KTH Electrical Engineering

Pattern Recognition

Fundamental Theory and Exercise Problems

ARNE LEIJON & GUSTAV EJE HENTER

Stockholm 2015

KTH – School of Electrical Engineering

Lecture Notes Course EQ2340

Contents

What is Pattern Recognition? **1**

1 Classification and Probability **3**
1.1 Example: First Binary Classifier Design 3
1.2 Gaussian Feature Vector 5
1.2.1 Two Independent Features 6
1.2.2 General Correlated Features 8
1.2.3 Coordinate Transformations and Eigenvectors 9
1.2.4 Random Vector Subspace 10
1.3 Gaussian Mixture Model 11
Summary 15
Problems 16

2 Conditional Probability and Bayes Rule **21**
Problems 24

3 Bayesian Pattern Classification **27**
3.1 Problem Definition 29
3.1.1 A Priori Source Category Distribution 29
3.1.2 Feature Vector Distribution 30
3.1.3 Decision Function 30
3.1.4 A Posteriori Source category Distribution 30
3.2 Bayes Minimum-Risk Decision Rule 31
3.2.1 Special case: Minimum Error Rate 33
3.2.2 Special case: Equal A-Priori Probabilities 34
3.3 Discriminant Functions 34
3.4 Decision Regions and Error Probability 35
3.5 Simple Example: Scalar Gaussian Feature 37
3.5.1 Decision Criterion 37
3.5.2 Discriminant Functions 37
3.5.3 Simplify Discriminant Functions 38
3.5.4 Decision Function 38
3.5.5 Classifier Performance 39

3.6 Example: Independent Gaussian Features 40
3.6.1 Decision Criterion 40
3.6.2 Discriminant Functions 41
3.6.3 Simplify Discriminant Functions 41
3.6.4 Decision Function 42
3.6.5 Classifier Performance 42
3.7 General Gaussian Feature Vector 43
3.7.1 Special case: Equal diagonal covariance matrices . . . 44
3.7.2 Special case: Equal covariance matrices 44
3.8 Discrete Feature Distribution 45
Summary 47
Problems 48

4 Classification in Practical Applications 57
4.1 Data Collection 58
4.2 Feature Extraction 59
4.3 Statistical Modelling 61
4.3.1 Parametric Models 62
4.3.2 The Importance of Simple Models 64
4.3.3 Some Common Feature Distributions 64
4.4 Parameter Estimation 65
4.4.1 Maximum Likelihood Parameter Estimation 66
4.4.2 Other Parameter Estimation Methods 70
4.5 Evaluating Classifier Performance 71
4.5.1 Cross-Validation 71
4.5.2 Improving the Classifier 72
4.5.3 Ensemble Methods 72
4.6 Feature Extraction Problems 73
4.6.1 The Curse of Dimensionality 73
4.7 Model Problems 74
4.7.1 Discriminative Training 75
4.7.2 Minimum Classification Error Training 75
4.7.3 Maximum Mutual Information Training 76
4.8 Training Data Problems 77
4.8.1 Scarce Data and Overfitting 77
4.8.2 Large Datasets 81
4.8.3 Mismatched Data 81
4.8.4 Novel Events 83
4.8.5 Errors in the Data 84
Summary 86
Problems 87

5 Hidden Markov Models for Sequence Classification 89
5.1 Hidden Markov Model – Definition . . . 91
5.1.1 Continuous-valued observations . . . 94
5.1.2 Discrete-valued (quantized) observations . . . 94
5.2 HMM Representations . . . 96
5.3 Hidden Markov Model Structures . . . 97
5.3.1 Infinite Duration . . . 98
5.3.2 Finite Duration . . . 98
5.3.3 Left-right HMM . . . 99
5.3.4 Stationary HMM . . . 100
5.3.5 Ergodic HMM . . . 101
5.4 Probability of an Observed Sequence – the Forward Algorithm 104
5.4.1 Problem Introduction . . . 104
5.4.2 Forward Variables . . . 107
5.4.3 Forward Calculation Procedure . . . 108
5.5 The Backward Algorithm . . . 110
5.5.1 Problem Introduction . . . 110
5.5.2 Backward Variables . . . 112
5.5.3 Backward Calculation Procedure . . . 113
5.6 Most Probable State Sequence – the Viterbi Algorithm . . . 115
5.6.1 Problem Formulation . . . 115
5.6.2 Viterbi Calculation Variables . . . 117
5.6.3 Viterbi Calculation Procedure . . . 118
Summary . . . 120
Problems . . . 121

6 Hidden Markov Model Training 127
6.1 The Baum-Welch Algorithm . . . 127
6.1.1 Updating the Initial Probability Vector . . . 129
6.1.2 Updating the Transition Probability Matrix . . . 130
6.1.3 Updating the Observation Probability Matrix . . . 132
6.2 HMM Training: Practical aspects . . . 133
6.2.1 Feature Extraction . . . 133
6.2.2 Model Size . . . 134
6.2.3 Initializing a Left-Right HMM . . . 134
6.2.4 Initializing an Ergodic HMM . . . 135
6.2.5 Termination of HMM Training . . . 136
6.2.6 Adjustment after Training . . . 136
Summary . . . 137
Problems . . . 138

7 Expectation Maximization **139**
7.1 Intuitive Introduction . . . 140
7.2 Algorithm Steps . . . 141
7.3 EM Algorithm Proof . . . 142
7.4 EM Examples with Solutions . . . 143
7.5 Training a Gaussian Mixture Model . . . 149
7.5.1 Updating the GMM Weight Factors . . . 151
7.5.2 Updating the Gaussian Parameters . . . 152
7.6 HMM Training . . . 153
7.6.1 Updating the Initial Probability Vector . . . 155
7.6.2 Updating the Transition Probability Matrix . . . 156
7.6.3 Updating the Output Probability Distributions . . . 157
7.6.4 Updating a Discrete Output Probability Distribution . 158
7.6.5 Updating GMM Output Distributions . . . 159
Summary . . . 161
Problems . . . 162

8 Bayesian Learning **167**
8.1 Introduction . . . 167
8.2 Bayesian Learning Procedure . . . 171
8.3 The Prior Distribution . . . 173
8.3.1 Subjective Informative Prior . . . 173
8.3.2 Subjective Non-Informative Prior . . . 174
8.3.3 Objective Non-Informative Prior . . . 176
8.4 Useful Conjugate Probability Distributions . . . 181
8.4.1 Beta Distribution . . . 181
8.4.2 Dirichlet Distribution . . . 182
8.4.3 Gamma Distribution . . . 182
Summary . . . 184
Problems . . . 185

9 Approximate Bayesian Learning **191**
9.1 Variational Inference – Notation . . . 192
9.2 Variational Inference – General Solution . . . 193
9.2.1 Kullback-Leibler Divergence . . . 194
9.3 Factorized Approximation . . . 195
9.4 VI Example with Solution . . . 197
9.5 EM – Special Case of Variational Inference . . . 200
Summary . . . 204
Problems . . . 205

A Exercise Project: Pattern Recognition System **209**
A.1 HMM Signal Source 210
A.1.1 HMM Random Source 211
A.1.2 Verify the MarkovChain and HMM Sources 212
A.2 Feature Extraction 215
A.2.1 Feature Extraction in General 215
A.2.2 Speech Recognition 216
A.2.3 Song Recognition 223
A.2.4 Character Recognition 232
A.3 Algorithm Implementation 237
A.3.1 The Forward Algorithm 237
A.3.2 The Backward Algorithm 239
A.4 Code Verification and Signal Database 242
A.4.1 Code Verification 242
A.4.2 Speech Database 243
A.4.3 Song Database 244
A.4.4 Character Database 245
A.5 System Demonstration 247

Bibliography **253**

Answers to some problems **255**

Index **273**

What is Pattern Recognition?

Pattern recognition is fundamental in many engineering applications, for example in decoding the information carried by radio signals corrupted by noise, automatic speech recognition, decoding of hand-written text, or biomedical signal analysis. The human brain has an outstanding capacity for pattern recognition, and pattern-recognition theory must therefore be used in any attempt to understand how the human brain decodes the very complex sensory data received from the eyes, ears, etc.

The following figure illustrates schematically the components of a general pattern recognition system. In this example the input signal is a sound wave, recorded by a microphone, but the same principle applies to any type of signal. "Recognition" of a signal pattern means that the input data is analysed and classified into one of a finite number of predetermined discrete categories. In this sense we can think of the classifier as a kind of decision-making machine. Thus, Pattern Recognition is the same thing as Signal Classification, with foundations in statistical Decision Theory.

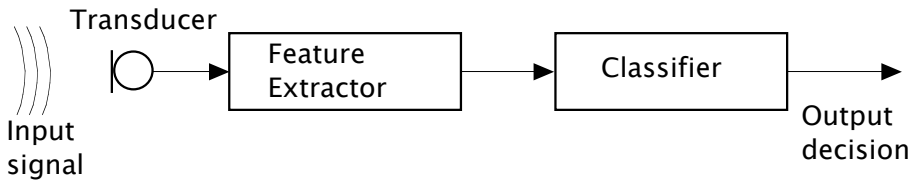


Figure 1: The main components of a general pattern recognition system: A *Transducer* records and transforms the signal pattern to be classified. The *Feature Extractor* reduces the amount of data in the input signal to a smaller set of relevant characteristics of the signal pattern. The *Classifier* selects one action that is the most appropriate, based on the received signal pattern.

This course covers three levels of classification problems, with increasing complexity, as outlined in the following table 1. These problems differ in the following aspects: Can the classifier be designed to make just one *single* decision, independently of previous decisions, or are decisions in a *sequence* dependent on each other? How many different source categories may have produced the observed features? How many feature data values are observed? How many different decision results are possible?

Table 1: Three different types of pattern-recognition problems.

Classifications	**Source**	**Observed Features**	**Decision**
Single, independent	Binary	Vector, any length	Binary
Single, independent	General	Vector, any length	General
Multiple, dependent	General	Sequence of vectors	General

The distinction between *single* or *multiple* classifications is only a question of statistical dependence. If a classifier system has been designed to perform any single classification optimally, it can of course still be used repeatedly, in every new instance of the problem it was designed for. However, the single-classification design assumes that each new decision can be made independently, without using any memory of previous classifications. An example could be a decision-support system for medical diagnosis based on laboratory test results for a patient.

A system designed for multiple dependent classifications must also account for the statistical dependence across a sequence of source states and feature vectors. Automatic speech recognition is a good example of this degree of complexity.

A simple binary classifier is introduced in Chapter 1. The general (possibly non-linear) single-classification problem is solved, in principle, in Chapters $2-3$, and Chapter 4 discusses important additional issues that must be considered when designing a practical classifier. A general sequential classifier can be designed using the theory presented in Chapters $5-7$. Some general aspects of *classifier training*, also called *machine learning*, are introduced in Chapters $6-8$.

Chapter 1

Classification and Probability

In this chapter we shall try to design a simple two-category signal classification system. At first we just argue intuitively how the automatic classifier algorithm should work. We will then express the problem in formal mathematical terms and also review some of the basic tools of probability theory that will be needed later. A complete theoretical presentation of optimal classifier design will have to wait until Chapter 3.

1.1 Example: First Binary Classifier Design

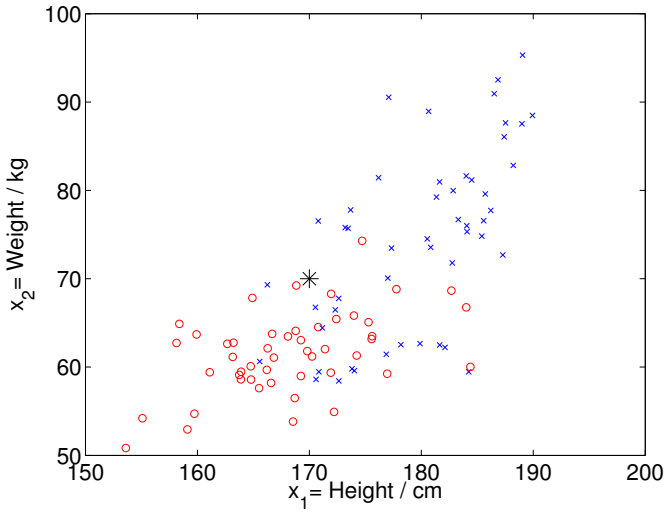


Figure 1.1: Observed data points indicating the distribution of (height, weight) value pairs for young adult women (circles) and men (crosses), for 50 random individuals of each gender. The point at (170, 70) marked (*) is the measured feature combination for an unknown test person. Should the classifier guess that this person is a man or a woman?

