exp(-2m\epsilon^2). \end{aligned}$$

where the first inequality follows from the union bound and the second from Hoeffding's inequality (Theorem 4.4) applied to each h_i separately, with the loss bounded in $[0, 1]$. Setting the right hand side to δ and solving for ϵ yields the final result □

This inequality is called a generalization bound, because it bounds the generalization error of any hypothesis $h \in \mathcal{H}$ in terms of its empirical error. We can derive a number of interesting consequences from this bound:

- The bound is uniform in the sense that it holds simultaneously for all hypotheses in $\mathcal{H}$.
- The bound is independent of the data distribution $\mathcal{D}$ and the loss function ℓ.
- The bound increases logarithmically with the size of the hypothesis set $|\mathcal{H}|$, that is, a richer hypothesis set is more likely to overfit.
- The bound decreases with the size of the training set m, that is, more data is less likely to overfit.
- For a fixed training set size m and two different hypothesis sets $\mathcal{H}_1$ and $\mathcal{H}_2$, the bound prefers the smaller hypothesis set. This is known as the **Occam's razor principle**. The principle is introduced by the 14th century theologian William of Ockham. It states that among competing hypotheses, the one with the fewest assumptions should be selected.
- The bound gives a **Probably Approximately Correct (PAC)** performance guarantee. The event that any hypothesis in $\mathcal{H}$ is **approximately correct** in the sense that its generalization error is at most ϵ with probability at least $1 - \delta$.

5.2 PAC Learnability

The formal notion below is due to (Valiant, 1984), who founded the **Probably Approximately Correct (PAC)** framework this whole chapter is built on.

Definition 5.2.1: Realizability

A hypothesis space $\mathcal{H}$ is realizable with respect to a loss ℓ and data distribution $\mathcal{D}$ if $\exists h^* \in \mathcal{H}$ such that $R(h^*) = 0$.

In words, realizability says the target concept itself is expressible inside $\mathcal{H}$, with no approximation error to worry about — the only thing standing between the learner and a perfect predictor is a finite sample. It trivially follows from this definition that $\min_{h \in \mathcal{H}} \widehat{R}_S(h) = 0$ with probability 1 for a sample set S collected i.i.d. from $\mathcal{D}$. Hence, under the realizability assumption, all ERM solutions $h_S \in \arg\min_{h \in \mathcal{H}} \widehat{R}_S(h)$ give zero error.

Definition 5.2.2: Representativeness

A training set S is called $\epsilon-$ representative if

$$\forall h \in \mathcal{H}, |R(h) - \widehat{R}_S(h)| \leq \epsilon.$$

In words, on an ϵ-representative sample, training error is a faithful proxy for true error, simultaneously for *every* hypothesis in $\mathcal{H}$ — not just the one a particular algorithm happens to output.

Lemma 5.2.3

Assume that a training set S is $\frac{\epsilon}{2}$-representative, then any ERM solution $h_S \in \arg\min_{h\in\mathcal{H}} \widehat{R}_S(h)$ satisfies

$$R(h_S) \leq \min_{h\in\mathcal{H}} R(h) + \epsilon.$$

Proof. For every $h \in \mathcal{H}$, $R(h_S) \leq \widehat{R}_S(h_S) + \frac{\epsilon}{2} \leq \widehat{R}_S(h) + \frac{\epsilon}{2} \leq R(h) + \frac{\epsilon}{2} + \frac{\epsilon}{2} = R(h) + \epsilon$, where the first and last steps use $\frac{\epsilon}{2}$-representativeness and the middle step uses that h_S minimizes $\widehat{R}_S$. Taking the minimum over $h \in \mathcal{H}$ on the right gives the claim □

In words: on a representative sample, ERM cannot be fooled — the hypothesis it picks by minimizing *empirical* risk is also near-optimal in *true* risk, because empirical and true risk cannot disagree by more than $\epsilon/2$ for any hypothesis, including both h_S and the true best one.

Definition 5.2.4: Uniform Convergence

A hypothesis class $\mathcal{H}$ has the **uniform convergence property** if there exists a function $m_{\mathcal{H}}^{\mathrm{UC}} : (0,1)^2 \to \mathbb{N}$ such that for every $\epsilon, \delta \in (0,1)$ and every distribution $\mathcal{D}$, any sampled data set $S = \{(x_i, y_i) \overset{i.i.d.}{\sim} D : i = 1, \ldots, m\}$ with $m \geq m_{\mathcal{H}}^{\mathrm{UC}}(\epsilon, \delta)$ is ϵ-representative with probability at least $1 - \delta$.

In words, $\mathcal{H}$ has uniform convergence if, given enough samples, a random data set is (with high probability) ϵ-representative — so that empirical risk minimization is safe to use on it, no matter which distribution $\mathcal{D}$ generated the data.

Definition 5.2.5: Agnostic PAC Learnability

A hypothesis class $\mathcal{H}$ is **agnostic PAC learnable** if there exist a function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a **learning algorithm** A such that for every $\epsilon, \delta \in (0,1)$ and every distribution $\mathcal{D}$, running the learning algorithm on data set $S = \{(x_i, y_i) \overset{i.i.d.}{\sim} D : i = 1, \ldots, m\}$ with $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ satisfies

$$P\big(\{S \sim \mathcal{D}^m : R(A(S)) \leq \min_{h'\in\mathcal{H}} R(h') + \epsilon\}\big) \geq 1 - \delta.$$

In words, an agnostic PAC learner is guaranteed, given enough data, to output a hypothesis nearly as good as the best one $\mathcal{H}$ has to offer — for *any* data distribution $\mathcal{D}$, and without assuming $\mathcal{H}$ contains a perfect predictor (hence "agnostic").

Corollary 5.2.6

If $|\mathcal{H}| < \infty$ (finite hypothesis class), then it has the uniform convergence property with sample complexity

$$m_{\mathcal{H}}^{\mathrm{UC}}(\epsilon, \delta) \leq \left\lceil \frac{\log(2|\mathcal{H}|/\delta)}{2\epsilon^2} \right\rceil.$$

and is agnostically PAC learnable using the ERM algorithm with sample complexity

$$m_{\mathcal{H}}(\epsilon, \delta) \leq m_{\mathcal{H}}^{\mathrm{UC}}(\epsilon/2, \delta) \leq \left\lceil \frac{2\log(2|\mathcal{H}|/\delta)}{\epsilon^2} \right\rceil.$$

Proof. Definition 5.4 asks for a *two-sided* bound, so run the proof of Theorem 5.1 over the $2|\mathcal{H}|$ bad events $\{R(h_i) - \widehat{R}_S(h_i) > \epsilon\}$ and $\{\widehat{R}_S(h_i) - R(h_i) > \epsilon\}$, $i = 1, \ldots, |\mathcal{H}|$. The union bound and Hoeffding's inequality (Theorem 4.4, applied to each tail separately) give

$$P\Big(\exists h \in \mathcal{H} : |R(h) - \widehat{R}_S(h)| > \epsilon\Big) \leq 2|\mathcal{H}|\, e^{-2m\epsilon^2}.$$

Requiring the right-hand side to be at most δ and solving for m yields $m \geq \log(2|\mathcal{H}|/\delta)/(2\epsilon^2)$, i.e. any such m makes S ϵ-representative with probability at least $1-\delta$: this is the first claim. For the second, Lemma 5.1 turns $\epsilon/2$-representativeness into the agnostic PAC guarantee for ERM, so it suffices to take $m \geq m_{\mathcal{H}}^{\mathrm{UC}}(\epsilon/2, \delta) = \log(2|\mathcal{H}|/\delta)/(2(\epsilon/2)^2) = 2\log(2|\mathcal{H}|/\delta)/\epsilon^2$ □

In short: finiteness of $\mathcal{H}$ alone is already enough to guarantee learnability, with the required sample size growing only *logarithmically* in $|\mathcal{H}|$ — doubling the hypothesis class costs only one extra bit of sample complexity.

Definition 5.2.7: Bayes Error

Given a data distribution $(x, y) \sim \mathcal{D}$, the **Bayes error** is the risk of the best possible predictor over *all* measurable functions $\mathcal{X} \to \mathcal{Y}$ — not merely the best one inside some hypothesis class $\mathcal{H}$:

$$R_{\mathcal{D}}^{*} := \inf_{h' : \mathcal{X} \to \mathcal{Y}} R(h').$$

A predictor h^* that achieves the Bayes error is called a **Bayes (optimal) predictor**. For binary classification with $\mathcal{Y} = \{0, 1\}$, the Bayes predictor is

$$h^*(x) = \arg\max_{y \in \{0,1\}} P(y \mid x),$$

with Bayes error obtained by averaging the pointwise error over the input distribution,

$$R_{\mathcal{D}}^{*} = \mathbb{E}_x\Big[1 - \max_{y \in \{0,1\}} P(y \mid x)\Big] = \mathbb{E}_x\Big[\min_{y \in \{0,1\}} P(y \mid x)\Big];$$

this floor is also called the **noise** level of $\mathcal{D}$, since it is the part of the risk that no predictor — however expressive — can remove. It is zero exactly when the label is a deterministic function of the input.

The Bayes error $R^*_{\mathcal{D}}$ is a property of the distribution $\mathcal{D}$ alone. It should not be confused with $\min_{h\in\mathcal{H}} R(h)$, the best risk achievable *within a fixed, possibly restricted* hypothesis class $\mathcal{H}$ — which we call the **approximation error** of $\mathcal{H}$ below. In general $\min_{h\in\mathcal{H}} R(h) \geq R^*_{\mathcal{D}}$, with equality only when $\mathcal{H}$ happens to contain a Bayes predictor for $\mathcal{D}$.

Definition 5.2.8: "Realizable" PAC learnability

A hypothesis class $\mathcal{H}$ is **PAC learnable** if there exist a function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm A such that for every $\epsilon, \delta \in (0,1)$ and every distribution $\mathcal{D}$ satisfying the realizability assumption (Definition 5.3), running A on $S = \{(x_i, y_i) \overset{i.i.d.}{\sim} \mathcal{D} : i = 1, \ldots, m\}$ with $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ satisfies

$$P\big(\{S \sim \mathcal{D} : R(A(S)) \leq \epsilon\}\big) \geq 1 - \delta.$$

Realizability forces $\min_{h\in\mathcal{H}} R(h) = 0$, and in particular $R^*_{\mathcal{D}} = 0$. Definition 5.8 is therefore a *weaker demand on the learner* than Definition 5.6: it is only required to succeed on the restricted family of distributions for which some hypothesis in $\mathcal{H}$ is perfect, whereas an agnostic learner must succeed on **every** distribution. Consequently, agnostic PAC learnability trivially implies PAC learnability — simply apply Definition 5.6 to a realizable $\mathcal{D}$, where $\min_{h'\in\mathcal{H}} R(h') = 0$ makes the two guarantees read identically.

What is far from trivial is that, for binary classification under the zero-one loss, the converse holds as well: the two notions turn out to be *equivalent*, and both are equivalent to $\mathcal{H}$ having a finite VC dimension. This is the content of the Fundamental Theorem of Statistical Learning (Theorem 5.4) below. The equivalence is specific to this setting; for general label spaces and loss functions the realizable and agnostic notions genuinely differ, and the sample complexities differ even in the binary case — $\Theta(1/\epsilon)$ in the realizable case against $\Theta(1/\epsilon^2)$ in the agnostic case, as Theorem 5.4 makes precise.

5.3 Bias-Complexity Dilemma (Trade-off)

The result below is in the spirit of (Wolpert, 1996).

Theorem 5.3.1: No free lunch

Let A be any learning algorithm for the task of binary classification with respect to the 0/1 loss over a domain $\mathcal{X}$ and $m < |\mathcal{X}|/2$ be a positive integer representing a training set size. Then, there exists a distribution $\mathcal{D}$ over $\mathcal{X} \times \{0,1\}$ such that:

- There exists a function $f : \mathcal{X} \to \{0,1\}$ with $R(f) = 0$.
- With probability of at least $1/7$ over the choice of $S \sim \mathcal{D}^m$ we have that $R(A(S)) \geq 1/8$.

Proof. Fix any $C \subseteq \mathcal{X}$ with $|C| = 2m$; this is possible since $m < |\mathcal{X}|/2$. There are exactly $T := 2^{2m}$ functions $f_1, \dots, f_T : C \to \{0,1\}$. For each i, let $\mathcal{D}_i$ be the distribution on $\mathcal{X} \times \{0,1\}$ that draws x uniformly from C and sets $y = f_i(x)$ deterministically. By construction $R_{\mathcal{D}_i}(f_i) = 0$, so the first bullet holds for every $\mathcal{D}_i$. It remains to exhibit one i for which A fails.

Step 1: reduce to an average. We show

$$\max_{i \in [T]} \mathbb{E}_{S \sim \mathcal{D}_i^m}\big[R_{\mathcal{D}_i}(A(S))\big] \;\geq\; \tfrac{1}{4}.$$

Enumerate the $k := (2m)^m$ possible input sequences $S_1, \dots, S_k$ drawable from C, and write S_j^i for the sequence S_j labeled by f_i. Since drawing $S \sim \mathcal{D}_i^m$ is exactly drawing one of the S_j uniformly and labeling it by f_i,

$$\mathbb{E}_{S \sim \mathcal{D}_i^m}\big[R_{\mathcal{D}_i}(A(S))\big] = \frac{1}{k}\sum_{j=1}^{k} R_{\mathcal{D}_i}\big(A(S_j^i)\big).$$

A maximum is at least an average, and averages may be exchanged, so

$$\begin{aligned}\max_i \frac{1}{k}\sum_j R_{\mathcal{D}_i}(A(S_j^i)) \;&\geq\; \frac{1}{T}\sum_i \frac{1}{k}\sum_j R_{\mathcal{D}_i}(A(S_j^i)) \\ &= \frac{1}{k}\sum_j \frac{1}{T}\sum_i R_{\mathcal{D}_i}(A(S_j^i)) \;\geq\; \min_j \frac{1}{T}\sum_i R_{\mathcal{D}_i}(A(S_j^i)).\end{aligned}$$

Step 2: bound the inner average for a fixed sequence. Fix j, write $S_j = (x_1, \dots, x_m)$, and let $v_1, \dots, v_p$ be the elements of C that **do not** appear in S_j. Since S_j contains at most m distinct elements and $|C| = 2m$, we have $p \geq m$. For any hypothesis h,

$$R_{\mathcal{D}_i}(h) = \frac{1}{2m}\sum_{x \in C} \mathbb{1}\big(h(x) \neq f_i(x)\big) \;\geq\; \frac{1}{2m}\sum_{r=1}^{p} \mathbb{1}\big(h(v_r) \neq f_i(v_r)\big) \;\geq\; \frac{1}{2p}\sum_{r=1}^{p} \mathbb{1}\big(h(v_r) \neq f_i(v_r)\big),$$

the last step using $p \geq m$. Averaging over i and exchanging the two finite averages,

$$\frac{1}{T}\sum_i R_{\mathcal{D}_i}(A(S_j^i)) \;\geq\; \frac{1}{2}\cdot\frac{1}{p}\sum_{r=1}^{p} \underbrace{\frac{1}{T}\sum_i \mathbb{1}\big(A(S_j^i)(v_r) \neq f_i(v_r)\big)}_{=:\,\Pi_r} \;\geq\; \frac{1}{2}\min_r \Pi_r.$$

Step 3: the unseen points are a coin flip. Fix r and pair up the T functions: partition $[T]$ into $T/2$ pairs (i, i') such that f_i and $f_{i'}$ agree everywhere on C **except** at v_r. Since v_r does not occur in S_j, the two labeled sequences coincide, $S_j^i = S_j^{i'}$, so A returns the *same* hypothesis h on both. But $f_i(v_r) \neq f_{i'}(v_r)$, so h disagrees with exactly one of them at v_r. Each pair therefore contributes exactly one to the sum, giving $\Pi_r = \frac{1}{2}$ for every r. Chaining Steps 1-3 gives $\max_i \mathbb{E}_{S \sim \mathcal{D}_i^m}[R_{\mathcal{D}_i}(A(S))] \geq \frac{1}{2}\cdot\frac{1}{2} = \frac{1}{4}$.

Step 4: from expectation to probability. Let $\mathcal{D} := \mathcal{D}_{i^*}$ for a maximizing i^* and let $\theta := R_{\mathcal{D}}(A(S)) \in [0,1]$, a random variable of the draw of S, with $\mathbb{E}[\theta] \geq \frac{1}{4}$. Apply Markov's inequality (Theorem 4.1) to the non-negative variable $1 - \theta$ with threshold $\frac{7}{8}$:

$$P\big(\theta < \tfrac{1}{8}\big) = P\big(1 - \theta > \tfrac{7}{8}\big) \leq \frac{\mathbb{E}[1-\theta]}{7/8} \leq \frac{3/4}{7/8} = \frac{6}{7},$$

hence $P(\theta \geq \frac{1}{8}) \geq \frac{1}{7}$ □

The proof is worth re-reading for what it does *not* assume: nothing about A beyond it being a function of the sample. Step 3 is the crux — on the at least m points of C the learner never saw, the label is, by construction of the family $\{f_i\}$, an independent fair coin flip that no amount of cleverness can predict.

Corollary 5.3.2

Let $\mathcal{X}$ be an infinite domain ($|\mathcal{X}| = \infty$, e.g. a real-valued feature space) and $\mathcal{H}$ be the set of all functions from $\mathcal{X}$ to $\{0, 1\}$. Then $\mathcal{H}$ is not PAC learnable.

Proof (Shalev-Shwartz book). Assume, by way of contradiction, that the class is learnable. Choose some $\epsilon < 1/8$ and $\delta < 1/7$. By the definition of PAC learnability, there must be some learning algorithm A and an integer $m = m(\epsilon, \delta)$, such that for any data-generating distribution over $\mathcal{X} \times \{0, 1\}$, if for some function $f : \mathcal{X} \to \{0, 1\}$, $R(f) = 0$, then with probability greater than $1 - \delta$ when A is applied to samples S of size m, generated i.i.d. by $\mathcal{D}$, $R(A(S)) \leq \epsilon$. However, applying the No-Free-Lunch theorem, since $|\mathcal{X}| > 2m$, for every learning algorithm (and in particular for the algorithm A), there exists a distribution $\mathcal{D}$ such that with probability greater than $1/7 > \delta$, $R(A(S)) > 1/8 > \epsilon$, which leads to the desired contradiction □

The take-home message of the above theorem and its corollary is that there is no universal learner that works well for all data distributions. There always exists a task for which the learner fails. This is a very important result. It tells us that we need to make assumptions about the data distribution in order to design a good learner. These assumptions will constitute the **inductive bias** of the learner. The no free lunch theorem tells us only that we need to induce a degree of bias to the learner. However, it does not tell anything about its consequences. Inducing too much bias limits the ability of the learner to explain the training observations. Inducing too little bias leads to overfitting. The goal is to find the right balance between the two. This dilemma is known as the **bias-complexity dilemma**.

5.3.1 Decomposing the Excess Risk

Let us describe the bias-complexity dilemma in more formal terms. Let $h_S \leftarrow A(S)$ be the hypothesis a learning algorithm returns on a sample S — for instance an ERM solution $h_S \in \arg\min_{h \in \mathcal{H}} \widehat{R}_S(h)$. Definition 5.7 already identified an irreducible floor $R^*_{\mathcal{D}}$ that no predictor can undercut. Subtracting it, and adding and subtracting the best risk achievable inside $\mathcal{H}$, splits the generalization error into three terms:

$$\underbrace{R(h_S)}_{\text{generalization error}} = \underbrace{R^*_{\mathcal{D}}}_{\text{Bayes error (noise)}} + \underbrace{\min_{h \in \mathcal{H}} R(h) - R^*_{\mathcal{D}}}_{\epsilon_{app}\ \text{(approximation error)}} + \underbrace{R(h_S) - \min_{h \in \mathcal{H}} R(h)}_{\epsilon_{est}\ \text{(estimation error)}}.$$

Each term is controlled by a different thing, and only the last two are ours to influence:

- The **Bayes error** $R^*_{\mathcal{D}}$ is a property of $\mathcal{D}$ alone (Definition 5.7). No choice of $\mathcal{H}$, algorithm, or sample size touches it.
- The **approximation error** ϵ_{app} measures how badly $\mathcal{H}$ fails to contain a Bayes predictor. It is a property of $\mathcal{H}$ and $\mathcal{D}$, and does *not* depend on S: enlarging the class can only reduce it, since $\mathcal{H}' \supset \mathcal{H}$ implies $\min_{h' \in \mathcal{H}'} R(h') \leq \min_{h \in \mathcal{H}} R(h)$.
- The **estimation error** ϵ_{est} measures how much worse the hypothesis we actually found is than the best one available in $\mathcal{H}$. This is the only term the generalization bounds of this chapter control: representativeness (Definition 5.4) with parameter $\epsilon/2$ gives $\epsilon_{est} \leq \epsilon$ by Lemma 5.1, and Theorem 5.1 bounds it by a quantity growing in $|\mathcal{H}|$ and shrinking in m.

Herein lies the dilemma. Enlarging $\mathcal{H}$ drives ϵ_{app} down but ϵ_{est} up; shrinking it does the reverse. Since we only ever observe $\widehat{R}_S(h_S)$, and a richer class drives *that* towards zero regardless of what happens to $R(h_S)$, an unconstrained search inflates ϵ_{est} invisibly — which is exactly overfitting. This is the **bias-complexity dilemma**, and the whole of Chapters 5 and 5a is machinery for pricing ϵ_{est} so that the trade-off can be made deliberately rather than accidentally.

5.3.2 Bias-Variance Decomposition

For the squared loss, the same excess risk admits a second, more classical split. Consider a regression problem with observations $(x, y) \sim \mathcal{D}$, and let $h_S \in \mathcal{H}$ be fitted by minimizing the mean squared error. For a fixed test point x, the expected squared error over both the noisy label y and the training sample S is

$$\begin{aligned}
\mathbb{E}_{S,y|x}\Big[\Big(y - h_S(x)\Big)^2\Big] &= \mathbb{E}_{S,y|x}\Big[y^2 - 2y h_S(x) + h_S(x)^2\Big] \\
&= \mathbb{E}_{y|x}[y^2] + \mathbb{E}_S[h_S(x)^2] - 2\mathbb{E}_{y|x}[y]\mathbb{E}_S[h_S(x)] \\
&= \mathbb{E}_{y|x}[y]^2 + \mathrm{Var}_{y|x}[y] + \mathbb{E}_S[h_S(x)]^2 + \mathrm{Var}_S[h_S(x)] - 2\mathbb{E}_{y|x}[y]\mathbb{E}_S[h_S(x)] \\
&= \Big(\underbrace{\mathbb{E}_{y|x}[y] - \mathbb{E}_S[h_S(x)]}_{\text{Estimator Bias}}\Big)^2 + \underbrace{\mathrm{Var}_S[h_S(x)]}_{\text{Estimator Variance}} + \underbrace{\mathrm{Var}_{y|x}[y]}_{\text{Label noise variance}}
\end{aligned}$$

The second line uses that y and S are independent given x, and we note that $\mathbb{E}_{S|x}[h_S(x)] = \mathbb{E}_S[h_S(x)]$ and $\mathrm{Var}_{S|x}[h_S(x)] = \mathrm{Var}_S[h_S(x)]$ since S is collected independently of the test point x.

> ***Example :***
> Take the k-nearest neighbours regressor,
>
> $$h_S(x) = \frac{1}{k}\sum_{i=1}^{k} y_{(i)},$$
>
> where $(x_{(1)}, y_{(1)}), \ldots, (x_{(k)}, y_{(k)})$ are the k training points nearest to x. Writing $f^*(x) := \mathbb{E}[y|x]$ and assuming label noise of variance σ^2, the three terms read

$$\mathbb{E}_{S,y|x}\Big[\Big(y - h_S(x)\Big)^2\Big] = \left(f^*(x) - \mathbb{E}_S\Big[\frac{1}{k}\sum_{i=1}^{k} f^*(x_{(i)})\Big]\right)^2 + \frac{\sigma^2}{k} + \sigma^2.$$

The middle term follows because averaging k independent noisy labels divides the noise variance by k. The two extremes are instructive:

- $k = m$: every training point is a "neighbour", so h_S predicts the global sample mean of the labels regardless of x. The variance term is at its smallest (σ^2/m), but the bias is large, since a constant cannot track how x influences y. The model **underfits**.
- $k = 1$: h_S predicts the label of the single nearest training point to x. The bias term is small (for dense enough data, $f^*(x_{(1)}) \approx f^*(x)$), but the variance is at its largest (σ^2), since the prediction rides on one noisy label. The model **overfits**: it incurs no error on S, while its test error depends entirely on how representative that particular S was.

5.3.3 The Two Decompositions Are the Same Decomposition

The two splits above look unrelated: one is stated in terms of risks of hypotheses, the other in terms of moments of a predicted value at a point. For the squared loss they are in fact two ways of cutting up one and the same quantity. Let $f^*(x) := \mathbb{E}[y|x]$ denote the Bayes predictor for the squared loss, write $\|g\|^2 := \mathbb{E}_x[g(x)^2]$ for the $L_2(\mathcal{D})$ norm on the input space, and let

$$\bar{h}(x) := \mathbb{E}_S[h_S(x)]$$

be the **average predictor**: the hypothesis we would obtain by averaging the outputs of our learning algorithm over all training samples of size m.

Step 1 (excess risk is a squared distance). Conditioning on x and expanding $(y - h(x))^2 = \big((y - f^*(x)) + (f^*(x) - h(x))\big)^2$, the cross term vanishes because $\mathbb{E}[y - f^*(x) \mid x] = 0$, leaving

$$R(h) = \|h - f^*\|^2 + \underbrace{\mathbb{E}_x\big[\mathrm{Var}[y|x]\big]}_{= R^*_{\mathcal{D}}},$$
$$\text{hence} \quad R(h) - R^*_{\mathcal{D}} = \|h - f^*\|^2.$$

So for the squared loss, excess risk over the Bayes error *is* squared distance to the Bayes predictor, and the label-noise term of the bias-variance decomposition is exactly the Bayes error of Definition 5.7.

Step 2 (average over samples). Taking $\mathbb{E}_S$ of Step 1 and splitting $h_S - f^* = (h_S - \bar{h}) + (\bar{h} - f^*)$, whose cross term vanishes by the definition of $\bar{h}$,

$$\mathbb{E}_S\big[R(h_S)\big] - R^*_{\mathcal{D}} = \underbrace{\|\bar{h} - f^*\|^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}_x\big[\mathrm{Var}_S[h_S(x)]\big]}_{\text{Variance}},$$

which is the bias-variance decomposition of the previous section, integrated over the test point x.

Step 3 (the identity). The very same left-hand side is, by the approximation-estimation decomposition,

$$\mathbb{E}_S\big[R(h_S)\big] - R^*_{\mathcal{D}} = \epsilon_{app} + \mathbb{E}_S[\epsilon_{est}].$$

Comparing Steps 2 and 3 gives the conceptual bridge we were after:

$$\boxed{\text{Bias}^2 + \text{Variance} \;=\; \epsilon_{app} + \mathbb{E}_S[\epsilon_{est}] \;=\; \mathbb{E}_S[R(h_S)] - R^*_{\mathcal{D}}}$$

Both decompositions carve up the *expected excess risk over the Bayes error*. They agree on the total and on the noise floor; they differ only in where they draw the internal line.

Where the line differs. Suppose $\mathcal{H}$ is convex, so that the average predictor $\bar{h}$ is itself a member of $\mathcal{H}$. Then

$$\text{Bias}^2 = \|\bar{h} - f^*\|^2 \;\geq\; \min_{h\in\mathcal{H}} \|h - f^*\|^2 = \epsilon_{app},$$
$$\text{and consequently} \quad \text{Variance} \;\leq\; \mathbb{E}_S[\epsilon_{est}].$$

In words: the bias term is the approximation error *plus* whatever part of the estimation error is systematic — the offset that survives averaging over samples. The variance term is the remaining, purely fluctuating part. The correspondence between the two vocabularies is therefore

plays the role of	approximation-estimation	bias-variance (squared loss)
irreducible noise	$R^*_{\mathcal{D}}$	$\mathbb{E}_x[\text{Var}[y \mid x]]$
class too restricted	ϵ_{app}	Bias^2 (which is $\geq \epsilon_{app}$)
finite sample	$\mathbb{E}_S[\epsilon_{est}]$	Variance (which is $\leq \mathbb{E}_S[\epsilon_{est}]$)
grows when $\mathcal{H}$ grows	ϵ_{est}	Variance
shrinks when $\mathcal{H}$ grows	ϵ_{app}	Bias^2

Two caveats, both instructive.

- *Convexity matters.* If $\mathcal{H}$ is not convex, $\bar{h}$ need not lie in $\mathcal{H}$, and it can then be strictly closer to f^* than any single member of $\mathcal{H}$ — so $\text{Bias}^2 < \epsilon_{app}$ becomes possible. This is not a pathology to be avoided but a mechanism to be exploited: it is precisely why averaging many predictors trained on resampled data can beat every individual predictor, which is the theoretical content of **bagging** in Chapter 10.
- *The exact identity is specific to the squared loss.* Step 1 relied on the squared loss to turn excess risk into a squared distance, and Step 2 relied on that distance being a variance. For the zero-one loss no analogous *additive* bias-variance decomposition exists — the various proposals in the literature are definition-dependent and do not decompose additively. The approximation-estimation decomposition, by contrast, is pure bookkeeping (add and subtract $\min_{h\in\mathcal{H}} R(h)$ and $R^*_{\mathcal{D}}$) and therefore holds for **any** loss function whatsoever. This is why statistical learning theory is built on the approximation-estimation split rather than on bias-variance, even though the latter is the more familiar of the two.

5.4 Vapnik - Chervonenkis (VC) Dimension

The generalization bound developed above was for a finite hypothesis set. However, in practice we often deal with infinite hypothesis sets. In order to develop generalization bounds for them, we need a measure of complexity that stays finite even when $|\mathcal{H}| = \infty$. (Vapnik and Chervonenkis, 1971) built such a measure on a remarkable observation: in a binary classification problem, a hypothesis set can report only a finite number of distinct labelings of a finite sample, even though it contains infinitely many hypotheses. The chain of definitions runs restriction → shattering → growth function → VC dimension; let us take them in turn.

Definition 5.4.1: Restriction

Given $S = \{x_1, \ldots, x_m\} \subset \mathcal{X}$, the following set

$$\mathcal{H}_S = \{(h(x_1), \ldots, h(x_m)) : h \in \mathcal{H}\}$$

is called a **restriction** of $\mathcal{H}$ to S.

In words, a restriction is the set of all unique label vectors a hypothesis class $\mathcal{H}$ can generate from a data set S.

Definition 5.4.2: Shattering

$\mathcal{H}$ is said to **shatter** S if $|\mathcal{H}_S| = 2^{|S|}$.

In words, a hypothesis class $\mathcal{H}$ shatters a data set S if the restriction of $\mathcal{H}$ to S is the set of all functions from S to $\{0, 1\}$.

Corollary 5.4.3

If a data set $C \subset \mathcal{X}$ with $2m$ data points (i.e. $|C| = 2m$) is shattered by $\mathcal{H}$, then for every learning algorithm A, there exists a distribution $\mathcal{D}$ over $\mathcal{X} \times \{0, 1\}$, supported on C, such that some $h \in \mathcal{H}$ satisfies $R(h) = 0$, yet $P\big(\{S' \sim \mathcal{D}^m : R(A(S')) \geq 1/8\}\big) \geq 1/7$.

Proof. Run the proof of Theorem 5.2 with $\mathcal{X}$ replaced by C itself, which is legitimate because $|C| = 2m > m$. That proof produced a distribution $\mathcal{D} = \mathcal{D}_{i^*}$, supported on C and labeled by one of the 2^{2m} functions $f_{i^*} : C \to \{0, 1\}$, on which A fails as claimed. The only thing left to check is that the perfect predictor may be taken *inside* $\mathcal{H}$: since $\mathcal{H}$ shatters C, its restriction $\mathcal{H}_C$ is all of $\{0, 1\}^C$ (Definition 5.10), so **every** f_i — f_{i^*} in particular — agrees on C with some $h \in \mathcal{H}$, and that h has $R(h) = 0$ because $\mathcal{D}$ is supported on C □

In words, if $\mathcal{H}$ is large enough to shatter a data set of $2m$ on a domain $\mathcal{X}$, then we cannot find a learning algorithm as good as we desire (arbitrarily good) that can be run on data sets of size m or smaller. The intuitive explanation is that due to the No Free Lunch theorem, under such circumstances, the hypothesis space $\mathcal{H}$ is so large that for any $2m$ data points, the true labels of the half that is not used in training can be perfectly explained by a hypothesis

inside $\mathcal{H}$. This is a manifestation of Karl Popper's falsifiability principle in machine learning theory. This principle sets a demarcation point between science and other types of knowledge generation. An explanation is scientific if there exists a set of observations that can falsify it.

Definition 5.4.4: Growth function

The growth function $\tau_{\mathcal{H}} : \mathbb{N}^+ \to \mathbb{N}^+$ of $\mathcal{H}$ is defined as

$$\tau_{\mathcal{H}}(m) = \max_{S \subseteq \mathcal{X}, |S|=m} |\mathcal{H}_S|.$$

In words, the growth function counts the maximum number of distinct labelings that $\mathcal{H}$ can report on any data set on domain $\mathcal{X}$ that contains m data points.

Note that $\mathcal{H}$ shatters some $S \subseteq \mathcal{X}$ with $|S| = m$ if and only if $\tau_{\mathcal{H}}(m) = 2^m$.

Definition 5.4.5: VC dimension

The VC dimension of a hypothesis set $\mathcal{H}$ is the size of the largest data set that $\mathcal{H}$ can shatter:

$$d_{VC}(\mathcal{H}) = \sup\{m : \tau_{\mathcal{H}}(m) = 2^m\},$$

where the supremum is ∞ if $\mathcal{H}$ shatters arbitrarily large sets.

Intuitively, $d_{VC}(\mathcal{H})$ is the largest number of points $\mathcal{H}$ can label in *every* conceivable way — a single number summarizing how much a class can "memorize" arbitrary data, and hence how easily it can overfit; it plays exactly the role $\log|\mathcal{H}|$ played for finite classes, but stays finite even when $\mathcal{H}$ itself is infinite. Let us now see what a finite VC dimension buys us — first qualitatively, then quantitatively.

Theorem 5.4.6: VC and PAC learnability

If $d_{VC}(\mathcal{H}) = \infty$ then $\mathcal{H}$ is not PAC learnable.

Proof. For any training set size m, the infinite VC dimension allows $\mathcal{H}$ to shatter a data set of size $2m$. Then the result follows from Corollary 5.3 □

This theorem is an analytical way of the argumentation presented above after the No Free Lunch theorem. Learning is only possible with some inductive bias. The bias-complexity dilemma stems from this obligation.