The purpose of this simple example is to introduce some fundamental concepts in classification theory. Let us assume you have collected data on individual body height and weight for a number of men and women, as illustrated in Fig. 1.1[1]. The data for each individual is defined numerically by a (column) vector $\boldsymbol{x} = (x_1, x_2)^T$ containing a pair of coordinates in the two-dimensional coordinate system outlined in the figure.

Your task is now to design an automatic classifier that will guess the gender of an unknown person, using only this person's body height and weight as a feature vector. If the observed feature vector was, say, $(190, 90)^T$, it seems rather clear that the classifier should output the result "man". If the observed feature vector is $(170, 70)^T$, as illustrated by the single point (*) in Fig. 1.1, the correct classification is not so obvious. How should this feature vector be classified? Right now your brain is performing as an automatic classifier system.

Let us generalize the question: For any observed feature vector $\boldsymbol{x} = (x_1, x_2)^T$ the classifier must produce a decision, either "woman" or "man". Exactly where in the figure should the boundary be drawn between these two decision results? The classification operation can obviously be mathematically defined as a scalar discrete-valued function of the observed feature vector, as

$$d(\boldsymbol{x}) = \begin{cases} 0, & \Leftrightarrow \text{guess "man"} \\ 1, & \Leftrightarrow \text{guess "woman"} \end{cases} \tag{1.1}$$

This *Decision Function* completely defines the classifier. The *domain* of the decision function is the set of all possible observed feature vectors. This set is conventionally called *Observation Space* in the context of pattern classification. In this example the Observation Space is the set of all pairs of positive real numbers. Here, the decision function has only two possible output values, corresponding to the two possible decisions of the classifier. Any observed $\boldsymbol{x}$ belongs to one of two *Decision Regions*. Each decision region is defined as the set of observed points $\boldsymbol{x}$ that result in the same decision:

$$\begin{cases} \Omega_0 = \{\boldsymbol{x} : d(\boldsymbol{x}) = 0\} \\ \Omega_1 = \{\boldsymbol{x} : d(\boldsymbol{x}) = 1\} \end{cases} \tag{1.2}$$

Thus, designing the automatic classifier algorithm is the same thing as defining the function $d(\boldsymbol{x})$, or equivalently, specifying the decision regions, or equivalently, defining the boundary curve between the decision regions.

How can we implement a machine version of this binary classifier? Later theoretical analysis will show that a good method is to first calculate a

[1] The total set of observed data used for classifier design is often called *Training Data*, because numerical parameters in the classifier must usually be "trained", i.e. adapted, to agree with the observed training data. In this simple example we do not need to introduce any special feature extraction. We just use the observed coordinate vector directly as a feature vector, without any transformation.

function value $g(\boldsymbol{x})$ that is somehow related to the degree of probability that a woman (or man) caused the observed feature vector:

$$\begin{cases} g(\boldsymbol{x}) < 0, & \Leftrightarrow \text{man more probable} \\ g(\boldsymbol{x}) = 0, & \Leftrightarrow \text{both equally probable} \\ g(\boldsymbol{x}) > 0, & \Leftrightarrow \text{woman more probable} \end{cases} \tag{1.3}$$

This type of function is called a *Discriminant Function*. It transforms any vector-valued observation $\boldsymbol{x}$ into a scalar real-valued result $y = g(\boldsymbol{x})$. The discriminant function combines all the feature-vector elements in a way that preserves all the information needed for the decision task, while discarding any feature characteristics that do not contribute to the decision. When we have defined the discriminant function, the decision function can of course be implemented simply by a *threshold*[2] operation on the output y:

$$d(\boldsymbol{x}) = \begin{cases} 0, & y = g(\boldsymbol{x}) \leq 0 \\ 1, & y = g(\boldsymbol{x}) > 0 \end{cases} \tag{1.4}$$

Obviously, there are many possible ways to define a discriminant function that leads to the same decision function. In a general classifier, a discriminant function takes a K-dimensional feature vector as input and calculates a scalar real output value. If there are more than two possible decision alternatives, we will usually need several different discriminant functions.

Chapter 3 will show general methods and design criteria to determine optimal discriminant functions for any classification problem. We will see that the discriminant functions can often be computationally quite simple. An optimal discriminant function can sometimes be defined simply as a *linear combination* of the elements in the observed K-dimensional feature vector, as

$$g(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0 \tag{1.5}$$

where $\boldsymbol{w}$ is a (column) vector of weight factors. In this special case the decision boundary is a *hyperplane* in K-dimensional observation space. If the observation space is two-dimensional, the boundary is a straight line.

1.2 Gaussian Feature Vector

Intuitive consideration of the simple example in the previous section shows that the discriminant function, and the decision function, must be somehow related to the statistical distributions of feature vectors for the two decision categories, as exemplified in Fig. 1.1. As this was an artificial example, the

[2]The boundary case $g(\boldsymbol{x}) = 0$ can be assigned arbitrarily to either of the two decisions, because this exact result will happen with zero probability.

"training data" in this case were generated from known probability distributions. The probability density functions used to generate a random sample of (Height/cm, Weight/kg) data for women and men are shown in Fig. 1.2 and 1.3. These density functions are examples of two-dimensional Gaussian (Normal) distributions. Usually we have no guarantee that the true feature-vector distribution is exactly Gaussian. Nevertheless, Gaussian distributions can often be used as good approximations in many practical situations. Therefore, it is necessary to become familiar with multi-dimensional Gaussian distributions.

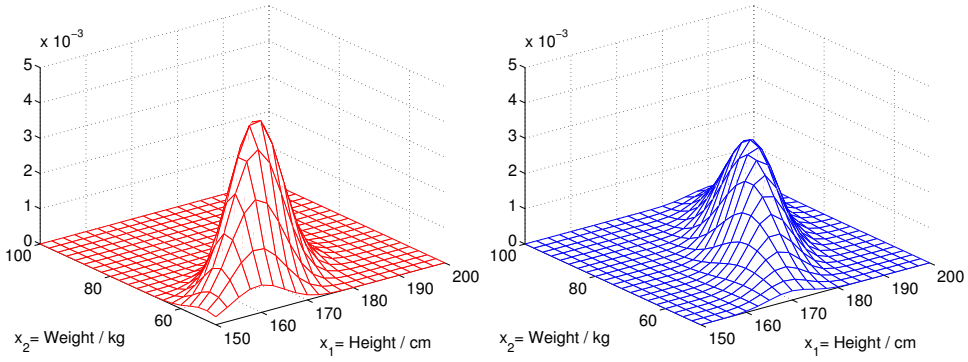


Figure 1.2: Probability density functions for (height, weight) features of women (left) and men (right), used to generate random sample data in Fig. 1.1.

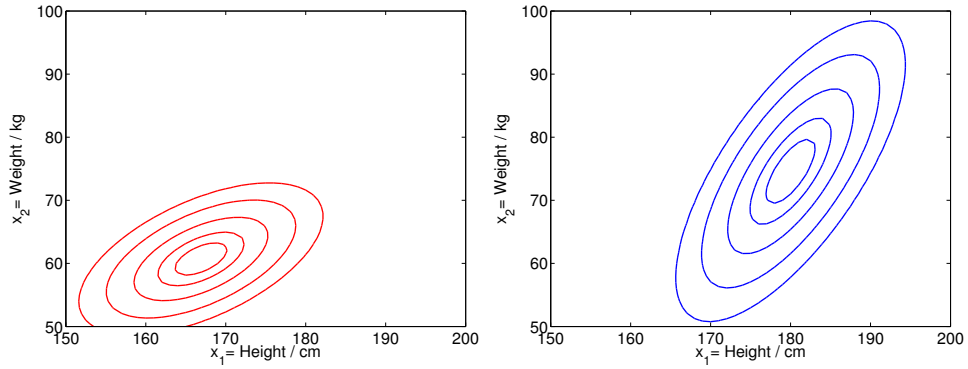


Figure 1.3: Different view of the same probability density functions for (height, weight) features of women (left) and men (right) as in Fig. 1.2. Contours of constant probability density are shown at 0.90, 0.75, 0.5, 0.25 and 0.10 of the peak probability density.

1.2.1 Two Independent Features

This section introduces the simple case of two-dimensional feature vectors with independent elements. The *probability density function* $f_X(\ )$ and the

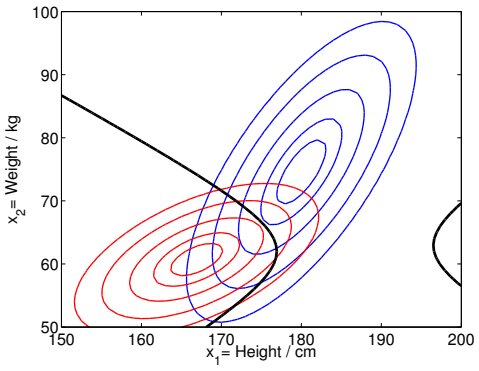


Figure 1.4: Decision regions for a classifier using the probability density functions for (height, weight) features of women and men shown in Fig. 1.3. The thick hyperbolic curve shows the *decision boundary* between the regions. Note that the decision region for women has two non-contiguous parts: an observation $(x_1, x_2) = (200, 65)$ would also be classified as a woman.

distribution function $F_X(\ )$ for a single scalar Gaussian random variable X are well known as

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{1.6}$$

$$F_X(x) = P\left[X \leq x\right] = \int_{-\infty}^{x} f_X(u)du \tag{1.7}$$

Gaussian distributions are easily extended to more than one random variable. If we have two *independent* Gaussian random variables, X_1 which is[3] $N(\mu_1, \sigma_1^2)$, and X_2 which is $N(\mu_2, \sigma_2^2)$, we know that the probability of two combined independent events is simply the product of the two event probabilities. Therefore, the distribution function $F_{\boldsymbol{X}}(\ )$ for the two-element random (column) vector $\boldsymbol{X} = (X_1, X_2)^T$ is simply

$$\begin{aligned} F_{\boldsymbol{X}}(x_1, x_2) &= P\left[X_1 \leq x_1 \cap X_2 \leq x_2\right] = \\ &= P\left[X_1 \leq x_1\right] P\left[X_2 \leq x_2\right] = \\ &= F_{X_1}(x_1) F_{X_2}(x_2) \end{aligned} \tag{1.8}$$

and the corresponding two-dimensional density function is, therefore,

$$\begin{aligned} f_{\boldsymbol{X}}(x_1, x_2) &= f_{X_1}(x_1) f_{X_2}(x_2) = \\ &= \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{(x_1-\mu_1)^2}{2\sigma_1^2}} \frac{1}{\sigma_2\sqrt{2\pi}} e^{-\frac{(x_2-\mu_2)^2}{2\sigma_2^2}} \end{aligned} \tag{1.9}$$

[3] Shorthand notation $N(\mu, \sigma^2)$ for a Gaussian variable with mean μ and variance σ^2.

This density function can be more conveniently expressed using matrix notation, as

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{1}{(2\pi)^{2/2}\sqrt{\det C}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T C^{-1}(\boldsymbol{x}-\boldsymbol{\mu})} \tag{1.10}$$

where

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}; \quad \boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}; \quad C = \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix}; \quad C^{-1} = \begin{pmatrix} 1/\sigma_1^2 & 0 \\ 0 & 1/\sigma_2^2 \end{pmatrix}.$$

1.2.2 General Correlated Features

Using the matrix notation from Eq. (1.10) it is easy to extend the Gaussian density to include correlated (dependent) feature elements. We only need to generalize the *covariance matrix*, as

$$C = \operatorname{cov}[\boldsymbol{X}] = E\left[(\boldsymbol{X}-\boldsymbol{\mu})(\boldsymbol{X}-\boldsymbol{\mu})^T\right] \tag{1.11}$$

Each element of the covariance matrix shows the scalar covariance between one pair of feature vector elements, as

$$c_{kl} = \operatorname{cov}[X_k, X_l] = E\left[(X_k-\mu_k)(X_l-\mu_l)\right] \tag{1.12}$$

The covariance matrix is obviously symmetric. A diagonal element of the covariance matrix indicates simply the usual *variance*, as

$$c_{kk} = \operatorname{cov}[X_k, X_k] = E\left[(X_k-\mu_k)^2\right] = \sigma_k^2 \tag{1.13}$$

The standard (Pearson) *correlation coefficient* ρ_{kl} is proportional to the covariance, but normalized to values in the range $-1 \leq \rho_{kl} \leq +1$, as

$$\rho_{kl} = \frac{c_{kl}}{\sigma_k \sigma_l} = \frac{c_{kl}}{\sqrt{c_{kk} c_{ll}}} \tag{1.14}$$

Assuming we know the mean vector $\boldsymbol{\mu}$ and the covariance matrix C, we can finally express the probability density function for any K-dimensional Gaussian random vector as (see Råde and Westergren, 1995, section 17.2)

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{1}{(2\pi)^{K/2}\sqrt{\det C}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T C^{-1}(\boldsymbol{x}-\boldsymbol{\mu})} \tag{1.15}$$

The constant factor is defined so that the total probability, integrated over the complete observation space, equals 1:

$$\int \cdots \int_{\text{all } \boldsymbol{x}} f_{\boldsymbol{X}}(\boldsymbol{x})\, d\boldsymbol{x} = 1 \tag{1.16}$$

If the covariance matrix is strictly diagonal, all correlations are zero. Then the Gaussian feature-vector elements are statistically independent[4], because the density function can again be written as a product of K density functions, one for each independent feature element.

[4] It is always true that *statistical independence* $\Rightarrow$ *zero correlation*, but zero correlation does *not* generally guarantee statistical independence. However, for *Gaussian* distributions, *statistical independence* $\Leftrightarrow$ *zero correlation*.

1.2.3 Coordinate Transformations and Eigenvectors

If the covariance matrix is diagonal, it is particularly easy to calculate the determinant and inverse matrix that are needed in Eq. (1.15) to evaluate the probability density for an observed K-dimensional feature vector $\boldsymbol{x}$. Fortunately, any general Gaussian random vector can be transformed by a coordinate-system rotation into a vector with diagonal covariance matrix.

We first assume that a random vector $\boldsymbol{X}$ is generated by a linear transformation of another Gaussian vector $\boldsymbol{Z}$ that has zero mean and independent elements. The covariance matrix for $\boldsymbol{Z}$ is diagonal

$$D = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \sigma_K^2 \end{pmatrix} \tag{1.17}$$

and the vector $\boldsymbol{X}$ is generated as[5]

$$\boldsymbol{X} = \boldsymbol{\mu} + P\boldsymbol{Z} \tag{1.18}$$

where P is a $K \times K$ matrix with orthonormal column vectors $\boldsymbol{p}_k$, i.e.

$$\boldsymbol{p}_k^T \boldsymbol{p}_l = \delta_{kl}; \quad PP^T = P^T P = I \tag{1.19}$$

where I is the unity matrix. The mean and covariance for $\boldsymbol{X}$ are (problem 1.1)

$$E[\boldsymbol{X}] = \boldsymbol{\mu}; \quad C = PDP^T \tag{1.20}$$

The column vectors of P form a complete K-dimensional basis. Therefore, any K-dimensional vector can be generated by a linear combination of the basis vectors as indicated by Eq. (1.18). By applying Eq. (1.19) we see that

$$\boldsymbol{Z} = P^T(\boldsymbol{X} - \boldsymbol{\mu}); \quad D = P^T C P \tag{1.21}$$

Now assume instead that we only know the mean $\boldsymbol{\mu}$ and covariance C for some general random vector $\boldsymbol{X}$ with possibly correlated elements. Then, using Eq. (1.21), we can always transform this vector into a vector $\boldsymbol{Z}$ with uncorrelated elements, i.e. diagonal covariance matrix. We only need to find the proper transformation matrix P that diagonalizes C.

As shown in introductory algebra courses, the columns of P are the normalized *eigenvectors* of C, and the elements along the diagonal of D are the corresponding *eigenvalues* (see Råde and Westergren, 1995, section 4.5).

[5] In MatLab, a general Gaussian random vector can be created as an instance of class `GaussD` in the code library supplied for this course. This class uses coordinate transformation to generate sample vectors (method `rand`) and to calculate probability densities (method `logprob`).

For any square matrix A, the scalar (possibly complex) number λ_k is called an *eigenvalue*, and the vector $\boldsymbol{e}_k \neq \mathbf{0}$ is a corresponding *eigenvector*, if

$$A\boldsymbol{e}_k = \lambda_k \boldsymbol{e}_k \tag{1.22}$$

This equation has non-zero solutions for $\boldsymbol{e}_k$, only if λ_k is a root of the *characteristic equation*

$$\det(A - \lambda_k I) = 0 \tag{1.23}$$

where I is the unity matrix with the same size as A. If A has size $K \times K$, the characteristic equation has K roots, because the left side is a polynomial of degree K. It is always possible to find K *orthonormal* eigenvectors corresponding to the eigenvalues[6]. Proofs are left as problem 1.2.

1.2.4 Random Vector Subspace

A random vector $\boldsymbol{X}$ with K elements can be generated by a transformation as in Eq. (1.18) even when the underlying random vector $\boldsymbol{Z}$ has only $L < K$ independent components. The transformation matrix P has size $K \times L$. In this case the random vector $\boldsymbol{X}$ is restricted to an L-dimensional *sub-space* of the full K-dimensional space. The subspace is defined by the basis vectors stored as the L columns of the transformation matrix P.

This representation can be very useful, for example if the observed feature "vector" $\boldsymbol{X}$ is actually an image with perhaps $K = 100 \times 100$ pixel elements, but only very few actual degrees of freedom.

However, the covariance matrix $C = PDP^T$ for $\boldsymbol{X}$ is now singular, i.e. $\det C = 0$. Therefore, any attempt to calculate a probability density in K-dimensional space, using the general Eq. (1.15), will fail. This is *not* a serious problem. The probability density appears to be infinite only because we are trying to calculate the density in K-dimensional space. The real probability density is finite as usual in the underlying L-dimensional subspace.

One practical way to handle this situation is to include the measurement noise in the theoretical model. If we measure K feature elements, there is always some random variability in each of the K feature values, even if the underlying subspace has only $L < K$ dimensions. Therefore, a more realistic model of the observed feature vector would be

$$\boldsymbol{X} = \boldsymbol{\mu} + P\boldsymbol{Z} + \boldsymbol{\epsilon} \tag{1.24}$$

where $\boldsymbol{\epsilon}$ is a random vector with K independent zero-mean noise values that all have the same finite variance σ_ϵ^2. Now, the covariance matrix for $\boldsymbol{X}$ is again non-singular. This model is the framework for *Principal-Component Analysis (PCA)*.

[6] MatLab function `eig` calculates eigenvalues and eigenvectors for any square matrix.

1.3 Gaussian Mixture Model

How can we know that the Gaussian density function in Eq. (1.15) gives a sufficiently accurate description of the K-dimensional feature-vector distribution? Usually we have only a training data set containing many observed feature vectors belonging to the same classification category. The designer of the classifier must then select a class of density functions and adapt function parameters (i.e. $\boldsymbol{\mu}$ and C in the Gaussian case) to agree with the given training data. In some situations we may use the central limit theorem and argue that the variability among observed feature vectors is caused by so many independent random factors that the Gaussian model should be adequate. In other situations it may be impossible to adapt a single Gaussian density function to the training data.

A very general form of probability density functions is provided by the *Gaussian Mixture Model* (GMM). Here, the probability density function is simply a weighted sum of M Gaussian components, each with the usual form. For a scalar random variable X the GMM density function is

$$f_X(x) = \sum_{m=1}^{M} w_m \frac{1}{\sigma_m \sqrt{2\pi}} e^{-\frac{(x-\mu_m)^2}{2\sigma_m^2}} \tag{1.25}$$

For a random vector $\boldsymbol{X}$, the GMM density function is

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \sum_{m=1}^{M} w_m \frac{1}{(2\pi)^{K/2}\sqrt{\det C_m}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_m)^T C_m^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_m)} \tag{1.26}$$

In order to normalize the probability density function properly, we must always choose the weight factors w_m such that

$$\sum_{m=1}^{M} w_m = 1 \tag{1.27}$$

A GMM can approximate any probability distribution with any desired accuracy, simply by increasing the number of components. Section 7.5 discusses how all parameters of the model can be estimated from training data.

How many GMM components do we need to characterize a given set of training data? By including more GMM components in the model, we can clearly achieve a better fit to the training data, but then perhaps the model becomes *overfitted* and does not agree with new feature vectors that should have been classified into the same category.

The GMM example in Fig. 1.6 used four Gaussian components. There is no way to know in advance which GMM size is the best. We may need to try several models with different sizes. There are formal methods to evaluate model assumptions, including a penalty for allowing too much flexibility in the model (e.g. MacKay, 2006, chapter 28). It is also possible to design learning algorithms that can automatically prune away model components that are not really needed.

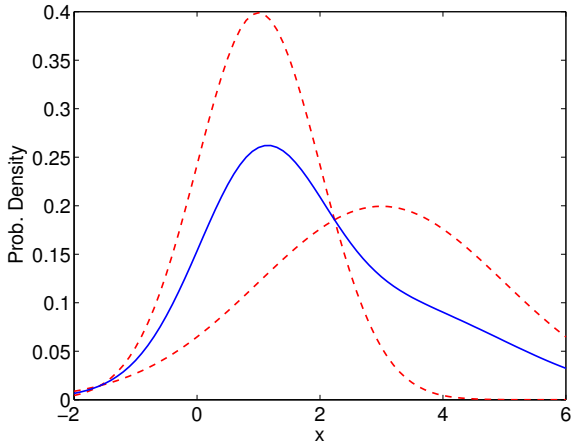


Figure 1.5: Example of a GMM probability density function (solid line) modelled as an equally weighted sum of two Gaussian component densities (dashed curves), for a scalar random variable X.

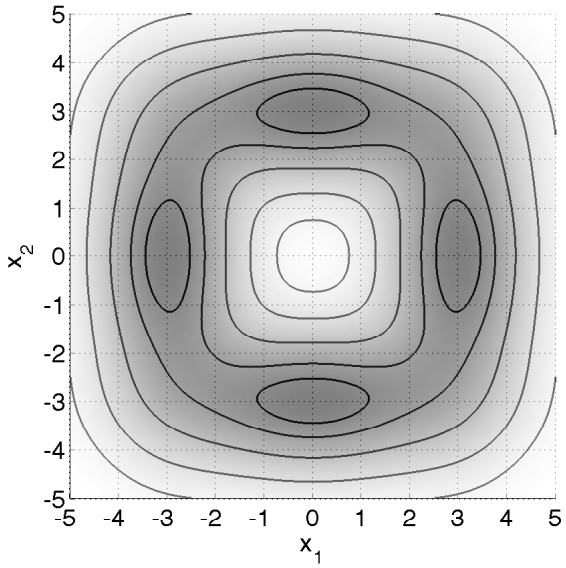


Figure 1.6: Example of a GMM probability density function in 2 dimensions. Contours of constant probability density are shown at 0.90, 0.75, 0.5, 0.25 and 0.10 of the peak probability density.

Example 1.1 (GMM): A scalar random variable X has a probability density function defined by a two-component GMM as

$$f_X(x) = w_1 \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} + w_2 \frac{1}{\sigma_2\sqrt{2\pi}} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} \tag{1.28}$$

Determine $\mu = E[X]$ and $\sigma^2 = \text{var}[X]$, expressed in terms of the given GMM parameters.