Theorem 5.4.7: The Fundamental Theorem of Statistical Learning

Assume that $d_{VC}(\mathcal{H}) = d < \infty$. Then, there exist $C_1, C_2 \in \mathbb{R}^+$ such that

- $\mathcal{H}$ has the uniform convergence property with sample complexity

$$C_1 \frac{d + \log(1/\delta)}{\epsilon^2} \leq m_{\mathcal{H}}^{\text{UC}}(\epsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\epsilon^2}$$

- $\mathcal{H}$ is agnostic PAC learnable with sample complexity $C_1 \frac{d+\log(1/\delta)}{\epsilon^2} \leq m_{\mathcal{H}}(\epsilon, \delta) \leq C_2 \frac{d+\log(1/\delta)}{\epsilon^2}$
- $\mathcal{H}$ is PAC learnable with sample complexity

$$C_1 \frac{d + \log(1/\delta)}{\epsilon} \leq m_{\mathcal{H}}(\epsilon, \delta) \leq C_2 \frac{d \log(1/\epsilon) + \log(1/\delta)}{\epsilon}$$

In words, the following statements are equal:

- $\mathcal{H}$ has the uniform convergence property.
- Any ERM rule is a successful agnostic PAC learner for $\mathcal{H}$.
- $\mathcal{H}$ is agnostic PAC learnable.
- $\mathcal{H}$ is PAC learnable.
- Any ERM rule is a successful PAC learner for $\mathcal{H}$.
- $\mathcal{H}$ has a finite VC-dimension.

This outcome is very useful because it allows one to achieve all these six nice properties by ensuring only one of them.

Proof. We prove the theorem in two parts: the **equivalence of the six statements**, which is where the conceptual content lies and which we establish completely; and the **sample complexity rates**, whose upper bounds we prove up to a logarithmic factor and whose lower bounds we prove in their $\Theta(d)$ part.

Part 1: the equivalence. It suffices to close the cycle

$$\begin{aligned} \text{finite } d_{VC} \;&\Rightarrow\; \text{uniform convergence} \;\Rightarrow\; \text{ERM is agnostic PAC} \\ &\Rightarrow\; \text{agnostic PAC learnable} \;\Rightarrow\; \text{PAC learnable} \;\Rightarrow\; \text{finite } d_{VC}, \end{aligned}$$

after which every statement implies every other, and the two ERM statements follow because the implications that produce them exhibit ERM itself as the successful learner.

1. *Finite $d_{VC} \Rightarrow$ uniform convergence.* By Lemma 5.2, $\tau_{\mathcal{H}}(m) \leq (em/d)^d$ is polynomial in m, so $\log \tau_{\mathcal{H}}(2m)/m \to 0$. Theorem 5.5 below (whose proof uses only Lemma 5.2, symmetrization, and Massart's lemma) then bounds the representativeness gap by a quantity tending to 0, which is exactly Definition 5.5.

2. *Uniform convergence ⇒ ERM is an agnostic PAC learner.* Immediate from Lemma 5.1: take $m \geq m_{\mathcal{H}}^{\mathrm{UC}}(\epsilon/2, \delta)$, so that S is $\epsilon/2$-representative with probability $\geq 1-\delta$, on which event every ERM output satisfies $R(h_S) \leq \min_h R(h) + \epsilon$.
3. *ERM is an agnostic PAC learner ⇒ agnostic PAC learnable.* By definition — a witness algorithm has been exhibited.
4. *Agnostic PAC learnable ⇒ PAC learnable.* Already argued after Definition 5.8: a realizable $\mathcal{D}$ has $\min_h R(h) = 0$, so Definition 5.6 specializes to Definition 5.8.
5. *PAC learnable ⇒ finite d_{VC}.* This is the contrapositive of Theorem 5.3, which was proved from Corollary 5.3 and hence from the No Free Lunch theorem.

Part 2: the upper bounds. Combining Lemma 5.2 with Massart's lemma bounds the empirical Rademacher complexity of $\mathcal{H}$ by $\sqrt{2d\log(m+1)/m}$ (this is Theorem 5a.3, proved in Chapter 5a), and Theorem 5a.2 then converts it into the uniform bound

$$\sup_{h\in\mathcal{H}} \left|R(h) - \widehat{R}_S(h)\right| \;\leq\; 2\sqrt{\frac{2d\log(m+1)}{m}} + 3\sqrt{\frac{\log(2/\delta)}{2m}} \quad \text{w.p.} \geq 1-\delta.$$

Setting the right-hand side to ϵ and solving for m gives $m_{\mathcal{H}}^{\mathrm{UC}}(\epsilon,\delta) = O\big(\frac{d\log(1/\epsilon)+\log(1/\delta)}{\epsilon^2}\big)$, and Part 1, step 2 turns this into the same rate for $m_{\mathcal{H}}(\epsilon,\delta)$ with an extra factor 4 from replacing ϵ by $\epsilon/2$. This matches the claimed $C_2\frac{d+\log(1/\delta)}{\epsilon^2}$ **up to the residual** $\log(1/\epsilon)$ **factor**. That last factor is an artifact of applying the finite-class bound at a single resolution; removing it requires the *chaining* argument (the Dudley entropy integral) discussed at the end of the covering-numbers section of Chapter 5a, which we do not develop here. For the realizable case, one exploits that an ERM hypothesis has $\widehat{R}_S(h_S) = 0$, for which the multiplicative ("relative deviation") form of the Chernoff bound replaces ϵ^2 by ϵ in the denominator, yielding the stated $O\big(\frac{d\log(1/\epsilon)+\log(1/\delta)}{\epsilon}\big)$.

Part 3: the lower bounds. Let $C \subseteq \mathcal{X}$ be a set of size d shattered by $\mathcal{H}$, and let $m < d/2$. Apply Theorem 5.2 with the domain taken to be C (legitimate since $m < |C|/2$) and then Corollary 5.3's observation that shattering places the perfect predictor inside $\mathcal{H}$: for **every** learning algorithm A there is a realizable distribution $\mathcal{D}$ on $C \times \{0,1\}$ with

$$P\big(\{S \sim \mathcal{D}^m : R(A(S)) \geq 1/8\}\big) \geq 1/7.$$

Hence for any $\epsilon < 1/8$ and $\delta < 1/7$ no sample size below $d/2$ can suffice, i.e. $m_{\mathcal{H}}(\epsilon,\delta) \geq d/2$: the sample complexity must grow at least linearly in the VC dimension, in the realizable case and therefore also in the agnostic case. A second, independent lower bound $m_{\mathcal{H}}(\epsilon,\delta) = \Omega(\log(1/\delta)/\epsilon^2)$ already holds for a class of a *single* hypothesis, by the two-point argument that a $\pm\epsilon$-biased coin needs $\Omega(\epsilon^{-2})$ flips to be told from a fair one. Adding the two gives the claimed shape $\Omega\big(\frac{d+\log(1/\delta)}{\epsilon^2}\big)$ **up to the joint dependence on** d **and** ϵ; sharpening the first bound from $\Omega(d)$ to $\Omega(d/\epsilon)$ (realizable) and $\Omega(d/\epsilon^2)$ (agnostic) requires a refinement of Theorem 5.2 in which the labels on the shattered set are not deterministic but ϵ-biased coins, so that the learner must additionally *estimate* d separate biases. That refinement is a genuine strengthening of the No Free Lunch argument and we omit it □

The VC dimension gives a binary characterization of the hypothesis class. One may want to analyze the capacity of a hypothesis class in relation to an arbitrary data set size smaller or larger than its VC dimension. The following lemma, independently due to (Sauer, 1972) and (Shelah, 1972), is instrumental to relate the growth function of a hypothesis class to sample size.

Lemma 5.4.8: Sauer-Shelah-Perles

If $d_{VC}(\mathcal{H}) = d < \infty$, then for all $m \geq d$, we have $\tau_{\mathcal{H}}(m) \leq \sum_{i=0}^{d} \binom{m}{i}$. In particular, for $m \geq d$, we have $\tau_{\mathcal{H}}(m) \leq \left(\frac{em}{d}\right)^d$.

Proof. The lemma follows from a stronger, purely combinatorial claim that makes no reference to d at all:

$$\textbf{Claim.} \quad |\mathcal{H}_S| \;\leq\; \big|\{B \subseteq S : \mathcal{H} \text{ shatters } B\}\big|, \quad \text{for every finite } S \subseteq \mathcal{X}.$$

Proof of the claim, by induction on $m = |S|$. For $m = 1$, say $S = \{x_1\}$: if $|\mathcal{H}_S| = 1$ the right-hand side is at least 1 (the empty set is shattered by any non-empty class), and if $|\mathcal{H}_S| = 2$ then $\mathcal{H}$ shatters both $\emptyset$ and $\{x_1\}$, so the right-hand side is 2.

For the inductive step, let $S = \{x_1, \ldots, x_m\}$ and $S' := \{x_2, \ldots, x_m\}$. Split the labelings in $\mathcal{H}_S$ according to whether their restriction to S' occurs with one or with both values of the first coordinate. Formally, set

$$Y_0 := \mathcal{H}_{S'},$$
$$Y_1 := \big\{(y_2, \ldots, y_m) : (0, y_2, \ldots, y_m) \in \mathcal{H}_S \text{ and } (1, y_2, \ldots, y_m) \in \mathcal{H}_S\big\},$$

so that every element of $\mathcal{H}_S$ is obtained either as the unique extension of an element of Y_0, or as one of the two extensions of an element of Y_1; counting gives exactly

$$|\mathcal{H}_S| = |Y_0| + |Y_1|.$$

The induction hypothesis applied to $\mathcal{H}$ on S' bounds the first term: $|Y_0| = |\mathcal{H}_{S'}| \leq |\{B \subseteq S' : \mathcal{H} \text{ shatters } B\}|$. For the second, define the subclass

$$\mathcal{H}' := \big\{h \in \mathcal{H} : \exists h' \in \mathcal{H},\ h'(x_1) = 1 - h(x_1) \text{ and } h'(x_i) = h(x_i)\ \forall i \geq 2\big\},$$

i.e. those hypotheses whose labeling of S' is realized with *both* labels of x_1. By construction $Y_1 = \mathcal{H}'_{S'}$, and crucially: **if $\mathcal{H}'$ shatters $B \subseteq S'$ then $\mathcal{H}'$, hence $\mathcal{H}$, shatters $B \cup \{x_1\}$** — because each of the $2^{|B|}$ labelings of B is available in $\mathcal{H}'$ with both possible labels of x_1. Therefore, by the induction hypothesis applied to $\mathcal{H}'$ on S',

$$\begin{aligned} |Y_1| = |\mathcal{H}'_{S'}| &\leq \big|\{B \subseteq S' : \mathcal{H}' \text{ shatters } B\}\big| \\ &= \big|\{B \subseteq S' : \mathcal{H}' \text{ shatters } B \cup \{x_1\}\}\big| \\ &\leq \big|\{B \subseteq S : x_1 \in B,\ \mathcal{H} \text{ shatters } B\}\big|. \end{aligned}$$

Adding the two bounds counts each shattered subset of S exactly once — those avoiding x_1 in the first term, those containing it in the second — which proves the claim.

From the claim to the lemma. Since $d_{VC}(\mathcal{H}) = d$, no subset of size $> d$ is shattered, so the right-hand side of the claim counts at most all subsets of size at most d:

$$\tau_{\mathcal{H}}(m) = \max_{|S|=m} |\mathcal{H}_S| \le \sum_{i=0}^{d} \binom{m}{i}.$$

For the closed form, assume $m \ge d$ so that $(d/m)^i \ge (d/m)^d$ for every $i \le d$, and apply the binomial theorem:

$$\Big(\frac{d}{m}\Big)^d \sum_{i=0}^{d} \binom{m}{i} \;\le\; \sum_{i=0}^{d} \binom{m}{i} \Big(\frac{d}{m}\Big)^i \;\le\; \sum_{i=0}^{m} \binom{m}{i} \Big(\frac{d}{m}\Big)^i = \Big(1 + \frac{d}{m}\Big)^m \le e^d,$$

the last step by $1 + u \le e^u$. Rearranging gives $\sum_{i=0}^{d} \binom{m}{i} \le (em/d)^d$ □

The lemma is the combinatorial heart of the whole theory: it converts a *binary* statement ("$\mathcal{H}$ cannot shatter $d+1$ points") into a *quantitative* one ("$\mathcal{H}$ produces at most polynomially many labelings"), and it is the polynomial-versus-exponential gap this opens that every subsequent bound exploits.

Theorem 5.4.9: Sample complexity wrt growth function

Let $\mathcal{H}$ be a hypothesis class with growth function $\tau_{\mathcal{H}}$. Then for every distribution $\mathcal{D}$ and every $\delta \in (0,1)$, a sample set $S = \{(x_i, y_i) \overset{i.i.d.}{\sim} \mathcal{D} : i = 1, \dots, m\}$ satisfies

$$P\left(\left\{ S \sim \mathcal{D}^m : \sup_{h \in \mathcal{H}} |R(h) - \widehat{R}_S(h)| \le \frac{1}{\delta}\sqrt{\frac{2\log\big(2\,\tau_{\mathcal{H}}(2m)\big)}{m}} \right\}\right) \ge 1 - \delta.$$

Proof. The proof has two steps: bound the *expected* representativeness gap, then convert to a high-probability statement by Markov's inequality.

Step 1 (expectation bound). Introduce a **ghost sample** $S' = \{(x'_i, y'_i)\}_{i=1}^m \sim \mathcal{D}^m$, independent of S. Since $R(h) = \mathbb{E}_{S'}[\widehat{R}_{S'}(h)]$, Jensen's inequality (an absolute value of an expectation is at most the expectation of the absolute value, and a supremum of expectations is at most the expectation of the supremum) gives

$$\mathbb{E}_S\Big[\sup_h \big|R(h) - \widehat{R}_S(h)\big|\Big] \;\le\; \mathbb{E}_{S,S'}\Big[\sup_h \big|\widehat{R}_{S'}(h) - \widehat{R}_S(h)\big|\Big].$$

Because S and S' are i.i.d. and mutually independent, exchanging the i-th pair of S with the i-th pair of S' leaves the joint law unchanged. Recording the exchanges with independent Rademacher signs $\sigma_i \in \{-1, +1\}$ therefore changes nothing:

$$\mathbb{E}_{S,S'}\Big[\sup_h \Big|\frac{1}{m}\sum_i \big(\ell(y'_i, h(x'_i)) - \ell(y_i, h(x_i))\big)\Big|\Big]$$
$$= \mathbb{E}_{S,S',\sigma}\Big[\sup_h \Big|\frac{1}{m}\sum_i \sigma_i\big(\ell(y'_i, h(x'_i)) - \ell(y_i, h(x_i))\big)\Big|\Big].$$

Now **condition on S and S'** and look at the inner expectation over σ alone. For each h define the vector $a^{(h)} \in \mathbb{R}^m$ with $a_i^{(h)} := \frac{1}{m}\big(\ell(y_i', h(x_i')) - \ell(y_i, h(x_i))\big)$. Two observations:

- $a^{(h)}$ depends on h only through h's restriction to the $2m$ points $\{x_1, \dots, x_m, x_1', \dots, x_m'\}$, so at most $\tau_{\mathcal{H}}(2m)$ distinct vectors arise (Definition 5.11);
- each coordinate lies in $[-1/m, 1/m]$ because the loss lies in $[0,1]$, hence $\|a^{(h)}\|_2 \le \sqrt{m}/m = 1/\sqrt{m}$.

Put $A := \{a^{(h)} : h \in \mathcal{H}\} \cup \{-a^{(h)} : h \in \mathcal{H}\}$, a finite set of at most $2\tau_{\mathcal{H}}(2m)$ vectors of Euclidean norm at most $1/\sqrt{m}$, and note $\sup_h |\langle \sigma, a^{(h)}\rangle| = \max_{a \in A} \langle \sigma, a\rangle$ — this is why the set is symmetrized. Massart's lemma (Lemma 5a.1, in the un-normalized form $\mathbb{E}_\sigma[\max_{a\in A}\langle\sigma,a\rangle] \le \max_a \|a\|_2 \sqrt{2\log|A|}$) gives

$$\mathbb{E}_\sigma\Big[\max_{a \in A} \langle \sigma, a\rangle\Big] \;\le\; \frac{1}{\sqrt{m}} \sqrt{2 \log\big(2\tau_{\mathcal{H}}(2m)\big)}.$$

The bound is uniform in S, S', so it survives taking the outer expectation:

$$\mathbb{E}_S\Big[\sup_h \big|R(h) - \widehat{R}_S(h)\big|\Big] \;\le\; \sqrt{\frac{2\log\big(2\tau_{\mathcal{H}}(2m)\big)}{m}} \;=:\; \bar{\epsilon}_m.$$

Step 2 (Markov). The random variable $\sup_h |R(h) - \widehat{R}_S(h)|$ is non-negative, so Theorem 4.1 with threshold $\bar{\epsilon}_m/\delta$ gives $P\big(\sup_h |R(h) - \widehat{R}_S(h)| \ge \bar{\epsilon}_m/\delta\big) \le \delta$, which is the claim $\square$

Two remarks on the shape of this bound. First, it depends on δ as $1/\delta$ rather than $\sqrt{\log(1/\delta)}$, which is the price of using Markov's inequality on the expectation rather than a concentration inequality for the supremum itself; Chapter 5a replaces this step with McDiarmid's inequality and recovers the $\sqrt{\log(1/\delta)}$ dependence. Second, the constants are not optimal — a more careful sub-Gaussian maximal inequality replaces the bound above by $\frac{4+\sqrt{\log \tau_{\mathcal{H}}(2m)}}{\delta\sqrt{2m}}$ — but the essential dependence, $\sqrt{\log \tau_{\mathcal{H}}(2m)/m}$, is the same and is all that the discussion below uses.

This result is consistent with the implications of the fundamental theorem of statistical learning. The bound is useful exactly when $\sqrt{\log \tau_{\mathcal{H}}(2m)}/\sqrt{2m} \to 0$, i.e. when $\log \tau_{\mathcal{H}}(m)/m \to 0$: the growth function may grow, but only *sub-exponentially* in m. By the Sauer-Shelah-Perles lemma this is precisely what a finite VC dimension guarantees, since $\tau_{\mathcal{H}}(m) \le (em/d)^d$ is polynomial in m, so $\log \tau_{\mathcal{H}}(m) = O(d \log m)$. Conversely, if $\mathcal{H}$ shatters arbitrarily large sets then $\tau_{\mathcal{H}}(m) = 2^m$ for every m, $\log \tau_{\mathcal{H}}(m)/m = \log 2$ does not vanish, and the bound degenerates — matching Theorem 5.3.

5.4.1 Nonuniform Learnability

Definition 5.4.10: Nonuniform Learnability

A hypothesis class $\mathcal{H}$ is **nonuniformly learnable** if there exist a function $m_{\mathcal{H}}^{NU} : (0,1)^2 \times \mathcal{H} \to \mathbb{N}$ and a **learning algorithm** A such that for every $\epsilon, \delta \in (0,1)$ and every distribution $\mathcal{D}$, running A on data set $S = \{(x_i, y_i) \overset{i.i.d.}{\sim} D : i = 1, \ldots, m\}$ satisfies, for every $h \in \mathcal{H}$ and every $m \geq m_{\mathcal{H}}^{NU}(\epsilon, \delta, h)$,

$$P(\{S \sim \mathcal{D}^m : R(A(S)) \leq R(h) + \epsilon\}) \geq 1 - \delta.$$

The difference from Definition 5.6 is that the required sample size may depend on the hypothesis h we are competing against, rather than being uniform over the whole class — hence the name.

An agnostic PAC learnable hypothesis class $\mathcal{H}$ is also nonuniform learnable because $R(A(S)) \leq \min_{h' \in \mathcal{H}} R(h') + \epsilon$ results trivially in $R(A(S)) \leq R(h) + \epsilon, \forall h \in \mathcal{H}$. Hence, the set of non-uniform learnable hypothesis classes is larger than the agnostic PAC learnable hypothesis classes. With this relaxation, our motivation is to increase the model capacity while sustaining its learnability.

Theorem 5.4.11

If $\mathcal{H} = \cup_{i \in \mathbb{N}} \mathcal{H}_i$ such that each $\mathcal{H}_i$ has the uniform convergence property, then $\mathcal{H}$ is nonuniformly learnable.

Proof. Fix the weights $w(i) := 6/(\pi^2 i^2)$, which satisfy $\sum_{i \geq 1} w(i) = 1$ by the Basel identity $\sum_{i \geq 1} i^{-2} = \pi^2/6$, and take A to be the SRM rule h_{SRM} defined below in this section. Theorem 5.7, proved next, guarantees that with probability at least $1 - \delta$,

$$R(h) \leq \widehat{R}_S(h) + \epsilon_{i(h)}\big(m, w(i(h))\delta\big)$$
$$\text{simultaneously for every } h \in \mathcal{H}.$$

On that event, for any fixed $h \in \mathcal{H}_i$, the definition of h_{SRM} as the minimizer of the right-hand side gives

$$R(h_{SRM}) \leq \widehat{R}_S(h_{SRM}) + \epsilon_{i(h_{SRM})}\big(m, w(i(h_{SRM}))\delta\big) \leq \widehat{R}_S(h) + \epsilon_i\big(m, w(i)\delta\big) \leq R(h) + 2\,\epsilon_i\big(m, w(i)\delta\big),$$

the last step using the uniform convergence of $\mathcal{H}_i$ once more, in the opposite direction. Since $\mathcal{H}_i$ has the uniform convergence property, $\epsilon_i(m, w(i)\delta) \to 0$ as $m \to \infty$ with i, δ fixed, so for every ϵ, δ and every h there is a finite

$$m_{\mathcal{H}}^{NU}(\epsilon, \delta, h) := \min\big\{m : 2\,\epsilon_{i(h)}\big(m, w(i(h))\delta\big) \leq \epsilon\big\}$$

beyond which $R(A(S)) \leq R(h) + \epsilon$ holds with probability at least $1 - \delta$. This is exactly Definition 5.13 □

Note precisely where nonuniformity enters: the sample size above depends on h through the index $i(h)$ of the subclass containing it. There is no single m that works for all h at once, because the penalty ϵ_i worsens without bound as i grows.

Vice versa also holds. If a hypothesis class $\mathcal{H}$ is nonuniformly learnable, then it can be expressed as a countable union of agnostic PAC learnable subclasses. Next let us see how we can use this result to build a more powerful learning algorithm than ERM. Remember that we have sample complexity guarantees at the hypothesis subclass level while we need to do our search in the hypothesis level. The following quantity is instrumental to bridge this gap:

$$\epsilon_i(m, \delta) = \min\{\epsilon \in (0, 1) : m^{UC}_{\mathcal{H}_i}(\epsilon, \delta) \leq m\}$$

which in effect inverts the complexity function $m^{UC}_{\mathcal{H}_i}$ of hypothesis subclass i. This quantity gives the smallest representativeness level attainable within subclass i from m data points at confidence $1 - \delta$. In other words, the best level of representativeness reachable within this hypothesis subclass, i.e., $P(\{S \sim \mathcal{D}^m : |R(h) - \widehat{R}_S(h)| \leq \epsilon_i(m, \delta)\}) \geq 1 - \delta$ for the event of collecting a data set S of size m i.i.d. from any data distribution $\mathcal{D}$.

Theorem 5.4.12

Let $w : \mathbb{N} \to [0, 1]$ be a function with $\sum_{i=1}^{\infty} w(i) < 1$ and $\mathcal{H} = \cup_{i \in \mathbb{N}} \mathcal{H}_i$ such that each $\mathcal{H}_i$ has the uniform convergence property with $m^{UC}_{\mathcal{H}_i}$. Then for every $\delta \in (0, 1)$, every data distribution $\mathcal{D}$, and every $m \in \mathbb{N}$, and every $i \in \mathbb{N}$ and $h \in \mathcal{H}_i$ we have

$$P\left(\left\{S \sim \mathcal{D}^m : |R(h) - \widehat{R}_S(h)| \leq \epsilon_i(m, w(i)\delta)\right\}\right) \geq 1 - \delta$$

where S is a data set of size m sampled i.i.d. from $\mathcal{D}$.

Proof. Fix i and let B_i be the bad event that S fails to be $\epsilon_i(m, w(i)\delta)$-representative for $\mathcal{H}_i$, i.e. $B_i := \{\exists h \in \mathcal{H}_i : |R(h) - \widehat{R}_S(h)| > \epsilon_i(m, w(i)\delta)\}$. By the definition of ϵ_i as the inverse of the sample complexity function $m^{UC}_{\mathcal{H}_i}$, applying Definition 5.5 to $\mathcal{H}_i$ with confidence parameter $w(i)\delta$ gives $P(B_i) \leq w(i)\,\delta$. The union bound over the countably many subclasses then gives

$$P\Big(\bigcup_{i \in \mathbb{N}} B_i\Big) \leq \sum_{i \in \mathbb{N}} P(B_i) \leq \delta \sum_{i \in \mathbb{N}} w(i) \;<\; \delta,$$

using the hypothesis $\sum_i w(i) < 1$. On the complement — an event of probability at least $1 - \delta$ — no subclass fails, which is the claim □

The proof shows exactly what the weights buy: a union bound over infinitely many events is finite only if the individual confidence levels are made summable, and w is the budget allocating a share $w(i)$ of the total failure probability δ to subclass i. Any summable allocation works; different choices of w simply express different prior preferences over the subclasses.

The goal of this theorem is to express the uniform convergence of a composite hypothesis class comprising subclasses with different uniform convergence guarantees. It only rescales the confidence level of the subclasses with an index function on the classes. As we will see next, this index will allow us to search the composite hypothesis space, thereby define a

learning rule that is more powerful than ERM. The above result can also be expressed as follows:

$$P\left(\left\{S \sim \mathcal{D}^m : R(h) \leq \widehat{R}_S(h) + \epsilon_i(m, w(i(h)) \cdot \delta)\right\}\right) \geq 1 - \delta, \ \forall h \in \mathcal{H}$$

where $i(h) = \min_{i:h\in\mathcal{H}_i}$ is a function that finds the most sample-efficient class a hypothesis h belongs. This result prescribes a searching rule for an extended set of hypotheses that satisfies PAC learnability. We can then use it to build a new learning algorithm

$$h_{SRM} \in \arg\min_{h\in\mathcal{H}} \left\{\widehat{R}_S(h) + \epsilon_{i(h)}(m, w(i(h)) \cdot \delta)\right\}$$

which we call **Structural Risk Minimization (SRM)**: it minimizes not the empirical risk itself, but the upper bound on $R(h)$ that the previous display certifies. Remember that by Hoeffding's inequality we have $m_h^{UC}(\epsilon, \delta) = \frac{\log(2/\delta)}{2\epsilon^2}$ for a single hypothesis h, which implies $\epsilon_i(m, \delta) = \sqrt{\frac{\log(2/\delta)}{2m}}$. Defining $w'(h) := w(i(h))$ we get

$$h_{SRM} \in \arg\min_{h\in\mathcal{H}} \widehat{R}_S(h) + \sqrt{\frac{-\log w'(h) + \log(2/\delta)}{2m}}$$

where we use the trivial identity that $\log(2/\delta \cdot w'(h)) = -\log w'(h) + \log(2/\delta)$. In effect, SRM applies a preference on the hypothesis subclasses represented in the learning rule by the weight $w'(h)$. The higher the weight, the stronger the preference. These preferences can be chosen to reflect prior knowledge coming from the expertise in the domain regarding the collected data.

In many real-world problems, we have little domain knowledge to be incorporated into the learning process. It would be useful to have a guideline for building machine learning algorithms without any domain knowledge. Remember that the only condition to combine individually agnostic PAC learnable hypothesis classes into a bigger class is to define a series of positive weights that converge to a value less than one. The following inequality provides an instrumental way of building such a series.

Lemma 5.4.13: Kraft's inequality

Let Σ^* be a set of finite-length strings over the alphabet $\Sigma = \{0, 1\}$ that is **prefix-free**, in the sense that no string $s \in \Sigma^*$ of length k coincides with the first k characters of a longer string in Σ^*. Then $\sum_{s\in\Sigma^*} 2^{-|s|} \leq 1$, where $|s|$ denotes the length of s.

Proof. Generate an infinite sequence of i.i.d. fair bits $b_1, b_2, \ldots$, so that $P(b_1 = c_1, \ldots, b_k = c_k) = 2^{-k}$ for any fixed pattern $c \in \{0, 1\}^k$. For each $s \in \Sigma^*$ define the event

$$A_s := \{b_1 = s_1, \ \ldots, \ b_{|s|} = s_{|s|}\},$$
$$P(A_s) = 2^{-|s|}.$$

The events are **pairwise disjoint**: if A_s and $A_{s'}$ both occurred for $s \neq s'$ with $|s| \leq |s'|$, then s would be the first $|s|$ characters of s', contradicting prefix-freeness. Disjointness and

the axioms of a probability measure (σ-additivity plus $P(\Omega) = 1$) give

$$\sum_{s\in\Sigma^*} 2^{-|s|} = \sum_{s\in\Sigma^*} P(A_s) = P\Big(\bigcup_{s\in\Sigma^*} A_s\Big) \leq 1. \qquad \square$$

In words: assigning short codewords is a scarce resource. Prefix-freeness is what makes a code uniquely decodable one symbol at a time, and it forces the codeword lengths to behave like an (unnormalized) probability distribution — you cannot give *everything* a short code, only give some items long codes to pay for others being short. This is exactly why $2^{-|s(h)|}$ below can be reused as the summable weight $w'(h)$ that Theorem 5.7 needs.

The lemma is precisely what Theorem 5.7 needs: a description language for hypotheses hands us, for free, a summable weight function $w'(h) := 2^{-|s(h)|}$, with no need to design one by hand. $\square$

Hence, whenever we can find an alphabet that uniquely describes all hypotheses in a class with a string $s(h)$, we can plug Kraft's inequality into Theorem 5.7 with $w'(h) := 2^{-|s(h)|}$ and get

$$P\Big(\Big\{S \sim \mathcal{D}^m : R(h) \leq \widehat{R}_S(h) + \sqrt{\frac{|s(h)| + \log(2/\delta)}{2m}}\Big\}\Big) \geq 1 - \delta$$

where we use the property $\log(2) < 1$. The right-hand side of this inequality prescribes the learning rule below

$$h_{MDL} \in \arg\min_{h\in\mathcal{H}} \widehat{R}_S(h) + \sqrt{\frac{|s(h)| + \log(2/\delta)}{2m}}.$$

Note that this rule suggests the hypothesis that can be expressed with minimum number of characters among the ones that fit the data best. Hence, it is referred to as the **Minimum Description Length(MDL)** rule. The MDL rule is yet another instance of the Occam's Razor principle in machine learning, which also has overarching consequences on hypothesis development in scientific investigation. This rule also sets a theoretical foundation for the idea of applying **regularizers** in machine learning.

Every result in this chapter — the finite-class bound, the VC-dimension-based Fundamental Theorem, the SRM and MDL rules built on top of it — measured the complexity of $\mathcal{H}$ through $|\mathcal{H}|$ or $d_{VC}(\mathcal{H})$: a raw count, or a worst-case combinatorial shattering capacity. Chapter 5a picks up right where the Fundamental Theorem (Theorem 5.4) leaves off, and asks what happens if we measure complexity differently — through how well $\mathcal{H}$ can correlate with pure noise (Rademacher complexity), through how finely $\mathcal{H}$ can be approximated by a finite grid (covering numbers), or through how much a data-dependent choice inside $\mathcal{H}$ must diverge from a fixed prior belief (PAC-Bayes bounds) — and shows that all of these, $|\mathcal{H}|$ and $d_{VC}(\mathcal{H})$ included, are instances of one and the same abstract idea.

CHAPTER 6

COMPLEXITY MEASURES AND GENERALIZATION

Chapter 5 built the entire theory of generalization on top of two concrete complexity measures: the raw count $|\mathcal{H}|$ for finite classes, and the VC dimension $d_{VC}(\mathcal{H})$ for infinite ones. Both did the same job: they compressed a hypothesis class down to a single number that a Hoeffding-type concentration inequality, combined with the union bound, could feed on. This chapter makes that role explicit and abstract, then instantiates it three more times.

The abstract recipe. Every bound in this course, past and future, has the shape

$$R(h) \le \widehat{R}_S(h) + \underbrace{\text{Complexity}(\mathcal{H}, m)}_{\text{shrinks as } m\to\infty} + \underbrace{\text{Confidence}(\delta, m)}_{\text{shrinks as } m\to\infty}, \qquad \forall h \in \mathcal{H},\ \text{w.p.} \ge 1-\delta,$$

i.e. a bound on the **representativeness** gap $\sup_{h\in\mathcal{H}} \left(R(h) - \widehat{R}_S(h)\right)$ from Definition 5.4. What is negotiable is only how Complexity$(\mathcal{H}, m)$ is defined. We already have two instances: Complexity $= \sqrt{\log|\mathcal{H}|/(2m)}$ (Theorem 5.1) and Complexity $= \sqrt{d_{VC}(\mathcal{H})/m}$ up to logarithmic factors (Theorem 5.4). We now add three more: **Rademacher complexity**, **covering numbers**, and **PAC-Bayes bounds**. All five are, in a precise sense we make explicit below, the same quantity viewed through different lenses — and by the end of this chapter, we will have shown algebraically that the very first bound of Chapter 5 (Theorem 5.1) is a special case of the PAC-Bayes bound derived last.

6.1 A Concentration Tool for Suprema: McDiarmid's Inequality

Hoeffding's inequality (Chapter 4) concentrates a *single* average $\widehat{\mu}_m$ around its mean. Bounding $\sup_{h\in\mathcal{H}} \left(R(h) - \widehat{R}_S(h)\right)$ requires concentrating a *function of the whole sample* — the supremum over (possibly infinitely many) hypotheses — around its mean. The right tool is a generalization, due to (McDiarmid, 1989), of Hoeffding's inequality to arbitrary functions of independent random variables, provided the function does not depend too sensitively on any single one of them.

Theorem 6.1.1: McDiarmid's bounded differences inequality

Let $Z_1, \dots, Z_m$ be independent random variables and let $f : \mathcal{Z}^m \to \mathbb{R}$ satisfy the **bounded differences** property: for every $i \in [m]$ and every $z_1, \dots, z_m, z_i' \in \mathcal{Z}$,

$$\big|f(z_1, \dots, z_i, \dots, z_m) - f(z_1, \dots, z_i', \dots, z_m)\big| \le c_i.$$

Then for any $t > 0$,

$$P\big(|f(Z_1, \dots, Z_m) - \mathbb{E}[f(Z_1, \dots, Z_m)]| \ge t\big) \le 2\exp\Big(\frac{-2t^2}{\sum_{i=1}^m c_i^2}\Big).$$

Proof. The argument is the Chernoff route of Theorem 4.4, applied not to the summands themselves — there are none — but to an artificial decomposition of f into m increments, one per input variable. Write $Z_{1:i} := (Z_1, \dots, Z_i)$, $F := f(Z_1, \dots, Z_m)$, and define the **increments**

$$V_i := \mathbb{E}\big[F \mid Z_{1:i}\big] - \mathbb{E}\big[F \mid Z_{1:i-1}\big], \qquad i = 1, \dots, m,$$

with the convention $\mathbb{E}[F \mid Z_{1:0}] := \mathbb{E}[F]$. The sum telescopes exactly:

$$\sum_{i=1}^m V_i = \mathbb{E}[F \mid Z_{1:m}] - \mathbb{E}[F] = F - \mathbb{E}[F].$$

Step 1: each increment has conditional mean zero and conditional range at most c_i. The tower property gives $\mathbb{E}[V_i \mid Z_{1:i-1}] = 0$. For the range, define

$$U_i := \sup_z \ \mathbb{E}\big[F \mid Z_{1:i-1}, Z_i = z\big] - \mathbb{E}[F \mid Z_{1:i-1}],$$
$$L_i := \inf_z \ \mathbb{E}\big[F \mid Z_{1:i-1}, Z_i = z\big] - \mathbb{E}[F \mid Z_{1:i-1}],$$

both functions of $Z_{1:i-1}$ alone, so that $L_i \le V_i \le U_i$ by construction. For any two values z, z', independence of the Z's lets us average the bounded-differences hypothesis over the *remaining* variables $Z_{i+1}, \dots, Z_m$:

$$\big|\mathbb{E}[F \mid Z_{1:i-1}, Z_i = z] - \mathbb{E}[F \mid Z_{1:i-1}, Z_i = z']\big| \le \mathbb{E}\big[\,\big|f(\dots, z, \dots) - f(\dots, z', \dots)\big|\,\big] \le c_i,$$

hence $U_i - L_i \le c_i$: conditionally on the past, V_i is a mean-zero random variable confined to an interval of width at most c_i.

Step 2: conditional Hoeffding's lemma, iterated. Lemma 4.1 applies verbatim to conditional expectations, so for every $s \in \mathbb{R}$,

$$\mathbb{E}\big[e^{sV_i} \mid Z_{1:i-1}\big] \le e^{s^2 c_i^2/8}.$$

Peeling the increments off one at a time with the tower property,

$$\mathbb{E}\big[e^{s\sum_{i=1}^m V_i}\big] = \mathbb{E}\Big[e^{s\sum_{i<m} V_i}\ \underbrace{\mathbb{E}\big[e^{sV_m} \mid Z_{1:m-1}\big]}_{\le\, e^{s^2 c_m^2/8}}\Big] \le \cdots \le \exp\Big(\frac{s^2}{8}\sum_{i=1}^m c_i^2\Big).$$

Step 3: Chernoff. Exactly as in Theorem 4.4, Markov's inequality on $e^{s(F-\mathbb{E}[F])}$ gives $P(F - \mathbb{E}[F] \geq t) \leq \exp(-st + \frac{s^2}{8}\sum_i c_i^2)$ for every $s > 0$; minimizing the exponent at $s^* = 4t/\sum_i c_i^2$ yields $\exp\big(-2t^2/\sum_i c_i^2\big)$. Applying the same argument to $-f$ bounds the other tail identically, and adding the two gives the factor 2 □

Setting $m = 1$, or taking $f(z_1, \ldots, z_m) = \frac{1}{m}\sum_i z_i$ with $c_i = (b_i - a_i)/m$, recovers Hoeffding's inequality — McDiarmid's inequality is its strict generalization, obtained by replacing "sum of independent variables" with "any function that no single variable can move very far".

For the choice $f(S) := \sup_{h\in\mathcal{H}}\big(R(h) - \widehat{R}_S(h)\big)$ with a loss bounded in $[0, 1]$, changing a single training example changes $\widehat{R}_S(h)$ by at most $1/m$ for every h simultaneously, hence f has bounded differences $c_i = 1/m$, and McDiarmid's inequality applies with $\sum_i c_i^2 = 1/m$. This is exactly the role Hoeffding's inequality played for a single hypothesis in Chapter 4 — McDiarmid is its supremum-over-a-class counterpart, and it is what turns a bound on $\mathbb{E}_S[\sup_h(R(h) - \widehat{R}_S(h))]$ into the high-probability statements below.

6.2 The Symmetrization Technique

The remaining question is how to bound $\mathbb{E}_S\big[\sup_{h\in\mathcal{H}}(R(h) - \widehat{R}_S(h))\big]$ itself. Every complexity measure in this chapter is obtained by the same device: introduce an independent **ghost sample** $S' = \{(x_i', y_i')\}_{i=1}^m$, drawn i.i.d. from $\mathcal{D}$ exactly like S, and note that $R(h) = \mathbb{E}_{S'}[\widehat{R}_{S'}(h)]$. Then, since sup of an expectation is at most the expectation of the sup (Jensen),

$$\begin{aligned}
\mathbb{E}_S\Big[\sup_{h\in\mathcal{H}}\big(R(h) - \widehat{R}_S(h)\big)\Big] &= \mathbb{E}_S\Big[\sup_{h\in\mathcal{H}}\mathbb{E}_{S'}\big[\widehat{R}_{S'}(h) - \widehat{R}_S(h)\big]\Big] \\
&\leq \mathbb{E}_{S,S'}\Big[\sup_{h\in\mathcal{H}}\big(\widehat{R}_{S'}(h) - \widehat{R}_S(h)\big)\Big] \\
&= \mathbb{E}_{S,S'}\Big[\sup_{h\in\mathcal{H}}\frac{1}{m}\sum_{i=1}^m\big(\ell(y_i', h(x_i')) - \ell(y_i, h(x_i))\big)\Big].
\end{aligned}$$

The last expression is symmetric in the pairing $(x_i, y_i) \leftrightarrow (x_i', y_i')$: since S and S' are i.i.d. and independent of each other, swapping the i-th pair of S with the i-th pair of S' leaves the joint distribution of (S, S') unchanged. Introducing an independent Rademacher variable $\sigma_i \in \{-1, +1\}$ (each sign equally likely) to record whether the i-th pair *was* swapped therefore leaves the expectation unchanged for any choice of signs, hence

$$\begin{aligned}
\mathbb{E}_{S,S'}\Big[\sup_h\frac{1}{m}\sum_i\big(\ell(y_i', h(x_i')) - \ell(y_i, h(x_i))\big)\Big] &= \mathbb{E}_{S,S',\sigma}\Big[\sup_h\frac{1}{m}\sum_i\sigma_i\big(\ell(y_i', h(x_i')) - \ell(y_i, h(x_i))\big)\Big] \\
&\leq 2\,\mathbb{E}_{S,\sigma}\Big[\sup_h\frac{1}{m}\sum_i\sigma_i\ell(y_i, h(x_i))\Big],
\end{aligned}$$

where the last step splits the supremum of a sum into the sum of two suprema (each bounded by the original, by symmetry of $\pm\sigma_i$). This is **symmetrization**, and the quantity on the right is, by definition, twice the (expected) Rademacher complexity of the loss class $\ell \circ \mathcal{H}$.

6.3 Rademacher Complexity

We follow the formulation of (Bartlett and Mendelson, 2002).

Definition 6.3.1: Rademacher complexity

For a set $A \subset \mathbb{R}^m$ and i.i.d. Rademacher signs $\sigma_1, \dots, \sigma_m \in \{-1, +1\}$ ($P(\sigma_i = 1) = P(\sigma_i = -1) = \frac{1}{2}$), define

$$\widehat{\mathfrak{R}}(A) := \mathbb{E}_\sigma\Big[\sup_{a \in A} \frac{1}{m} \sum_{i=1}^{m} \sigma_i a_i\Big].$$

For a hypothesis class $\mathcal{H}$ and a fixed sample $S = \{(x_i, y_i)\}_{i=1}^m$, write $\ell \circ \mathcal{H} \circ S := \big\{(\ell(y_1, h(x_1)), \dots, \ell(y_m, h(x_m))) : h \in \mathcal{H}\big\} \subset \mathbb{R}^m$ for the loss class evaluated on S, and define the **empirical Rademacher complexity** $\widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H}) := \widehat{\mathfrak{R}}(\ell \circ \mathcal{H} \circ S)$ — a data-dependent quantity, computable from S alone — and the **(expected) Rademacher complexity** $\mathfrak{R}_m(\ell \circ \mathcal{H}) := \mathbb{E}_S\big[\widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H})\big]$. When we want the complexity of the hypothesis class itself rather than of the loss class, we write $\widehat{\mathfrak{R}}_S(\mathcal{H}) := \widehat{\mathfrak{R}}(\mathcal{H}_S)$, evaluated on the restriction $\mathcal{H}_S$ of Definition 5.9.

Intuitively, $\widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H})$ measures how well $\mathcal{H}$ can fit **pure random noise**: σ_i is an independent coin flip uncorrelated with everything, so $\sup_h \frac{1}{m}\sum_i \sigma_i \ell(y_i, h(x_i))$ asks how large a spurious correlation the richest hypothesis in $\mathcal{H}$ can manufacture with random labels. A class rich enough to realize any loss vector in $\{0,1\}^m$ (as in the memorization hypothesis of Chapter 1) can match every sign, giving $\widehat{\mathfrak{R}}_S \approx \frac{1}{2}$. A class with a single hypothesis has no supremum to exploit, so $\widehat{\mathfrak{R}}_S = \frac{1}{m}\sum_i \mathbb{E}[\sigma_i]\, a_i = 0$ exactly.

Theorem 6.3.2: Rademacher generalization bound

Let the loss be bounded in $[0,1]$. For any $\delta \in (0,1)$, with probability at least $1-\delta$ over $S \sim \mathcal{D}^m$, simultaneously for all $h \in \mathcal{H}$:

$$R(h) \le \widehat{R}_S(h) + 2\,\widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H}) + 3\sqrt{\frac{\log(2/\delta)}{2m}}.$$

Proof. Write $\Phi(S) := \sup_{h \in \mathcal{H}} \big(R(h) - \widehat{R}_S(h)\big)$, so the claim is a high-probability upper bound on $\Phi(S)$. The proof applies McDiarmid's inequality twice, with symmetrization in between.

Step 1: concentrate Φ around its mean. Replacing one example of S changes $\widehat{R}_S(h)$ by at most $1/m$ for every h at once, hence changes the supremum by at most $1/m$: Φ has bounded differences $c_i = 1/m$, so $\sum_i c_i^2 = 1/m$. The one-sided form of Theorem 5a.1 at confidence $\delta/2$ gives, with probability at least $1 - \delta/2$,

$$\Phi(S) \;\le\; \mathbb{E}_S[\Phi(S)] + \sqrt{\frac{\log(2/\delta)}{2m}}.$$

Step 2: bound the mean by the expected Rademacher complexity. This is the symmetrization argument of the previous section, verbatim: $\mathbb{E}_S[\Phi(S)] \le 2\,\mathfrak{R}_m(\ell \circ \mathcal{H})$.

Step 3: replace the expected complexity by its empirical counterpart. The map $S \mapsto \widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H})$ also has bounded differences $1/m$: changing one example changes each coordinate of the loss vector by at most 1, hence changes $\frac{1}{m}\sum_i \sigma_i a_i$ by at most $1/m$ uniformly, and a supremum of functions that all move by at most $1/m$ moves by at most $1/m$. Applying Theorem 5a.1 once more at confidence $\delta/2$, with probability at least $1 - \delta/2$,

$$\mathfrak{R}_m(\ell \circ \mathcal{H}) = \mathbb{E}_S\big[\widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H})\big] \;\le\; \widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H}) + \sqrt{\frac{\log(2/\delta)}{2m}}.$$

Step 4: union bound. Both events hold simultaneously with probability at least $1-\delta$, and on their intersection

$$\sup_h \big(R(h) - \widehat{R}_S(h)\big) \le 2\,\widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H}) + 2\sqrt{\frac{\log(2/\delta)}{2m}} + \sqrt{\frac{\log(2/\delta)}{2m}},$$

where the factor 2 from Step 2 multiplies the deviation of Step 3. Rearranging the supremum gives the claim for every h □

This is the Rademacher-complexity instance of the abstract recipe stated at the top of this chapter, with $\text{Complexity}(\mathcal{H}, m) = 2\widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H})$.

6.3.1 Massart's Lemma, and Theorem 5.1 revisited

Lemma 6.3.3: Massart's lemma

For a finite set $A = \{a^{(1)}, \dots, a^{(N)}\} \subset \mathbb{R}^m$ with $\|a^{(j)}\|_2 \le B$ for all j,

$$\widehat{\mathfrak{R}}(A) \le \frac{B\sqrt{2\log N}}{m},$$
$$\text{equivalently} \quad \mathbb{E}_\sigma\Big[\max_{j\in[N]} \langle \sigma, a^{(j)}\rangle\Big] \le B\sqrt{2\log N}.$$

Proof. Write $X_j := \langle \sigma, a^{(j)}\rangle = \sum_{i=1}^m \sigma_i a_i^{(j)}$ and assume $N \ge 2$ (for $N = 1$ the left side is $\mathbb{E}_\sigma[X_1] = 0$ and there is nothing to prove). Fix $s > 0$. Because $u \mapsto e^{su}$ is convex and increasing, Jensen's inequality and then the crude bound "a maximum of non-negative terms is at most their sum" give

$$\exp\Big(s\,\mathbb{E}_\sigma\big[\max_j X_j\big]\Big) \;\le\; \mathbb{E}_\sigma\Big[e^{s\max_j X_j}\Big] = \mathbb{E}_\sigma\Big[\max_j e^{sX_j}\Big] \;\le\; \sum_{j=1}^N \mathbb{E}_\sigma\big[e^{sX_j}\big].$$

Each X_j is a sum of m independent, mean-zero terms $\sigma_i a_i^{(j)}$, the i-th confined to the interval $[-|a_i^{(j)}|, +|a_i^{(j)}|]$ of width $2|a_i^{(j)}|$. Hoeffding's lemma (Lemma 4.1) applied to each term and independence give

$$\mathbb{E}_\sigma\big[e^{sX_j}\big] = \prod_{i=1}^{m} \mathbb{E}\big[e^{s\sigma_i a_i^{(j)}}\big] \le \prod_{i=1}^{m} \exp\Big(\frac{s^2(2a_i^{(j)})^2}{8}\Big) = \exp\Big(\frac{s^2\|a^{(j)}\|_2^2}{2}\Big) \le \exp\Big(\frac{s^2B^2}{2}\Big).$$

Substituting and taking logarithms, $s\,\mathbb{E}_\sigma[\max_j X_j] \le \log N + \frac{s^2B^2}{2}$, i.e.

$$\mathbb{E}_\sigma\big[\max_j X_j\big] \le \frac{\log N}{s} + \frac{sB^2}{2}$$
$$\text{for every } s > 0.$$

The right-hand side is minimized at $s^* = \sqrt{2\log N}/B$, where its value is $B\sqrt{2\log N}$. Dividing by m turns the left-hand side into $\widehat{\mathfrak{R}}(A)$ □

Note the shape of the result: the dependence on the number of elements is only $\sqrt{\log N}$, which is what makes the union-bound-style arguments of Chapter 5 as cheap as they are. Note also that the bound is *not* improved by adding a point to A that is far from the others — only the count and the worst-case norm matter.

Apply this to $\ell \circ \mathcal{H} \circ S$ for a **finite** hypothesis class, $N = |\mathcal{H}|$, with loss bounded in $[0,1]$ so $\|a^{(j)}\|_2 \le \sqrt{m}$: this gives $\widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H}) \le \sqrt{2\log|\mathcal{H}|/m}$, and plugging into Theorem 5a.2 recovers, up to constants, exactly the finite-hypothesis-class bound of Theorem 5.1. **The very first generalization bound of Chapter 5 was already an instance of a Rademacher-complexity bound** — we simply proved it directly via Hoeffding and the union bound there, without naming the general machinery.

6.3.2 The Contraction Lemma

Lemma 6.3.4: Contraction

If $\phi_i : \mathbb{R} \to \mathbb{R}$ is ρ-Lipschitz for each i, then $\widehat{\mathfrak{R}}\big(\{(\phi_1(a_1), \dots, \phi_m(a_m)) : a \in A\}\big) \le \rho\,\widehat{\mathfrak{R}}(A)$.

Proof. The whole content is a **one-coordinate** statement, which we prove first and then apply m times:

> **One-coordinate step.** Let $\psi_1, \dots, \psi_{m-1}$ be arbitrary functions and ϕ be ρ-Lipschitz. Then
>
> $$\mathbb{E}_\sigma\Big[\sup_{a\in A}\Big(\sum_{i<m}\sigma_i\psi_i(a_i) + \sigma_m\phi(a_m)\Big)\Big] \;\le\; \mathbb{E}_\sigma\Big[\sup_{a\in A}\Big(\sum_{i<m}\sigma_i\psi_i(a_i) + \sigma_m\,\rho\,a_m\Big)\Big].$$

Granting this, apply it to coordinate m (with $\psi_i := \phi_i$), replacing ϕ_m by $\rho \cdot \mathrm{id}$; then to coordinate $m-1$ of the resulting set, and so on. After m steps every ϕ_i has been replaced by $\rho \cdot \mathrm{id}$, so the left-hand side of the lemma is at most $\widehat{\mathfrak{R}}(\{\rho a : a \in A\}) = \rho\,\widehat{\mathfrak{R}}(A)$, the last

equality by homogeneity of the definition. Each step transforms a *different* coordinate, so the factor ρ is collected once in total, not m times.

To prove the one-coordinate step, condition on $\sigma_1, \dots, \sigma_{m-1}$ and abbreviate $u(a) := \sum_{i<m} \sigma_i \psi_i(a_i)$, which does not involve the m-th coordinate's transform. Averaging over the two equally likely values of σ_m,

$$\begin{aligned}
\mathbb{E}_{\sigma_m}\Big[\sup_{a\in A}\big(u(a)+\sigma_m\phi(a_m)\big)\Big] &= \tfrac{1}{2}\sup_{a\in A}\big(u(a)+\phi(a_m)\big) + \tfrac{1}{2}\sup_{a'\in A}\big(u(a')-\phi(a'_m)\big) \\
&= \tfrac{1}{2}\sup_{a,a'\in A}\Big(u(a)+u(a')+\phi(a_m)-\phi(a'_m)\Big) \\
&\le \tfrac{1}{2}\sup_{a,a'\in A}\Big(u(a)+u(a')+\rho\,\big|a_m-a'_m\big|\Big) \\
&= \tfrac{1}{2}\sup_{a,a'\in A}\Big(u(a)+u(a')+\rho\,\big(a_m-a'_m\big)\Big) \\
&= \mathbb{E}_{\sigma_m}\Big[\sup_{a\in A}\big(u(a)+\sigma_m\rho\,a_m\big)\Big],
\end{aligned}$$

where the second line merges the two independent suprema into one over the pair (a, a'); the third applies the ρ-Lipschitz property of ϕ; the fourth drops the absolute value because the expression is symmetric under swapping $a \leftrightarrow a'$, so the supremum is attained at a pair for which $a_m - a'_m \ge 0$ anyway; and the fifth reverses the first two steps with $\rho\, a_m$ in place of $\phi(a_m)$. Taking the expectation over $\sigma_1, \dots, \sigma_{m-1}$ gives the one-coordinate step □

Note where the argument would break for the *signed* variant $\sup_a |\frac{1}{m}\sum_i \sigma_i a_i|$: the second line merges two suprema into one, which is valid for a supremum of a set but not for a supremum of absolute values — this is the source of the extra factor of 2 mentioned below.

This lets us pass, in the definition of $\ell \circ \mathcal{H}$, from the Rademacher complexity of the *loss-composed* class directly to the Rademacher complexity of $\mathcal{H}$ itself, whenever the loss $\ell(\cdot, y)$ is ρ-Lipschitz in its first argument: $\widehat{\mathfrak{R}}_S(\ell \circ \mathcal{H}) \le \rho\, \widehat{\mathfrak{R}}_S(\mathcal{H})$. A word of caution, since this is a place where different textbooks state slightly different constants: the *clean* version above, with no extra factor, holds for the **unsigned** Rademacher complexity (where the supremum in Definition 5a.1 is *not* wrapped in an absolute value, exactly as we defined it). If instead one works with the signed variant $\sup_a |\frac{1}{m}\sum_i \sigma_i a_i|$, the contraction inequality picks up an extra factor of 2, and the standard proof of even that weaker bound requires $\phi_i(0) = 0$ for all i. Since our Definition 5a.1 already uses the unsigned form throughout, the clean statement above applies directly to everything in this chapter.

6.3.3 The Bridge to VC Dimension

Theorem 6.3.5: VC dimension bounds Rademacher complexity

For a hypothesis class $\mathcal{H}$ of binary classifiers with $d_{VC}(\mathcal{H}) = d < \infty$, and the zero-one loss,

$$\widehat{\mathfrak{R}}_S(\mathcal{H}) \leq 2\sqrt{\frac{d\log(m+1)}{m}},$$

for every sample S of size m.

Proof. The restriction $\mathcal{H}_S = \{(h(x_1), \ldots, h(x_m)) : h \in \mathcal{H}\}$ is a *finite* set (Definition 5.9), of size $|\mathcal{H}_S| = \tau_{\mathcal{H}}(m) \leq (m+1)^d$ by the Sauer-Shelah-Perles lemma (Lemma 5.2, using the simplified polynomial form). Apply Massart's lemma (Lemma 5a.1) with $N = (m+1)^d$ and $B = \sqrt{m}$ (binary values in $\{0,1\}$, so $\|a^{(j)}\|_2 \leq \sqrt{m}$):

$$\widehat{\mathfrak{R}}(\mathcal{H}_S) \leq \frac{\sqrt{m}\sqrt{2\log\left((m+1)^d\right)}}{m} = \sqrt{\frac{2d\log(m+1)}{m}} \leq 2\sqrt{\frac{d\log(m+1)}{m}},$$

the last step simply using $\sqrt{2} \leq 2$, which is where the slack in the stated constant comes from. □

This is the promised unification made concrete: **VC dimension is not a separate theory from Rademacher complexity — it is a specific, combinatorial way of bounding it**, via Sauer-Shelah-Perles turning an infinite hypothesis class into a finite restriction on any given sample, to which Massart's finite-class bound then applies.

6.3.4 A VC-Dimension-Free Example: Linear Predictors

One reason Rademacher complexity is worth the extra abstraction: it can be **finite and small even when** $d_{VC}(\mathcal{H}) = \infty$, provided we constrain the hypothesis class by a *norm* rather than by dimension.

Theorem 6.3.6

Let $\mathcal{H} = \{x \mapsto w^\top x : \|w\|_2 \leq B\}$ and suppose $\|x\|_2 \leq R$ almost surely under $\mathcal{D}$. Then

$$\widehat{\mathfrak{R}}_S(\mathcal{H}) \leq \frac{BR}{\sqrt{m}}.$$

Proof. By definition and linearity, for a fixed realization of the signs,

$$\sup_{\|w\|_2 \leq B} \frac{1}{m}\sum_{i=1}^m \sigma_i\, w^\top x_i = \frac{1}{m}\sup_{\|w\|_2 \leq B}\left\langle w, \sum_{i=1}^m \sigma_i x_i\right\rangle = \frac{B}{m}\left\|\sum_{i=1}^m \sigma_i x_i\right\|_2,$$

where the second equality is the equality case of the Cauchy-Schwarz inequality: the linear functional $w \mapsto \langle w, v\rangle$ is maximized over the ball $\|w\|_2 \le B$ at $w = B\,v/\|v\|_2$, with value $B\|v\|_2$. Taking $\mathbb{E}_\sigma$ and applying Jensen's inequality to the concave map $u \mapsto \sqrt{u}$,

$$\widehat{\mathfrak{R}}_S(\mathcal{H}) = \frac{B}{m}\,\mathbb{E}_\sigma\Big[\Big\|\sum_i \sigma_i x_i\Big\|_2\Big] \le \frac{B}{m}\sqrt{\mathbb{E}_\sigma\Big[\Big\|\sum_i \sigma_i x_i\Big\|_2^2\Big]}.$$

Expanding the squared norm and using $\mathbb{E}[\sigma_i\sigma_{i'}] = \mathbb{1}(i = i')$ (the signs are independent and mean-zero, and $\sigma_i^2 = 1$),

$$\mathbb{E}_\sigma\Big[\Big\|\sum_i \sigma_i x_i\Big\|_2^2\Big] = \sum_{i,i'} \mathbb{E}[\sigma_i\sigma_{i'}]\, x_i^\top x_{i'} = \sum_{i=1}^m \|x_i\|_2^2 \le mR^2.$$

Combining, $\widehat{\mathfrak{R}}_S(\mathcal{H}) \le \frac{B}{m}\sqrt{mR^2} = \frac{BR}{\sqrt{m}}$ □

Notice this bound does not depend on the input dimension d at all — only on the norm bound B and the data norm bound R. This matters directly for the kernel methods of Chapter 9, where $\phi(x)$ can live in an infinite-dimensional feature space (for which d_{VC} is typically infinite) while $\|w\|_2$ and $\|\phi(x)\|_2$ remain finite: Rademacher complexity, unlike VC dimension, gives a meaningful, finite generalization guarantee in exactly this regime.

6.4 Covering Numbers

Rademacher complexity measures $\mathcal{H}$'s capacity to correlate with noise. **Covering numbers** measure something more literal: how many hypotheses it takes to approximate every hypothesis in $\mathcal{H}$ to a given tolerance.

Definition 6.4.1: ϵ-cover, covering number

Let $\mathcal{H}$ be a hypothesis class equipped with a (pseudo-)metric $\mathrm{dist}(\cdot,\cdot)$, in the sense of Chapter 2. A finite set $\{h_1, \dots, h_N\} \subset \mathcal{H}$ is an **ϵ-cover** of $\mathcal{H}$ if for every $h \in \mathcal{H}$ there exists $j \in [N]$ with $\mathrm{dist}(h, h_j) \le \epsilon$. The **covering number** $N(\epsilon, \mathcal{H}, \mathrm{dist})$ is the size of the smallest such cover.

In words, an ϵ-cover is a finite "grid" of representative hypotheses such that every hypothesis in $\mathcal{H}$, however large or infinite the class, is within ϵ of some grid point — turning an infinite class into a finite proxy at the cost of an ϵ-sized approximation error.

The relevant metric for generalization is one under which the risk is Lipschitz: if $\ell(\cdot, y)$ is G-Lipschitz in its first argument for every y, and $\mathrm{dist}(h, h') := \mathbb{E}_{x\sim\mathcal{D}}|h(x) - h'(x)|$, then $|R(h) - R(h')| \le G\,\mathrm{dist}(h, h')$ for all $h, h' \in \mathcal{H}$ — replacing h by the nearest cover element changes the risk by only $G\epsilon$.

Theorem 6.4.2: Covering-number generalization bound

Let $\{h_1, \dots, h_{N(\epsilon)}\}$ be an ϵ-cover of $\mathcal{H}$ under dist as above, with loss bounded in $[0, 1]$. Then, for any $h \in \mathcal{H}$, choosing h_j within ϵ of h,

$$\begin{aligned}\left|\widehat{R}_S(h) - R(h)\right| &\le \left|\widehat{R}_S(h) - \widehat{R}_S(h_j)\right| + \left|\widehat{R}_S(h_j) - R(h_j)\right| + \left|R(h_j) - R(h)\right| \\ &\le 2G\epsilon + \left|\widehat{R}_S(h_j) - R(h_j)\right|,\end{aligned}$$

using the Lipschitz bound on both the empirical and true risk. Applying the finite-hypothesis-class bound (Theorem 5.1) to the right-hand side, over the $N(\epsilon)$ cover centers, gives: with probability at least $1 - \delta$, simultaneously for all $h \in \mathcal{H}$,

$$R(h) \le \widehat{R}_S(h) + 2G\epsilon + \sqrt{\frac{\log(2N(\epsilon)/\delta)}{2m}}.$$

Proof. The displayed chain of inequalities is the triangle inequality for $|\cdot|$, applied twice, with $|\widehat{R}_S(h) - \widehat{R}_S(h_j)| \le G\epsilon$ and $|R(h) - R(h_j)| \le G\epsilon$ from the Lipschitz property just established (the same argument bounds the empirical risk, since $\widehat{R}_S$ is the risk under the empirical distribution and the Lipschitz bound holds pointwise in x). The remaining term $|\widehat{R}_S(h_j) - R(h_j)|$ concerns only the $N(\epsilon)$ **fixed, data-independent** cover centers, so Theorem 5.1 applies to the finite class $\{h_1, \dots, h_{N(\epsilon)}\}$ and bounds it, uniformly over j, by $\sqrt{\log(2N(\epsilon)/\delta)/(2m)}$ with probability at least $1 - \delta$ □

This is exactly the finite-hypothesis-class recipe of Theorem 5.1, applied not to $\mathcal{H}$ itself but to a *finite proxy* for it — the ϵ-cover — with an extra $2G\epsilon$ term paying for the approximation error of using the proxy.

Lemma 6.4.3: Volumetric covering bound

Let $\mathcal{B}_B := \{v \in \mathbb{R}^d : \|v\|_2 \le B\}$ with the Euclidean metric. Then $N(\epsilon, \mathcal{B}_B, \|\cdot\|_2) \le \big(1 + 2B/\epsilon\big)^d$.

Proof. Let $\{v_1, \dots, v_N\} \subseteq \mathcal{B}_B$ be a **maximal ϵ-separated** set, i.e. $\|v_j - v_{j'}\|_2 > \epsilon$ for $j \ne j'$ and no further point of $\mathcal{B}_B$ can be added without violating this. Maximality means every $v \in \mathcal{B}_B$ is within ϵ of some v_j — otherwise it could be added — so the set is an ϵ-cover and $N(\epsilon, \mathcal{B}_B, \|\cdot\|_2) \le N$.

To bound N, note the open balls of radius $\epsilon/2$ around the v_j are pairwise disjoint (their centers are more than ϵ apart) and all contained in $\mathcal{B}_{B+\epsilon/2}$. Since the volume of a Euclidean ball of radius r in $\mathbb{R}^d$ is $V_d\, r^d$ for a constant V_d depending only on d, comparing volumes gives

$$N \cdot V_d\Big(\frac{\epsilon}{2}\Big)^d \le V_d\Big(B + \frac{\epsilon}{2}\Big)^d \quad \Longrightarrow \quad N \le \Big(\frac{B + \epsilon/2}{\epsilon/2}\Big)^d = \Big(1 + \frac{2B}{\epsilon}\Big)^d. \qquad \square$$

The exponent d is the entire story: covering numbers grow *exponentially* in the dimension, so $\log N(\epsilon) = O(d \log(1/\epsilon))$ grows only linearly in it — and it is $\log N(\epsilon)$, not $N(\epsilon)$, that

enters the bound. This is the same phenomenon as the logarithmic dependence on $|\mathcal{H}|$ in Theorem 5.1, and the reason both bounds remain useful in high dimension. □

Balancing $\epsilon \propto 1/\sqrt{m}$ against the second term yields an overall rate of $O\big(\sqrt{(d/m)\log m}\big)$ — the same $\sqrt{d/m}$ rate as the VC/Rademacher bounds above, up to a $\sqrt{\log m}$ factor. That residual log factor can be removed by not fixing a single resolution ϵ but summing the bound over a geometric sequence of resolutions $\epsilon_k = 2^{-k}$ — a technique called **chaining**, which produces the sharper **Dudley entropy integral** bound $\widehat{\mathfrak{R}}_S(\mathcal{H}) \lesssim \frac{1}{\sqrt{m}} \int_0^\infty \sqrt{\log N(\epsilon, \mathcal{H}, \mathrm{dist})}\, d\epsilon$. Chaining is beyond the scope of this course; it is mentioned here only so that the $\sqrt{\log m}$ gap between the simple covering bound above and the sharp VC-dimension rate of Theorem 5a.3 does not look like an unexplained discrepancy.

6.5 PAC-Bayes Bounds

The bounds so far all fix a hypothesis class $\mathcal{H}$ first, and then ask for a guarantee that holds *uniformly*, for every $h \in \mathcal{H}$ at once — this is what makes it valid to then plug in whichever h our learning algorithm happens to output. **PAC-Bayes bounds** (McAllester, 1999) relax this: instead of a single hypothesis, they bound the risk of a whole *distribution* over hypotheses, and let that distribution depend on the data.

Setup. Fix a **prior** distribution q over $\mathcal{H}$, chosen *before* seeing S (it may encode domain knowledge, or simply be uniform/uninformative). After seeing S, we are free to choose any **posterior** distribution ρ over $\mathcal{H}$ — informed by the data in any way whatsoever, including via an algorithm's output — and consider the *randomized* predictor that draws a fresh $h \sim \rho$ at prediction time. Define $R(\rho) := \mathbb{E}_{h\sim\rho}[R(h)]$ and $\widehat{R}_S(\rho) := \mathbb{E}_{h\sim\rho}[\widehat{R}_S(h)]$.

Theorem 6.5.1: PAC-Bayes bound

Assume the loss is bounded in $[0,1]$. For any prior q fixed before S, any $\beta > 0$, and any $\delta \in (0,1)$, with probability at least $1-\delta$ over $S \sim \mathcal{D}^m$, **simultaneously for every posterior** ρ (chosen after seeing S, in any way):

$$R(\rho) \le \widehat{R}_S(\rho) + \frac{1}{\beta}\,\mathrm{KL}(\rho\|q) + \frac{1}{\beta}\log\frac{1}{\delta} + \frac{\beta}{8m}.$$

Proof. Fix $\beta > 0$ and define, for each hypothesis, the random variable $X(h) := \beta\big(R(h) - \widehat{R}_S(h)\big)$, a function of S.

Step 1: a moment generating function bound for a single, fixed h. Write $\ell_i := \ell(y_i, h(x_i)) \in [0,1]$, independent across i with mean $R(h)$. Then $X(h) = \sum_{i=1}^m \frac{\beta}{m}\big(R(h) - \ell_i\big)$ is a sum of m independent, mean-zero terms, the i-th confined to an interval of width β/m. Hoeffding's lemma (Lemma 4.1) and independence give

$$\mathbb{E}_S\big[e^{X(h)}\big] \le \prod_{i=1}^m \exp\left(\frac{(\beta/m)^2}{8}\right) = \exp\left(\frac{\beta^2}{8m}\right).$$

Step 2: average over the prior, then apply Markov. The bound of Step 1 holds for every h, and q does not depend on S, so the two expectations may be exchanged:

$$\mathbb{E}_S\Big[\mathbb{E}_{h\sim q}\big[e^{X(h)}\big]\Big] = \mathbb{E}_{h\sim q}\Big[\mathbb{E}_S\big[e^{X(h)}\big]\Big] \le e^{\beta^2/(8m)}.$$

The inner quantity $\mathbb{E}_{h\sim q}[e^{X(h)}]$ is a non-negative random variable of S, so Markov's inequality (Theorem 4.1) gives: with probability at least $1-\delta$ over S,

$$\mathbb{E}_{h\sim q}\big[e^{X(h)}\big] \;\le\; \frac{1}{\delta}\, e^{\beta^2/(8m)}. \qquad (*)$$

Note carefully that $(*)$ involves **only the prior** q, so it holds on a single event of probability $1-\delta$ that is fixed before any posterior is chosen. This is the entire trick of PAC-Bayes: all the probabilistic work is done under q, and the data-dependent ρ is introduced afterwards, deterministically, on that same event.