Solution:

$$\begin{aligned} E[X] &= \int_{-\infty}^{\infty} x f_X(x) dx = \\ &= w_1 \underbrace{\int_{-\infty}^{\infty} x \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} dx}_{\mu_1} + w_2 \underbrace{\int_{-\infty}^{\infty} x \frac{1}{\sigma_2\sqrt{2\pi}} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} dx}_{\mu_2} = \\ &= w_1\mu_1 + w_2\mu_2 = \mu \end{aligned} \tag{1.29}$$

Here we just identified the usual integral expressions for the single-component means μ_1 and μ_2. A similar approach is used for the variance:

$$\begin{aligned} \text{var}[X] &= \int_{-\infty}^{\infty} (x-\mu)^2 f_X(x) dx = \\ &= w_1 \underbrace{\int_{-\infty}^{\infty} (x-\mu)^2 \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} dx}_{I_1} + \\ &\quad + w_2 \underbrace{\int_{-\infty}^{\infty} (x-\mu)^2 \frac{1}{\sigma_2\sqrt{2\pi}} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} dx}_{I_2} \end{aligned} \tag{1.30}$$

The two integrals I_1 and I_2 are similar to, but not quite identical to, the single-component variances σ_1^2 and σ_2^2. To identify the component-variance

integrals more easily, we apply a common and useful trick:

$$\begin{aligned} I_1 &= \int_{-\infty}^{\infty} (x-\mu_1+\mu_1-\mu)^2 \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} dx = \\ &= \underbrace{\int_{-\infty}^{\infty} (x-\mu_1)^2 \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} dx}_{\sigma_1^2} + \underbrace{\int_{-\infty}^{\infty} (\mu_1-\mu)^2 \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} dx}_{(\mu_1-\mu)^2} + \\ &\quad + 2(\mu_1-\mu) \underbrace{\int_{-\infty}^{\infty} (x-\mu_1) \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} dx}_{0} = \\ &= \sigma_1^2 + (\mu_1-\mu)^2 \end{aligned} \tag{1.31}$$

Combining the result with a similar solution for I_2, we find

$$\text{var}\,[X] = w_1\sigma_1^2 + w_2\sigma_2^2 + w_1(\mu_1-\mu)^2 + w_2(\mu_2-\mu)^2 \tag{1.32}$$

Alternative Solution: Let us consider the random variable X as dependent on a hidden discrete random variable Z that represents the choice of mixture component no. 1 or 2. The probability-mass distribution for Z is

$$\begin{cases} P\,[Z=1] = w_1 \\ P\,[Z=2] = w_2 \end{cases} \tag{1.33}$$

The conditional distribution of X, given Z, has the following known characteristics:

$$E_X\,[X \mid Z=1] = \mu_1; \qquad \text{var}_X\,[X \mid Z=1] = \sigma_1^2 \tag{1.34}$$

$$E_X\,[X \mid Z=2] = \mu_2; \qquad \text{var}_X\,[X \mid Z=2] = \sigma_2^2 \tag{1.35}$$

For the unconditional mean and variance of X we can then apply general formulas (Råde and Westergren, 1995, sec. 17.1, pg. 408):

$$\begin{aligned} E\,[X] &= E_Z\,[E_X\,[X \mid Z]] = \\ &= w_1 E_X\,[X \mid Z=1] + w_2 E_X\,[X \mid Z=2] = w_1\mu_1 + w_2\mu_2 = \mu \end{aligned} \tag{1.36}$$

$$\begin{aligned} \text{var}\,[X] &= E_Z\,[\text{var}_X\,[X \mid Z]] + \text{var}_Z\,[E_X\,[X \mid Z]] = \\ &= \underbrace{w_1\,\text{var}_X\,[X \mid Z=1] + w_2\,\text{var}_X\,[X \mid Z=2]}_{E_Z[\text{var}_X[X \mid Z]]} + \\ &\quad + \underbrace{w_1(E_X\,[X \mid Z=1]-\mu)^2 + w_2(E_X\,[X \mid Z=2]-\mu)^2}_{\text{var}_Z[E_X[X \mid Z]]} = \\ &= w_1\sigma_1^2 + w_2\sigma_2^2 + w_1(\mu_1-\mu)^2 + w_2(\mu_2-\mu)^2 \end{aligned} \tag{1.37}$$

Example 1.1

Summary

In this chapter we designed a simple two-category classifier using only some intuitive reasoning. Main concepts introduced in this chapter:

- A *Feature Vector* is a collection of all measured (and possibly transformed) quantities that are provided as input to the classifier. Feature vectors are modeled as random variables because both the data source and the measurement devices can introduce random variations. Measurement results may be pre-processed by a *feature extractor* to reduce the amount of data.
- The *Observation Space* is the set of all possible feature-vector outcomes in a given classification situation.
- The *Decision Function* $d(\boldsymbol{x})$ maps any observed feature vector $\boldsymbol{x}$ into a discrete output value representing the result of the classification.
- A *Discriminant Function* $g(\boldsymbol{x})$ maps any observed feature vector $\boldsymbol{x}$ into a scalar real number that preserves sufficient information for the classification task.
- A *Binary* (two-category) Classifier can be optimally designed using a *single* discriminant function followed by a simple threshold mechanism.
- A multi-dimensional *Gaussian* density function can sometimes be used to model the feature-vector distribution.
- A *Gaussian Mixture Model (GMM)* is flexible enough to approximate any feature-vector distribution.

Remaining Questions

This introduction made some assumptions without formal proofs. The following important questions will be answered in Chapter 3:

- Exactly which mappings should discriminant functions perform?
- When and why are *linear* discriminant functions optimal?
- How can classifier performance be predicted?

The final question is related to classifier training and will be discussed in Chapters $6-8$:

- How can a classifier be designed when the statistical distributions of feature vectors are not known exactly?

Problems

1.1 Two scalar Gaussian random variables, Z_1 and Z_2, have zero mean, variances σ_1^2 and σ_2^2 respectively, and are statistically independent of each other. A random vector $\boldsymbol{X}$ is formed as a weighted sum

$$\boldsymbol{X} = \boldsymbol{p}_1 Z_1 + \boldsymbol{p}_2 Z_2$$

using constant (column) vectors $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$, both with the same length K.

1.1.a Calculate the mean vector $\boldsymbol{\mu}_X = E[\boldsymbol{X}]$ and the covariance matrix $C_X = \text{cov}[\boldsymbol{X}] = E\left[(\boldsymbol{X} - \boldsymbol{\mu}_X)(\boldsymbol{X} - \boldsymbol{\mu}_X)^T\right]$ for the random vector $\boldsymbol{X}$. Express the mean and covariance in general matrix form, using the given variances, and combining the given constant column vectors into a single matrix $P = (\boldsymbol{p}_1, \boldsymbol{p}_2)$.

1.1.b Calculate the mean and covariance for $\boldsymbol{X}$ numerically, in the special case with $\sigma_1 = 3$, $\sigma_2 = 1$, and vectors

$$\boldsymbol{p}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \boldsymbol{p}_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

and sketch the distribution of random points in the (x_1, x_2) plane.

1.2 For any square matrix A, the scalar number λ_k is called an *eigenvalue*, and the vector $\boldsymbol{e}_k \neq \boldsymbol{0}$ is a corresponding *eigenvector*, if

$$A\boldsymbol{e}_k = \lambda_k \boldsymbol{e}_k$$

1.2.a Given that $\boldsymbol{e}_k$ is an eigenvector of a matrix A, show that any scaled vector $c\boldsymbol{e}_k$ is also an eigenvector for the same eigenvalue. This means it is possible to normalize any eigenvector so that $\|\boldsymbol{e}_k\| = 1$.

1.2.b Determine eigenvalues and normalized eigenvectors for the symmetric matrix

$$C = \begin{pmatrix} 10 & 8 \\ 8 & 10 \end{pmatrix}$$

Hint: First solve the characteristic equation $\det(C - \lambda I) = 0$ for eigenvalues, and then find the eigenvectors. Double-check with function `eig` in MatLab.

1.2.c Given that λ is an eigenvalue of a matrix C, show that λ^{-1} is an eigenvalue of the inverse matrix C^{-1}.

1.2.d Show that all eigenvalues λ_k are real-valued for a symmetric and real-valued matrix C.
Hint: Form the scalar product between the complex conjugate of the eigenvector $\boldsymbol{e}_k$ and the expressions on each side of the original eigenvalue-eigenvector definition equation $C\boldsymbol{e}_k = \lambda\boldsymbol{e}_k$, to obtain $\overline{\boldsymbol{e}}_k^T C\boldsymbol{e}_k = \overline{\boldsymbol{e}}_k^T \lambda\boldsymbol{e}_k$. Then form a new equation as the transposed complex conjugate of this scalar equation. Finally, observe that the original and the transposed, conjugated, scalar equations are identical, because C is real and symmetric.

1.2.e For any real symmetric matrix C, use the symmetry property to show that eigenvectors $\boldsymbol{e}_k$ and $\boldsymbol{e}_l$ must be *orthogonal*, i.e. $\boldsymbol{e}_k^T\boldsymbol{e}_l = 0$, if the corresponding eigenvalues are *different*, i.e. $\lambda_k \neq \lambda_l$. Show also, that eigenvectors can be chosen to be orthogonal, if the eigenvalues are equal.
Hint: Form the scalar product between one eigenvector $\boldsymbol{e}_k$ and the expressions on each side of the original eigenvalue-eigenvector equation $C\boldsymbol{e}_l = \lambda_l\boldsymbol{e}_l$ for a different eigenvector $\boldsymbol{e}_l$, to obtain $\boldsymbol{e}_k^T C\boldsymbol{e}_l = \boldsymbol{e}_k^T\lambda_l\boldsymbol{e}_l$. Form a similar equation again, with exchanged indices k and l. Then transpose both sides of the second equation, and show that $(\lambda_k - \lambda_l)\boldsymbol{e}_k^T\boldsymbol{e}_l = 0$, which proves the statement about unequal eigenvalues. Then, if two different eigenvectors correspond to equal eigenvalues, $\lambda_k = \lambda_l$, show that any linear combination $\alpha\boldsymbol{e}_k + \beta\boldsymbol{e}_l$ is also an eigenvector. Then, it is always possible to form two eigenvectors to be orthogonal, by proper choices of α and β.

1.2.f A real symmetric matrix C, with size $K \times K$, has eigenvalues $\lambda_k; k = 1, \ldots, K$, and corresponding *orthonormal* eigenvectors $\boldsymbol{e}_k; k = 1, \ldots, K$. Collect all the eigenvalues along the main diagonal in a matrix Λ, and collect all the orthonormal eigenvectors as columns in a matrix

$$P = (\boldsymbol{e}_1, \boldsymbol{e}_2, \cdots, \boldsymbol{e}_K)$$

with orthonormality property $P^T P = I$. Show the following equalities:

$$\begin{aligned} P^{-1} &= P^T \\ PP^T &= I \\ CP &= P\Lambda \\ P^T CP &= \Lambda \\ C &= P\Lambda P^T = \sum_{k=1}^{K} \lambda_k \boldsymbol{e}_k \boldsymbol{e}_k^T \\ C^{-1} &= P\Lambda^{-1}P^T \end{aligned}$$

Hint: To show the first statement, determine $P^T P P^{-1}$.

1.3 A Gaussian random vector $\boldsymbol{X}$ is defined as

$$\boldsymbol{X} = \boldsymbol{a} + Q\begin{pmatrix} U_1 \\ U_2 \end{pmatrix}, \text{where } \boldsymbol{a} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad Q = \begin{pmatrix} 2 & 1 \\ 2 & 0 \end{pmatrix},$$

and U_1 and U_2 are statistically independent scalar random variables with $N(0,1)$ distributions.

1.3.a Determine the mean vector $\boldsymbol{\mu}_X = E[\boldsymbol{X}]$ and covariance matrix $C_X = \text{cov}[\boldsymbol{X}] = E\left[(\boldsymbol{X} - \boldsymbol{\mu}_X)(\boldsymbol{X} - \boldsymbol{\mu}_X)^T\right]$.

1.3.b Use Matlab to generate a sequence of $T = 1000$ random sample vectors $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ with the distribution of $\boldsymbol{X}$ and plot the samples as a cloud of points.