Step 3: the Donsker-Varadhan change of measure. For any distribution ρ absolutely continuous with respect to q, and any function X,

$$\mathbb{E}_{h\sim\rho}\big[X(h)\big] \;\le\; \mathrm{KL}(\rho\|q) + \log \mathbb{E}_{h\sim q}\big[e^{X(h)}\big].$$

To see this, define the **tilted** distribution $\tilde{q}(h) := q(h)e^{X(h)}/\mathbb{E}_{h\sim q}[e^{X(h)}]$, which is a valid probability distribution by construction. Expanding the Kullback-Leibler divergence of ρ from $\tilde{q}$,

$$\mathrm{KL}(\rho\|\tilde{q}) = \mathbb{E}_{h\sim\rho}\Big[\log\frac{\rho(h)}{\tilde{q}(h)}\Big] = \underbrace{\mathbb{E}_{h\sim\rho}\Big[\log\frac{\rho(h)}{q(h)}\Big]}_{=\,\mathrm{KL}(\rho\|q)} - \mathbb{E}_{h\sim\rho}\big[X(h)\big] + \log\mathbb{E}_{h\sim q}\big[e^{X(h)}\big],$$

so the claimed inequality is exactly the statement $\mathrm{KL}(\rho\|\tilde{q}) \ge 0$, which holds for any pair of distributions by Jensen's inequality applied to the convex function $u \mapsto -\log u$.

Step 4: combine. On the event of $(*)$, for **every** ρ simultaneously,

$$\beta\big(R(\rho) - \widehat{R}_S(\rho)\big) = \mathbb{E}_{h\sim\rho}\big[X(h)\big] \le \mathrm{KL}(\rho\|q) + \log\Big(\tfrac{1}{\delta}\, e^{\beta^2/(8m)}\Big) = \mathrm{KL}(\rho\|q) + \log\tfrac{1}{\delta} + \frac{\beta^2}{8m},$$

the first equality by linearity of $\mathbb{E}_\rho$ together with the definitions $R(\rho) := \mathbb{E}_\rho[R(h)]$ and $\widehat{R}_S(\rho) := \mathbb{E}_\rho[\widehat{R}_S(h)]$. Dividing by $\beta > 0$ gives the theorem □

The parameter β — an **inverse temperature** — trades off the two error terms and can be optimized; we do this explicitly below. The minimizer of the right-hand side over ρ is the **Gibbs posterior** $\widehat{\rho}_\beta(h) \propto \exp(-\beta\,\widehat{R}_S(h))\, q(h)$, i.e. exactly the tilted distribution $\tilde{q}$ of Step 3 for the choice $X = -\beta\widehat{R}_S$ — note that $\widehat{\rho}_\beta$ coincides with the *Bayesian* posterior of Chapter 6 exactly when $\beta = m$; other choices of β are read as different inference temperatures.

6.5.1 Recovering Theorem 5.1 as a Special Case

Take $\mathcal{H} = \{h_1, \ldots, h_N\}$ finite, the **uniform prior** $q(h_j) = 1/N$, and restrict attention to **Dirac posteriors** $\rho = \delta_h$ that place all their mass on a single hypothesis $h \in \mathcal{H}$ — precisely the deterministic predictors we have used throughout this course. Then $R(\rho) = R(h)$, $\widehat{R}_S(\rho) = \widehat{R}_S(h)$, and $\mathrm{KL}(\delta_h \| q) = \log(1/q(h)) = \log N$, so Theorem 5a.6 gives: with probability $\geq 1 - \delta$, simultaneously for **every** $h \in \mathcal{H}$,

$$R(h) \leq \widehat{R}_S(h) + \frac{\log N}{\beta} + \frac{1}{\beta} \log \frac{1}{\delta} + \frac{\beta}{8m},$$
$$\forall \beta > 0.$$

Minimizing the right-hand side over β (setting the derivative in β to zero gives $\beta^* = \sqrt{8m(\log N + \log(1/\delta))}$) yields exactly

$$R(h) \leq \widehat{R}_S(h) + \sqrt{\frac{\log N + \log(1/\delta)}{2m}} = \widehat{R}_S(h) + \sqrt{\frac{\log |\mathcal{H}| + \log(1/\delta)}{2m}},$$

which is **precisely Theorem 5.1**, recovered as the special case of PAC-Bayes with a uniform prior and Dirac posteriors over a finite $\mathcal{H}$. This closes the loop promised at the start of this chapter: $|\mathcal{H}|$-counting, VC dimension, Rademacher complexity, covering numbers, and PAC-Bayes bounds are not five different theories of generalization — they are one theory, instantiated through five different choices of how to measure "the complexity of $\mathcal{H}$," and each of the first four turns out to be reachable as a special case of the last, more general one.

6.6 Beyond Complexity: What This Chapter Does Not Cover

Three further directions extend the picture above; we state them only as pointers, since a full treatment is beyond a BSc-level course.

- **Minimax lower bounds.** Every bound above is an *upper* bound: it shows some algorithm (e.g. ERM) cannot do worse than a certain rate. A **minimax lower bound** shows the converse — that *no* algorithm can do better, in the worst case over $\mathcal{D}$, than a matching rate (typically $\Omega(\sqrt{d_{VC}/m})$ under no further assumptions). Such results are proved by reducing estimation to hypothesis testing over a carefully chosen finite set of hard distributions, then invoking information-theoretic tools (Fano's inequality or its relatives). They certify that the rates derived in this chapter are not an artifact of a loose proof technique, but essentially unimprovable.
- **Margin theory and fast rates.** All bounds above scale as $O(1/\sqrt{m})$. Under a **low-noise (margin) condition** — informally, when $\mathcal{D}$ rarely places points close to the decision boundary — the *same* complexity measures support sharper $O(1/m)$ "fast rates," interpolating continuously between the two regimes as the margin condition strengthens. This refines, rather than replaces, the bias-complexity dilemma of Chapter 5: near the decision boundary is exactly where estimation error concentrates, so a

distribution that avoids that region is intrinsically easier to learn, beyond what its VC dimension alone would suggest.

- **Algorithmic stability.** Every bound in this chapter is a property of the hypothesis *class* $\mathcal{H}$, independent of which algorithm searches it. A complementary theory bounds generalization instead via the **stability** of the specific learning algorithm — informally, how much its output hypothesis can change if a single training point is swapped out — and can certify generalization even in settings (e.g. certain non-ERM algorithms, or classes with poorly behaved uniform convergence) where the complexity-based bounds of this chapter are vacuous or inapplicable. It is the standard tool, for instance, for analyzing the generalization behavior of stochastic gradient descent itself (Chapter 8), independently of the capacity of the neural network it is training.

CHAPTER 7

BAYESIAN LEARNING

7.1 Maximum Likelihood Estimation

Machine learning is the desired tool especially in real-world problems where there are factors of uncertainty and randomness. Having learned the principled way of accounting for uncertainty via probability theory, next we will see how we can use this theory to build machine learning models. Every machine learning problem has two key components: the data and the model. Let our data set be $S = \{x_1, x_2, \ldots, x_m\}$, where x_i are the observed data points. At the stage of modeling, we draw a hypothesis about how this data could have been generated. Let our hypothesis be that the data is generated as independent samples from a distribution following a parametric probability function $P(x \mid \theta)$, where θ is the parameter of the distribution. Then the joint probability of the random variables representing the occurrence of the data set is given by

$$P(S \mid \theta) = P(x_1, x_2, \ldots, x_m \mid \theta) = \prod_{i=1}^{m} P(x_i \mid \theta).$$

This quantity, viewed as a function of θ for the fixed, observed S, is called the **likelihood** of θ given the data set S. The second equality follows from the assumption that the data points are independent. The maximum likelihood estimation (MLE) is a method of estimating the parameters θ by maximizing the likelihood $P(S \mid \theta)$. This way we aim to find the parameters that are most likely to have generated the data set S. We are up to solving the optimization problem below:

$$\theta_{MLE} = \arg\max_{\theta} P(S \mid \theta) = \arg\max_{\theta} \prod_{i=1}^{m} P(x_i \mid \theta).$$

Formulating the objective as a product of terms is numerically brittle: if each factor is small, the product of many such terms underflows the floating point system. It is also harder to differentiate a product than a sum. We therefore maximize the **log-likelihood** instead. Since the logarithm is monotonically increasing, its maximizer coincides with that of the likelihood itself:

$$\theta_{MLE} = \arg\max_{\theta} \sum_{i=1}^{m} \log P(x_i \mid \theta).$$

7.1.1 Example: MLE for the Bernoulli distribution

Let $x \in \{0, 1\}$ be the outcome of a coin toss (1 for heads, 0 for tails), modeled as $P(x \mid \theta) := \theta^x (1-\theta)^{1-x}$, where $\theta \in [0, 1]$ is the probability of heads. This is the **Bernoulli distribution**, and θ is its only parameter. Given $S = \{x_1, \ldots, x_m\}$, i.i.d. samples of x, the log-likelihood is

$$\log P(S \mid \theta) = \sum_{i=1}^{m} \log \theta^{x_i}(1-\theta)^{1-x_i} = \Big(\sum_{i=1}^{m} x_i\Big) \log \theta + \Big(m - \sum_{i=1}^{m} x_i\Big) \log(1-\theta).$$

Setting the derivative with respect to θ to zero,

$$\begin{aligned} \frac{\partial}{\partial \theta} \log P(S \mid \theta) = \frac{\sum_{i=1}^{m} x_i}{\theta} - \frac{m - \sum_{i=1}^{m} x_i}{1-\theta} &\triangleq 0 \\ \Rightarrow (1-\theta) \sum_{i=1}^{m} x_i &= \theta\Big(m - \sum_{i=1}^{m} x_i\Big) \\ \Rightarrow \theta_{MLE} &= \frac{1}{m} \sum_{i=1}^{m} x_i. \end{aligned}$$

Unsurprisingly, the MLE of a Bernoulli parameter is simply the observed frequency of heads, i.e. the sample mean of S. This is the estimator we implicitly used throughout Chapter 4 whenever we wrote $\widehat{\mu}_m$ for a sample of Bernoulli random variables.

7.2 Bayesian Learning

The MLE approach commits to a single value of θ and does not represent how uncertain we are about it — an uncertainty that is naturally large when m is small (imagine estimating θ from three coin tosses) and shrinks as m grows. This ansatz, where θ is treated as an unknown but fixed constant, is called **frequentist learning**. **Bayesian learning** instead treats θ itself as a random variable, with a distribution that captures our belief about its likely values before and after seeing data.

Concretely, the modeling assumption — the **generative process** or **data generating process** — becomes

$$\begin{aligned} \theta &\sim p(\theta) \\ x_i \mid \theta &\sim \text{Bernoulli}(\theta), \quad i = 1, \ldots, m. \end{aligned}$$

We start with a **prior distribution** $p(\theta)$ that encodes our belief about θ before seeing any data, and update it to a **posterior distribution** $p(\theta \mid S)$ after observing S, via Bayes' rule:

$$p(\theta \mid S) = \frac{P(S \mid \theta)\, p(\theta)}{P(S)} = \frac{P(S \mid \theta)\, p(\theta)}{\int_0^1 P(S \mid \theta')\, p(\theta')\, d\theta'}.$$

Note the case of the letters, which is not decoration: $P(S \mid \theta)$ and $P(S)$ are **probabilities** of the observed discrete data, while $p(\theta)$ and $p(\theta \mid S)$ are **densities** over the continuous parameter — the latter are not probabilities and can exceed 1. Every mixed expression in this chapter obeys this rule, which is worth checking against as the formulas grow.

The denominator $P(S) = \int_0^1 P(S \mid \theta')p(\theta')\, d\theta'$ is the **evidence**. Two facts about it matter for the rest of this course:

- In almost every real-world model, the evidence is intractable to compute in closed form, forcing us to approximate the posterior. The whole field of Bayesian machine learning is, in large part, about finding good posterior approximations.
- The evidence measures how well the whole model family fits the data, averaged over the prior; it can therefore be used for model selection, by comparing $P(S)$ across candidate models.

The Bernoulli-parameter case is one of the rare instances that admits an exact, closed-form posterior. We choose the prior to be a **Beta distribution**, $\theta \sim \mathrm{Beta}(\alpha, \beta) \propto \theta^{\alpha-1}(1-\theta)^{\beta-1}$ for hyperparameters $\alpha, \beta > 0$, because it is **conjugate** to the Bernoulli likelihood: the posterior comes out in the same family, which makes the update mechanical.

$$\begin{aligned} p(\theta \mid S) &\propto P(S \mid \theta)\, p(\theta) \\ &= \Big[\prod_{i=1}^{m} \theta^{x_i}(1-\theta)^{1-x_i}\Big] \cdot \theta^{\alpha-1}(1-\theta)^{\beta-1} \\ &= \theta^{\sum_{i=1}^m x_i + \alpha - 1}(1-\theta)^{m - \sum_{i=1}^m x_i + \beta - 1}. \end{aligned}$$

This is, up to a normalizing constant, exactly the density of a $\mathrm{Beta}(\alpha', \beta')$ distribution with

$$\alpha' := \alpha + \sum_{i=1}^{m} x_i,$$
$$\beta' := \beta + m - \sum_{i=1}^{m} x_i.$$

Since a probability density integrates to one, the normalizing constant is forced to be the one that makes $\mathrm{Beta}(\alpha', \beta')$ a valid density, so $p(\theta \mid S) = \mathrm{Beta}(\theta \mid \alpha', \beta')$ exactly, with no approximation needed. Note the intuitive interpretation of the hyperparameters: α and β act as **pseudo-counts** of heads and tails observed prior to seeing any real data, and the posterior simply adds the real counts $\sum_i x_i$ and $m - \sum_i x_i$ on top.

For a new observation x_*, the Bayesian model predicts a distribution over outcomes rather than a single value:

$$P(x_* = 1 \mid S) = \int_0^1 P(x_* = 1 \mid \theta)\, p(\theta \mid S)\, d\theta = \int_0^1 \theta \cdot \mathrm{Beta}(\theta \mid \alpha', \beta')\, d\theta$$
$$= \mathbb{E}_{\theta \sim p(\theta|S)}[\theta] = \frac{\alpha'}{\alpha' + \beta'},$$

using the known mean of the Beta distribution. This is the **posterior predictive distribution**. Two of its properties are worth highlighting, both instances of a more general phenomenon in Bayesian models:

- It **averages over all possible values of** θ, weighted by their posterior probability — a form of **model averaging** that accounts for parameter uncertainty, rather than committing to a single point estimate.
- As $m \to \infty$, the pseudo-counts α, β become negligible relative to $\sum_i x_i$ and $m - \sum_i x_i$, so $P(x_* = 1 \mid S) \to \theta_{MLE}$: the Bayesian and frequentist predictions coincide in the large-sample limit, while differing — often substantially — for small m, exactly where accounting for uncertainty matters most.

Given a predictive distribution, there is more than one way to commit to a single prediction. The **Bayes predictor** takes the mode of the predictive distribution, $\widehat{x} := \arg\max_x P(x_* = x \mid S)$; it is optimal in the sense of minimizing expected 0/1 loss. The **Gibbs predictor** instead samples a prediction, $\widehat{x} \sim P(x_* \mid S)$.

7.3 Maximum A-Posteriori Estimation

In most real-world applications, the posterior distribution is intractable because of the integral in the evidence term. One common approximation is the **maximum a-posteriori (MAP)** estimate, the mode of the posterior:

$$\theta_{MAP} = \arg\max_\theta p(\theta \mid S) = \arg\max_\theta P(S \mid \theta) p(\theta) = \arg\max_\theta \log P(S \mid \theta) + \log p(\theta).$$

The intractable evidence $P(S)$ drops out since it does not depend on θ, making the objective tractable — this is the entire appeal of MAP over full Bayesian inference. For the Beta-Bernoulli model, since $\mathrm{Beta}(\alpha', \beta')$ has mode $\frac{\alpha'-1}{\alpha'+\beta'-2}$ (for $\alpha', \beta' > 1$),

$$\theta_{MAP} = \frac{\alpha + \sum_{i=1}^m x_i - 1}{\alpha + \beta + m - 2}.$$

Comparing this to $\theta_{MLE} = \frac{1}{m}\sum_i x_i$ makes the role of the prior explicit: MAP behaves like MLE run on an *augmented* data set that includes $\alpha - 1$ extra pseudo-heads and $\beta - 1$ extra pseudo-tails. As $m \to \infty$, $\theta_{MAP} \to \theta_{MLE}$, the same way the full posterior predictive did above.

For a test input, the MAP estimate predicts as a **point estimate**, $x_* \sim P(x_* \mid \theta_{MAP})$, and does not perform model averaging: unlike the fully Bayesian predictive distribution above, it is not a truly Bayesian approach, even though it is derived from the posterior.

7.4 Monte Carlo Integration

Bayesian quantities — evidences, predictive distributions, posterior expectations — are often integrals of the form

$$I = \int f(x)\, p(x)\, dx$$

for some function f and probability density $p(x)$ (with the integral replaced by a sum against a probability mass function $P(x)$ in the discrete case), which is exactly what we needed to evaluate $\mathbb{E}_{\theta \sim p(\theta|S)}[\theta]$ above. When this integral has no closed form (as is typical outside conjugate models like Beta-Bernoulli), and we can draw m samples $x_1, \ldots, x_m \sim p(x)$, we can approximate it by the sample average

$$I \approx \frac{1}{m} \sum_{i=1}^{m} f(x_i).$$

This is called **Monte Carlo integration**. By the weak law of large numbers (Chapter 4), this approximation is consistent: it converges to I as $m \to \infty$.

7.5 Generative Models

Consider a supervised learning problem with $(x, y) \sim \mathcal{D}$ for an unknown data distribution $\mathcal{D}$. Take the label y to be discrete and the features x to be continuous, so that — following the convention of Chapter 1 — the joint object is written $p(x, y)$ while its discrete factors are written with a capital P. There are two ways to factorize $p(x, y)$, and each suggests a different modeling strategy:

- $p(x, y) = P(y \mid x)\, p(x)$: model $P(y \mid x)$ directly and treat $p(x)$ as a nuisance (e.g. approximated via Monte Carlo integration on the training inputs). This is called **discriminative modeling** — the logistic regression model of Chapter 3 is an example.
- $p(x, y) = p(x \mid y)\, P(y)$: model both $P(y)$ and $p(x \mid y)$, i.e. infer the whole data-generating process where a label is chosen first and the corresponding input is generated conditioned on it. This is called **generative modeling**.

7.5.1 Example: a generative Bernoulli classifier

Let us build the simplest possible generative classifier: both the label $y \in \{0, 1\}$ and a single binary feature $x \in \{0, 1\}$ are Bernoulli. We choose

$$y \sim \text{Bernoulli}(\pi),$$
$$x \mid y \sim \text{Bernoulli}(\theta_y),$$

so π is the marginal probability of $y = 1$, and θ_0, θ_1 are the class-conditional probabilities that $x = 1$ given $y = 0$ and $y = 1$ respectively. Given $S = \{(x_i, y_i)\}_{i=1}^{m}$, all three parameters are Bernoulli parameters and each is fit by the MLE recipe derived above, applied to the relevant subset of the data:

$$\widehat{\pi} = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}(y_i{=}1),$$
$$\widehat{\theta}_c = \frac{\sum_{i:y_i=c} x_i}{\sum_{i:y_i=c} 1}, \quad c \in \{0,1\}.$$

The second formula is exactly θ_{MLE} from before, restricted to the subset of the data with $y_i = c$: fitting a generative model reduces to running the same single-distribution MLE machinery once per class. Given a new x_*, the model predicts

$$\begin{aligned} P(y{=}1 \mid x_*) &= \frac{P(x_* \mid y{=}1)\,P(y{=}1)}{P(x_* \mid y{=}0)P(y{=}0) + P(x_* \mid y{=}1)P(y{=}1)} \\ &= \frac{\widehat{\theta}_1^{x_*}(1-\widehat{\theta}_1)^{1-x_*}\,\widehat{\pi}}{\widehat{\theta}_0^{x_*}(1-\widehat{\theta}_0)^{1-x_*}(1-\widehat{\pi}) + \widehat{\theta}_1^{x_*}(1-\widehat{\theta}_1)^{1-x_*}\widehat{\pi}}. \end{aligned}$$

This is a complete, if minimal, generative classifier. Real problems have more than one feature, which brings us to Naive Bayes.

7.5.2 Naive Bayes Classifier

Assume now d discrete features $x = (x_1, \ldots, x_d)$, each taking values in a finite alphabet, and a label $y \in \{1, \ldots, C\}$. Modeling the *joint* class-conditional distribution $P(x \mid y{=}c)$ exactly would require a full contingency table with one entry per combination of feature values — exponential in d, and infeasible to estimate from any realistic amount of data. The **naive Bayes assumption** sidesteps this by assuming the features are conditionally independent given the class:

$$P(x \mid y{=}c) = \prod_{j=1}^{d} P(x_j \mid y{=}c).$$

This is "naive" because it is almost always false in the real world (features are typically correlated even within a class) — yet the resulting classifier is often highly effective in practice, and always tractable. For each feature j with alphabet $\{1, \ldots, V_j\}$, we only need to estimate a small **tabular** distribution $P(x_j = v \mid y{=}c)$ per class c and value v, i.e. a $C \times V_j$ table of probabilities. For binary features ($V_j = 2$) this is again exactly the Bernoulli-MLE machinery from before, one Bernoulli parameter $\theta_{j,c}$ per (feature, class) pair:

$$\widehat{\theta}_{j,c} := \widehat{P}(x_j{=}1 \mid y{=}c) = \frac{\sum_{i:y_i=c} x_{ij}}{m_c},$$
$$m_c := \sum_{i:y_i=c} 1,$$

together with the class prior $\widehat{\pi}_c = m_c/m$. Given a query x_*, the classifier predicts

$$\widehat{y} = \arg\max_{c\in[C]} \widehat{\pi}_c \prod_{j=1}^{d} \widehat{\theta}_{j,c}^{x_{*j}}(1-\widehat{\theta}_{j,c})^{1-x_{*j}}.$$

The total number of estimated parameters is $O(Cd)$ — linear in the number of features, in stark contrast to the exponential table size of the unrestricted model. This tabular form generalizes to categorical (non-binary) features and to continuous features (e.g. via a per-class Gaussian $p(x_j \mid y{=}c) = \mathcal{N}(x_j \mid \mu_{j,c}, \sigma^2_{j,c})$) by the same recipe, replacing the Bernoulli MLE with the MLE of the chosen per-feature family.

PAC analysis of Naive Bayes. Since every one of the $\widehat{\theta}_{j,c}$ (and $\widehat{\pi}_c$) is a Bernoulli-parameter MLE — a sample mean of a $\{0,1\}$-valued quantity — Hoeffding's inequality (Theorem 4.4) applies to each of them individually. We have $K = Cd$ such parameters $\widehat{\theta}_{j,c}$, each averaged over m_c i.i.d. samples of a $[0,1]$-valued random variable, so the *uniform* Hoeffding bound of Theorem 4.5 applies directly with $K = Cd$ estimators and common range $b - a = 1$:

$$P\Big(\exists (j,c) \in [d] \times [C] : |\widehat{\theta}_{j,c} - \theta_{j,c}| \geq \epsilon\Big) \leq 2\, Cd\, e^{-2 m_{\min} \epsilon^2},$$

where $m_{\min} := \min_c m_c$ is the smallest class sample size, and the factor 2 accounts for bounding the two-sided deviation $|\widehat{\theta}_{j,c} - \theta_{j,c}| \geq \epsilon$ by a union of the "too high" and "too low" one-sided events, each covered by Theorem 4.5. Setting the right-hand side to δ and solving for ϵ, with probability at least $1 - \delta$, **every** one of the Cd estimated parameters is simultaneously within

$$\epsilon = \sqrt{\frac{\log(2Cd/\delta)}{2 m_{\min}}}$$

of its true value. This is a genuine PAC-style guarantee: the sample complexity to achieve a target accuracy ϵ with confidence $1 - \delta$ grows only as $O\big(\frac{1}{\epsilon^2} \log(Cd/\delta)\big)$ — logarithmically in the number of features d and classes C, the direct benefit of the naive independence assumption reducing the effective number of parameters from exponential to linear in d. A uniform bound on the individual parameters like this one does not immediately give an equally tight bound on the resulting classifier's generalization error $R(\widehat{y})$ — the classification decision is a nonlinear function of all Cd parameters at once — but it does show that the *building blocks* of the Naive Bayes classifier concentrate fast, which is the source of its practical robustness even from modest amounts of data.

7.6 Implementation: Naive Bayes on a Real Data Set

Let us implement the tabular Naive Bayes classifier from scratch and test it on the **Wisconsin Breast Cancer** data set: 569 real tumor samples, each with 30 real-valued measurements (radius, texture, smoothness, ...) of a cell nucleus, labeled malignant or benign. Since our model above is built for *binary* features, we first binarize each measurement at its training-set median — a common, simple discretization strategy that brings a continuous data set into the tabular setting this chapter's math was derived for. Every other step — the train/test split, the class-conditional MLEs $\widehat{\theta}_{j,c}$, and the prediction rule — is implemented directly from the formulas above, with no black-box model classes.

```
import torch as th
from sklearn.datasets import load_breast_cancer

# The only library-provided piece is the raw data set itself; the
# splitting, estimation, and prediction logic are all implemented below.
data = load_breast_cancer()
X_all = th.tensor(data.data, dtype=th.float64)
y_all = th.tensor(data.target, dtype=th.long)

th.manual_seed(0)
idx = th.randperm(y_all.numel())
n_train = int(0.7 * y_all.numel())
train_idx, test_idx = idx[:n_train], idx[n_train:]

# Binarize each feature at its training-set median.
median = X_all[train_idx].median(dim=0).values
X_train = (X_all[train_idx] > median).double()
X_test = (X_all[test_idx] > median).double()
y_train, y_test = y_all[train_idx], y_all[test_idx]

class BernoulliNaiveBayes:
    def __init__(self, num_classes=2, laplace=1.0):
        self.C = num_classes
        self.alpha = laplace  # Laplace (add-alpha) smoothing pseudo-count

    def learn(self, X, y):
        n, d = X.shape
        self.pi = th.zeros(self.C, dtype=th.float64)
        self.theta = th.zeros(self.C, d, dtype=th.float64)
        for c in range(self.C):
            Xc = X[y == c]
            n_c = Xc.shape[0]
            self.pi[c] = n_c / n
            # theta_{j,c} = (sum_i x_ij + alpha) / (n_c + 2*alpha): the MAP
            # estimate under a Beta(alpha+1, alpha+1) prior derived earlier
            # in this chapter, with alpha=1 recovering add-one smoothing.
            self.theta[c] = (Xc.sum(dim=0) + self.alpha) / (n_c + 2 * self.alpha)

    def predict(self, X):
        # Work in log-space for numerical stability, exactly as motivated
        # by the log-likelihood discussion at the start of this chapter.
        log_theta = th.log(self.theta)
        log_1m_theta = th.log(1 - self.theta)
        log_class_cond = X @ log_theta.T + (1 - X) @ log_1m_theta.T
        log_prior = th.log(self.pi)
        log_joint = log_class_cond + log_prior
        return th.argmax(log_joint, dim=1)
```

```
model = BernoulliNaiveBayes()
model.learn(X_train, y_train)

train_preds = model.predict(X_train)
test_preds = model.predict(X_test)

train_acc = (train_preds == y_train).double().mean().item()
test_acc = (test_preds == y_test).double().mean().item()

print(f"Naive Bayes train accuracy: {train_acc:.3f}")
print(f"Naive Bayes test accuracy:  {test_acc:.3f}")
```

```
Naive Bayes train accuracy: 0.910
Naive Bayes test accuracy:  0.918
```

Despite the crudeness of median binarization (which discards all information about *how far* a measurement is from the median) and the naive independence assumption across all 30 features, the classifier reaches over 90% test accuracy with a training procedure that is nothing more than counting — no gradient descent, no iterative optimization, just the closed-form MLE/MAP formulas of this chapter applied once per class.

CHAPTER 8

DECISION TREES

Every hypothesis class we have used so far predicts through a fixed algebraic form: a dot product for linear predictors (Chapter 2), a sigmoid or softmax of a dot product for classifiers (Chapter 3), a weighted sum of similarities for kernel methods (Chapter 9). A **decision tree** (Breiman et al., 1984) predicts through a different mechanism entirely: a sequence of "if-then" questions about individual features, organized in a tree, that recursively partitions the feature space $\mathcal{X}$ into regions, each assigned a single output.

8.1 The Hypothesis Class

A decision tree is a rooted tree in which:

- every **internal node** v is labeled with a **splitting rule** $\phi_v : \mathcal{X} \to \{1, \dots, k_v\}$ that routes an input x to one of v's k_v children — for the binary trees we focus on here, $k_v = 2$ and $\phi_v(x) := \mathbb{1}(x_j \le t)$ for some feature index j and threshold t (or $\phi_v(x) := \mathbb{1}(x_j = a)$ for a categorical feature taking the value a),
- every **leaf** u is labeled with a prediction $c_u \in \mathcal{Y}$.

A tree T computes a hypothesis $h_T : \mathcal{X} \to \mathcal{Y}$ by routing x from the root, following $\phi_v(x)$ at each internal node v, until it reaches a leaf u, and outputting $h_T(x) := c_u$. Since each leaf's region is exactly the intersection of the splitting rules on the path from the root, the collection of leaves of T partitions $\mathcal{X}$ into disjoint axis-aligned regions, and h_T is piecewise constant on this partition. The **hypothesis class of decision trees**, $\mathcal{H}_{\text{tree}}$, is the union of h_T over all finite binary trees T built from the allowed splitting rules — an infinite, nonuniform hypothesis class in the sense of Chapter 5, since larger trees induce finer partitions and hence more expressive hypotheses.

Formally, then, a decision tree instantiates the general recursive partitioning scheme

$$h_T(x) = \sum_{u \in \text{leaves}(T)} c_u \cdot \mathbb{1}\big(x \in R_u\big),$$

$$R_u := \bigcap_{v \in \text{path}(u)} \phi_v(x)\text{-consistent half-space at } v,$$

where path(u) is the sequence of internal nodes from the root to leaf u, and $\{R_u\}_{u \in \text{leaves}(T)}$ partitions $\mathcal{X}$.

8.2 Growing a Tree: A Greedy ERM Surrogate

Given a training set S, ERM over $\mathcal{H}_{\text{tree}}$ would search over all trees for the one minimizing $\widehat{R}_S(h_T)$. This is computationally intractable — even fixing the tree's shape, choosing the best splits jointly is NP-hard. In practice, we build the tree by a **greedy, recursive** procedure:

1. Start with a single leaf holding all of S.
2. At each leaf currently holding a subset $S_v \subseteq S$, consider splitting it into two children by some rule ϕ_v. Among all candidate rules (all features, all thresholds), pick the one that most reduces an **impurity** measure of the label distribution within the node (defined below).
3. Recurse into each child with its corresponding subset of S_v, and repeat, until a stopping criterion is met (e.g. a maximum depth, a minimum leaf size, or zero remaining impurity).
4. Label each final leaf u with the majority class of S_v restricted to that leaf (classification) or the mean label (regression).

The key design choice is the **impurity measure**, which stands in for $\widehat{R}_S(h)$ as a differentiable-in-spirit, locally computable proxy that the greedy procedure can optimize one split at a time. For a node v holding a subset S_v of size $m_v := |S_v|$, with class proportions $\widehat{p}_c := \frac{|\{(x,y) \in S_v : y = c\}|}{m_v}$ for $c \in \{1, \dots, C\}$, two common choices are:

- **Entropy** (of the empirical label distribution at the node): $H(S_v) := -\sum_{c=1}^{C} \widehat{p}_c \log \widehat{p}_c$, borrowed directly from information theory — it is maximized when S_v's labels are uniformly spread across classes (there is the most "uncertainty" about the label at this node) and zero when S_v is pure (a single class). Following the convention of these notes, log here is the natural logarithm, so entropy is measured in nats rather than the more familiar bits; the base only rescales H by a positive constant and therefore never changes which split is selected.
- **Gini index**: $G(S_v) := 1 - \sum_{c=1}^{C} \widehat{p}_c^2 = \sum_{c=1}^{C} \widehat{p}_c(1 - \widehat{p}_c)$, the probability that two labels drawn independently (with replacement) from S_v's empirical distribution disagree. It behaves similarly to entropy — zero for a pure node, maximal for a uniform one — but is cheaper to compute and does not involve a logarithm.

Given an impurity measure $\text{Imp}(\cdot) \in \{H, G\}$ and a candidate split of S_v into $S_{v,\text{left}}$ and $S_{v,\text{right}}$, the **information gain** of the split is the reduction in impurity, weighted by the resulting subset sizes:

$$\text{Gain}(S_v, \phi_v) := \text{Imp}(S_v) - \frac{|S_{v,\text{left}}|}{m_v}\text{Imp}(S_{v,\text{left}}) - \frac{|S_{v,\text{right}}|}{m_v}\text{Imp}(S_{v,\text{right}}).$$

At each node, the greedy algorithm picks the split ϕ_v maximizing $\text{Gain}(S_v, \phi_v)$ over all candidate features and thresholds. This greedy, node-by-node maximization is what makes

decision tree induction tractable: it never revisits an earlier split, and never jointly optimizes two splits together, at the cost of no longer being guaranteed to find the ERM-optimal tree of a given size.

8.3 Overfitting and the Bias-Complexity Dilemma

An unconstrained tree can always drive $\widehat{R}_S(h_T) = 0$: grow it until every leaf holds a single training point (or a set of points sharing one label). This is the tree-induction analogue of the memorization hypothesis from Chapter 1 — it exhibits the same overfitting behavior, with tree size (roughly, the number of leaves) playing the role that polynomial degree M played in the curve-fitting example, or that $|\mathcal{H}|$ played in the generalization bound of Chapter 5. Two standard remedies, both instances of the Structural Risk Minimization principle from Chapter 5:

- **Early stopping.** Halt the recursive splitting once a stopping criterion is met (maximum depth, minimum node size, or a minimum required Gain), directly capping the tree's complexity during growth.
- **Pruning.** Grow a large tree first, then remove (merge) subtrees whose removal does not substantially hurt performance on a held-out validation set — trading a controlled increase in $\widehat{R}_S$ for a reduction in complexity, in the hope of decreasing $R(h_T)$.

Consistent with the fundamental theorem of statistical learning (Chapter 5), the capacity of $\mathcal{H}_{\text{tree}}$ can also be bounded formally.

Proposition 8.3.1

VC dimension of size-limited trees. Let $\mathcal{H}_{\text{tree},L} \subset \mathcal{H}_{\text{tree}}$ be the subclass of binary trees with axis-aligned threshold splits on d real-valued features and at most L leaves, predicting in $\{0,1\}$. Then

$$\tau_{\mathcal{H}_{\text{tree},L}}(m) \;\le\; \underbrace{C_{L-1}}_{\text{tree shapes}} \cdot \underbrace{\big(d(m+1)\big)^{L-1}}_{\text{splits}} \cdot \underbrace{2^{L}}_{\text{leaf labels}},$$

$$\text{hence} \qquad d_{VC}(\mathcal{H}_{\text{tree},L}) = O\big(L\log(Ld)\big),$$

where $C_k \le 4^k$ is the k-th Catalan number, counting the shapes of a binary tree with k internal nodes.