1.3.c Show that an equivalent definition of $\boldsymbol{X}$ is

$$\boldsymbol{X} = \boldsymbol{\mu}_X + P\begin{pmatrix} Z_1 \\ Z_2 \end{pmatrix}$$

where the columns of P are normalized eigenvectors of C_X, and Z_1 and Z_2 are independent Gaussian random variables with zero mean and variances equal to the eigenvalues of C_X. Use MatLab to determine P numerically, for the given Q matrix.

1.3.d Write a MatLab function that generates a sequence of random Gaussian vectors with any mean and covariance matrix, given as function arguments.

1.4 A random vector $\boldsymbol{X}$ with K elements has a multivariate normal probability density with parameters

$$\boldsymbol{\mu} = E[\boldsymbol{X}] = \begin{pmatrix} \mu_1 \\ \vdots \\ \mu_K \end{pmatrix}$$
$$\boldsymbol{C} = \text{cov}[\boldsymbol{X}] = diag(\sigma_1^2, \ldots, \sigma_K^2)$$

1.4.a Show that

$$f_{\boldsymbol{X}}(x_1, \ldots, x_K) = \frac{1}{\sqrt{2\pi}^K \sigma_1 \cdots \sigma_K} e^{-\frac{1}{2}\sum_{i=1}^{K}\left(\frac{x_i - \mu_i}{\sigma_i}\right)^2}$$

1.4.b Determine an equation for the "surfaces" of constant probability density in K-dimensional feature space. Describe these surfaces qualitatively for K =1, 2, and 3.

1.5 A Gaussian two-dimensional distribution has mean vector $(0, 0)^T$ and covariance matrix

$$C = \begin{pmatrix} 6.4 & 7.2 \\ 7.2 & 10.6 \end{pmatrix}$$

Curves of equal probability density for this distribution take the shape of ellipses in two-dimensional space. What are the two main axes of these ellipses? What is the ratio between the maximal and minimal diameters of these equal-density ellipses?
Hint: Rotate the coordinate system so that the covariance matrix becomes diagonal in the rotated coordinate system. Use MatLab as calculator.

1.6 The probability density functions for the features (Height in cm, Weight in kg) for women and men, shown in Fig. 1.2 and 1.3, were defined as Gaussian with means and standard deviations listed in the following table. Means were obtained for Swedish men and women aged 16–24, and standard deviations were taken from U.S. military statistics for army soldiers. Correlation coefficients between height and weight were set at $\rho_{12} = 0.71$.

Table 1.1: Statistical parameters for men and women

Parameters	Women	Men
Height, Mean	166.9	180.0
Weight, Mean	60.7	74.6
Height, St.Dev.	6.36	6.68
Weight, St.Dev.	8.35	11.1

1.6.a Why is the peak probability density in Fig. 1.2 higher for women than for men?

1.6.b Create MatLab `GaussD` objects for the two distributions and calculate the two probability densities for an observed feature vector $\boldsymbol{x} = (170, 70)^T$.

1.6.c Plot the boundary curve where both probability densities are exactly equal.

1.7 Estimate all GMM parameters approximately for the density function shown in Fig. 1.6.

1.8 A random variable X has a Gaussian (normal) distribution $N(\mu, \sigma^2)$. Determine the probability density function $f_Y(y)$ for the transformed variable

$$Y = X^2$$

expressed using the parameters of the density function $f_X(x)$ for X. Simplify the result in the special case with $\mu = 0$.
Hint: First express the cumulative distribution function $F_Y(y) = P[Y \leq y]$ in terms of the cumulative distribution function $F_X(x) = P[X \leq x]$, and then find the density function as

$$f_Y(y) = \frac{d}{dy} F_Y(y)$$

Chapter 2

Conditional Probability and Bayes Rule

Pattern Classification relies mainly on one single mathematical concept: *conditional probability*. This chapter is intended as a review of this concept, with focus on the applications in later chapters. Conditional probability is introduced in basic course in Probability Theory, and you may want to skip this review if you feel confident that you remember everything about conditional probability.

Let us start with a very simple example: In your pocket you have two coins. Coin#0 is a *normal* coin, and coin#1 is a *fake* coin. The fake coin is prepared to be heavier on the tail side, but both coins look exactly identical.

You select one coin S at random out of your pocket. This means there is equal probability that you choose either of the coins:

$$P_S(normal) = P_S(fake) = 0.5 \tag{2.1}$$

You toss the selected coin once and observe the random result X which can be either $X = head$ or $X = tail$. If you happened to use the normal coin, you know that both results are equally probable. This fact can be expressed by the conditional probabilities[1]

$$P_{X|S}(head \mid normal) = P_{X|S}(tail \mid normal) = 0.5 \tag{2.2}$$

Let us now assume that you also know the characteristics of the fake coin:

$$P_{X|S}(head \mid fake) = 0.7; \quad P_{X|S}(tail \mid fake) = 0.3 \tag{2.3}$$

Now, at one particular game you chose a coin, tossed it and found the *tail* facing upward. What is the probability that you used the normal coin this time?

[1] Writing convention: $P_{X|S}(x \mid s)$ means the same as $P[X = x \mid S = s]$

Let us consider all the four possible combinations and calculate all the probabilities for all combined events $X \cap S$ as shown in table 2.1. For example, to obtain the event $(X = head) \cap (S = fake)$, you would first have to choose the fake coin, which happens with probability 0.5, and then obtain a head, which has probability 0.7 when you are using the fake coin. It is easy to understand that this combined event has probability $0.5 \cdot 0.7 = 0.35$.

Now assume you observe a *tail* but you do not know which of the two coins was used. This happens with probability 0.4. If you repeat the game 1000 times, putting the used coin back into your pocket each time, then you would see tails in about $0.4 \cdot 1000 = 400$ trials. In about $0.25 \cdot 1000 = 250$ of these trials you were using the normal coin. Thus, the normal coin was used in a proportion of about $250/400 = 0.625$ of all those trials where you observed a tail. Therefore, it makes sense to say that the conditional probability that you used the normal coin, given that you observed a tail, is $0.25/0.4 = 0.625$.

Table 2.1: Probabilities for all combinations of events involving the random variables X and S.

$P_{X \cap S}(\,)$	$X = head$	$X = tail$	Sum
$S = normal$	$0.5 \cdot 0.5 = 0.25$	$0.5 \cdot 0.5 = 0.25$	$0.5 = P_S(normal)$
$S = fake$	$0.5 \cdot 0.7 = 0.35$	$0.5 \cdot 0.3 = 0.15$	$0.5 = P_S(fake)$
Sum	$0.6 = P_X(head)$	$0.4 = P_X(tail)$	1

Every observation reduces the set of possible remaining event combinations. The conditional probability, given the observation, involves only these remaining event combinations. *Before* the observation, the probability $P_S(normal)$ is 0.5. This value is called the *A Priori* probability. *After* the observation, the probability that the normal coin was used has a different value: $P_{S|X}(normal \mid tail) = 0.625$. This is called the *A Posteriori*[2] probability. All similar calculations are elegantly formulated by *Bayes Rule*

$$P_{X \cap S}(x, s) = P_{X|S}(x \mid s) P_S(s) = P_{S|X}(s \mid x) P_X(x) \tag{2.4}$$

where the function arguments x and s represent any outcome value that the random variables X and S can have. As exemplified in table 2.1 we note that the previously unknown quantity $P_X(x)$ can always be calculated as

$$P_X(x) = \sum_{\text{all } s} P_{X \cap S}(x, s) = \sum_{\text{all } s} P_{X|S}(x \mid s) P_S(s) \tag{2.5}$$

This is just a slightly different way of writing the well-known formula for any events A and B:

$$P\left[A \cap B\right] = P\left[A \mid B\right] P\left[B\right] = P\left[B \mid A\right] P\left[A\right] \tag{2.6}$$

[2] *Prior* means before, and *posterior* means after.

Until now both X and S were *discrete* random variables. However, in the next chapters we will often need a corresponding relation for the situation where X is a *continuous* random variable, and S is discrete. In this case X is characterised by a continuous probability density function $f_X(\ )$ that can be evaluated as $f_X(a)$ for any particular outcome $X = a$. The dependence on S is expressed by a conditional density function $f_{X|S}(\ )$, that can be evaluated as $f_{X|S}(a \mid j)$ for any outcome $X = a$, given any $S = j$. Then, to determine the probability of an event $S = j$, given an observation $X = a$, the most useful version of Bayes rule is

$$P\left[S = j \mid X = a\right] = P_{S|X}(j \mid a) = \frac{f_{X|S}(a \mid j)P_S(j)}{f_X(a)} \tag{2.7}$$

To prove this relation, we must think about the probability for the event that X would fall in a small interval around a, i.e., $(a - \delta/2 < X < a + \delta/2)$, and calculate:

$$\begin{aligned} P\left[S = j \mid a - \delta/2 < X < a + \delta/2\right] &= \frac{\int_{a-\delta/2}^{a+\delta/2} f_{X|S}(x \mid j)dx\, P_S(j)}{\int_{a-\delta/2}^{a+\delta/2} f_X(x)dx} \\ &\xrightarrow{\delta \to 0} \frac{\delta f_{X|S}(a \mid j)P_S(j)}{\delta f_X(a)} = \frac{f_{X|S}(a \mid j)P_S(j)}{f_X(a)} \end{aligned} \tag{2.8}$$

If the conditional density functions $f_{X|S}(\ )$ are known for all possible outcomes $S = j$, then the unconditional density function for X can be easily evaluated as

$$f_X(a) = \sum_{\text{all } j} f_{X|S}(a \mid j)P_S(j) \tag{2.9}$$

Thus, the conditional discrete probability-mass distribution for S, given X, has the usual property of any discrete probability-mass distribution:

$$\sum_{\text{all } j} P_{S|X}(j \mid a) = 1 \tag{2.10}$$

Problems

2.1 (The "Monty Hall Problem") You are invited to participate in a TV game show where you can win an expensive new car. The car is parked behind one of three closed doors.

First you are told to choose one door, where you believe the car is hidden. You stand in front of this door to mark your choice.

Then the game show host, who knows where the car is, opens one of the remaining two doors and shows that there is no car behind it. Now you know the car must be either behind the door you first selected or behind the remaining closed door.

Next, the game show host asks you if you want to change your mind and choose the other door instead. Do you accept this offer, or do you stay where you are? Finally, the door you have chosen is opened, and you get the car, if it is there! Can you improve your chances by moving to that other door?

Hint: Do not rely on your intuition! Do a formal calculation! Let us denote the doors as 1, 2, and 3, where, arbitrarily, door number 1 is the one you first blocked. Model the hidden place of the car as a discrete "state of nature" S that can have values 1, 2, or 3. Model your observation of which door the host first opens as another discrete random variable X,which can have values 2 or 3. Calculate the conditional probabilities $P(S = i|X = 2)$ and $P(S = i|X = 3)$ for all the possible states i =1, 2, or 3.

2.2 Two continuous random variables X and Y are statistically independent, and both have a Gaussian (normal) distribution. X is $N(\mu_X, \sigma_X^2)$ and Y is $N(\mu_Y, \sigma_Y^2)$. This means that their probability density functions can be written as

$$f_X(x) = \frac{1}{\sigma_X\sqrt{2\pi}} e^{-\frac{(x-\mu_X)^2}{2\sigma_X^2}}$$

and similarly for Y. A new random variable is created as $Z = X + Y$.

2.2.a What is the probability density function $f_Z(\ )$ for the random variable Z?
Hint: A sum of Gaussian random variables is also Gaussian.

2.2.b What is the conditional probability density function $f_{Z|X}(z|x_1)$ for the random variable Z, given a specific observed outcome $X = x_1$?
Hint: Bayes rule can always be used, but there is a simpler method.

2.2.c What is the conditional probability density function $f_{X|Z}(x|z_1)$ for the random variable X, given a specific observed outcome $Z = z_1$?
Hint: Bayes' rule can always be used, and it is equivalent to this direct approach: Consider the joint random variable (z, x), and express

$f_{Z,X}(z,x) = f_{Z|X}(z|x)f_X(x)$. Then regard $z = z_1$ as known, and normalize the remaining function $f_{X|Z}(x|z_1) \propto f_{Z,X}(z_1,x)$ to be a proper density function for x.