Proof. Fix m points. A tree with at most L leaves has at most $L-1$ internal nodes, and its restriction to the sample is determined by three independent choices:

- the **shape** of the tree, i.e. which binary tree with $L-1$ internal nodes it is: $C_{L-1} \le 4^{L-1}$ possibilities;
- the **splitting rule** at each internal node. A rule $\mathbb{1}(x_j \le t)$ is determined, *as far as its behavior on the sample is concerned*, by the feature index $j \in [d]$ and by the position

of t among the m observed values of feature j — at most $m+1$ distinct positions. So at most $d(m+1)$ effective rules per node, and at most $\big(d(m+1)\big)^{L-1}$ jointly;
- the **label** $c_u \in \{0,1\}$ of each of the at most L leaves: at most 2^L possibilities.

Multiplying gives the stated bound on the growth function. Shattering m points requires $2^m \le \tau_{\mathcal{H}_{\text{tree},L}}(m)$, i.e.

$$m \log 2 \;\le\; (L-1)\log 4 + (L-1)\log\big(d(m+1)\big) + L\log 2,$$

so $m = O\big(L\log(d) + L\log m\big)$, which forces $m = O\big(L\log(Ld)\big)$ □

Complexity therefore grows essentially linearly with the number of leaves — up to a logarithmic factor for choosing which feature and threshold to split on at each internal node — a formal restatement of the intuition that more leaves means more capacity to overfit. The proof is also worth noting for what it does *not* count: the thresholds t range over an uncountable set, yet only $m+1$ of them are distinguishable on a given sample. This is the same "infinite class, finite restriction" phenomenon that Definition 5.9 was introduced to capture.

8.4 PAC Analysis of Decision Trees

$\mathcal{H}_{\text{tree}}$ itself is not PAC learnable. Since a tree can grow arbitrarily many leaves, $d_{VC}(\mathcal{H}_{\text{tree}}) = \lim_{L\to\infty} d_{VC}(\mathcal{H}_{\text{tree},L}) = \infty$ (a tree with 2^k leaves can, on real-valued features, always be grown to shatter any k points by isolating each into its own leaf). By Theorem 5.3, an infinite VC dimension rules out PAC learnability for the *unconstrained* class: no sample size m guarantees good generalization uniformly over all of $\mathcal{H}_{\text{tree}}$, matching the memorization behavior already noted above. Just as with k-nearest neighbors (Chapter 3) and unconstrained polynomial regression (Chapter 1), the fix is not to abandon the hypothesis class, but to control which part of it we search — exactly the role that early stopping and pruning play informally, and which we now make precise.

Fixed leaf budget: a standard agnostic PAC bound. Fix a leaf budget L and restrict to $\mathcal{H}_{\text{tree},L}$, which has finite VC dimension $d_L := d_{VC}(\mathcal{H}_{\text{tree},L}) = O(L\log(Ld))$. The Fundamental Theorem of Statistical Learning (Theorem 5.4) then applies directly: $\mathcal{H}_{\text{tree},L}$ is agnostic PAC learnable, and with probability at least $1-\delta$, every tree $h \in \mathcal{H}_{\text{tree},L}$ satisfies

$$R(h) \le \widehat{R}_S(h) + O\left(\sqrt{\frac{L\log(Ld) + \log(1/\delta)}{m}}\right).$$

This already formalizes the early-stopping heuristic: fixing a maximum depth (hence an upper bound on L) in advance and running ERM (i.e. the greedy growth procedure, as a tractable surrogate for exact ERM) within $\mathcal{H}_{\text{tree},L}$ inherits this generalization guarantee.

Unknown leaf budget: decision trees as an instance of SRM. In practice we do not fix L in advance — the greedy algorithm decides how many leaves to grow from the data itself, effectively searching $\mathcal{H}_{\text{tree}} = \bigcup_{L=1}^{\infty} \mathcal{H}_{\text{tree},L}$. This is precisely the nonuniform learning setting of Theorem 5.7: assign each subclass $\mathcal{H}_{\text{tree},L}$ a weight $w(L)$ with $\sum_{L=1}^{\infty} w(L) < 1$ (e.g.

$w(L) := 6/(\pi^2 L^2)$, or a prefix-free code on L as in the MDL construction of Lemma 5.3), and Theorem 5.7 gives, with probability at least $1-\delta$, simultaneously for **every** tree $h \in \mathcal{H}_{\text{tree}}$ of any size $L(h)$,

$$R(h) \leq \widehat{R}_S(h) + O\left(\sqrt{\frac{L(h)\log(L(h)d) + \log(1/(w(L(h))\delta))}{m}}\right).$$

The resulting h_{SRM} rule — minimize the right-hand side jointly over tree structure and size — makes explicit what pruning and early stopping approximate: a term that strictly decreases with L (the greedy algorithm's $\widehat{R}_S$, which can always be driven towards 0 by adding leaves) traded off against a term that strictly increases with L (the complexity penalty $\sqrt{L\log(Ld)/m}$). A pruning rule that stops merging leaves once further merges would raise a held-out validation error is, in effect, estimating this trade-off directly from data rather than from the closed-form penalty above — a data-driven proxy for the same SRM objective.

8.5 Illustrative Example

Consider a tiny data set for deciding whether to play outdoor tennis, with two binary features — `Outlook` $\in \{\text{Sunny}, \text{Rainy}\}$ and `Wind` $\in \{\text{Weak}, \text{Strong}\}$ — and label `Play` $\in \{\text{Yes}, \text{No}\}$:

`Outlook`	`Wind`	`Play`
Sunny	Weak	Yes
Sunny	Strong	Yes
Sunny	Weak	Yes
Sunny	Strong	No
Rainy	Weak	No
Rainy	Weak	No
Rainy	Strong	No
Rainy	Strong	No

Here $m = 8$, with 4 "Yes" and 4 "No", so the root's Gini index is $G(S) = 1 - (\frac{1}{2})^2 - (\frac{1}{2})^2 = 0.5$.

Candidate split on `Outlook`. This sends the 4 Sunny points (3 Yes, 1 No) to the left child and the 4 Rainy points (0 Yes, 4 No) to the right child:

$$G(S_{\text{left}}) = 1 - \left(\tfrac{3}{4}\right)^2 - \left(\tfrac{1}{4}\right)^2 = 0.375,$$
$$G(S_{\text{right}}) = 1 - 1^2 - 0^2 = 0.$$

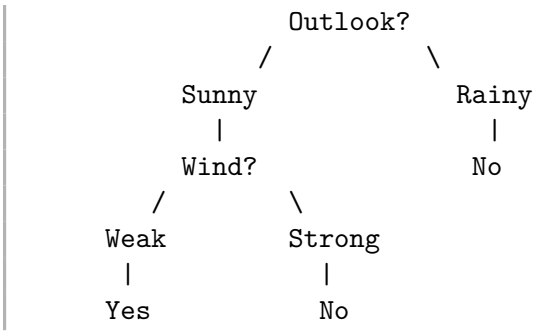
$$\text{Gain}(S, \texttt{Outlook}) = 0.5 - \tfrac{4}{8}(0.375) - \tfrac{4}{8}(0) = 0.5 - 0.1875 = 0.3125.$$

Candidate split on `Wind`. This sends the 4 Weak points (2 Yes, 2 No) to the left child and the 4 Strong points (1 Yes, 3 No) to the right child:

$$G(S_{\text{left}}) = 1 - \left(\tfrac{1}{2}\right)^2 - \left(\tfrac{1}{2}\right)^2 = 0.5,$$
$$G(S_{\text{right}}) = 1 - \left(\tfrac{1}{4}\right)^2 - \left(\tfrac{3}{4}\right)^2 = 0.375.$$

$$\text{Gain}(S, \texttt{Wind}) = 0.5 - \tfrac{4}{8}(0.5) - \tfrac{4}{8}(0.375) = 0.5 - 0.4375 = 0.0625.$$

Since $\text{Gain}(S, \texttt{Outlook}) = 0.3125 > 0.0625 = \text{Gain}(S, \texttt{Wind})$, the greedy algorithm splits on `Outlook` first. The right child (Rainy) is already pure ($G = 0$) and becomes a leaf predicting No. The left child (Sunny, 3 Yes/1 No) is not pure, so we recurse: splitting it further on `Wind` sends its single Strong/No point to its own leaf and the remaining 3 Weak/Yes points to another, both pure, terminating the recursion. The resulting tree is



Observe that this tree achieves $\widehat{R}_S(h_T) = 0$ using only 3 leaves out of the $2^2 = 4$ possible input combinations having been observed — the greedy split ordering (largest information gain first) is exactly what let it reach a pure partition using the *fewest* splits, which is the practical mechanism by which impurity-driven greedy growth tends to produce small, and hence better-generalizing, trees compared to growing in an arbitrary order.

8.6 Implementation: Growing a Tree on a Real Data Set

Let us implement the greedy tree-growing algorithm of this chapter from scratch — the Gini impurity, the exhaustive split search, and the recursive growth procedure, all exactly as derived above — and test it on the **Wisconsin Breast Cancer** data set: 569 real tumor samples, each with 30 real-valued measurements of a cell nucleus, labeled malignant or benign. Unlike the Naive Bayes example of Chapter 6, decision trees natively handle real-valued features via threshold splits, so no discretization is needed here.

```
import torch as th
from sklearn.datasets import load_breast_cancer

# The only library-provided piece is the raw data set itself; the
# splitting, growth, and prediction logic are all implemented below.
data = load_breast_cancer()
X_all = th.tensor(data.data, dtype=th.float64)
y_all = th.tensor(data.target, dtype=th.long)

th.manual_seed(0)
```

```
idx = th.randperm(y_all.numel())
n_train = int(0.7 * y_all.numel())
train_idx, test_idx = idx[:n_train], idx[n_train:]

X_train, y_train = X_all[train_idx], y_all[train_idx]
X_test, y_test = X_all[test_idx], y_all[test_idx]

def gini(labels, num_classes=2):
    # G(S) = 1 - sum_c p_c^2
    if labels.numel() == 0:
        return th.tensor(0.0)
    counts = th.bincount(labels, minlength=num_classes).double()
    p = counts / labels.numel()
    return 1.0 - th.sum(p ** 2)

def best_split(X, y):
    # Exhaustively search every (feature, threshold) candidate and keep
    # the one with the highest information gain, Gain(S_v, phi_v).
    m, d = X.shape
    parent_impurity = gini(y)
    best_gain, best_feat, best_thresh = 0.0, None, None
    for j in range(d):
        values = th.unique(X[:, j])
        if values.numel() < 2:
            continue
        # midpoints between consecutive values
        thresholds = (values[:-1] + values[1:]) / 2
        for t in thresholds:
            left_mask = X[:, j] <= t
            n_left = left_mask.sum().item()
            n_right = m - n_left
            if n_left == 0 or n_right == 0:
                continue
            weighted_impurity = (n_left / m) * gini(y[left_mask]) + \
                                (n_right / m) * gini(y[~left_mask])
            gain = (parent_impurity - weighted_impurity).item()
            if gain > best_gain:
                best_gain, best_feat, best_thresh = gain, j, t.item()
    return best_feat, best_thresh, best_gain

class Node:
    def __init__(self, feature=None, threshold=None,
                 left=None, right=None, prediction=None):
        self.feature, self.threshold = feature, threshold
        self.left, self.right = left, right
        self.prediction = prediction  # set only for leaves
```

```
def build_tree(X, y, depth=0, max_depth=3, min_leaf_size=10):
    majority = th.bincount(y, minlength=2).argmax().item()
    # Early-stopping criteria: depth cap, minimum leaf size, or purity.
    if depth == max_depth or y.numel() < min_leaf_size or gini(y).item() < 1e-12:
        return Node(prediction=majority)
    feat, thresh, gain = best_split(X, y)
    if feat is None or gain <= 1e-12:
        return Node(prediction=majority)
    left_mask = X[:, feat] <= thresh
    left = build_tree(X[left_mask], y[left_mask],
                      depth + 1, max_depth, min_leaf_size)
    right = build_tree(X[~left_mask], y[~left_mask],
                       depth + 1, max_depth, min_leaf_size)
    return Node(feature=feat, threshold=thresh, left=left, right=right)

def predict_one(node, x):
    while node.prediction is None:  # route x from the root to a leaf
        node = node.left if x[node.feature] <= node.threshold else node.right
    return node.prediction

def predict(tree, X):
    return th.tensor([predict_one(tree, x) for x in X])

tree = build_tree(X_train, y_train, max_depth=3)

train_preds = predict(tree, X_train)
test_preds = predict(tree, X_test)
train_acc = (train_preds == y_train).double().mean().item()
test_acc = (test_preds == y_test).double().mean().item()

print(f"Decision tree train accuracy: {train_acc:.3f}")
print(f"Decision tree test accuracy:  {test_acc:.3f}")
```

```
Decision tree train accuracy: 0.965
Decision tree test accuracy:  0.918
```

A tree with at most $2^3 = 8$ leaves (`max_depth=3`) already reaches 91.8% test accuracy — close to its 96.5% training accuracy, indicating that the depth cap from Section "PAC Analysis of Decision Trees" above is doing its job of controlling L (and hence $d_{VC}(\mathcal{H}_{\text{tree},L})$) without starving the tree of the capacity it needs to separate the two classes.

CHAPTER 9

NEURAL NETWORKS

9.1 Neuron (Perceptron)

A neuron, in the original form introduced as the **perceptron** (Rosenblatt, 1958), is the basic unit of a neural network. A neuron identified by index j has the following structure:

$$z = w_j^T x + b_j$$
$$h = g(z).$$

Above,

- x is the input vector,
- w_j is the weight vector, which correspond to the synapses of the neuron that admit input signals from the previous layer,
- g is the activation function that determines whether the neuron is activated or not,
- b_j is the bias, which is a constant,
- z is the linear activation of the neuron,
- h is the activation output of the neuron.

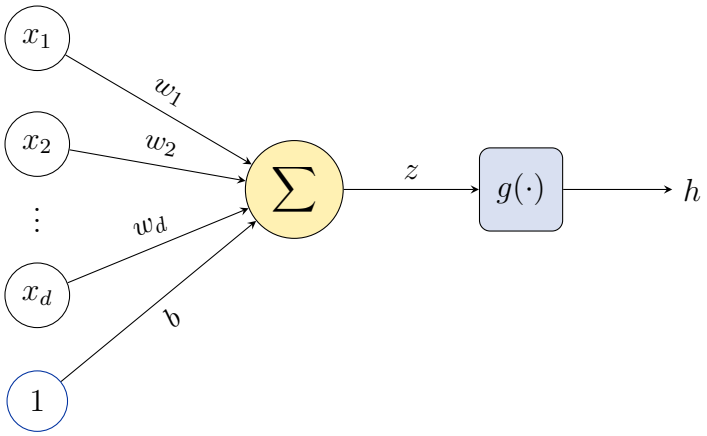


Figure 9.1: A single neuron (perceptron). The inputs $x_1, \ldots, x_d$ are combined through the weights $w_1, \ldots, w_d$ and a bias b into the linear activation $z = w^\top x + b$, which the activation function g maps to the output $h = g(z)$.

The first-generation activation function was the Heaviside step function:

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

This construction, called a **perceptron**, is not popular nowadays since it is not differentiable. Modern neural networks use differentiable activation functions such as:

- Sigmoid function: $g(z) = \frac{1}{1+e^{-z}}$,
- Hyperbolic tangent function: $g(z) = \tanh(z)$,
- Rectified linear unit (ReLU): $g(z) = \max(0, z)$.

```
import torch as th
import matplotlib.pyplot as plt

# The ReLU, Sigmoid, and Tanh activation functions can be plotted in
# different panels of a figure as below:

x = th.linspace(-10, 10, 100)
x.requires_grad = True  # Enable gradient tracking for x

# Create a figure with two rows and three columns of subplots
fig, axes = plt.subplots(2, 3, figsize=(12, 8))

# Plot the original functions in the top row
# ReLU
y_relu = th.relu(x)
axes[0, 0].plot(x.detach(), y_relu.detach())
axes[0, 0].set_title('ReLU')
axes[0, 0].set_ylim(-1, 10.1)

# Sigmoid
y_sigmoid = th.sigmoid(x)
axes[0, 1].plot(x.detach(), y_sigmoid.detach())
axes[0, 1].set_title('Sigmoid')
axes[0, 1].set_ylim(-0.1, 1.1)

# Tanh
y_tanh = th.tanh(x)
axes[0, 2].plot(x.detach(), y_tanh.detach())
axes[0, 2].set_title('Tanh')
axes[0, 2].set_ylim(-1.1, 1.1)

# Compute gradients
# Compute gradients using the automatic differentiation provided by PyTorch
grad_relu = th.autograd.grad(y_relu.sum(), x)[0].detach()
grad_sigmoid = th.autograd.grad(y_sigmoid.sum(), x)[0].detach()
```

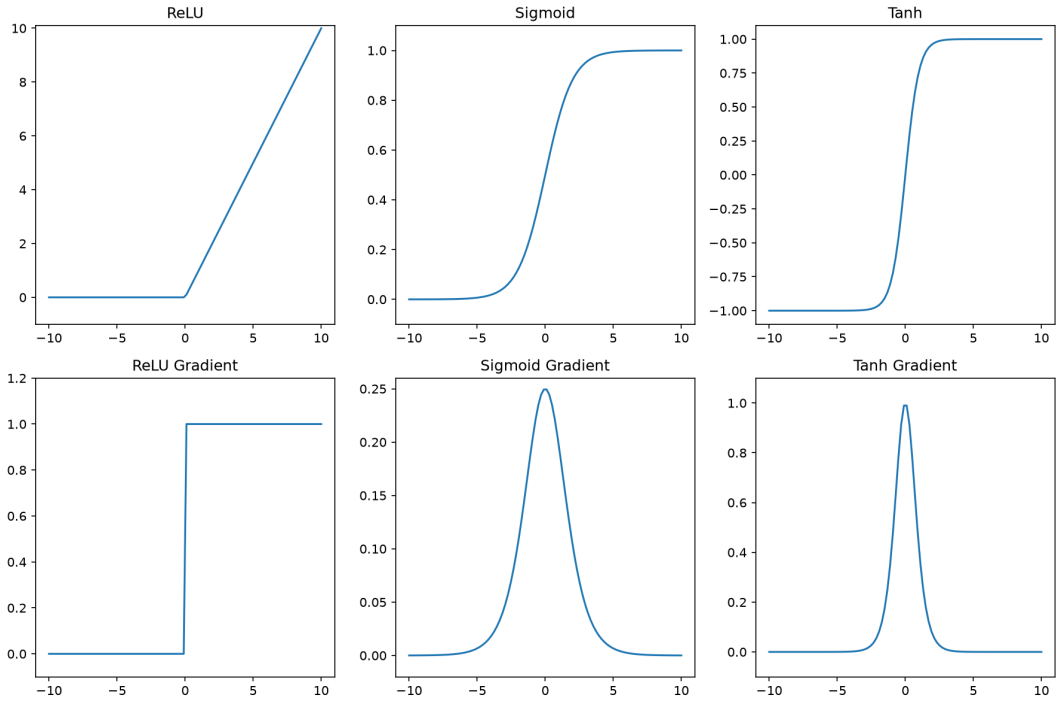


Figure 9.2: The ReLU, sigmoid, and tanh activation functions (top row) and their gradients (bottom row).

```
grad_tanh = th.autograd.grad(y_tanh.sum(), x)[0].detach()

# Plot the gradients in the bottom row
axes[1, 0].plot(x.detach(), grad_relu)
axes[1, 0].set_title('ReLU Gradient')
axes[1, 0].set_ylim(-0.1, 1.2)

axes[1, 1].plot(x.detach(), grad_sigmoid)
axes[1, 1].set_title('Sigmoid Gradient')
axes[1, 1].set_ylim(-0.02, 0.26)  # Adjusted y-axis limits for sigmoid gradient

axes[1, 2].plot(x.detach(), grad_tanh)
axes[1, 2].set_title('Tanh Gradient')
axes[1, 2].set_ylim(-0.1, 1.1)

plt.tight_layout()

plt.show()
```

9.2 Neural Network (Multilayer Perceptron)

A neural network is a collection of neurons. The neurons are organized in layers such that the activation output of the neurons in one layer become the synaptic inputs of the next layer.

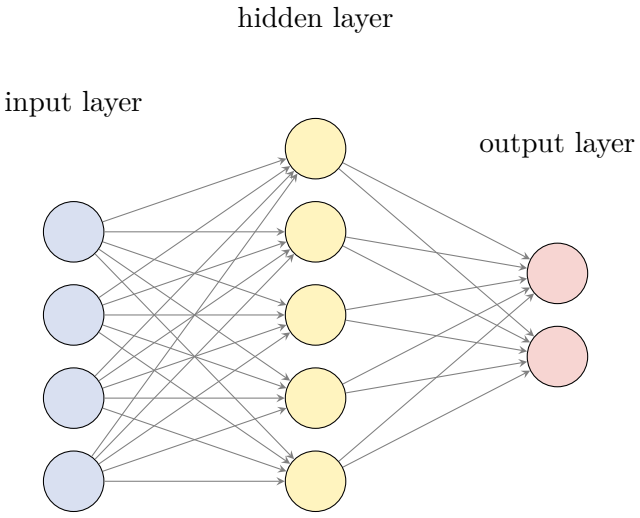


Figure 9.3: A feedforward neural network (multilayer perceptron). Each node is a perceptron unit as in Figure 9.1, organized into layers, with every neuron of one layer feeding into every neuron of the next.

The operation of k neurons sharing the same layer l on the i th can be given in the vector form below:

$$z_n^l = W_l^\top h_n^{l-1} + b^l,$$
$$h_n^l = g(z_n^l),$$

where

$$W_l = \begin{bmatrix} w_1^l & w_2^l & \cdots & w_k^l \end{bmatrix}$$

is the matrix of weight vectors and

$$b^l = (b_1^l, \ldots, b_k^l)$$

is the vector of biases. The vector h_n^l is called the **activation map** of layer l for the n-th input. For $l = 0$, the activation map is the input vector x, i.e. $h_n^0 := x_n$. The final pre-activation z_n^H is the output $\widehat{y}_n$ of the neural network.

Assume we have a neural network with two layers. We can express its whole operation as:

$$\widehat{y}_n = W_2^\top g(W_1^\top x_n + b^1) + b^2.$$

Likewise, we can express the operation of a neural network with three layers as:

$$\widehat{y}_n = W_3^\top g(W_2^\top g(W_1^\top x_n + b^1) + b^2) + b^3.$$

9.3 Training a Neural Network: Backpropagation

As other machine learning models we covered thus far, neural networks are also trained to minimize a loss function. Denote the loss function by $\ell(y, \widehat{y})$ for a prediction $\widehat{y}$ and the true value y. Due to the cascaded application of the nonlinear activation functions, the loss function is not convex. Therefore, we cannot find the optimal solution analytically. Instead, we use gradient descent:

$$w_{ij}^l := w_{ij}^l - \alpha \frac{\partial \ell}{\partial w_{ij}^l},$$

for all synaptic connections $i \to j$ in all layers l. Given a neural network with H layers, define $\widehat{y}_j = z_j^H$ as the output of the j th neuron in the last layer. Also define

$$\delta_j^l = \frac{\partial \ell}{\partial z_j^l}.$$

This expression is called the **error** of the j th neuron in the l th layer. To see why, consider the case where $l = H$ and the loss function is the mean squared error: $\ell(y, \widehat{y}) = \frac{1}{2}\sum_j (y_j - \widehat{y}_j)^2$. Then we have

$$\begin{aligned}
\delta_j^H &= \frac{\partial \frac{1}{2}\sum_j (y_j - \widehat{y}_j)^2}{\partial z_j^H} \\
&= \frac{\partial \frac{1}{2}(y_j - \widehat{y}_j)^2}{\partial \widehat{y}_j} \\
&= \widehat{y}_j - y_j.
\end{aligned}$$

This is the error of the prediction of the neural network on the j output channel. The gradient of the loss with respect to an arbitrary weight is

$$\begin{aligned}
\frac{\partial \ell}{\partial w_{ij}^l} &= \frac{\partial \ell}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{ij}^l} \\
&= \delta_j^l \frac{\partial z_j^l}{\partial w_{ij}^l} \\
&= \delta_j^l h_i^{l-1}.
\end{aligned}$$

Let us next calculate the error for an intermediate layer l. We have

$$\begin{aligned}
\delta_j^l &= \frac{\partial \ell}{\partial z_j^l} \\
&= \sum_k \frac{\partial \ell}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\
&= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l}
\end{aligned}$$

where k ranges over the neurons in the next layer. Place the definition of z_k^{l+1} into the above equation, we have

$$\begin{aligned}\frac{\partial z_k^{l+1}}{\partial z_j^l} &= \frac{\partial}{\partial z_j^l}\left(\sum_{j'} w_{j'k}^{l+1} g(z_{j'}^l) + b_k^{l+1}\right) \\ &= w_{jk}^{l+1}\frac{\partial g(z_j^l)}{\partial z_j^l}.\end{aligned}$$

Placing this result into the above expression about the derivative of the error, we get

$$\delta_j^l = \frac{\partial g(z_j^l)}{\partial z_j^l}\sum_k \delta_k^{l+1} w_{jk}^{l+1}$$

This way we obtain a recursive formula for the error of a neuron in an intermediate layer, where the recursion applies in the backward direction of the layers. There are remarkable facts regarding the computational efficiency of this formula that are first observed by Rumelhart et al. in 1986 and made the training of multilayer neural networks feasible:

- It requires only z_j^l, which is already calculated during the forward pass.
- Its computational cost is linear to the number of neurons in the network, i.e. $O(|W|)$ where $|W|$ is the number of weights in the network.

The findings above can be put together into an algorithm and expressed as below.

1. Set $h^0 = x$ (the activation map of the input layer is the input itself).
2. **Forward Pass.** For $l = 1 \to H$ do:
   - Compute and save $z_j^l = \sum_i w_{ij}^l h_i^{l-1} + b_j^l$ for every neuron j in layer l.
   - If $l < H$, compute $h_j^l = g(z_j^l)$. If $l = H$, the network output is $\widehat{y}_j := z_j^H$.
3. **Backward Pass.** For $l = H \to 1$ do:
   - Compute $\delta_j^l = \frac{\partial g(z_j^l)}{\partial z_j^l}\sum_k \delta_k^{l+1} w_{jk}^{l+1}$ for every neuron j in layer l (for $l = H$, use $\delta_j^H = \widehat{y}_j - y_j$ directly, as derived above).
   - Compute $\frac{\partial \ell}{\partial w_{ij}^l} = \delta_j^l h_i^{l-1}$ for every synaptic connection $i \to j$.

This algorithm is called **backpropagation** (Rumelhart et al., 1986). The name comes after the notion that the error term δ_j^l is propagated backward from the output layer to the input layer.

9.3.1 Stochastic Gradient Descent

Above we explained the backpropagation algorithm for a single data point for simplicity. For a data set with m samples, the gradient descent algorithm becomes:

$$w_{ij}^l := w_{ij}^l - \alpha \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \ell(y_i, \widehat{y}_i)}{\partial w_{ij}^l},$$

This means we need to do a forward pass and a backward pass for all weights and all data points. This is computationally expensive for large data sets. Instead, we can use a technique called **stochastic gradient descent**. In this technique, we use a subset of the data set, called a **mini-batch**, to calculate the gradient. The size of the mini-batch is denoted by b. The stochastic gradient descent algorithm can be expressed as:

$$B \subseteq S, \quad |B| = b, \quad B \text{ drawn uniformly at random}$$
$$w_{ij}^l := w_{ij}^l - \alpha \frac{1}{b} \sum_{i=1}^{b} \frac{\partial \ell(y_i, \widehat{y}_i)}{\partial w_{ij}^l},$$

where the first step draws a mini-batch B uniformly at random from the **training set** S — not from the data distribution $\mathcal{D}$, which we cannot sample from — and the second step performs a gradient descent update using the backpropagation algorithm. The random selection makes the gradient signal a random variable. This is why the algorithm is called stochastic gradient descent. Because B is drawn uniformly from S, this random variable is an unbiased estimate of the full-batch gradient over S, i.e. of the gradient of the empirical risk $\widehat{R}_S$. Hence, under mild conditions (a suitably decaying learning rate α, among others) clarified by (Robbins and Monro, 1951), the algorithm is guaranteed to converge to a stationary point of the loss, just as deterministic gradient descent is. Since the neural network loss is generally non-convex, owing to the cascaded, nonlinear composition of layers described above, neither algorithm is guaranteed to reach the *same* stationary point, or the global optimum; in practice the noise injected by the mini-batch sampling is even considered beneficial, as it helps the iterate escape poor local stationary points that plain gradient descent could get stuck in.

9.3.2 Backpropagation in Matrix Form

The scalar recursion above is what one implements on paper; what one implements on a computer is its **batched matrix** form, which processes all b examples of a mini-batch B in one pass and maps one-to-one onto the code of the next section. Stack the activation maps of the mini-batch as rows,

$$\mathbf{H}^l := \begin{bmatrix} (h_1^l)^\top \\ \vdots \\ (h_b^l)^\top \end{bmatrix} \in \mathbb{R}^{b \times k_l},$$
$$\mathbf{H}^0 := \mathbf{X} \in \mathbb{R}^{b \times d},$$

and collect the layer errors in the same layout, $\mathbf{\Delta}^l := \partial \widehat{R}_B / \partial \mathbf{Z}^l \in \mathbb{R}^{b \times k_l}$, whose (n, j)-th entry is the error δ_j^l of neuron j on example n. Then the entire algorithm is six matrix

identities:

$$\begin{aligned}
\textbf{(F1)}\quad & \mathbf{Z}^l = \mathbf{H}^{l-1}W_l + \mathbf{1}_b(b^l)^\top, && W_l \in \mathbb{R}^{k_{l-1}\times k_l},\\
\textbf{(F2)}\quad & \mathbf{H}^l = g(\mathbf{Z}^l) && \text{(elementwise)},\\
\textbf{(B1)}\quad & \boldsymbol{\Delta}^H = \tfrac{1}{b}\big(\widehat{\mathbf{Y}} - \mathbf{Y}\big), && \text{(output layer)}\\
\textbf{(B2)}\quad & \boldsymbol{\Delta}^{l-1} = \big(\boldsymbol{\Delta}^l W_l^\top\big) \odot g'(\mathbf{Z}^{l-1}), && \text{(recursion)}\\
\textbf{(B3)}\quad & \partial\widehat{R}_B/\partial W_l = (\mathbf{H}^{l-1})^\top\boldsymbol{\Delta}^l,\\
\textbf{(B4)}\quad & \partial\widehat{R}_B/\partial b^l = (\boldsymbol{\Delta}^l)^\top\mathbf{1}_b,
\end{aligned}$$

where $\odot$ is the elementwise (Hadamard) product and $\mathbf{1}_b \in \mathbb{R}^b$ is the all-ones vector. (F1)-(F2) are the forward pass; (B1)-(B4) are the backward pass. Two observations:

- **(B3) is the scalar rule $\partial\ell/\partial w_{ij}^l = \delta_j^l h_i^{l-1}$, summed over the mini-batch** — the matrix product $(\mathbf{H}^{l-1})^\top\boldsymbol{\Delta}^l$ performs exactly that sum, which is why one gradient step costs one matrix multiplication per layer rather than $|W|$ scalar operations. (The averaging factor $1/b$ that turns the sum into the mini-batch mean has already been folded into $\boldsymbol{\Delta}^H$ by (B1), so it needs no separate mention here.)
- **(B2) is the scalar recursion $\delta_j^l = g'(z_j^l)\sum_k \delta_k^{l+1} w_{jk}^{l+1}$** — the sum over the next layer's neurons k is the product with $W_l^\top$, and multiplying by g' is the Hadamard product. Every layer therefore needs exactly two things from its forward pass: its input $\mathbf{H}^{l-1}$ (for B3) and its pre-activation $\mathbf{Z}^l$ (for g' in B2). This is the entire content of the "cache the forward pass" instruction in the algorithm above.

(B1) for the softmax cross-entropy loss. We derived $\delta_j^H = \widehat{y}_j - y_j$ for the squared loss. For C-class classification the output layer is instead passed through a **softmax**,

$$\widehat{y}_j := \frac{\exp(z_j^H)}{\sum_{c=1}^C \exp(z_c^H)},$$
$$\ell(y,\widehat{y}) := -\sum_{c=1}^C y_c \log \widehat{y}_c \quad (y \text{ one-hot}),$$

and a short calculation gives the *same* form. Writing $\widehat{y}_j = \exp(z_j)/Z$ with $Z := \sum_c \exp(z_c)$,

$$\frac{\partial \widehat{y}_c}{\partial z_j^H} = \widehat{y}_c\big(\mathbb{1}(c{=}j) - \widehat{y}_j\big)$$
$$\Longrightarrow \quad \frac{\partial \ell}{\partial z_j^H} = -\sum_c \frac{y_c}{\widehat{y}_c}\,\widehat{y}_c\big(\mathbb{1}(c{=}j) - \widehat{y}_j\big) = -y_j + \widehat{y}_j\sum_c y_c = \widehat{y}_j - y_j,$$

using $\sum_c y_c = 1$. So $\delta_j^H = \widehat{y}_j - y_j$ holds for the squared loss with a linear output *and* for the cross-entropy loss with a softmax output: in both cases the error that starts the backward recursion is simply **prediction minus target**. This is not a coincidence — it holds for every matched pair of an exponential-family output layer and its negative log-likelihood loss.

9.4 Implementation: A Multilayer Perceptron From Scratch

We now implement (F1)-(F2) and (B1)-(B4) literally, with no automatic differentiation anywhere: each layer is an object exposing a `forward` map and the corresponding explicitly coded `backward` map, and the optimizer is the four-line stochastic gradient step of the previous section. Two things are worth watching for in the code below.

- **Every `backward` is one of the identities above**, in the same order: `Affine.backward` implements (B3), (B4) and the factor $\mathbf{\Delta}^l W_l^\top$ of (B2); `ReLU.backward` supplies the remaining $\odot\, g'(\mathbf{Z})$ factor; `SoftmaxCrossEntropy.backward` is (B1).
- **The gradients are verified against finite differences.** Since we claim to have differentiated the network by hand, we check that claim numerically: for a random weight w, the central difference $\big(\widehat{R}(w{+}\epsilon) - \widehat{R}(w{-}\epsilon)\big)/(2\epsilon)$ must agree with the analytic gradient to within the $O(\epsilon^2)$ truncation error. This **gradient check** is the standard way to debug a hand-written backward pass, and is worth internalizing as a technique in its own right.