2.3 A continous random variable X has a Gaussian (normal) distribution, but its mean and variance are known to depend on another discrete random variable S which can be either 0 or 1 with equal probability. X is $N(0,1)$, if $S = 0$, and $N(1,2^2)$, if $S = 1$.

2.3.a Express the conditional probability density functions $f_{X|S}(x|i)$ for X, given outcomes $S = i \in \{0,1\}$.

2.3.b What is the overall probability density function $f_X(\,)$ for X, without any observed value of S.

2.3.c What is $P_{S|X}(0|0.3)$ i.e. the conditional probability that $S = 0$, given an observed outcome $X = 0.3$?

2.4 A continous random variable X has a Gaussian (normal) distribution $N(1,1)$, and another independent discrete random variable S can have values either 1 or 2 with a probability mass distribution

$$P_S(i) = \begin{cases} 0.8, & i = 1 \\ 0.2, & i = 2 \end{cases}$$

A new random variable is created as $Z = SX$.

2.4.a What is the conditional probability density function $f_{Z|S}(z|i)$ for the random variable Z, given a specific observed outcome $S = i$?

2.4.b What is the probability density function $f_Z(\,)$ for the random variable Z?

2.4.c What is the conditional probability mass distribution $P_{S|Z}(i|z_1)$ for the random variable S, given a specific observed outcome $Z = z_1$?

2.4.d What is the conditional probability density function $f_{X|Z}(x|z_1)$ for the random variable X, given a specific observed outcome $Z = z_1$?

2.5 A time-sequence of discrete random variables $(S_t; t = 1 \ldots \infty)$ forms a first-order Markov chain. This means that the conditional probability distribution for S_t depends only on the outcome of the previous variable S_{t-1} in the sequence. This conditional dependence can be completely specified by a *transition probability matrix*,

$$A = \begin{pmatrix} 0.7 & 0.1 & 0.2 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0 & 0.9 \end{pmatrix}, \text{ with elements } a_{ij} = P(S_t = j | S_{t-1} = i).$$

2.5.a What is the conditional probability distribution $P_{S_{13}|S_{12}}(k|2)$ for S_{13}, given an observed outcome $S_{12} = 2$?

2.5.b What is the conditional probability distribution $P_{S_{13}|S_{11},S_{12}}(k|1,2)$ for S_{13}, given observed outcomes $S_{11} = 1$ and $S_{12} = 2$?

2.5.c Assume that we have previous knowledge that the Markov chain was initialized so that the unconditional probabilities $P_{S_{11}}(i)$ are equal for $i = 1 \ldots 3$. What is then the conditional probability distribution $P_{S_{11}|S_{12}}(k|2)$, given an observed outcome $S_{12} = 2$?

2.5.d What is the conditional probability distribution $P_{S_{11}|S_{10},S_{12}}(k|3,2)$ for S_{11}, given observed outcomes $S_{10} = 3$ and $S_{12} = 2$?

2.6 Prove the following useful versions of Bayes' rule:

$$P(A \cap B \cap C) = P(A|B \cap C)P(B|C)P(C)$$
$$P(A \cap B|C) = P(A|B \cap C)P(B|C)$$

Chapter 3

Bayesian Pattern Classification

A few years ago the Swedish weekly magazine "Ny Teknik" presented the following problem:

> Someone shows you two number sequences, {24513} and {12345}, and tells you that one of the sequences was generated by a random number generator in a computer and the other was generated by a person. Most people with no special education in probability theory would say spontaneuously that the sequence {12345} was most probably the one written down by a human being. However, many well-educated engineers would argue, some of them very strongly, that both sequences are exactly equally probable to have been generated by a human or by a computer. Nevertheless, in this case the uneducated naive guess is the correct answer. Why?

As predicted by the problem inventor, a debate article appeared a couple of weeks later in the magazine. It was written by an engineer who took a course in mathematical statistics as part of his university degree, maybe at KTH. He argued very emphatically that the uneducated guess is wrong, because it is influenced by "number magic" assigning a special "subjective meaning" to one of the arbitrary number sequences.

This problem is an example of signal classification, although slightly artificial. We all make decisions of a similar type all the time, every day. This section will show how to make any such decision in the best possible way. The presented method of thinking will illustrate what was wrong with the "well-educated" engineer's argument about the problem in "Ny Teknik". The material in this chapter is based mainly on the excellent reference book by Duda et al. (2000).

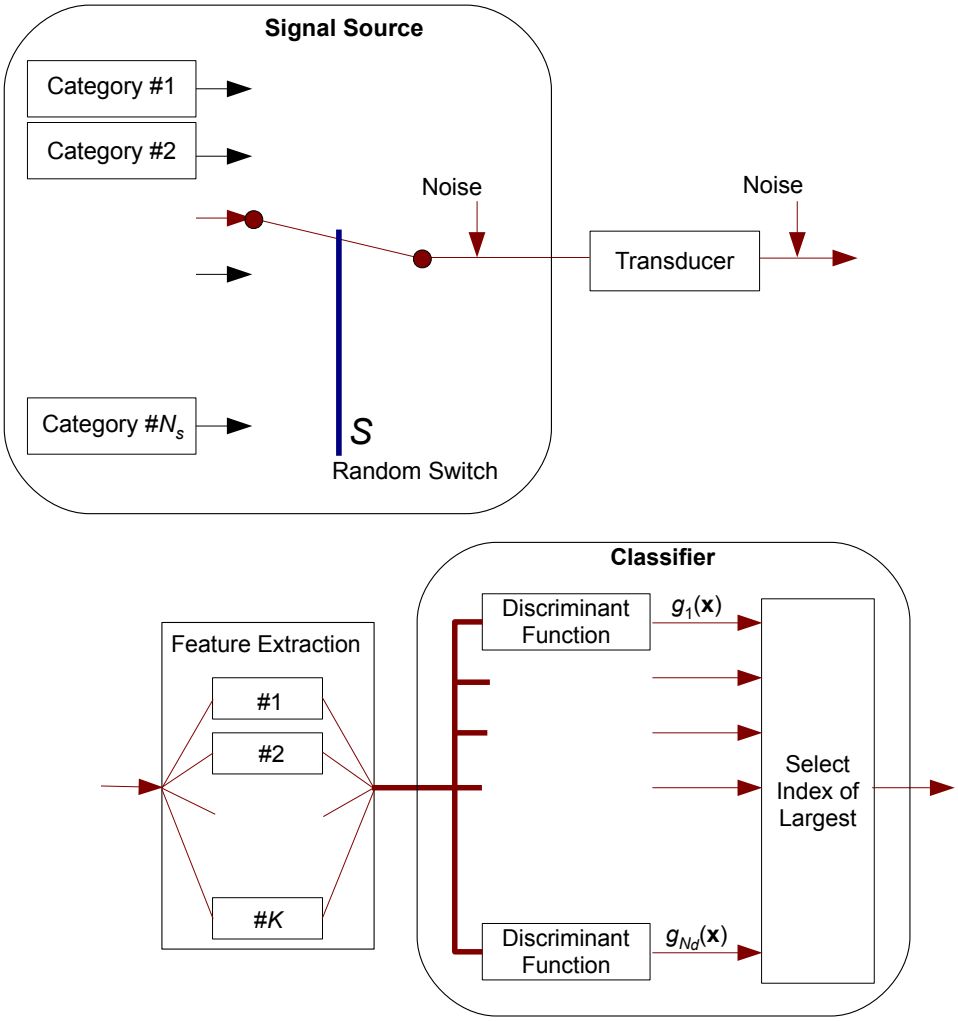


Figure 3.1: General signal classification. A signal source randomly selects an internal source category $S \in \{1 \dots N_s\}$. The generated signal may then be transduced and transformed. Noise may be introduced at several points. The source category is unknown to the external world. The only evidence available to the classifier is a feature vector $\boldsymbol{x}$ containing K separate features, i.e. signal characteristics. Depending on the observed feature vector the classifier selects one action from a set of N_d alternatives. The classifier compares the scalar output values from N_d different *discriminant functions* and searches for the largest value. The output is an index $d \in \{1 \dots N_d\}$ of the selected action.

3.1 Problem Definition

The general classification problem is presented in Fig. 3.1. It is easy to see that the simple binary classifier discussed in chapter 1 was just a special case of this general classifier. Now the problem will be extended in several aspects:

- The source is chosen from a set of arbitrarily many categories, not just two.
- The decision is chosen from a set of arbitrarily many decision categories, not just two.
- A very general performance criterion is defined.

Although Fig. 3.1 looks like a signal-processing block diagram, a little imagination should reveal that the illustrated process of pattern classification is, in fact, very general. The figure might as well illustrate how a doctor decides on a diagnosis based on medical examination evidence $\boldsymbol{x}$. Or maybe the diagram shows how an accused prisoner, who might be either innocent ($S = 1$), or a murderer ($S = 2$), is tried in a criminal court on the basis of police evidence $\boldsymbol{x}$, and sentenced to "life time in prison" ($d = 1$), or "psychiatric treatment"($d = 2$), or "not guilty" ($d = 3$).

The block diagram in Fig. 3.1 illustrates how one particular classifier may behave in one particular situation. The source category has some definite, but unknown, value $S = j$, and the feature vector has some particular outcome value $\boldsymbol{x}$. However, the design of the classifier must take account of all possible situations that can occur. The industrial designer of a classifier cannot, of course, know exactly what will happen with a particular product with serial number *nn.nnn*. The design must be optimized so that all individual products of the design perform as well as possible, on average. Therefore, the designer must use a statistical model to define the conditions where the classifier will work.

3.1.1 A Priori Source Category Distribution

The source category is regarded as a discrete-valued random variable S which takes a particular value $S = j$ with probability $P_S(j)$. This distribution is often called the *a priori* probability distribution for the source category. This is the unconditional source-category probability distribution, before any observation has been made. We assume that this probability distribution is known by the designer of the classifier. The sum over all possible categories is, as it should be for any probability distribution,

$$\sum_{j=1}^{N_s} P_S(j) = 1 \tag{3.1}$$

3.1.2 Feature Vector Distribution

The *observed feature vector* $\boldsymbol{x}$ is regarded as a particular outcome of a random vector $\boldsymbol{X}$. The probability distribution for $\boldsymbol{X}$ must depend on the source category. (Otherwise, the feature vector would not convey any information about the unknown source type.) This dependence is defined by N_s different conditional probability density functions for $\boldsymbol{X}$, one for each sub-source. Therefore, these density functions are called conditional density functions, denoted as

$$f_{\boldsymbol{X}|S}(\ )$$

The function value $f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j)$ defines the conditional probability density for observing an outcome $\boldsymbol{X} = \boldsymbol{x}$, given that the source category was known to be $S = j$. Note that the function subscript indicates the random variables, whereas the function arguments are the actual outcome of the random variables. We assume that these density functions are completely known, although this is seldom true in reality. (Usually, these density functions must be estimated from training data, but this discussion will have to wait until Ch. 7.) As usual, all the probability density functions are normalized so that the multi-dimensional integral over all possible values equals 1, i.e.

$$\int_{\text{all } \boldsymbol{x}} f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j) d\boldsymbol{x} = 1, \text{ for any } j \in \{1 \ldots N_s\} \tag{3.2}$$

3.1.3 Decision Function

The classifier must be constructed to automatically select a specific action for any actual observation $\boldsymbol{x}$. This means we must implement a deterministic decision function $d(\ )$, that maps any observed feature vector $\boldsymbol{x}$ to an integer number $d(\boldsymbol{x}) \in \{1 \ldots N_d\}$, which is the classifier result. Designing the classifier is the same as determining the decision function.

$$d(\boldsymbol{x}) = i \quad \Leftrightarrow \quad \text{"Perform action no. } i \text{ in the list of actions"}$$

This formalism is very general and allows any set of possible actions without making things any more difficult. However, in many important applications the task of the classifier is simply to guess the hidden source category S. In this special case we have $N_d = N_s$, and the classifier's output integer should be interpreted as

$$d(\boldsymbol{x}) = i \quad \Leftrightarrow \quad \text{"I believe that } S = i\text{"}$$