The data set is `digits`: 1797 real handwritten digits scanned as 8×8 grayscale images ($d = 64$ features, $C = 10$ classes) — the same task as the MNIST example below, at a scale that trains from scratch in seconds.

```
# A multilayer perceptron implemented from scratch: every forward map, every
# backward map, and every gradient step is written out explicitly as a tensor
# operation. No automatic differentiation is used anywhere in this block --
# not a single tensor below carries requires_grad.

import torch as th
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits

th.manual_seed(0)

# Every tensor below is float64: the finite-difference gradient check needs
# more precision than the float32 that the libraries use by default.

# --- Affine layer: (F1) Z = H W + 1 b^T -----------------------------------

class Affine:
    def __init__(self, k_in, k_out):
        # He initialisation: Var[w] = 2/k_in keeps Var[z^l] ~ Var[z^{l-1}]
        self.W = th.randn(k_in, k_out, dtype=th.float64) * (2.0 / k_in) ** 0.5
        self.b = th.zeros(k_out, dtype=th.float64)

    def forward(self, H):
        self.H = H                          # cached: needed by (B3)
```

```
        return H @ self.W + self.b       # (b,k_in)(k_in,k_out) -> (b,k_out)

    def backward(self, Delta):
        # Delta = dR/dZ^l, shape (b,k_out). One line per identity:
        self.dW = self.H.T @ Delta       # (B3)  dR/dW_l = (H^{l-1})^T Delta^l
        self.db = Delta.sum(dim=0)       # (B4)  dR/db^l = (Delta^l)^T 1
        return Delta @ self.W.T          # (B2)  first factor: Delta^l W_l^T

    def step(self, alpha):
        self.W -= alpha * self.dW        # gradient step on the weights
        self.b -= alpha * self.db        # gradient step on the biases

# --- Activation layer: (F2) H = g(Z), with g the ReLU -----------------------

class ReLU:
    def forward(self, Z):
        self.mask = (Z > 0.0).to(Z.dtype)  # cached: this IS g'(Z)
        return Z * self.mask               # g(z) = max(0,z)

    def backward(self, dH):
        return dH * self.mask            # (B2) second factor: o g'(Z^{l-1})

    def step(self, alpha):
        pass                             # no parameters to update

# --- Loss layer: softmax + cross entropy, differentiated jointly ------------

class SoftmaxCrossEntropy:
    def forward(self, Z, Y):
        Z = Z - Z.max(dim=1, keepdim=True).values     # shift: exp cannot
↪   overflow
        E = Z.exp()
        self.Yhat = E / E.sum(dim=1, keepdim=True)    # softmax probabilities
        self.Y = Y                                    # one-hot targets
        return -(Y * (self.Yhat + 1e-12).log()).sum() / Z.shape[0]

    def backward(self):
        return (self.Yhat - self.Y) / self.Y.shape[0] # (B1) delta^H = yhat - y

# --- The network: a list of layers, traversed forwards, then backwards ------

class MLP:
    def __init__(self, sizes):
        self.layers = []
        for l in range(len(sizes) - 1):
            self.layers.append(Affine(sizes[l], sizes[l + 1]))
            if l < len(sizes) - 2:                    # no activation on output
```

```
                self.layers.append(ReLU())
        self.loss = SoftmaxCrossEntropy()

    def forward(self, X, Y):
        H = X                                          # H^0 := X
        for layer in self.layers:                      # l = 1 -> H
            H = layer.forward(H)
        return self.loss.forward(H, Y)

    def backward(self):
        Delta = self.loss.backward()                   # delta^H
        for layer in reversed(self.layers):            # l = H -> 1
            Delta = layer.backward(Delta)

    def step(self, alpha):
        for layer in self.layers:
            layer.step(alpha)

    def predict(self, X):
        H = X
        for layer in self.layers:
            H = layer.forward(H)
        return H.argmax(dim=1)

# --- Gradient check: analytic backward pass vs. central finite differences --

net = MLP([5, 4, 3])
Xc = th.randn(6, 5, dtype=th.float64)
Yc = th.eye(3, dtype=th.float64)[th.randint(0, 3, (6,))]
net.forward(Xc, Yc)
net.backward()
W = net.layers[0].W                                    # differentiate w.r.t. W_1
dW_analytic = net.layers[0].dW.clone()
eps, dW_numeric = 1e-6, th.zeros_like(W)
for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        W[i, j] += eps;     Rp = net.forward(Xc, Yc)
        W[i, j] -= 2 * eps; Rm = net.forward(Xc, Yc)
        W[i, j] += eps
        dW_numeric[i, j] = (Rp - Rm) / (2 * eps)
print("max |analytic - numeric| gradient error:",
      f"{(dW_analytic - dW_numeric).abs().max():.2e}")

# --- The pipeline: 8x8 handwritten digits, 1797 samples, 10 classes ---------

digits = load_digits()
X_all = th.tensor(digits.data, dtype=th.float64) / 16.0  # pixel range [0,1]
```

```
y_all = th.tensor(digits.target, dtype=th.long)

idx = th.randperm(y_all.numel())
n_tr = int(0.8 * y_all.numel())
tr, te = idx[:n_tr], idx[n_tr:]
mu, sd = X_all[tr].mean(dim=0), X_all[tr].std(dim=0) + 1e-8
X_tr, X_te = (X_all[tr] - mu) / sd, (X_all[te] - mu) / sd
y_tr, y_te = y_all[tr], y_all[te]
Y_tr = th.eye(10, dtype=th.float64)[y_tr]  # one-hot targets

net = MLP([64, 64, 32, 10])                          # d=64 -> 64 -> 32 -> C=10
alpha, batch_size, epochs = 0.5, 32, 60
curve_tr, curve_te = [], []

for epoch in range(epochs):
    order = th.randperm(n_tr)                        # B ~ Uniform(S), |B| = b
    for start in range(0, n_tr, batch_size):
        B = order[start:start + batch_size]
        net.forward(X_tr[B], Y_tr[B])                # forward pass  (F1),(F2)
        net.backward()                               # backward pass (B1)-(B4)
        net.step(alpha)                              # stochastic gradient step
    curve_tr.append((net.predict(X_tr) != y_tr).double().mean())
    curve_te.append((net.predict(X_te) != y_te).double().mean())

print(f"train error: {curve_tr[-1]:.4f}   test error: {curve_te[-1]:.4f}")

plt.figure(figsize=(6, 4))
plt.plot(curve_tr, label="training error $\\widehat{R}_S(h)$")
plt.plot(curve_te, label="test error (estimate of $R(h)$)")
plt.xlabel("epoch"); plt.ylabel("zero-one error"); plt.legend();
↪  plt.grid(alpha=.3)
plt.tight_layout()
plt.show()
```

```
max |analytic - numeric| gradient error: 9.15e-11
train error: 0.0000   test error: 0.0278
```

The analytic and numerical gradients agree to within 10^{-10}, confirming that the four backward identities are implemented correctly. The learning curve shows the training error driven to exactly 0 — the network has $64 \cdot 64 + 64 \cdot 32 + 32 \cdot 10 = 6{,}464$ weights against $m = 1437$ training examples, so it can interpolate the training set outright — while the test error settles near 3%. This is the empirical form of the puzzle the next section addresses: a hypothesis class whose VC dimension far exceeds m, achieving $\widehat{R}_S(h) = 0$, and generalizing anyway.

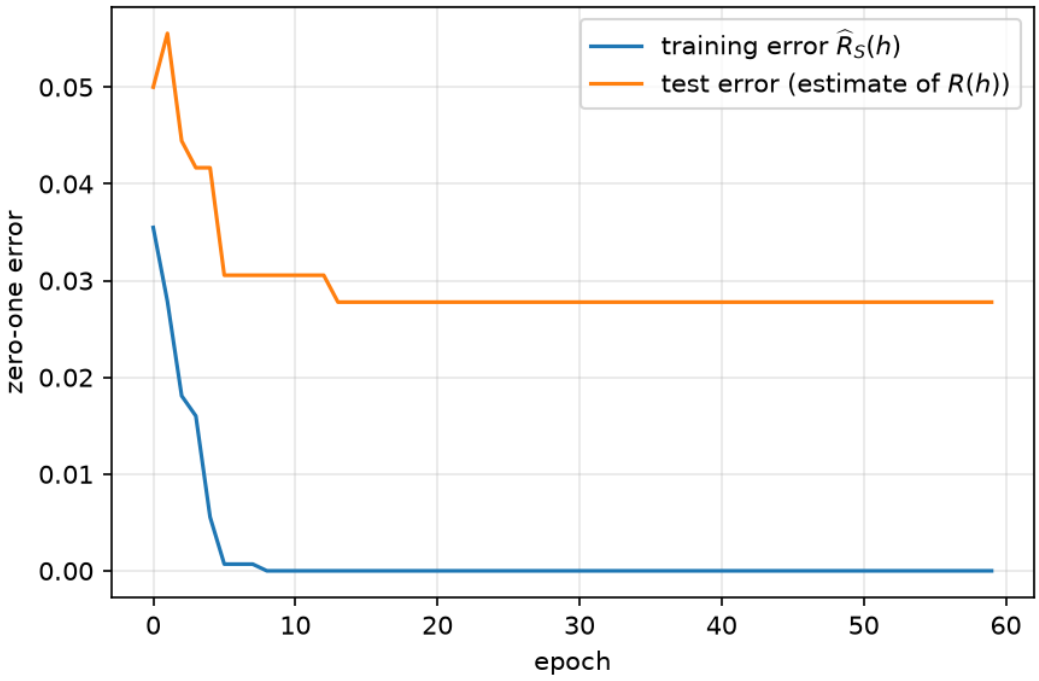


Figure 9.4: Learning curve of the from-scratch multilayer perceptron on the digits data set: training error reaches zero while test error settles near 3%.

9.5 PAC Analysis of Neural Networks

A two-layer network with k hidden units and unconstrained real-valued weights has $O(k \cdot (d+1)+k)$ trainable parameters, and its VC dimension grows accordingly — for the sign-thresholded output of a network with piecewise-linear activations (e.g. ReLU), d_{VC} scales at least linearly, and can scale super-linearly, in the number of weights. For any network wide enough to be useful in practice, this VC dimension vastly exceeds the number of training examples m, so the VC-dimension bound of Theorem 5.4 is **vacuous**: it promises nothing, since its right-hand side exceeds 1. Yet, deep networks with far more parameters than training examples routinely generalize well. Resolving this puzzle needs a complexity measure that does not grow with the number of parameters — exactly the motivation behind Rademacher complexity (Chapter 5a). We now derive such a bound for a simplified network, using only the tools already built in Chapter 5a.

Simplified setup. Consider a network with a single hidden layer of k units and a scalar output,

$$h_{W,w}(x) = w^\top g(Wx),$$
$$W \in \mathbb{R}^{k\times d},\ w \in \mathbb{R}^k,$$

where g is applied elementwise. We assume:

- g is 1-Lipschitz with $g(0)=0$ — true for ReLU (and for tanh, but *not* for the sigmoid, whose offset $g(0)=\frac{1}{2}$ would need to be carried through the argument separately);

- every row $w_j^{(1)}$ of W (the incoming weight vector of hidden unit j) satisfies $\|w_j^{(1)}\|_2 \le B_1$;
- the output weights satisfy $\|w\|_2 \le B_2$;
- $\|x\|_2 \le R$ almost surely under $\mathcal{D}$.

Write $\mathcal{H}_{k,B_1,B_2}$ for the resulting hypothesis class. Note that k, B_1, B_2 are hyperparameters we are free to fix in advance — capping them plays exactly the role that capping the leaf budget L played for decision trees in Chapter 7's own PAC analysis.

Theorem 9.5.1: Rademacher complexity of a two-layer network

$$\widehat{\mathfrak{R}}_S(\mathcal{H}_{k,B_1,B_2}) \le \frac{\sqrt{k}\, B_1 B_2 R}{\sqrt{m}}.$$

Proof. Fix the sample $S = \{x_1, \dots, x_m\}$. Since $h_{W,w}(x) = \sum_{j=1}^{k} w_j\, g(w_j^{(1)\top} x)$ is linear in w, the supremum over $\|w\|_2 \le B_2$ of a linear functional is attained by Cauchy-Schwarz, for every fixed realization of the Rademacher signs $\sigma_1, \dots, \sigma_m$:

$$\widehat{\mathfrak{R}}_S(\mathcal{H}_{k,B_1,B_2}) = \mathbb{E}_\sigma\Big[\sup_{W,w} \frac{1}{m}\sum_{i=1}^{m} \sigma_i\, w^\top g(W x_i)\Big] = \frac{B_2}{m}\,\mathbb{E}_\sigma\Big[\sup_{W} \Big\|\sum_{i=1}^{m} \sigma_i\, g(W x_i)\Big\|_2\Big].$$

Write $v_j(W) := \sum_i \sigma_i\, g(w_j^{(1)\top} x_i)$ for the j-th coordinate of the inner vector, where $w_j^{(1)}$ is the j-th row of W. Bounding the Euclidean norm by $\sqrt{k}$ times the sup-norm, and using that the k rows range independently over identical B_1-balls (so the outer supremum and the coordinate-wise maximum commute),

$$\begin{aligned}\sup_W \big\|v(W)\big\|_2 &\le \sqrt{k}\, \sup_W \max_j |v_j(W)| = \sqrt{k}\, \max_j \sup_{\|w_j^{(1)}\|_2 \le B_1} |v_j(W)| \\ &= \sqrt{k}\, \sup_{\|u\|_2 \le B_1} \Big|\sum_i \sigma_i\, g(u^\top x_i)\Big|,\end{aligned}$$

where the last equality holds because every one of the k terms being maximized over is, by construction, the *same* scalar quantity — the row index j only labels which weight vector we call u, the constraint set $\{\|u\|_2 \le B_1\}$ and the data $x_1, \dots, x_m$ being identical for every hidden unit. This collapses a k-fold maximum into a single scalar supremum, over a class with a single B_1-norm-bounded weight vector. Taking $\mathbb{E}_\sigma$ of both sides and recalling that g is 1-Lipschitz with $g(0) = 0$, the contraction lemma (Lemma 5a.2, in the unsigned-but-$\phi(0)$=0 form noted there) now applies validly — because it is invoked *after* the maximum has collapsed to a single scalar class, not pointwise inside a supremum over k simultaneously varying rows:

$$\mathbb{E}_\sigma\Big[\sup_{\|u\|_2 \le B_1} \Big|\sum_i \sigma_i\, g(u^\top x_i)\Big|\Big] \le \mathbb{E}_\sigma\Big[\sup_{\|u\|_2 \le B_1} \Big|\sum_i \sigma_i\, u^\top x_i\Big|\Big] = B_1\, \mathbb{E}_\sigma\Big[\Big\|\sum_i \sigma_i x_i\Big\|_2\Big],$$

the last equality by Cauchy-Schwarz. Finally, by Jensen's inequality and $\mathbb{E}[\sigma_i \sigma_{i'}] = 0$ for $i \ne i'$,

$$\mathbb{E}_\sigma\Big[\Big\|\sum_i \sigma_i x_i\Big\|_2\Big] \le \sqrt{\mathbb{E}_\sigma\Big[\Big\|\sum_i \sigma_i x_i\Big\|_2^2\Big]} = \sqrt{\sum_i \|x_i\|_2^2} \le R\sqrt{m}.$$

Assembling the pieces, $\widehat{\mathfrak{R}}_S(\mathcal{H}_{k,B_1,B_2}) \le \frac{B_2}{m} \cdot \sqrt{k}\, B_1 \cdot R\sqrt{m} = \frac{\sqrt{k}\, B_1 B_2 R}{\sqrt{m}}$. □

Consequences. Plugging Theorem 8.1 into Theorem 5a.2 (after one more application of the contraction lemma to pass from $\mathcal{H}_{k,B_1,B_2}$ to a Lipschitz loss composed with it) gives, with probability at least $1-\delta$, simultaneously for every $h \in \mathcal{H}_{k,B_1,B_2}$,

$$R(h) \le \widehat{R}_S(h) + O\Big(\frac{\sqrt{k}\, B_1 B_2 R}{\sqrt{m}}\Big) + O\Big(\sqrt{\frac{\log(1/\delta)}{m}}\Big).$$

Three things are worth noting about this bound:

- It **does not depend on the input dimension** d — only on the per-neuron weight-norm bound B_1, the output-norm bound B_2, the data-norm bound R, and the width k. This is precisely the phenomenon flagged in Theorem 5a.4: capacity control by norm, rather than by counting parameters, is what makes the bound meaningful for models with far more parameters than VC-dimension arguments could ever tolerate.
- It scales with $\sqrt{k}$, not with the number of parameters $k(d+1)$. Width still costs something — but far less than a naive parameter-counting argument (à la Theorem 5.4) would suggest, and this residual $\sqrt{k}$ is itself an artifact of the crude row-independent decomposition step above; sharper arguments based on matrix covering numbers (Golowich, Rakhlin & Shamir, 2018) remove even this factor, at the cost of machinery beyond this course's scope.
- The bound gives no credit to gradient descent for finding a good hypothesis — it only certifies that *if* training happens to land on a hypothesis with small weight norms, that hypothesis generalizes well. This matches an empirical regularity of trained networks: gradient descent on over-parameterized networks tends to find solutions with small weight norms even without explicit regularization, an implicit-bias phenomenon that is an active research topic and, again, outside our scope — but it is exactly why this norm-based, width-independent view of capacity, rather than the VC-dimension view, is the one that matches practice. We revisit this network in a further-simplified, exactly solvable form — where only the output layer is trained and the hidden layer is fixed — in the Neural Tangent Kernel discussion of Chapter 9.

Nearly every neural network used nowadays is trained by minimizing a loss function using stochastic gradient descent with backpropagation. The common practice is to use an automatic differentiation library such as TensorFlow or PyTorch that automates the process of calculating the gradients. These libraries also provide a variety of neural network architectures and optimization algorithms. Below, we give an example implementation of a neural network using PyTorch on a handwritten digit classification problem. The used data set is among the most famous data sets in machine learning, called MNIST. It contains 70,000 images of handwritten digits. Each image is a 28x28 grayscale image. The goal is to classify the images into 10 classes, one for each digit.

```
# MNIST digits classification with a simple deep neural network,
# this time written the way it is written in practice: with the library's
# own layer, loss and optimizer objects standing in for the code above.

import torch as th
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Load the data from torchvision

# Define the transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Download the data
train_ds = datasets.MNIST('data', train=True,
            download=True, transform=transform)
test_ds = datasets.MNIST('data', train=False,
            download=True, transform=transform)

# Create the data loaders
batch_size = 64 # Define the batch size
train_dl = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
test_dl = DataLoader(test_ds, batch_size=batch_size, shuffle=False)

# Define the model
model = nn.Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 512),
    nn.ReLU(),
    nn.Linear(512, 1024),
    nn.ReLU(),
    nn.Linear(1024, 10)
)

# Define the loss function
loss_fn = nn.CrossEntropyLoss()

# Define the optimizer
optimizer = th.optim.SGD(model.parameters(), lr=1e-4)

# Train the model, tracking the training loss and test accuracy after
# every epoch so that we can plot a proper learning curve instead of a
```

```
# wall of per-batch print statements.
num_epochs = 40
train_losses, test_accuracies = [], []

def evaluate(model, test_dl):
    model.eval()
    correct = 0
    for xb, yb in test_dl:
        y_pred = model(xb.view(-1, 784))
        preds = y_pred.argmax(dim=1)
        correct += (preds == yb).float().mean().item()
    model.train()
    return correct / len(test_dl)

for epoch in range(num_epochs):
    epoch_loss = 0.0
    for xb, yb in train_dl:
        # Forward pass
        y_pred = model(xb.view(-1, 784))
        loss = loss_fn(y_pred, yb)

        # Backward pass
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        epoch_loss += loss.item()

    train_losses.append(epoch_loss / len(train_dl))
    test_accuracies.append(evaluate(model, test_dl))

print(f"final training loss: {train_losses[-1]:.4f}   "
      f"final test accuracy: {test_accuracies[-1]:.4f}")

fig, axes = plt.subplots(ncols=2, figsize=(11, 4))
axes[0].plot(range(1, num_epochs + 1), train_losses)
axes[0].set_xlabel("epoch"); axes[0].set_ylabel("training loss");
↪  axes[0].grid(alpha=.3)
axes[1].plot(range(1, num_epochs + 1), test_accuracies, color="tab:red")
axes[1].set_xlabel("epoch"); axes[1].set_ylabel("test accuracy");
↪  axes[1].grid(alpha=.3)
plt.tight_layout()
plt.show()
```

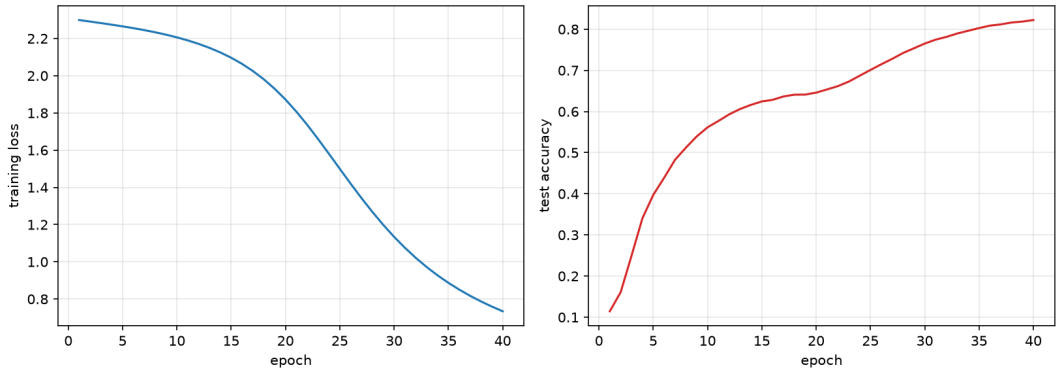


Figure 9.5: Learning curve of the library-based MLP on MNIST: training cross-entropy loss (left) and test accuracy (right) across epochs of plain SGD.

CHAPTER 10

KERNEL METHODS

10.1 Transforming the input space

Linear predictors have many desirable properties. They:

- can be trained by analytical methods,
- are easy to interpret for humans,
- are theoretically well understood.

However, their expressiveness is limited. For example, they cannot represent the XOR function. The first solution we presented to improve the expressiveness of linear predictors was to pass them through an activation function and build a network of them. The second is **kernel methods**. The core idea is to transform the input space to another space where the problem can be solved by linear methods. For instance, in classification, we expect from the transformed space that there exists a hyperplane that perfectly separates the classes. This way, nonlinearity is handled by the transformation and all the nice properties of linear predictors are preserved.

Consider the case that 2D feature space $x = (x_1, x_2) \in \mathbb{R}^2$ contains data points from two classes, one is concentrated around the origin in the shape of a sphere and the other surrounds this sphere in the shape of a ring. The data is not linearly separable in the original space. However, if we transform the data to a new space $\phi(x) = (x_1^2, x_2^2, \sqrt{2}x_1x_2) \in \mathbb{R}^3$, the classes become linearly separable as illustrated below.

```
import torch as th
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

plt.rcParams['text.usetex'] = True # Enable Latex in plots

# Step 1: Generate random samples for two classes
th.manual_seed(0)

# Number of samples for each class
```

```
num_samples = 100

# Class 1:
radius1 = 3
theta1 = 2 * th.pi * th.rand(num_samples)
r1 = th.rand(num_samples).sqrt() * radius1
class1_x = r1 * th.cos(theta1)
class1_y = r1 * th.sin(theta1)

# Class 2
radius2 = 5
theta2 = 2 * th.pi * th.rand(num_samples)
r2 = (-0.5 + th.rand(num_samples)) + radius2
class2_x = r2 * th.cos(theta2)
class2_y = r2 * th.sin(theta2)

# Step 2: Define the transformation function to 3D
def transform_to_3d(x, y):
    x_3d = x**2
    y_3d = y**2
    z_3d = (2 ** 0.5) * x * y
    return x_3d, y_3d, z_3d

# Apply the transformation to both classes
class1_x_3d, class1_y_3d, class1_z_3d = transform_to_3d(class1_x, class1_y)
class2_x_3d, class2_y_3d, class2_z_3d = transform_to_3d(class2_x, class2_y)

# Create a figure with two subplots
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# The classes are not linearly separable in 2D
axes[0].scatter(class1_x, class1_y, label='Class 1', c='b')
axes[0].scatter(class2_x, class2_y, label='Class 2', c='r')
axes[0].set_xlabel('$x_1$', fontsize=14)
axes[0].set_ylabel('$x_2$', fontsize=14)
axes[0].set_title('Original 2D Data')
axes[0].legend(loc='best')

# The classes are linearly separable in 3D after transformation
ax = fig.add_subplot(1, 2, 2, projection='3d', proj_type='ortho')

ax.scatter(class1_x_3d, class1_y_3d, class1_z_3d, label='Class 1', c='b')
ax.scatter(class2_x_3d, class2_y_3d, class2_z_3d, label='Class 2', c='r')
ax.set_xlabel('$x_1^2$', fontsize=14)
ax.set_ylabel('$x_2^2$', fontsize=14)
ax.set_zlabel('$\sqrt{2}  x_1  x_2$')
```

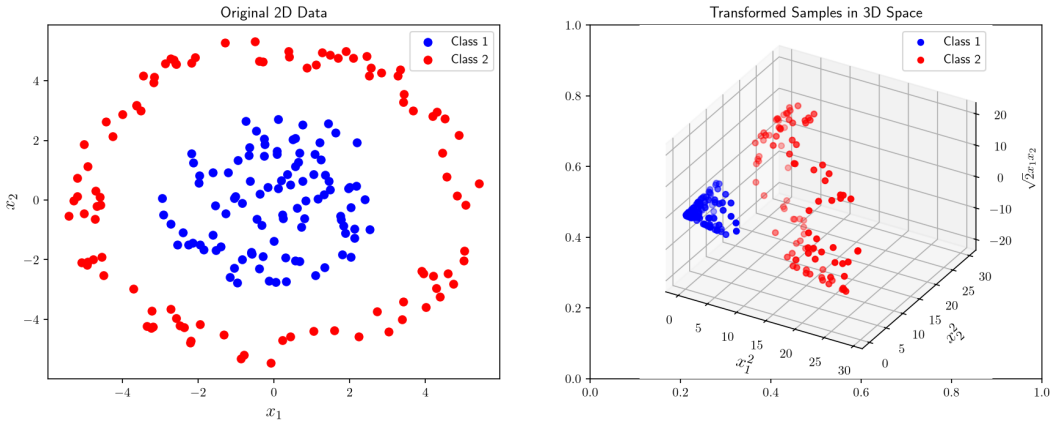


Figure 10.1: Two classes that are not linearly separable in the original 2D input space (left) become linearly separable after the feature transformation $\phi(x) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$ (right).

```
ax.legend(loc='best')
ax.set_title('Transformed Samples in 3D Space')

plt.show()
```

10.2 Kernel Trick

In real-world applications, it is not as straightforward to find a suitable transformation space as in the example above that would make the problem linearly separable. The best one can often do is to transform the data to an as high dimensional space as possible. However, this is computationally expensive. There are attractive mathematical tools that allow us to model the **similarity** of a pair of data points in the transformation space without explicitly computing the representations of these data points in the transformation space. This can be achieved by a special family of functions, called **kernel functions** with certain characteristics. Given an input space $\mathcal{X}$, we would like to have a kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ with the following property:

$$k(x_i, x_j) = \phi(x_i)^\top \phi(x_j).$$

In words, we would like to design directly the left hand side of the equation above without going through the computational steps of the right hand side. This is called the **kernel trick**. This is possible for only certain types of functions that have specific properties. The following result, due to (Mercer, 1909), gives a necessary and sufficient condition for a function to be a valid kernel function.

Theorem 10.2.1: Mercer

A symmetric function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ admits a feature map ϕ with $k(x, x') = \phi(x)^\top \phi(x')$ **if and only if** the Gram matrix it generates (i.e. the matrix generated by evaluating the function on all pairs of data points in an arbitrary data set with m data points)

$$K = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_m) \\ \vdots & \ddots & \vdots \\ k(x_m, x_1) & \cdots & k(x_m, x_m) \end{bmatrix}$$

is positive semi-definite, i.e. $v^\top K v \geq 0$ for all $v \in \mathbb{R}^m$.

Proof (of the "only if" direction). Suppose $k(x, x') = \phi(x)^\top \phi(x')$ for some feature map ϕ. Then for any $x_1, \ldots, x_m$ and any $v \in \mathbb{R}^m$,

$$v^\top K v = \sum_{i=1}^{m} \sum_{j=1}^{m} v_i v_j \, \phi(x_i)^\top \phi(x_j) = \Big\| \sum_{i=1}^{m} v_i \, \phi(x_i) \Big\|_2^2 \;\geq\; 0,$$

since the double sum is exactly the squared norm of the single vector $\sum_i v_i \phi(x_i)$. Symmetry of K is immediate from symmetry of the inner product □

The converse — that positive semi-definiteness *suffices*, i.e. that some feature map always exists — is the substantial half of the theorem, and it is proved constructively in the section on reproducing kernel Hilbert spaces below (Theorem 9.2), where the feature map is built as $\phi(x) := k(x, \cdot)$ and the feature space as a space of functions on $\mathcal{X}$. Everything in that construction is carried out here except one analytic step — completing a pre-Hilbert space — which we state but do not perform. Until then, the "only if" direction just proved, together with the composition rules below, is already enough to certify every kernel we actually employ.

The most commonly used kernel functions are:

- **Linear Kernel:** $k(x, y) = x^\top y$
- **Polynomial Kernel:** $k(x, y) = \left(x^\top y + c\right)^d$
- **Radial Basis Function (RBF) Kernel:** $k(x, y) = \exp(-\frac{\|x-y\|^2}{2\sigma^2})$
- **Sigmoid Kernel:** $k(x, y) = \tanh\left(\gamma x^\top y + c\right)$

There also exist kernels that compute the similarity of a pair of graphs, strings, or other complex objects. The nice property of the kernel functions is that one can compute these similarities directly, without needing to define suitable feature spaces for these object types.

One can imagine that it is not straightforward to devise a function that satisfies this condition. However, there are certain rules that allow us to construct new valid kernel functions from existing ones. These rules are called **kernel composition rules**.

Proposition 10.2.2

Kernel composition rules. Let k_1, k_2 be valid kernels, $a \in \mathbb{R}^+$ a constant, and B a positive semi-definite matrix. Then each of the following is a valid kernel:

- $k(x, x') = k_1(x, x') + k_2(x, x')$
- $k(x, x') = a \cdot k_1(x, x')$
- $k(x, x') = k_1(x, x') \cdot k_2(x, x')$
- $k(x, x') = x^\top B x'$

Proof. By Theorem 9.1 it suffices to verify that each rule produces a positive semi-definite Gram matrix. Write K_1, K_2 for the Gram matrices of k_1, k_2 on an arbitrary sample, both positive semi-definite by hypothesis.

Sum. $v^\top (K_1 + K_2) v = v^\top K_1 v + v^\top K_2 v \geq 0$.

Positive scaling. $v^\top (aK_1) v = a\,(v^\top K_1 v) \geq 0$ since $a > 0$.

Product. The Gram matrix of the product kernel is the entrywise (Hadamard) product $K_1 \odot K_2$, and the **Schur product theorem** states that the Hadamard product of two positive semi-definite matrices is positive semi-definite. Directly: write $K_2 = \sum_r \lambda_r u_r u_r^\top$ with $\lambda_r \geq 0$ (spectral decomposition of a symmetric positive semi-definite matrix). Then for any v,

$$v^\top (K_1 \odot K_2) v = \sum_{i,j} v_i v_j (K_1)_{ij} \sum_r \lambda_r u_{r,i} u_{r,j} = \sum_r \lambda_r \underbrace{\big(v \odot u_r\big)^\top K_1 \big(v \odot u_r\big)}_{\geq 0} \;\geq\; 0.$$

Bilinear form. Since B is symmetric positive semi-definite, it has a square root $B^{1/2}$ with $B = B^{1/2} B^{1/2}$, so $x^\top B x' = (B^{1/2} x)^\top (B^{1/2} x')$ exhibits the feature map $\phi(x) := B^{1/2} x$ directly $\square$

These four rules already certify most kernels used in practice. The polynomial kernel $(x^\top x' + c)^d$, for instance, is built from the linear kernel by d applications of the product rule together with the sum rule and positive scaling (expanding the binomial), and the RBF kernel is obtained as a limit of such combinations.

10.3 The Reproducing Kernel Hilbert Space

Mercer's theorem guarantees that a positive semi-definite kernel *has* a feature map, but says nothing about which one. There are in general many: the polynomial kernel $(x^\top x')^2$ on $\mathbb{R}^2$ is reproduced by $\phi(x) = (x_1^2, x_2^2, \sqrt{2} x_1 x_2)$ and equally by any rotation of it. One choice is canonical, and it turns out to be the one that lets us stop talking about ϕ altogether and talk instead about a **space of functions** — which is what a hypothesis class is. That space is the RKHS of k, following the formulation of (Aronszajn, 1950).

Definition 10.3.1: Reproducing kernel Hilbert space

A **reproducing kernel Hilbert space (RKHS)** for a kernel k on $\mathcal{X}$ is a Hilbert space $\mathcal{H}_k$ of functions $f : \mathcal{X} \to \mathbb{R}$, with inner product $\langle \cdot, \cdot \rangle_{\mathcal{H}_k}$ and induced norm $\|f\|_{\mathcal{H}_k} := \sqrt{\langle f, f \rangle_{\mathcal{H}_k}}$, such that

1. $k(x, \cdot) \in \mathcal{H}_k$ for every $x \in \mathcal{X}$, and
2. (**reproducing property**) $\langle f, k(x, \cdot) \rangle_{\mathcal{H}_k} = f(x)$ for every $f \in \mathcal{H}_k$ and every $x \in \mathcal{X}$.

The reproducing property is the whole point: it says that *evaluating* a function at a point is the same operation as taking an inner product against a fixed element of the space. Applying it twice with $f = k(x', \cdot)$ gives

$$\langle k(x, \cdot),\, k(x', \cdot) \rangle_{\mathcal{H}_k} = k(x, x'),$$

so $\phi(x) := k(x, \cdot)$ is a feature map for k — the **canonical feature map** — whose feature space is $\mathcal{H}_k$ itself. Every function in $\mathcal{H}_k$ is simultaneously a hypothesis and a point of the feature space.

Theorem 10.3.2: Moore-Aronszajn

Every positive semi-definite kernel k has exactly one RKHS.

Proof (construction; completeness omitted). Start from the linear span of the kernel sections,

$$\mathcal{V} := \Big\{ f = \sum_{i=1}^{n} a_i\, k(x_i, \cdot) \;:\; n \in \mathbb{N},\; a_i \in \mathbb{R},\; x_i \in \mathcal{X} \Big\},$$

and define, for $f = \sum_i a_i k(x_i, \cdot)$ and $g = \sum_j b_j k(x'_j, \cdot)$,

$$\langle f, g \rangle_{\mathcal{V}} := \sum_{i=1}^{n} \sum_{j=1}^{n'} a_i b_j\, k(x_i, x'_j). \tag{9.1}$$

Well defined. The right-hand side of (9.1) refers to the coefficients a_i and points x_i, which are not unique for a given f. But rewriting it two ways,

$$\sum_{i,j} a_i b_j k(x_i, x'_j) = \sum_{j=1}^{n'} b_j\, f(x'_j) = \sum_{i=1}^{n} a_i\, g(x_i),$$

exhibits it once as a quantity depending on f only through its *values*, and once as a quantity depending on g only through its values. Hence it does not depend on either representation.