3.1.4 A Posteriori Source category Distribution

Even if the final goal of the classifier is *not* to guess the source category, we will soon see that a crucial step in the classifier design process is to calculate

the conditional probability-mass distribution for the source category S, given any particular observation $\boldsymbol{X} = \boldsymbol{x}$:

$$P_{S|\boldsymbol{X}}(j \mid \boldsymbol{x}), \quad j = 1 \ldots N_s \tag{3.3}$$

This discrete distribution accurately represents all the knowledge about the source category that is available in the classifier after observing the feature vector. Therefore, this distribution is often called the *a posteriori* probability distribution for the source category. It can be obtained using the extremely important *Bayes Rule*:

$$\boxed{P_{S|\boldsymbol{X}}(j \mid \boldsymbol{x}) = \frac{f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j) P_S(j)}{f_{\boldsymbol{X}}(\boldsymbol{x})}} \tag{3.4}$$

$$\text{where } f_{\boldsymbol{X}}(\boldsymbol{x}) = \sum_{j=1}^{N_s} f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j) P_S(j) \tag{3.5}$$

Here, $f_{\boldsymbol{X}}(\boldsymbol{x})$ is the overall non-conditional feature density, averaged over all possible source categories. It is easy to see that the sum of the a posteriori probabilities over all source categories is, as it should be,

$$\sum_{j=1}^{N_s} P_{S|X}(j \mid \boldsymbol{x}) = 1, \text{for any } \boldsymbol{x} \tag{3.6}$$

It is easy to remember Bayes rule mechanically, especially in its symmetric form

$$P_{S|X}(j \mid \boldsymbol{x}) f_{\boldsymbol{X}}(\boldsymbol{x}) = f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j) P_S(j) \tag{3.7}$$

However, as the calculation of conditional probabilities is so vital in many applications, it is important to also understand this concept deep in one's heart. The review examples discussed in chapter 2 may be of some help.

3.2 Bayes Minimum-Risk Decision Rule

In order to determine an optimal decision rule $d(\boldsymbol{x})$, we must first define a general performance criterion for the classifier. Then we can optimize the classifier design with regard to the chosen criterion. It would, of course, be nonsense to say that something is "optimal", if the criterion is not clearly defined. In this section we define a completely general performance criterion that can later be adapted to any kind of classification problem.

If the task of the classifier is simply to guess the hidden source category, the probability of decision error is a good design criterion. This probability should, of course, be as small as possible. However, in some applications the different types of error may not be equally disastrous. For example, most

people would say that criminal courts should always require very strong evidence "beyond reasonable doubt" before deciding that somebody is "guilty". It is better to let a murderer go free than to put an innocent person in jail, if the evidence is not clear. In this case the simple probability of any type of decision error is not a suitable criterion.

Let us define a numerical *Loss Matrix* of size $N_d \times N_s$ as

$$\boldsymbol{L} = \begin{pmatrix} L(d=1 \mid S=1) & \cdots & L(d=1 \mid S=N_s) \\ \vdots & \ddots & \vdots \\ L(d=N_d \mid S=1) & \cdots & L(d=N_d \mid S=N_s) \end{pmatrix} \tag{3.8}$$

with elements $L_{ij} = L(d = i \mid S = j)$ indicating the loss we suffer if the classifier would decide $d = i$, when the true source category was $S = j$. All the $N_d N_s$ loss values are assumed to be known and tabulated in the loss matrix. The performance criterion will then be the overall expected loss, calculated as an average over all possible events the classifier might encounter.

Suppose now that the classifier observes a particular feature vector $\boldsymbol{x}$, and we consider designing the classifier to output $d(\boldsymbol{x}) = i$ in this event. If the true source category were actually $S = j$, this decision would cause the loss L_{ij} defined in the loss matrix. However, we cannot know the actual source category. Therefore, we must apply Bayes Rule from Eq. (3.4) to calculate instead the *Conditional Expected Loss* $R(i \mid \boldsymbol{x})$ as an average over all possible source categories, given the observed feature vector $\boldsymbol{x}$:

$$R(i \mid \boldsymbol{x}) = \sum_{j=1}^{N_s} L(d = i \mid S = j) P_{S|\boldsymbol{X}}(j \mid \boldsymbol{x}) \tag{3.9}$$

This quantity is a measure of the average loss we make if we would design the classifier to decide $d(\boldsymbol{x}) = i$. The conditional expected loss is often called the *Conditional Risk*.

Whenever the classifier has received a particular observation $\boldsymbol{x}$ we minimize the average loss if the classifier computes $R(i \mid \boldsymbol{x})$ for all possible actions $i = 1 \ldots N_d$, and selects the particular action that gives the smallest risk value. This means we can design an optimal classifier by implementing *Bayes Minimum-Risk Decision Rule*:

$$\boxed{d(\boldsymbol{x}) = \underset{i}{\operatorname{argmin}}\, R(i \mid \boldsymbol{x})} \tag{3.10}$$

To see that this rule actually minimizes the total expected loss Q we can calculate the average loss across all possible outcomes of the random feature vector $\boldsymbol{X}$, as

$$Q = \int_{\text{all } \boldsymbol{x}} R(d(\boldsymbol{x}) \mid \boldsymbol{x}) f_{\boldsymbol{X}}(\boldsymbol{x}) d\boldsymbol{x} \tag{3.11}$$

Obviously this integral is minimized if the integrand is chosen as small as possible for every $\boldsymbol{x}$. This is exactly what Bayes Minimum-Risk Decision Rule in Eq. (3.10) does.

3.2.1 Special case: Minimum Error Rate

The minimum-risk decision rule in Eq. (3.10) can be greatly simplified, if the task of the classifier is to guess the hidden source category with the smallest possible probability of error. In this case $N_d = N_s$ and the decision is correct if $d(\boldsymbol{x})$ is equal to the source category S. Let us therefore define a loss matrix

$$\boldsymbol{L} = \begin{pmatrix} 0 & 1 & \cdots & 1 \\ 1 & 0 & \ddots & \vdots \\ \vdots & 1 & \ddots & 1 \\ 1 & \cdots & 1 & 0 \end{pmatrix}, \text{i.e. } L(d = i \mid S = j) = \begin{cases} 0, & i = j \\ 1, & \text{otherwise} \end{cases} \tag{3.12}$$

This means that correct decisions cost nothing and all types of error cost 1 unit. Suppose now that the classifier observes a feature vector $\boldsymbol{x}$, and we consider designing it to choose output $d(\boldsymbol{x}) = i$ in this event. The conditional risk would then be

$$\begin{aligned} R(i \mid \boldsymbol{x}) &= \sum_{j=1}^{N_s} L(d = i \mid S = j) P_{S|\boldsymbol{X}}(j \mid \boldsymbol{x}) = \sum_{j \neq i} P_{S|\boldsymbol{X}}(j \mid \boldsymbol{x}) \\ &= 1 - P_{S|\boldsymbol{X}}(i \mid \boldsymbol{x}) \end{aligned} \tag{3.13}$$

To minimize the risk the classifier should obviously guess that the source was actually of type i with the greatest value of the a posteriori conditional probability $P_{S|\boldsymbol{X}}(i \mid \boldsymbol{x})$. This is called the *Maximum a Posteriori probability (MAP) Decision Rule*

$$d(\boldsymbol{x}) = \operatorname*{argmax}_i P_{S|\boldsymbol{X}}(i \mid \boldsymbol{x}) = \operatorname*{argmax}_i \frac{f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) P_S(i)}{f_{\boldsymbol{X}}(\boldsymbol{x})} \tag{3.14}$$

Fortunately, it is not necessary to implement the complete computation of the a posteriori probability using Bayes rule. For any given observation $\boldsymbol{x}$ the denominator $f_{\boldsymbol{X}}(\boldsymbol{x})$ in Eq. (3.14) is the same for all i and it is not necessary to perform the division. The same decision will be obtained by the following simplified form of the *MAP Decision Rule*

$$\boxed{d(\boldsymbol{x}) = \operatorname*{argmax}_i f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) P_S(i)} \tag{3.15}$$

3.2.2 Special case: Equal A-Priori Probabilities

The MAP decision rule in Eq. (3.15) allows a further simplification if all the source categories are a priori equally probable, i.e. $P_S(i) = 1/N_s$ for all i. Then we can achieve minimum error probability by the even simpler *Maximum Likelihood (ML) Decision Rule*

$$\boxed{d(\boldsymbol{x}) = \underset{i}{\operatorname{argmax}}\, f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i)} \tag{3.16}$$

3.3 Discriminant Functions

It is easy to see that all the three decision rules defined so far, the Mimimum-Risk rule in Eq. (3.10), the MAP rule in Eq. (3.15), and the ML rule in Eq. (3.16), imply that the classifier must be designed to calculate the values of N_d different scalar real-valued functions of the observed feature vector $\boldsymbol{x}$ and then choose the output corresponding to the function which has the largest value. All the rules can be expressed in a common form, as the *General Decision Rule*

$$\boxed{d(\boldsymbol{x}) = \underset{i=1\ldots N_d}{\operatorname{argmax}}\, g_i(\boldsymbol{x})} \tag{3.17}$$

The functions $g_i(\boldsymbol{x})$ are called *discriminant functions*. The special decision rules defined in the previous section are easily obtained by defining

$$\begin{aligned} \text{Min. Risk Rule:} \quad & g_i(\boldsymbol{x}) = -R(i \mid \boldsymbol{x}) \\ \text{MAP Rule:} \quad & g_i(\boldsymbol{x}) = f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) P_S(i) \\ \text{ML Rule:} \quad & g_i(\boldsymbol{x}) = f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) \end{aligned} \tag{3.18}$$

This proves that the general classifier structure illustrated in Fig. 3.1 is indeed always an optimal design, if the discriminant functions are chosen properly.

Of course, the result of the decision will be the same, if all discriminant functions are multiplied by a common positive scale factor, or if a constant term is added to all of them. We can obtain a set of new discriminant functions, with identical decision results, by applying a transformation

$$g_i'(\boldsymbol{x}) = h(g_i(\boldsymbol{x})) \tag{3.19}$$

where $h(\ )$ is any monotonically increasing function. In many cases such a transformation can simplify the design considerably. For example, the conditonal density functions often have an exponential form, like

$$f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) = ce^{-\gamma_i(\boldsymbol{x})} \tag{3.20}$$

As the logarithm function is monotonically increasing, we can choose a transformation $h(\ ) = \ln(\ )$. We can also remove the positive constant c, because it has no effect on the classifier performance. We then define a much simpler set of discriminant functions, for example for the MAP rule, by choosing

$$g_i(\boldsymbol{x}) = \ln \frac{1}{c} f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) P_S(i) = -\gamma_i(\boldsymbol{x}) + \ln P_S(i) \tag{3.21}$$

For a two-category classifier, i.e. $N_d = 2$, the general decision rule in Eq. (3.17) implies that we should define two discriminant functions, $g_1(\boldsymbol{x})$ and $g_2(\boldsymbol{x})$ and then choose the one with the largest value. However, in this case the classifier can be simplified even more, if we define just a single discriminant function

$$g(\boldsymbol{x}) = g_1(\boldsymbol{x}) - g_2(\boldsymbol{x}) \tag{3.22}$$

and then use the decision rule

$$d(\boldsymbol{x}) = \begin{cases} 1, & g(\boldsymbol{x}) > 0 \\ 2, & \text{otherwise} \end{cases} \tag{3.23}$$

Thus, we have now proved that an optimal *two-category classifier* can always be designed using a *single threshold device* as the final block in the classifier. This justifies the intuitive discussion of the binary classifier in chapter 1.

3.4 Decision Regions and Error Probability

For any observed feature vector $\boldsymbol{x}$ the decision function $d(\boldsymbol{x})$ defines a unique classifier output. We can think of the decision rule as a mechanism which divides feature space into disjoint *Decision Regions*

$$\Omega_i = \{\boldsymbol{x} :\ d(\boldsymbol{x}) = i\} \tag{3.24}$$

Any observation $\boldsymbol{x}$ belonging to a particular decision region leads to the same decision. Thus, the decision mechanism can also be seen as a quantization of the continuous observation space, representing an observed feature vector $\boldsymbol{x}$ only by the index of the decision region it belongs to. Fig. 3.2 illustrates decision regions in a two-dimensional observation space and 3.3 in a one-dimensional space. In these examples the decision regions are contiguous, but this is not necessary.