Inner product. Bilinearity is immediate from (9.1) and symmetry from $k(x, x') = k(x', x)$. For positive semi-definiteness, $\langle f, f \rangle_{\mathcal{V}} = \sum_{i,j} a_i a_j k(x_i, x_j) = a^\top K a \geq 0$, which is exactly the

Gram-matrix condition of Theorem 9.1 — this is the *only* place the assumption on k is used, and it is used in full.

Reproducing property. Taking $g = k(x, \cdot)$ (i.e. $n' = 1$, $b_1 = 1$, $x'_1 = x$) in (9.1) gives $\langle f, k(x, \cdot)\rangle_{\mathcal{V}} = \sum_i a_i k(x_i, x) = f(x)$.

Definiteness. We must still rule out a non-zero f with $\|f\|_{\mathcal{V}} = 0$. The Cauchy-Schwarz inequality holds for any positive semi-definite bilinear form, so combining it with the reproducing property, for every x,

$$|f(x)| = \big|\langle f, k(x, \cdot)\rangle_{\mathcal{V}}\big| \;\le\; \|f\|_{\mathcal{V}}\, \|k(x, \cdot)\|_{\mathcal{V}} = \|f\|_{\mathcal{V}} \sqrt{k(x, x)}. \tag{9.2}$$

If $\|f\|_{\mathcal{V}} = 0$ then $f(x) = 0$ for every x, i.e. f is the zero *function*. So $\langle \cdot, \cdot\rangle_{\mathcal{V}}$ is a genuine inner product on $\mathcal{V}$, and $(\mathcal{V}, \langle \cdot, \cdot\rangle_{\mathcal{V}})$ is a pre-Hilbert space satisfying both conditions of Definition 9.1.

The remaining step is to **complete** $\mathcal{V}$ — adjoin the limits of its Cauchy sequences — and to check that the completion still consists of functions on $\mathcal{X}$ and still reproduces k. Bound (9.2) is what makes this work: convergence in $\|\cdot\|_{\mathcal{V}}$ forces pointwise convergence, so a Cauchy sequence of functions has a well-defined pointwise limit to attach to. Carrying this out, together with the uniqueness claim, is a standard exercise in functional analysis that we do not reproduce here □

Two consequences are worth recording immediately. First, the construction **completes the proof of Mercer's theorem**: the converse direction we deferred in Theorem 9.1 is exactly the statement that a positive semi-definite k admits a feature map, and $\phi(x) := k(x, \cdot)$ into $\mathcal{H}_k$ is one. Second, every element of $\mathcal{H}_k$ is a limit of finite kernel expansions $\sum_i a_i k(x_i, \cdot)$ — so the hypothesis class generated by a kernel is precisely the closure of the set of weighted similarity scores to a finite set of points. This is the formal content of the informal description of kernel methods as "predicting by weighted similarity to remembered examples".

Proposition 10.3.3

The RKHS norm controls the function. For every $f \in \mathcal{H}_k$ and every $x \in \mathcal{X}$, $|f(x)| \le \|f\|_{\mathcal{H}_k} \sqrt{k(x, x)}$. In particular, if $k(x, x) \le R^2$ uniformly, then $\sup_x |f(x)| \le R\|f\|_{\mathcal{H}_k}$.

Proof. This is (9.2), which used only the reproducing property and Cauchy-Schwarz, both available in $\mathcal{H}_k$ □

Proposition 9.2 is the reason $\|\cdot\|_{\mathcal{H}_k}$ is the right complexity measure for kernel methods, and it identifies the hypothesis class of the generalization section below: for a linear predictor $x \mapsto w^\top \phi(x)$ written in the canonical feature map, w *is* a function $f \in \mathcal{H}_k$ and $\|w\|_2$ *is* $\|f\|_{\mathcal{H}_k}$, so

$$\mathcal{H}_B \;=\; \big\{ f \in \mathcal{H}_k : \|f\|_{\mathcal{H}_k} \le B \big\}$$

is a class of uniformly bounded functions, whatever the dimension of the feature space happens to be. That is precisely the norm-constrained, dimension-free regime that Theorem 5a.4 was built for.

What the norm measures. For the RBF kernel the RKHS norm penalizes wiggliness: one can show that $\|f\|_{\mathcal{H}_k}^2$ equals a weighted integral of $|\widehat{f}(\omega)|^2$ against $e^{\sigma^2\|\omega\|^2/2}$ over the frequency domain, so high-frequency components are charged exponentially, the more so the larger σ is. This makes the bandwidth σ and the regularization coefficient λ two different dials on the same quantity — σ decides *which* functions are expensive, λ decides *how much* we are willing to pay — and explains, before any experiment, why the fits below vary so sharply with σ.

10.3.1 The Representer Theorem

The RKHS is typically infinite-dimensional, so minimizing an objective over all of $\mathcal{H}_k$ looks like an infinite-dimensional optimization problem. It is not: the solution is always a finite kernel expansion on the training inputs, with one coefficient per training example. This is the single result that makes kernel methods computable, due to (Kimeldorf and Wahba, 1971).

Theorem 10.3.4: Representer theorem

Let $S = \{(x_i, y_i)\}_{i=1}^m$, let $\Psi : \mathbb{R}^m \to \mathbb{R}$ be an arbitrary function of the m predicted values (a "data-fit" term), and let $\text{pen} : [0, \infty) \to \mathbb{R}$ be a **strictly increasing** regularizer, in the sense of Chapter 2. Then every minimizer f^* of

$$\min_{f \in \mathcal{H}_k} \; \Psi\big(f(x_1), \dots, f(x_m)\big) + \text{pen}\big(\|f\|_{\mathcal{H}_k}\big)$$

admits a representation

$$f^*(\cdot) = \sum_{i=1}^{m} a_i \, k(x_i, \cdot), \qquad a \in \mathbb{R}^m.$$

Proof. Let $\mathcal{V}_S := \text{span}\{k(x_1, \cdot), \dots, k(x_m, \cdot)\} \subseteq \mathcal{H}_k$, a finite-dimensional and therefore closed subspace, and decompose any $f \in \mathcal{H}_k$ orthogonally as

$$f = f_\parallel + f_\perp, \qquad f_\parallel \in \mathcal{V}_S, \quad f_\perp \perp \mathcal{V}_S.$$

The data-fit term Ψ cannot see $f_\perp$. By the reproducing property, for every training input x_j,

$$f(x_j) = \langle f, k(x_j, \cdot)\rangle_{\mathcal{H}_k} = \langle f_\parallel, k(x_j, \cdot)\rangle_{\mathcal{H}_k} + \underbrace{\langle f_\perp, k(x_j, \cdot)\rangle_{\mathcal{H}_k}}_{=0, \text{ since } k(x_j,\cdot) \in \mathcal{V}_S} = f_\parallel(x_j),$$

so $\Psi(f(x_1), \dots, f(x_m)) = \Psi(f_\parallel(x_1), \dots, f_\parallel(x_m))$ exactly.

The regularizer can only be hurt by $f_\perp$. By the Pythagorean identity in a Hilbert space,

$$\|f\|_{\mathcal{H}_k}^2 = \|f_\parallel\|_{\mathcal{H}_k}^2 + \|f_\perp\|_{\mathcal{H}_k}^2 \;\geq\; \|f_\parallel\|_{\mathcal{H}_k}^2,$$

with equality if and only if $f_\perp = 0$. Since pen is strictly increasing, $\mathrm{pen}(\|f\|_{\mathcal{H}_k}) \geq \mathrm{pen}(\|f_\parallel\|_{\mathcal{H}_k})$, strictly so unless $f_\perp = 0$.

Adding the two, the objective at $f_\parallel$ is no larger than at f, and is **strictly** smaller whenever $f_\perp \neq 0$. A minimizer therefore has $f^*_\perp = 0$, i.e. $f^* = f^*_\parallel \in \mathcal{V}_S$, which is the claimed form $\square$

Read the proof again for what it does *not* assume. Nothing about Ψ: it may be built from the squared loss, the hinge loss of an SVM, the logistic loss, or a non-convex loss with many minima; it need not even be continuous, and it need not decompose across examples at all. The theorem is driven entirely by the reproducing property (which makes Ψ blind to $f_\perp$) and by orthogonality (which makes pen charge for $f_\perp$ without getting anything in return). What *is* essential is that the regularizer be a strictly increasing function of the RKHS norm specifically — this is why kernel methods regularize with $\|f\|_{\mathcal{H}_k}$ and not with something else.

Corollary 10.3.5

Kernel ridge regression. Take $\Psi(u_1, \ldots, u_m) := \sum_{i=1}^m (y_i - u_i)^2$ and $\mathrm{pen}(t) := \lambda t^2$ with $\lambda > 0$. Then the minimizer over $\mathcal{H}_k$ is $f^*(\cdot) = \sum_i a_i k(x_i, \cdot)$ with

$$a = (K + \lambda I)^{-1} y.$$

Proof. By Theorem 9.3 we may restrict the search to $f = \sum_i a_i k(x_i, \cdot)$, at which point the reproducing property and (9.1) evaluate both terms in terms of the Gram matrix alone:

$$f(x_j) = \sum_i a_i k(x_i, x_j) = (Ka)_j,$$
$$\|f\|^2_{\mathcal{H}_k} = \sum_{i,j} a_i a_j k(x_i, x_j) = a^\top K a.$$

The infinite-dimensional problem has become the m-dimensional one

$$\min_{a \in \mathbb{R}^m} \ \|y - Ka\|_2^2 + \lambda\, a^\top K a,$$

whose gradient is $-2Ky + 2KKa + 2\lambda Ka = 2K\big[(K + \lambda I)a - y\big]$. Setting it to zero is solved by $(K + \lambda I)a = y$, and $K + \lambda I \succ 0$ for $\lambda > 0$ makes that system uniquely solvable $\square$

This is exactly the solution the dual derivation of the previous section arrived at, and exactly what `precompute_a` computes in the implementation below — but obtained without ever mentioning ϕ, Φ or w. The dual derivation had to *discover* that w lies in the span of the training features, by inspecting the stationarity condition of one particular objective; the representer theorem says it in advance, for every objective of that shape. The practical readings are:

- **Cost.** Training is $O(m^3)$ (one linear solve) and prediction $O(m)$ per query, *independent of the dimension of the feature space* — including when that dimension is infinite, as for the RBF kernel. What is traded away is a cost that grows with the sample size instead, which is the precise sense in which kernel methods are "non-parametric": the model that must be stored is the training set itself.

- **Where the guarantee comes from.** The trained predictor satisfies $\|f^*\|_{\mathcal{H}_k}^2 = a^\top K a \le \|y\|_2^2/\lambda$ by comparing the objective at f^* against the candidate $f = 0$, so λ directly caps the RKHS norm — hence, via Proposition 9.2 and Theorem 5a.4, the Rademacher complexity of the class the solution came from. The section on generalization below makes this quantitative.

10.4 Kernel regression

Let us remember the linear methods . It is possible to apply the kernel trick to all of them, i.e. to **kernelize** them. Let us take **ridge regression** as an example. Consider the case where the raw inputs x_i are passed through a feature extraction step $\phi(x_i)$. Call the matrix that contains $\phi(x_i)^\top$ in its rows as $\Phi \in \mathbb{R}^{m\times d}$. The ridge regression objective can then be expressed as

$$\begin{aligned}\mathcal{L}(w) &= \sum_{i=1}^{m}(y_i - w^\top\phi(x_i))^2 + \lambda||w||_2^2 \\ &= y^\top y - 2y^\top\Phi w + w^\top\Phi^\top\Phi w + \lambda w^\top w,\end{aligned}$$

where $y \in \mathbb{R}^m$ is the target vector and $w \in \mathbb{R}^d$ is the weight vector. We perform learning by minimizing this objective with respect to the weights:

$$\begin{aligned}\nabla_w\mathcal{L}(w) &= -2\Phi^\top y + 2\Phi^\top\Phi w + 2\lambda w \triangleq 0 \\ &\Rightarrow (\Phi^\top\Phi + \lambda I)w = \Phi^\top y \\ &\Rightarrow w = (\Phi^\top\Phi + \lambda I)^{-1}\Phi^\top y.\end{aligned}$$

This primal solution still requires inverting a $d \times d$ matrix and computing $\phi(\cdot)$ explicitly, both of which can be prohibitive when ϕ maps into a very high (or infinite) dimensional space. Since the gradient stationarity condition $\Phi^\top\Phi w + \lambda w = \Phi^\top y$ implies $w = \frac{1}{\lambda}\Phi^\top(y - \Phi w)$, the optimal w always lies in the span of the training feature vectors $\phi(x_1), \ldots, \phi(x_m)$. We can therefore reparametrize it as

$$w = \Phi^\top a, \qquad a \in \mathbb{R}^m,$$

and search for the optimal a instead. Placing this expression into the objective gives its so-called **dual representation**, which contains data only through inner products in the transformed space:

$$\mathcal{L}(a) = y^\top y - 2y^\top\Phi\Phi^\top a + a^\top\Phi\Phi^\top\Phi\Phi^\top a + \lambda a^\top\Phi\Phi^\top a.$$

Note that $K := \Phi\Phi^\top$ is the $m \times m$ dimensional **Gram** matrix with $K_{ij} = \phi(x_i)^\top\phi(x_j)$. This inner product can be replaced by any kernel function, generating a Gram matrix with entries $K_{ij} = k(x_i, x_j)$. The kernelized objective then reads

$$\mathcal{L}(a) = y^\top y - 2y^\top K a + a^\top K K a + \lambda a^\top K a.$$

Setting the gradient of $\mathcal{L}$ with respect to a to zero and solving for a gives

$$\begin{aligned}\nabla_a \mathcal{L}(a) &= -2Ky + 2KKa + 2\lambda Ka \triangleq 0\\ &\Rightarrow K\big[(K+\lambda I)a - y\big] = 0\\ &\Rightarrow a = (K+\lambda I)^{-1}y,\end{aligned}$$

where the last step assumes K is invertible (it always is a valid solution regardless, since any a satisfying $(K+\lambda I)a = y$ also solves $K\big[(K+\lambda I)a - y\big] = 0$).

To reiterate, we first redefined the parameters w by an expression that involves a newly introduced variable a, and then found the optimal value for this variable that minimizes the training loss. The resulting objective does not have any trainable parameters left that are not summarized by the Gram matrix K. Furthermore, the optimal value of a requires generation of a Gram matrix using the whole training set, and it appears as an intermediate result for calculation of the prediction function. We predict the label y_* of a query input x_* by

$$\begin{aligned}f(x_*) &= \phi(x_*)^\top w = \phi(x_*)^\top \Phi^\top a\\ &= k_*^\top (K+\lambda I)^{-1} y,\end{aligned}$$

where

$$k_* = \begin{bmatrix} k(x_*, x_1)\\ k(x_*, x_2)\\ \vdots,\\ k(x_*, x_m)\end{bmatrix}.$$

Because the prediction function requires computations on the whole training set, kernel regression is a lazy learner similar to the k-nearest neighbor approach. It is also a **non-parametric** method. As seen in the example implementation given below, its "predict" function takes the whole training set as an input to be able to calculate the k_* vector.

A major weakness of the kernel methods is that many common kernel functions have tunable hyperparameters. For instance, the RBF kernel has a hyperparameter σ that controls the length scale of the kernel — and, as the RKHS reading above showed, thereby decides how expensive high-frequency components are in $\|f\|_{\mathcal{H}_k}$. The performance of the model is sensitive to the choice of this hyperparameter. The example below performs kernel regression with three different choices of σ, each giving dramatically different outcomes. While $\sigma = 0.1$ gives a reasonable fit, $\sigma = 0.01$ **overfits** — the kernel sections are so narrow that the fit spikes at each training point and returns to zero between them — and $\sigma = 1$ **underfits**, since at that length scale the RKHS charges so much for curvature that the fit cannot follow a full period of the sine.

```
import matplotlib.pyplot as plt
import torch as th

th.manual_seed(6)

# Generate the data
inputs = th.linspace(0, 1, 50, dtype=th.float64).unsqueeze(1)
```

```
outputs = th.sin(2 * th.pi * inputs)
labels = outputs + th.randn(inputs.shape, dtype=th.float64) * 0.25

num_samples = inputs.shape[0]
num_train_samples = num_samples // 2

idx = th.randperm(num_samples)
inputs_train = inputs[idx[:num_train_samples]]
labels_train = labels[idx[:num_train_samples]]

class KernelLinearRegression:
    def __init__(self, lambda_reg=0.1, length_scale=0.1):
        self.lambda_reg = lambda_reg
        self.sigma = length_scale

    def compute_rbf_kernel(self, X, Xp):
        # k(x,x') = exp(-||x-x'||^2 / (2 sigma^2))
        dist = th.cdist(X, Xp)
        return th.exp(-dist**2 / (2 * self.sigma**2))

    def precompute_a(self, inputs_train, labels_train):
        # a = (K + lambda I)^{-1} y, solved as the linear system
        # (K + lambda I) a = y
        K = self.compute_rbf_kernel(inputs_train, inputs_train)
        A = K + self.lambda_reg * th.eye(inputs_train.shape[0], dtype=K.dtype)
        self.a = th.linalg.solve(A, labels_train)

    def predict(self, inputs_train, inputs):
        # h(x) = sum_i a_i k(x_i, x)
        Kp = self.compute_rbf_kernel(inputs_train, inputs)
        return Kp.T @ self.a

# Create three models with different kernel hyperparameters
model1 = KernelLinearRegression(length_scale=0.01)
model1.precompute_a(inputs_train, labels_train)

model2 = KernelLinearRegression(length_scale=0.1)
model2.precompute_a(inputs_train, labels_train)

model3 = KernelLinearRegression(length_scale=1)
model3.precompute_a(inputs_train, labels_train)

# This is a "nonparametric model", so
#   i)  There is no "learn" function. "precompute_a" only
#       computes a value that is reused across different predictions
#   ii) Predict function takes the training data as input
```

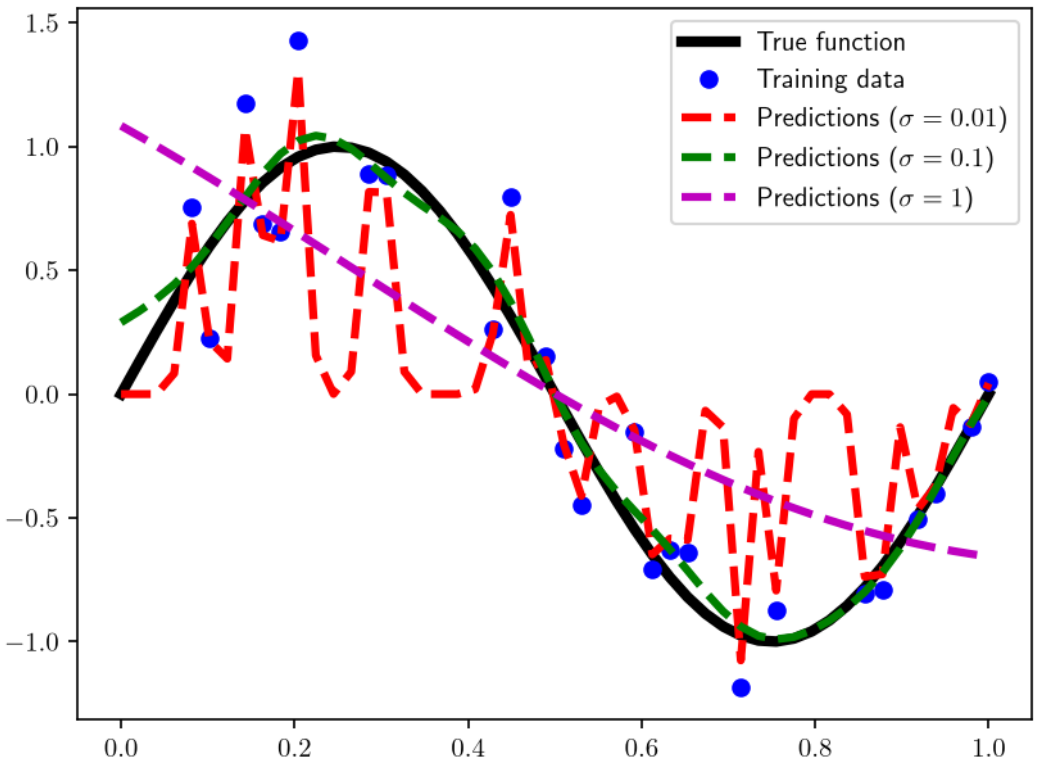


Figure 10.2: Kernel ridge regression fits with an RBF kernel of bandwidth $\sigma \in \{0.01, 0.1, 1\}$, illustrating overfitting, a good fit, and underfitting.

```
predictions1 = model1.predict(inputs_train, inputs)
predictions2 = model2.predict(inputs_train, inputs)
predictions3 = model3.predict(inputs_train, inputs)

plt.plot(inputs, outputs, 'k-',
         label="True function", linewidth=4)
plt.plot(inputs_train, labels_train, 'bo',
         label="Training data")
plt.plot(inputs, predictions1, 'r--',
         label=r"Predictions ($\sigma=0.01$)", linewidth=3)
plt.plot(inputs, predictions2, 'g--',
         label=r"Predictions ($\sigma=0.1$)", linewidth=3)
plt.plot(inputs, predictions3, 'm--',
         label=r"Predictions ($\sigma=1$)", linewidth=3)
plt.legend(loc="upper right")
plt.show()
```

10.4.1 Generalization of Kernel Methods

Kernel methods look, at first glance, like exactly the setting Theorem 5a.4 warned us about: the feature map $\phi(x)$ can live in an arbitrarily high — even infinite — dimensional space (the RBF kernel above is a standard example), for which d_{VC} is typically infinite, making the VC-dimension bounds of Chapter 5 vacuous. But Theorem 5a.4 already told us this does not matter: for a norm-constrained linear predictor $x \mapsto w^\top \phi(x)$ with $\|w\|_2 \le B$, the Rademacher complexity is $BR/\sqrt{m}$, where R bounds $\|\phi(x)\|_2$ — with **no dependence on the dimension of $\phi(x)$ at all**. This is precisely the regime kernel methods create by design, and in the canonical feature map of Definition 9.1 the constrained class is exactly the RKHS ball $\mathcal{H}_B = \{f \in \mathcal{H}_k : \|f\|_{\mathcal{H}_k} \le B\}$ of Proposition 9.2.

We can, in fact, do slightly better and eliminate ϕ from the bound entirely, expressing it purely in terms of the kernel — a small derivation worth doing explicitly, since it shows the kernel trick paying off even inside the generalization theory itself, not only inside the training algorithm. Let $\mathcal{H}_B := \{x \mapsto w^\top \phi(x) : \|w\|_2 \le B\}$. Since $\|\phi(x)\|_2^2 = \phi(x)^\top \phi(x) = k(x,x)$,

$$\widehat{\mathfrak{R}}_S(\mathcal{H}_B) = \frac{B}{m}\,\mathbb{E}_\sigma\Big[\Big\|\sum_{i=1}^m \sigma_i \phi(x_i)\Big\|_2\Big] \le \frac{B}{m}\sqrt{\mathbb{E}_\sigma\Big[\Big\|\sum_i \sigma_i \phi(x_i)\Big\|_2^2\Big]}$$
$$= \frac{B}{m}\sqrt{\sum_{i=1}^m \|\phi(x_i)\|_2^2} = \frac{B}{m}\sqrt{\sum_{i=1}^m k(x_i,x_i)} = \frac{B\sqrt{\mathrm{tr}(K)}}{m},$$

using Jensen's inequality and $\mathbb{E}[\sigma_i \sigma_{i'}] = 0$ for $i \ne i'$ (exactly as in the proof of Theorem 5a.4), where K is the Gram matrix of Mercer's theorem. This **trace bound** is computable directly from the kernel matrix — never requiring ϕ to be written down — and is always at least as tight as the generic $BR/\sqrt{m}$ bound, since $\mathrm{tr}(K) = \sum_i k(x_i,x_i) \le mR^2$ whenever $k(x,x) \le R^2$ uniformly.

This connects directly to the regularization coefficient λ already introduced for ridge regression in Chapter 2 and reused for kernel regression above. At the optimum w^* of the ridge objective $\mathcal{L}(w) = \|y - \Phi w\|_2^2 + \lambda \|w\|_2^2$, optimality against the trivial candidate $w = 0$ gives $\lambda \|w^*\|_2^2 \le \mathcal{L}(w^*) \le \mathcal{L}(0) = \|y\|_2^2$, hence

$$\|w^*\|_2 \le \frac{\|y\|_2}{\sqrt{\lambda}} =: B(\lambda).$$

Plugging $B(\lambda)$ into the trace bound above makes quantitative exactly what Chapter 2 promised only qualitatively when it introduced ridge regression as an instance of Structural Risk Minimization: increasing λ shrinks $B(\lambda)$, which shrinks $\widehat{\mathfrak{R}}_S(\mathcal{H}_{B(\lambda)})$, which — through Theorem 5a.2 — tightens the generalization guarantee. The regularization coefficient is not merely a heuristic device against overfitting; it is a direct dial on the Rademacher complexity of the space the ERM solution is allowed to come from.

10.4.2 The Neural Tangent Kernel

Chapter 8 analyzed a two-layer network $h_{W,w}(x) = w^\top g(Wx)$ and bounded its Rademacher complexity by controlling the norms B_1, B_2 of *both* layers — a bound that holds regardless of

how W and w are jointly trained. There is an illustrative special case of that same network in which training reduces *exactly* to the kernel methods of this chapter, and it is worth making explicit, since it is the seed of one of the more influential ideas in modern deep learning theory: the **Neural Tangent Kernel (NTK)**.

A simplified case: random features. Suppose we freeze the hidden layer at its random initialization W_0 (drawn once, e.g. with i.i.d. $\mathcal{N}(0, 1/d)$ entries per row) and train *only* the output layer w. Then $\phi(x) := g(W_0 x)$ is a fixed feature map, and $h_{W_0,w}(x) = w^\top \phi(x)$ is exactly a kernel method with kernel $k_{W_0}(x, x') := \phi(x)^\top \phi(x') = \sum_{j=1}^{k} g(w_{0,j}^{(1)\top} x)\, g(w_{0,j}^{(1)\top} x')$ — every tool from this chapter, including the generalization bound just derived, applies to it unchanged. As the width $k \to \infty$, the weak law of large numbers (Theorem 4.3, applied to the k i.i.d. terms $g(w_{0,j}^{(1)\top} x)\, g(w_{0,j}^{(1)\top} x')$ as j ranges over hidden units) gives

$$\frac{1}{k} k_{W_0}(x, x') \xrightarrow{P} k_\infty(x, x') := \mathbb{E}_{u \sim \mathcal{N}(0, I/d)} \big[g(u^\top x)\, g(u^\top x') \big],$$

a **deterministic kernel that no longer depends on the random draw of** W_0 at all, only on the architecture (g) and the input distribution of u. This is the "random features" or "infinite-width random network" kernel, and it already illustrates the core NTK phenomenon in the easiest possible case: a wide random network with a trained linear readout *is*, in the limit, exactly kernel regression with a fixed, explicitly computable kernel.

The general case. The full Neural Tangent Kernel result — which trains *all* parameters $\theta = (W, w)$, not just the output layer — is more subtle but built on the same idea. Near the initialization θ_0, a first-order Taylor expansion of the network output in its parameters gives

$$f_\theta(x) \approx f_{\theta_0}(x) + \nabla_\theta f_{\theta_0}(x)^\top (\theta - \theta_0),$$

which is *affine* in θ: exactly the linear-predictor template of this chapter once more, but now with the **tangent feature map** $\phi_{\mathrm{NTK}}(x) := \nabla_\theta f_{\theta_0}(x)$ (one feature per network parameter) and its associated kernel $k_{\mathrm{NTK}}(x, x') := \nabla_\theta f_{\theta_0}(x)^\top \nabla_\theta f_{\theta_0}(x')$. Jacot, Gabriel & Hongler (2018) show two facts that make this linearization exact rather than merely suggestive, in the infinite-width limit and for appropriately scaled initializations: (i) k_{NTK} converges, by the same law-of-large-numbers mechanism as above (now averaging over a growing number of parameters instead of a growing number of hidden units), to a fixed deterministic kernel depending only on the architecture; and (ii) gradient descent moves θ so little from θ_0 relative to the network's width that the linearization above remains accurate for the *entire* training trajectory, not just near initialization. Under these conditions, training an infinitely wide neural network by gradient descent is equivalent to kernel regression (as derived earlier in this chapter) against the fixed kernel k_{NTK}.

Why this matters for generalization. This equivalence resolves, for the infinite-width regime, the very puzzle Chapter 8's PAC analysis raised: an infinitely wide network has infinite VC dimension, yet the NTK correspondence shows it is *simultaneously* a kernel predictor of the exact form Theorem 5a.4 and the trace bound above already know how to handle — capacity controlled by $\|w\|_{\mathcal{H}}$ (the RKHS norm under k_{NTK}) and by $\mathrm{tr}(K_{\mathrm{NTK}})$, neither of which explodes with the width. This is the precise sense in which Chapter 8's remark that "capacity control by norm, rather than by counting parameters, is what makes the bound meaningful" for neural networks is not a coincidence but a theorem: in the wide-network limit, a neural network trained by gradient descent literally *is* a norm-constrained

kernel predictor, and the same Rademacher-complexity machinery developed once, in Chapter 5a, governs both.

10.5 Support Vector Machines

A prime application of kernel methods, and the historical reason the representer theorem is stated in the generality it is, is the **Support Vector Machine (SVM)** (Boser et al., 1992). Among many possible hyperplanes that may separate the data points of two classes, SVM chooses the one that maximizes the smallest distance between a data point and the hyperplane. This distance is called the **margin**. Because of this property, SVM is also called a **maximum margin classifier**. The data points that are closest to the hyperplane are called **support vectors**. The margin-maximizing hyperplane is defined by only these support vectors. In the language of Theorem 9.3, the SVM objective is an instance of the representer template with the **hinge** data-fit term $\Psi(u) = \sum_i \max(0, 1 - y_i u_i)$ and $\mathrm{pen}(t) = \lambda t^2$, so its solution is again $f^* = \sum_i a_i k(x_i, \cdot)$ — but the hinge loss, unlike the squared loss, is flat on correctly classified points beyond the margin, which forces $a_i = 0$ for all of them. The support vectors are exactly the indices with $a_i \neq 0$. The other data points do not play any role in prediction. This is a desirable property especially when the SVM is kernelized, as the kernel function is evaluated only on the support vectors instead of the whole training set as was the case for kernel regression above. The figure below illustrates the core idea behind the SVM. We leave the mathematical details out of our scope, since SVMs are greatly overshadowed by deep learning and Gaussian processes in the modern machine learning practice.

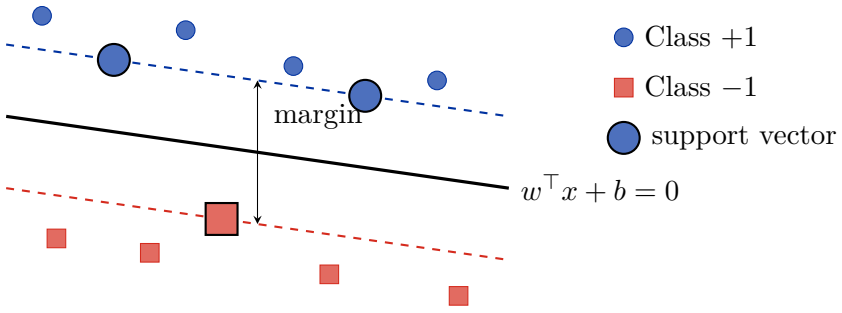


Figure 10.3: The maximum-margin hyperplane of a Support Vector Machine. The decision boundary $w^\top x + b = 0$ is placed to maximize the margin between the two classes; the support vectors (outlined) are the points that lie exactly on the margin boundary and alone determine the solution.

CHAPTER 11

ENSEMBLE METHODS

An **ensemble** aggregates the predictions of several hypotheses $h_1, \dots, h_T$ into a single predictor. The two constructions in this chapter differ in *how* the members are produced and in *which* term of the excess-risk decomposition of Chapter 5 they attack:

	members produced	aggregation	attacks
Bagging	in parallel, on resampled data	unweighted average	the **variance** term
Boosting	in sequence, on reweighted data	weighted vote $\sum_t w_t h_t$	the **approximation error** ϵ_{app}

Bagging takes a low-bias, high-variance base learner and averages its fluctuations away. Boosting takes a high-bias, low-variance base learner — one barely better than a coin flip — and composes many of them into a predictor of much greater capacity. The second is the more surprising of the two, and it is where the theory of this chapter is concentrated.

11.1 Bootstrap Aggregation (Bagging)

Bagging (Breiman, 1996) is the technique this section formalizes.

Definition 11.1.1: Bootstrap sample

Given $S = \{(x_i, y_i)\}_{i=1}^m$, a **bootstrap sample** of S is

$$S' := \{(x_{j_k}, y_{j_k}) : j_k \sim \mathcal{U}(\{1, \dots, m\}),\ k = 1, \dots, m\},$$

i.e. m indices drawn uniformly **with replacement**, so an index may repeat.

Definition 11.1.2: Bagged predictor

Let $S_1, \dots, S_T$ be bootstrap samples of S and $h_{S_t} \leftarrow A(S_t)$. The **bagged predictor** is

$$h_{S_1,\dots,S_T}(x) := \frac{1}{T}\sum_{t=1}^{T} h_{S_t}(x).$$

Two elementary facts explain what averaging buys, both stated for the idealized case of T **independent** samples $S_1, \dots, S_T \sim \mathcal{D}^m$; bootstrapping approximates this by reusing one S, at the cost of correlation between the members, which is why the gains below are real but attenuated in practice.

Proposition 11.1.3

Averaging preserves bias and divides variance. Let $h_{S_1}, \dots, h_{S_T}$ be i.i.d. as functions of independent samples $S_1, \dots, S_T \sim \mathcal{D}^m$. Then for every fixed x,

$$\mathbb{E}\Big[\tfrac{1}{T}\textstyle\sum_t h_{S_t}(x)\Big] = \mathbb{E}_S[h_S(x)],$$
$$\mathrm{Var}\Big[\tfrac{1}{T}\textstyle\sum_t h_{S_t}(x)\Big] = \tfrac{1}{T}\,\mathrm{Var}_S[h_S(x)].$$

Proof. The first claim is linearity of expectation. For the second, independence makes all cross-covariances vanish, so

$$\mathrm{Var}\Big[\tfrac{1}{T}\sum_{t} h_{S_t}(x)\Big] = \tfrac{1}{T^2}\sum_{t=1}^{T}\mathrm{Var}[h_{S_t}(x)] + \tfrac{1}{T^2}\sum_{t\neq t'}\underbrace{\mathrm{Cov}[h_{S_t}(x), h_{S_{t'}}(x)]}_{=0} = \tfrac{1}{T^2}\cdot T\,\mathrm{Var}_S[h_S(x)]. \qquad \square$$

In the vocabulary of Chapter 5's bias-variance decomposition: the Bias^2 term is untouched and the Variance term is divided by T. Letting $T \to \infty$, the weak law of large numbers (Theorem 4.3) gives $h_{S_1,\dots,S_T}(x) \to \bar{h}(x) := \mathbb{E}_S[h_S(x)]$, the **average predictor** of Chapter 5. $\square$

Proposition 11.1.4

Bagging never hurts, in expectation. For the squared loss, $R(\bar{h}) \le \mathbb{E}_S[R(h_S)]$, with equality iff $\mathrm{Var}_S[h_S(x)] = 0$ for $\mathcal{D}$-almost every x.

Proof. Chapter 5 established $\mathbb{E}_S[R(h_S)] - R^*_{\mathcal{D}} = \mathrm{Bias}^2 + \mathrm{Variance}$ and $R(\bar{h}) - R^*_{\mathcal{D}} = \mathrm{Bias}^2 = \|\bar{h} - f^*\|^2$. Subtracting, $\mathbb{E}_S[R(h_S)] - R(\bar{h}) = \mathrm{Variance} = \mathbb{E}_x[\mathrm{Var}_S[h_S(x)]] \ge 0$. Equality holds exactly when the integrand vanishes almost everywhere. (Equivalently: this is Jensen's inequality for the convex map $t \mapsto t^2$.) $\square$

Proposition 10.2 compares $\bar{h}$ only to the *average* member. It can do better. If $\mathcal{H}$ is **not convex**, then $\bar{h} \notin \mathcal{H}$ in general, and $\bar{h}$ may be strictly closer to f^* than *every* $h \in \mathcal{H}$ — the caveat already flagged in Chapter 5. Decision trees (Chapter 7) are such a class: an average of trees is not a tree. Bagged trees, with an extra randomization over the features considered at each split, are **random forests**, and this non-convexity is exactly the source of their strength. Bagging a *convex* class such as linear predictors, by contrast, cannot beat its best member, since there $\bar{h} \in \mathcal{H}$ already.

11.2 Boosting

Boosting starts from the opposite premise: the base class is deliberately **too weak** to fit the data, and the ensemble must manufacture capacity rather than stability. Throughout this section $\mathcal{Y} = \{-1, +1\}$, so that the sign of a real-valued score is a prediction and $y\,f(x) > 0$ means "correct".

11.2.1 Weak Learnability

Definition 11.2.1: γ-weak learner

Let $\gamma \in (0, 1/2)$ and let $\mathcal{H}_0 \subseteq \{-1, +1\}^{\mathcal{X}}$ be a **base class**. A **γ-weak learner** for $\mathcal{H}_0$ is a procedure WL that maps a sample S and a probability vector $D \in \Delta_m := \{D \in \mathbb{R}^m_{\geq 0} : \sum_i D_i = 1\}$ to some $h = \mathrm{WL}(S, D) \in \mathcal{H}_0$ whose **D-weighted empirical error** satisfies

$$\widehat{R}_{S,D}(h) := \sum_{i=1}^{m} D_i \, \mathbb{1}\big(h(x_i) \neq y_i\big) \;\leq\; \tfrac{1}{2} - \gamma.$$

The demand is minimal: a coin flip achieves $1/2$ in expectation, so WL must merely beat guessing by a fixed margin γ — but it must do so for **every** reweighting D of the sample, including adversarial ones that concentrate on the points previous rounds got wrong. This uniformity over D is the whole content of the assumption, and it is what boosting consumes.

Definition 10.3 is the *empirical* form of weak learnability, stated over reweightings of a fixed sample because that is exactly the interface AdaBoost calls. Its distributional counterpart replaces "D on $[m]$" by "any distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$ realizable by $\mathcal{H}_0$" and asks for $R(h) \leq \frac{1}{2} - \gamma$ with high probability from finitely many samples. The two are linked by the Fundamental Theorem (Theorem 5.4): if $d_{VC}(\mathcal{H}_0) < \infty$ then ERM over $\mathcal{H}_0$ is a γ-weak learner in the distributional sense for every $\gamma < 1/2$, given enough samples — so weak learnability is not a weaker *combinatorial* condition than PAC learnability, only a weaker *accuracy* requirement. Corollary 10.1 below shows that even that weakening buys nothing: the two are equivalent.

Definition 11.2.2: Class of T-term linear combinations

For a base class $\mathcal{H}_0$,

$$L(\mathcal{H}_0, T) := \Big\{ x \mapsto \operatorname{sign}\Big(\sum_{t=1}^{T} w_t h_t(x) \Big) \;:\; w \in \mathbb{R}^T,\; h_1, \dots, h_T \in \mathcal{H}_0 \Big\}.$$

$\mathcal{H}_0 \subseteq L(\mathcal{H}_0, 1) \subseteq L(\mathcal{H}_0, 2) \subseteq \cdots$, so the ensemble class is a **nested family** indexed by T — precisely the setting of nonuniform learnability and SRM (Chapter 5), with the number of rounds T playing the role that the leaf budget played for decision trees in Chapter 7.

Example : decision stumps

$\mathcal{H}_{\text{stump}} := \{x \mapsto s\,\operatorname{sign}(x_j - \theta) : j \in [d],\ \theta \in \mathbb{R},\ s \in \{-1, +1\}\}$. On any m points, a stump's labeling is determined by the feature j (d choices), the position of θ among the m observed values of that feature ($m+1$ choices), and the polarity s (2 choices), so

$$\tau_{\mathcal{H}_{\text{stump}}}(m) \le 2d(m+1).$$

Since $\tau(m) = 2^m$ is needed to shatter, $d_{VC}(\mathcal{H}_{\text{stump}}) = O(\log d)$: a stump is a genuinely weak hypothesis, unable even to represent a two-feature XOR. Yet $L(\mathcal{H}_{\text{stump}}, T)$ grows rich with T, which is what makes stumps the canonical base class for boosting. ■

11.2.2 AdaBoost

AdaBoost (Freund and Schapire, 1997) is the algorithm below.

Input: $S = \{(x_i, y_i)\}_{i=1}^m$ with $y_i \in \{-1, +1\}$; a weak learner WL; a number of rounds T.

1. $D^{(1)} := \big(\frac{1}{m}, \dots, \frac{1}{m}\big)$.
2. For $t = 1, \dots, T$:
   1. $h_t := \mathrm{WL}\big(S, D^{(t)}\big)$
   2. $\epsilon_t := \sum_{i=1}^{m} D_i^{(t)}\, \mathbb{1}\big(h_t(x_i) \neq y_i\big)$
   3. $w_t := \frac{1}{2} \log\Big(\frac{1-\epsilon_t}{\epsilon_t}\Big)$
   4. $D_i^{(t+1)} := \dfrac{D_i^{(t)}\, e^{-w_t y_i h_t(x_i)}}{\sum_{j=1}^m D_j^{(t)}\, e^{-w_t y_j h_t(x_j)}}, \qquad i = 1, \dots, m$
3. **Output** $h_S := \operatorname{sign}(f_T) \in L(\mathcal{H}_0, T)$, where $f_T(x) := \sum_{t=1}^{T} w_t\, h_t(x)$.

Step 2.4 is one line because $y_i h_t(x_i) = +1$ on correct points and -1 on incorrect ones: the update multiplies a correct point's weight by e^{-w_t} and an incorrect point's by e^{+w_t}, then renormalizes. The choice of w_t in step 2.3 is not a guess; it is a minimization, as the proof below shows in one line.

Theorem 11.2.3: AdaBoost drives the training error down exponentially

If $\epsilon_t \le \frac{1}{2} - \gamma$ at every round, then the output of AdaBoost satisfies

$$\widehat{R}_S(h_S) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}\big(h_S(x_i) \neq y_i\big) \;\le\; e^{-2\gamma^2 T}.$$

Proof. Write $f_t := \sum_{p \le t} w_p h_p$, $f_0 := 0$, and define the **exponential loss**

$$Z_t := \frac{1}{m}\sum_{i=1}^{m} e^{-y_i f_t(x_i)}.$$

Step 1: the exponential loss dominates the zero-one loss. Since $\mathbb{1}(u \le 0) \le e^{-u}$ for all $u \in \mathbb{R}$, and $h_S(x_i) \neq y_i$ exactly when $y_i f_T(x_i) \le 0$,

$$\widehat{R}_S(h_S) \le Z_T.$$

Step 2: the weights are the normalized exponential losses. We claim

$$D_i^{(t)} = e^{-y_i f_{t-1}(x_i)} \Big/ \sum_j e^{-y_j f_{t-1}(x_j)}.$$

For $t = 1$ both sides are $1/m$ since $f_0 = 0$. Assuming it at t, step 2.4 gives

$$D_i^{(t+1)} \;\propto\; D_i^{(t)} e^{-w_t y_i h_t(x_i)} \;\propto\; e^{-y_i f_{t-1}(x_i)} e^{-w_t y_i h_t(x_i)} = e^{-y_i f_t(x_i)},$$

and normalizing proves the claim at $t+1$.

Step 3: the per-round progress factor. Using Step 2, and splitting the sum according to whether h_t is correct,

$$\frac{Z_t}{Z_{t-1}} = \frac{\sum_i e^{-y_i f_{t-1}(x_i)}\, e^{-w_t y_i h_t(x_i)}}{\sum_i e^{-y_i f_{t-1}(x_i)}} = \sum_{i=1}^{m} D_i^{(t)} e^{-w_t y_i h_t(x_i)} = \underbrace{(1-\epsilon_t)\, e^{-w_t} + \epsilon_t\, e^{w_t}}_{=:\ \psi(w_t)}.$$

Now $\psi'(w) = -(1-\epsilon_t)e^{-w} + \epsilon_t e^{w} = 0$ gives $e^{2w} = (1-\epsilon_t)/\epsilon_t$, i.e. **exactly the w_t of step 2.3** — this is where that formula comes from, and ψ is convex, so it is a minimum. Substituting $e^{w_t} = \sqrt{(1-\epsilon_t)/\epsilon_t}$,

$$\frac{Z_t}{Z_{t-1}} = (1-\epsilon_t)\sqrt{\tfrac{\epsilon_t}{1-\epsilon_t}} + \epsilon_t\sqrt{\tfrac{1-\epsilon_t}{\epsilon_t}} = 2\sqrt{\epsilon_t(1-\epsilon_t)}.$$

Step 4: telescoping. Writing $\epsilon_t = \frac{1}{2} - \gamma_t$ with $\gamma_t \ge \gamma$, we get $2\sqrt{\epsilon_t(1-\epsilon_t)} = \sqrt{1-4\gamma_t^2} \le e^{-2\gamma_t^2} \le e^{-2\gamma^2}$, using $1 - u \le e^{-u}$ with $u = 4\gamma_t^2$ under the square root. Since $Z_0 = 1$,

$$\widehat{R}_S(h_S) \le Z_T = \prod_{t=1}^{T} \frac{Z_t}{Z_{t-1}} \le e^{-2\gamma^2 T}. \qquad \square$$

Two readings of the proof are worth keeping. First, AdaBoost is **coordinate-wise greedy minimization of the exponential loss** Z_t: step 2.1 picks the direction h_t and step 2.3 picks the optimal step size w_t along it. Second, the weighting scheme is not an extra heuristic layered on top — Step 2 shows $D^{(t)}$ *is* the exponential loss, normalized, so up-weighting the currently misclassified points is exactly what greedy minimization of Z_t prescribes. □

Corollary 11.2.4

Weak learnability implies strong learnability. Let $\mathcal{H}_0$ admit a γ-weak learner and run AdaBoost for $T \geq \frac{\log m}{2\gamma^2}$ rounds. Then $\widehat{R}_S(h_S) = 0$.

Proof. By Theorem 10.1, $\widehat{R}_S(h_S) \leq e^{-2\gamma^2 T} \leq e^{-\log m} = 1/m$. But $\widehat{R}_S(h_S)$ is a multiple of $1/m$, and the only such value strictly below $1/m$ is 0. □

This is the theorem that made boosting famous: a learner that can only beat a coin flip by γ, on every reweighting of the sample, can be compiled into one that fits the sample perfectly, in $O(\gamma^{-2} \log m)$ rounds. **Weak** and **strong** learnability are the same notion, up to computation. What remains is to price the generalization of the resulting predictor — and this is where two different complexity measures give two genuinely different answers.

11.2.3 What Linear Combinations Cost: Two Accounts

Account 1: the VC dimension of $L(\mathcal{H}_0, T)$ grows with T

Lemma 11.2.5

Let $d_0 := d_{VC}(\mathcal{H}_0) < \infty$ (not to be confused with the feature dimension d). Then

$$\tau_{L(\mathcal{H}_0,T)}(m) \;\leq\; \tau_{\mathcal{H}_0}(m)^T \cdot \left(\frac{em}{T}\right)^T \;\leq\; \left(\frac{em}{d_0}\right)^{d_0 T} \left(\frac{em}{T}\right)^T,$$

$$\text{hence} \qquad d_{VC}\big(L(\mathcal{H}_0, T)\big) = O\big(d_0 T \log(d_0 T)\big).$$

Proof. Fix $S = \{x_1, \ldots, x_m\}$. A hypothesis of $L(\mathcal{H}_0, T)$ is specified by base hypotheses $h_1, \ldots, h_T$ and weights w. Its restriction to S depends on the h_t only through their own restrictions to S, and there are at most $\tau_{\mathcal{H}_0}(m)$ of those per slot, hence at most $\tau_{\mathcal{H}_0}(m)^T$ jointly. Having fixed them, map each point to $v_i := (h_1(x_i), \ldots, h_T(x_i)) \in \{-1, +1\}^T$; the remaining freedom is the labeling $\operatorname{sign}(w^\top v_i)$, i.e. a homogeneous halfspace in $\mathbb{R}^T$ applied to m fixed points. The class of homogeneous halfspaces in $\mathbb{R}^T$ has VC dimension T, so by Sauer-Shelah-Perles (Lemma 5.2) it realizes at most $(em/T)^T$ labelings. Multiplying the two counts gives the first claim, and the second follows from Lemma 5.2 applied to $\mathcal{H}_0$. For the VC bound, shattering m points requires $2^m \leq \tau_{L(\mathcal{H}_0,T)}(m)$, i.e. $m \log 2 \leq (d_0 T + T) \log(em)$, which forces $m = O(d_0 T \log(d_0 T))$. □

Feeding $d_{VC}(L(\mathcal{H}_0, T)) = O(d_0 T \log(d_0 T))$ into the Fundamental Theorem (Theorem 5.4) gives, with probability at least $1 - \delta$, for every $h \in L(\mathcal{H}_0, T)$,

$$R(h) \;\le\; \widehat{R}_S(h) + O\left(\sqrt{\frac{d_0 T \log(d_0 T) + \log(1/\delta)}{m}}\right). \tag{10.1}$$

Combined with Corollary 10.1 this already proves boosting works: choose $T \asymp \gamma^{-2} \log m$, get $\widehat{R}_S = 0$, and read off a nonvacuous bound as soon as $m \gtrsim d_0 \gamma^{-2} \log^2 m$. It also makes a **prediction**: the bound degrades as $\sqrt{T}$, so running AdaBoost past the point where it fits the sample should start to overfit. Empirically, this prediction fails — AdaBoost's test error typically keeps *improving* for hundreds of rounds after the training error hits zero. The bound is not wrong; it is measuring the wrong thing.

Account 2: the Rademacher complexity of the convex hull does not grow at all

Assume $\mathcal{H}_0$ is **symmetric** ($h \in \mathcal{H}_0 \Rightarrow -h \in \mathcal{H}_0$; true for stumps, by flipping s). Then we may take every $w_t \ge 0$ without loss of generality, and the **normalized ensemble**

$$\bar{f}_T(x) := \frac{f_T(x)}{\sum_{t=1}^{T} w_t} = \sum_{t=1}^{T} \lambda_t h_t(x),$$
$$\lambda_t := \frac{w_t}{\sum_p w_p} \ge 0, \quad \sum_t \lambda_t = 1,$$

is a **convex combination** of members of $\mathcal{H}_0$, i.e. $\bar{f}_T \in \operatorname{conv}(\mathcal{H}_0)$, with $\bar{f}_T(x) \in [-1, 1]$. Note $\operatorname{sign}(\bar{f}_T) = \operatorname{sign}(f_T) = h_S$: normalizing changes no prediction, only the scale on which we may measure *how confidently* each prediction was made.

Theorem 11.2.6: Convexification is free

For any $A \subset \mathbb{R}^m$, $\widehat{\mathfrak{R}}(\operatorname{conv}(A)) = \widehat{\mathfrak{R}}(A)$. In particular $\widehat{\mathfrak{R}}_S(\operatorname{conv}(\mathcal{H}_0)) = \widehat{\mathfrak{R}}_S(\mathcal{H}_0)$ for every S, **independently of** T.

Proof. $A \subseteq \operatorname{conv}(A)$ gives "$\ge$". For "$\le$", fix the signs $\sigma \in \{-1, +1\}^m$ and let $a = \sum_j \lambda_j a^{(j)} \in \operatorname{conv}(A)$ with $\lambda_j \ge 0$, $\sum_j \lambda_j = 1$. By linearity of the inner product,

$$\langle \sigma, a \rangle = \sum_j \lambda_j \langle \sigma, a^{(j)} \rangle \le \left(\sum_j \lambda_j\right) \max_j \langle \sigma, a^{(j)} \rangle \le \sup_{a' \in A} \langle \sigma, a' \rangle.$$

Taking the supremum over $a \in \operatorname{conv}(A)$ and then $\mathbb{E}_\sigma$ of both sides gives the claim. □

Theorem 11.2.7: Margin bound for boosting

Let $\mathcal{H}_0$ be symmetric with values in $\{-1,+1\}$, fix a **margin threshold** $\nu \in (0,1]$ — a free parameter of the bound, unrelated to the weak-learning parameter γ of Definition 10.3 — and let $\bar{f} \in \operatorname{conv}(\mathcal{H}_0)$ be arbitrary. Then with probability at least $1-\delta$ over $S \sim \mathcal{D}^m$, simultaneously for every such $\bar{f}$,

$$R\big(\operatorname{sign}(\bar{f})\big) \;=\; P_{(x,y)\sim\mathcal{D}}\big(y\,\bar{f}(x) \le 0\big) \;\le\; \underbrace{\frac{1}{m}\sum_{i=1}^{m} \mathbb{1}\big(y_i \bar{f}(x_i) \le \nu\big)}_{\text{fraction of training margins} \le \nu} + \frac{2}{\nu}\,\widehat{\mathfrak{R}}_S(\mathcal{H}_0) \;+\; 3\sqrt{\frac{\log(2/\delta)}{2m}}.$$

Proof. Call $y\bar{f}(x) \in [-1,1]$ the **margin** of the prediction — positive exactly when the weighted vote is correct, and large when it is correct by a wide majority. Define the **ν-margin loss**

$$\phi_\nu(u) := \min\big(1, \max(0, 1 - u/\nu)\big),$$

which is $\frac{1}{\nu}$-Lipschitz, takes values in $[0,1]$, and satisfies the sandwich $\mathbb{1}(u \le 0) \le \phi_\nu(u) \le \mathbb{1}(u \le \nu)$.

Apply the Rademacher generalization bound (Theorem 5a.2) to the loss class $\big\{(x,y) \mapsto \phi_\nu\big(y\bar{f}(x)\big) : \bar{f} \in \operatorname{conv}(\mathcal{H}_0)\big\}$, whose values lie in $[0,1]$: with probability $\ge 1-\delta$, for all $\bar{f}$,

$$\mathbb{E}_{\mathcal{D}}\big[\phi_\nu(y\bar{f}(x))\big] \le \frac{1}{m}\sum_i \phi_\nu\big(y_i\bar{f}(x_i)\big) + 2\,\widehat{\mathfrak{R}}_S\big(\phi_\nu \circ \operatorname{conv}(\mathcal{H}_0)\big) + 3\sqrt{\tfrac{\log(2/\delta)}{2m}}.$$

The left side is $\ge P_{\mathcal{D}}\big(y\bar{f}(x) \le 0\big)$ and the empirical average is $\le \frac{1}{m}\sum_i \mathbb{1}(y_i\bar{f}(x_i) \le \nu)$, both by the sandwich. For the middle term, note that the contraction lemma (Lemma 5a.2) permits a *different* function in each coordinate: take $\phi_i(u) := \phi_\nu(y_i u)$, which is $\frac{1}{\nu}$-Lipschitz for either sign of y_i, so contracting with these removes both the label multiplication and the margin loss in one step. Theorem 10.2 then removes the convex hull:

$$\widehat{\mathfrak{R}}_S\big(\phi_\nu \circ \operatorname{conv}(\mathcal{H}_0)\big) \;\le\; \tfrac{1}{\nu}\,\widehat{\mathfrak{R}}_S\big(\operatorname{conv}(\mathcal{H}_0)\big) \;=\; \tfrac{1}{\nu}\,\widehat{\mathfrak{R}}_S(\mathcal{H}_0). \qquad \square$$

$\square$

The resolution. Compare (10.1) with Theorem 10.3, term by term:

	Account 1 (VC of $L(\mathcal{H}_0, T)$)	Account 2 (Rademacher of $\operatorname{conv}(\mathcal{H}_0)$)
fit term	$\widehat{R}_S(h_S)$, hits 0 and then stops moving	$\frac{1}{m}\sum_i \mathbb{1}(y_i \bar{f}_T(x_i) \le \nu)$, **keeps shrinking**

	Account 1 (VC of $L(\mathcal{H}_0, T)$)	Account 2 (Rademacher of $\mathrm{conv}(\mathcal{H}_0)$)
complexity term	$O(\sqrt{d_0 T \log(d_0 T)/m})$, **grows with** T	$\frac{2}{\nu}\widehat{\mathfrak{R}}_S(\mathcal{H}_0)$, **independent of** T

Account 2 says that additional boosting rounds cost nothing in complexity — a convex combination of T members is no richer, in the Rademacher sense, than a single member — while they continue to buy something the zero-one training error is too coarse to see: Theorem 10.1's proof shows AdaBoost keeps driving the exponential loss Z_t down after $\widehat{R}_S = 0$, and Z_t is small only when the margins $y_i f_t(x_i)$ are *large*, not merely positive. Boosting past zero training error is therefore not idle: it is converting correct-but-marginal predictions into correct-and-confident ones, which is precisely what the first term of Theorem 10.3 rewards. This is the same lesson as Chapter 8's: the complexity measure that grows with the number of parameters is the wrong one, and a scale-sensitive measure — margins here, weight norms there — is the right one.

The trade-off has not disappeared, it has moved into ν: a larger ν shrinks the complexity term but makes the margin-violation term larger. The bound holds for all ν simultaneously (up to a union bound over a grid of ν's), so one may read it at whichever ν is most favorable for the ensemble actually obtained.

11.2.4 Implementation: AdaBoost with Decision Stumps

We implement Definition 10.3 and the AdaBoost pseudocode literally: the weak learner is an exhaustive search over $\mathcal{H}_{\text{stump}}$ for the stump of least D-weighted error, and the boosting loop is the four numbered steps. The data set is the **Wisconsin Breast Cancer** set of Chapters 6 and 7 (569 samples, $d = 30$ features), with labels recoded to $\{-1, +1\}$. Alongside the two error curves we track $\min_i y_i \bar{f}_T(x_i)$, the smallest training margin — the quantity Theorem 10.3 says should keep improving after the training error stops.

```
# AdaBoost implemented from scratch: the base class is decision stumps, the
# weak learner is an exhaustive weighted-error search over them, and the
# boosting loop is line-for-line the pseudocode above.

import torch as th
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer

th.manual_seed(0)

data = load_breast_cancer()
X_all = th.tensor(data.data, dtype=th.float64)
y_all = 2 * th.tensor(data.target, dtype=th.float64) - 1   # labels in {-1,+1}
idx = th.randperm(y_all.numel())
n_tr = int(0.7 * y_all.numel())
tr, te = idx[:n_tr], idx[n_tr:]
```

```
X, y = X_all[tr], y_all[tr]
X_te, y_te = X_all[te], y_all[te]
m, d = X.shape
order = th.argsort(X, dim=0)                              # sort once, reuse

# --- The base class: decision stumps  h(x) = s * sign(x_j - theta) ----------

def weak_learner(D):
    """Return the stump of least D-weighted empirical error, by exhaustive
    search over all d features and all m+1 distinct thresholds of each."""
    best = (float('inf'), 0, -float('inf'), 1)            # (err, j, theta, s)
    for j in range(d):
        o = order[:, j]
        xs, ys, Ds = X[o, j], y[o], D[o]
        # Sweep theta from left to right. At theta = -inf the stump predicts
        # +1 everywhere, so its error is the weight of the negative points;
        # each time theta passes a point, that point's prediction flips,
        # changing the error by +D_i if y_i = +1 and by -D_i if y_i = -1.
        errs = Ds[ys == -1].sum() + th.cat([th.zeros(1, dtype=D.dtype),
                                            th.cumsum(Ds * ys, dim=0)])
        thetas = th.cat([th.tensor([-float('inf')], dtype=xs.dtype), xs])
        for s in (1, -1):                                 # both leaf polarities
            e = errs if s == 1 else 1.0 - errs
            k = int(e.argmin())
            if e[k] < best[0]:
                best = (e[k].item(), j, thetas[k].item(), s)
    return best[1:]                                       # (j, theta, s)

def stump_predict(Xq, stump):
    j, theta, s = stump
    return s * th.where(Xq[:, j] > theta, 1.0, -1.0).double()

# --- AdaBoost ---------------------------------------------------------------

T = 200
D = th.full((m,), 1.0 / m, dtype=th.float64)              # step 1: D^(1)
w_sum = 0.0
F, F_te = th.zeros(m, dtype=th.float64), th.zeros(y_te.numel(), dtype=th.float64)
err_tr, err_te, min_margin = [], [], []

for t in range(T):
    stump = weak_learner(D)                               # 2.1  h_t = WL(S,D^(t))
    pred = stump_predict(X, stump)
    eps = D[pred != y].sum()                              # 2.2  epsilon_t
    eps = eps.clamp(1e-10, 1 - 1e-10)                     #      guard log(0)
    w_t = 0.5 * th.log(1.0 / eps - 1.0)                   # 2.3  w_t
    D = D * th.exp(-w_t * y * pred)                       # 2.4  reweight ...
```

```
    D = D / D.sum()                                  #      ... and normalise

    w_sum += w_t.item()                              # bookkeeping only:
    F += w_t * pred                                  #   f_t on the train set
    F_te += w_t * stump_predict(X_te, stump)         #   f_t on the test set
    err_tr.append((th.sign(F) != y).double().mean().item())
    err_te.append((th.sign(F_te) != y_te).double().mean().item())
    min_margin.append((y * F / w_sum).min().item())    # min_i y_i bar-f_t(x_i)

t0 = next(t for t, e in enumerate(err_tr) if e == 0.0)
print("       T   train err   test err   min margin")
for t in [0, 4, 9, t0, 49, 99, T - 1]:
    print(f"{t+1:8d}    {err_tr[t]:.4f}     {err_te[t]:.4f}
    ↪  {min_margin[t]:+.4f}")
print(f"\ntraining error first reaches 0 at T = {t0+1}")

rounds = th.arange(1, T + 1)
fig, ax = plt.subplots(1, 2, figsize=(11, 4))
ax[0].plot(rounds, err_tr, label=r"training error $\widehat{R}_S(h_S)$")
ax[0].plot(rounds, err_te, label=r"test error (estimate of $R(h_S)$)")
ax[0].axvline(t0 + 1, ls=":", c="k")
ax[0].set_xscale("log"); ax[0].set_xlabel("boosting rounds $T$")
ax[0].set_ylabel("zero-one error"); ax[0].legend(); ax[0].grid(alpha=.3)
ax[1].plot(rounds, min_margin, c="C2")
ax[1].axvline(t0 + 1, ls=":", c="k"); ax[1].axhline(0, lw=.5, c="k")
ax[1].set_xscale("log"); ax[1].set_xlabel("boosting rounds $T$")
ax[1].set_ylabel(r"minimum training margin")
ax[1].grid(alpha=.3)
plt.tight_layout(); plt.show()
```

```
       T   train err   test err   min margin
       1    0.0804     0.0819     -1.0000
       5    0.0251     0.0585     -0.6193
      10    0.0201     0.0585     -0.1861
      27    0.0000     0.0292     +0.0253
      50    0.0000     0.0351     +0.0599
     100    0.0000     0.0292     +0.1175
     200    0.0000     0.0234     +0.1268

training error first reaches 0 at T = 27
```

The numbers track the theory. Across the 200 rounds the worst weighted error the stump search ever returns is $\epsilon_t = 0.436$, so the weak-learning parameter realized on this data set is $\gamma = 0.065$. Theorem 10.1 then guarantees only $\widehat{R}_S \le e^{-2(0.065)^2 \cdot 27} \approx 0.80$ after 27 rounds, and Corollary 10.1's sufficient budget is $\log(398)/(2 \cdot 0.065^2) \approx 720$ rounds — both valid but, as worst-case bounds usually are, far more conservative than what actually happens: the training error is already exactly 0 at $T = 27$. From $T = 27$ onward, the first term of Account

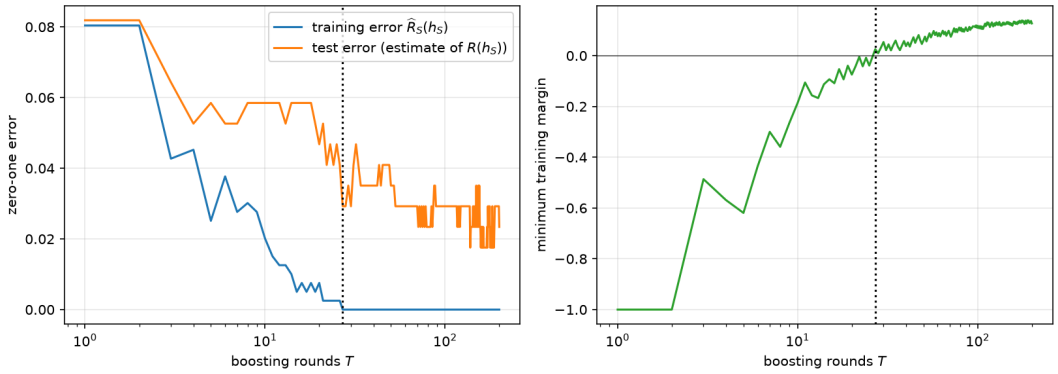


Figure 11.1: AdaBoost's training and test error (left) and minimum training margin (right) as a function of the number of boosting rounds T, on a logarithmic axis; the dotted line marks the round at which training error first reaches zero.

1 is frozen at 0 and its complexity term keeps growing, so (10.1) predicts deterioration; instead the test error *falls further*, from 2.9% to 2.3%. Account 2 explains why: over the same rounds the smallest training margin climbs from $+0.025$ to $+0.127$, so the margin-violation term of Theorem 10.3 keeps shrinking at any fixed threshold $\nu \lesssim 0.12$, while its complexity term $\frac{2}{\nu}\widehat{\mathfrak{R}}_S(\mathcal{H}_{\text{stump}})$ never moved.

REFERENCES

Aronszajn, Nachman (1950). Theory of Reproducing Kernels. *Transactions of the American Mathematical Society, 68(3), 337–404.*

Bartlett, Peter L. and Mendelson, Shahar (2002). Rademacher and Gaussian Complexities: Risk Bounds and Structural Results. *Journal of Machine Learning Research, 3, 463–482.*

Bayes, Thomas and Price, Richard (1763). An Essay towards Solving a Problem in the Doctrine of Chances. *Philosophical Transactions of the Royal Society of London, 53, 370–418.*

Boser, Bernhard E., Guyon, Isabelle M. and Vapnik, Vladimir N. (1992). A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory (COLT)*, pp. 144–152.

Breiman, Leo (1996). Bagging Predictors. *Machine Learning, 24(2), 123–140.*

Breiman, Leo, Friedman, Jerome H., Olshen, Richard A. and Stone, Charles J. (1984). *Classification and Regression Trees.* Wadsworth.

Cox, D. R. (1958). The Regression Analysis of Binary Sequences. *Journal of the Royal Statistical Society: Series B, 20(2), 215–242.*

Fix, Evelyn and Hodges, Joseph L. (1951). Discriminatory Analysis, Nonparametric Discrimination: Consistency Properties. Technical report, USAF School of Aviation Medicine, Randolph Field, Texas.

Freund, Yoav and Schapire, Robert E. (1997). A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences, 55(1), 119–139.*

Hoeffding, Wassily (1963). Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association, 58(301), 13–30.*

Hoerl, Arthur E. and Kennard, Robert W. (1970). Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics, 12(1), 55–67.*

Kimeldorf, George S. and Wahba, Grace (1971). Some Results on Tchebycheffian Spline Functions. *Journal of Mathematical Analysis and Applications, 33(1), 82–95.*

McAllester, David A. (1999). PAC-Bayesian Model Averaging. In *Proceedings of the 12th Annual Conference on Computational Learning Theory (COLT)*, pp. 164–170.

McDiarmid, Colin (1989). On the Method of Bounded Differences. *Surveys in Combinatorics, 141, 148–188.*

Mercer, James (1909). Functions of Positive and Negative Type, and Their Connection with the Theory of Integral Equations. *Philosophical Transactions of the Royal Society A, 209, 415–446.*

Mitchell, Tom M. (1997). *Machine Learning.* McGraw-Hill.

Robbins, Herbert and Monro, Sutton (1951). A Stochastic Approximation Method. *The Annals of Mathematical Statistics, 22(3), 400–407.*

Rosenblatt, Frank (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review, 65(6), 386–408.*

Rumelhart, David E., Hinton, Geoffrey E. and Williams, Ronald J. (1986). Learning Representations by Back-Propagating Errors. *Nature, 323, 533–536.*

Sauer, N. (1972). On the Density of Families of Sets. *Journal of Combinatorial Theory, Series A, 13(1), 145–147.*

Shelah, Saharon (1972). A Combinatorial Problem; Stability and Order for Models and Theories in Infinitary Languages. *Pacific Journal of Mathematics, 41(1), 247–261.*

Tibshirani, Robert (1996). Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B, 58(1), 267–288.*

Valiant, Leslie G. (1984). A Theory of the Learnable. *Communications of the ACM, 27(11), 1134–1142.*

Vapnik, V. N. and Chervonenkis, A. Ya. (1971). On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities. *Theory of Probability and its Applications, 16(2), 264–280.*

Wolpert, David H. (1996). The Lack of A Priori Distinctions Between Learning Algorithms. *Neural Computation, 8(7), 1341–1390.*