If the task of the classifier is to guess the hidden category of the source, we consider any result $d(\boldsymbol{x}) = i$, when $S = i$, as a correct decision. A two-category classifier can then make two types of errors: It can decide 1 when the source category was actually 2, or vice versa, as illustrated in Fig. 3.3. The total probability of error is then

$$\begin{aligned} P[error] &= P[\boldsymbol{X} \in \Omega_2 \cap S = 1] + P[\boldsymbol{X} \in \Omega_1 \cap S = 2] \\ &= P_S(1) \int_{\Omega_2} f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid 1) d\boldsymbol{x} + P_S(2) \int_{\Omega_1} f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid 2) d\boldsymbol{x} \end{aligned} \tag{3.25}$$

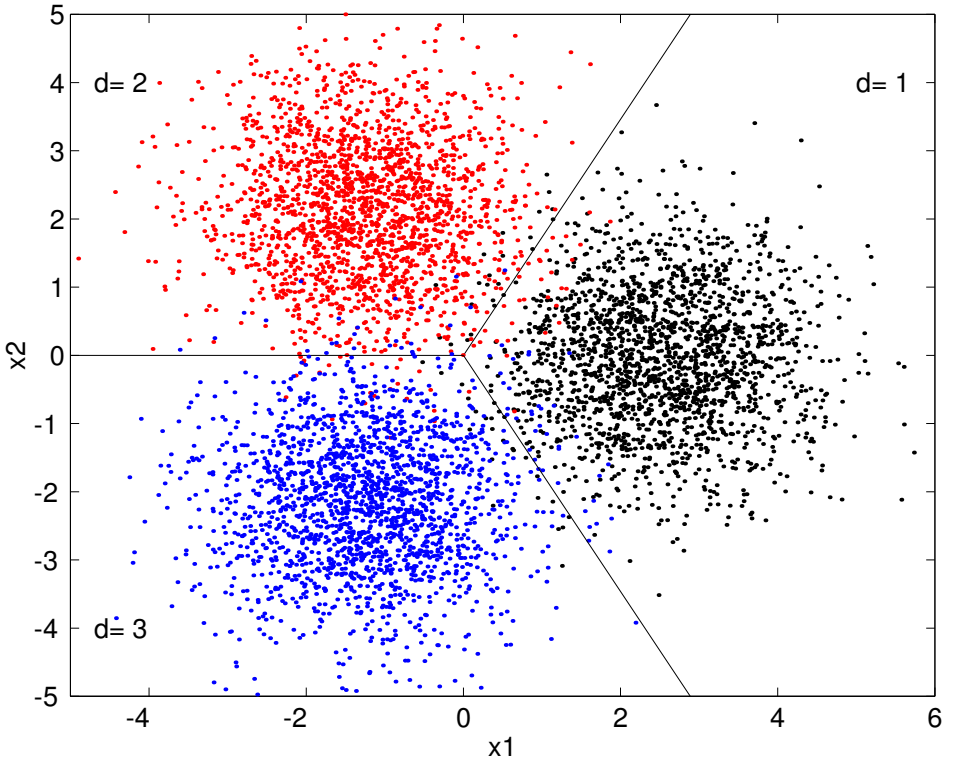


Figure 3.2: Three-category classification in a two-dimensional feature space. Scatter plots show, for each source category, a large number of dots representing random outcomes of the feature vector. Optimal decision regions are shown as defined by the ML decision rule.

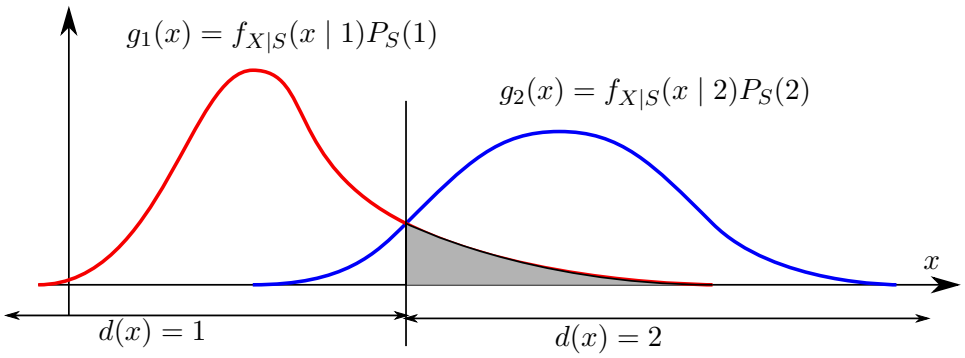


Figure 3.3: Optimal two-category classification in a one-dimensional feature space. The vertical axis indicates values of the discriminant functions for the MAP decision rule. The shaded area represents the probability of one type of error: decision $d(x) = 2$ when $S = 1$.

In a more general case where $N_d = N_s > 2$, there are more ways to be wrong than to be right. Then it is easier to first calculate the probability of correct

decisions, as

$$P\left[correct\right] = \sum_{i=1}^{N_s} P\left[\boldsymbol{X} \in \Omega_i \cap S = i\right] = \sum_{i=1}^{N_s} P_S(i) \int_{\Omega_i} f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) d\boldsymbol{x} \quad (3.26)$$

and then obtain $P\left[error\right] = 1 - P\left[correct\right]$.

3.5 Simple Example: Scalar Gaussian Feature

In many practical situations it is reasonable to assume that the feature variables have Gaussian (Normal) conditional distributions. Even if this is not exactly true, the Gaussian distribution can often be a good approximation. This section presents an example to show that classifier design can be very simple.

Problem: Let us design a two-category classifier to guess the source category with minimum error probability, by observing a single scalar feature variable X. We have reasons to assume that X is conditionally $N(\mu_i, \sigma^2)$, given that the source is $S = i$, with i either 1 or 2. The variance is caused only by additive external noise which is independent of the source category. Therefore, it is safe to assume that the variance of the feature variable is the same for both source types. The a priori probabilities $P_S(1)$ and $P_S(2)$ are assumed to be known, but not necessarily equal.

Solution: A classifier can always be designed in five distinct steps, that will be discussed in detail in following sub-sections:

1. Choose decision criterion.
2. Formulate discriminant functions.
3. Simplify the discriminant functions, if possible.
4. Formulate the decision function.
5. Calculate expected performance.

3.5.1 Decision Criterion

To obtain minimum error probability, when the a priori source probabilities are not equal, we must use the *MAP rule* as defined in Eq. (3.15).

3.5.2 Discriminant Functions

In this special example the conditional density functions of the feature variable can be explicitly written as

$$f_{X|S}(x \mid i) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu_i)^2}{2\sigma^2}}, \quad i \in \{1, 2\} \quad (3.27)$$

To implement the MAP decision rule of Eq. (3.15) we begin by defining two preliminary discriminant functions directly according to the general prescription in Eq. (3.18):

$$\begin{aligned} g_i'(x) &= \ln f_{X|S}(x \mid i) P_S(i) \\ &= \ln \frac{1}{\sqrt{2\pi}\sigma} - \frac{(x-\mu_i)^2}{2\sigma^2} + \ln P_S(i), \quad i \in \{1,2\} \end{aligned} \tag{3.28}$$

3.5.3 Simplify Discriminant Functions

We can immediately omit the first constant term in the two preliminary discriminant functions and get the same decisions. Define new discriminant functions as

$$\begin{cases} g_1''(x) = -\frac{1}{2\sigma^2}(x^2 - 2x\mu_1 + \mu_1^2) + \ln P_S(1) \\ g_2''(x) = -\frac{1}{2\sigma^2}(x^2 - 2x\mu_2 + \mu_2^2) + \ln P_S(2) \end{cases} \tag{3.29}$$

Furthermore, the x^2 term occurs identically in both discriminant functions and can therefore be omitted. We also multiply both discriminant functions by the positive scale factor σ^2 and redefine them again as

$$\begin{cases} g_1(x) = x\mu_1 - \mu_1^2/2 + \sigma^2 \ln P_S(1) \\ g_2(x) = x\mu_2 - \mu_2^2/2 + \sigma^2 \ln P_S(2) \end{cases} \tag{3.30}$$

Now we see that the discriminant functions involve only linear operations on x. As there are only two categories, we can simplify the design even further by defining a single discriminant function

$$\begin{aligned} g(x) &= g_1(x) - g_2(x) \\ &= (\mu_1 - \mu_2) \left(x - \underbrace{\left(\frac{\mu_1 + \mu_2}{2} - \frac{\sigma^2}{\mu_1 - \mu_2} \ln \frac{P_S(1)}{P_S(2)} \right)}_{x_{th}} \right) \end{aligned} \tag{3.31}$$

This single discriminant function is *linear* and will obviously change sign at the threshold value $x = x_{th}$.

3.5.4 Decision Function

The classifier can now be implemented with a simple threshold decision

$$d(x) = \begin{cases} 1, & g(x) > 0 \\ 2, & \text{otherwise} \end{cases} \tag{3.32}$$

Optimal decisions can obviously be obtained simply by comparing the observed value of x with a predetermined threshold x_{th}. The direction of the decision depends on the sign of the scale factor $(\mu_1 - \mu_2)$:

$$d(x) = \begin{cases} 1, & \operatorname{sgn}(x - x_{th}) = \operatorname{sgn}(\mu_1 - \mu_2) \\ 2, & \text{otherwise} \end{cases} \qquad \text{where } x_{th} = \frac{\mu_1 + \mu_2}{2} - \frac{\sigma^2}{\mu_1 - \mu_2} \ln \frac{P_S(1)}{P_S(2)} \tag{3.33}$$

This example shows that the optimal MAP decision rule can lead to extremely simple classifier design. If the a priori probabilities are equal, i.e. $P_S(1) = P_S(2) = 0.5$, the threshold is simply the midpoint between the two means. This seems intuitively correct for symmetry reasons. If $P_S(1) > P_S(2)$, the threshold is offset to favor decisions $d(x) = 1$.

3.5.5 Classifier Performance

In this simple example the decision was based on the scalar output value $y = g(x)$ produced by the single discriminant function $g(\ )$ defined in Eq. (3.31). The classifier always guesses that the source category was $S = 1$ whenever the discriminant function outputs a value $y = g(x) > 0$. This event leads to a correct decision, if the true source category was actually $S = 1$, but this outcome can also happen, although less often, when the true source was $S = 2$, and then the decision is incorrect.

To calculate the probability of correct classifications, we must regard any observed value of the decision variable y as an outcome of a random variable defined as $Y = g(X)$. This variable is also Gaussian, because it is just a linearly transformed version of the Gaussian random variable X. Knowing the distribution it is easy to calculate any desired probability values. For example, we might want to calculate *conditional* probabilites of correct classifications, given the source category:

$$P[\mathit{correct} \mid S = 1] = P[Y > 0 \mid S = 1]$$
$$P[\mathit{correct} \mid S = 2] = P[Y < 0 \mid S = 2]$$

The most interesting measure of classifier performance is of course the *total average probability of correct decisions*, for any source category:

$$P[\mathit{correct}] = P[Y > 0 \mid S = 1]\, P[S = 1] + P[Y < 0 \mid S = 2]\, P[S = 2] \tag{3.34}$$

To calculate this probability, we can temporarily assume that $\mu_1 > \mu_2$,

without loss of generality. Then

$$\begin{aligned} P\left[correct\right] &= P\left[X > x_{th} \mid S = 1\right] P_S(1)) + P\left[X \leq x_{th} \mid S = 2\right] P_S(2)) \\ &= \left(1 - \Phi\left(\frac{x_{th} - \mu_1}{\sigma}\right)\right) P_S(1) + \Phi\left(\frac{x_{th} - \mu_2}{\sigma}\right) P_S(2) \\ &= \Phi\left(\frac{\mu_1 - x_{th}}{\sigma}\right) P_S(1) + \Phi\left(\frac{x_{th} - \mu_2}{\sigma}\right) P_S(2) \end{aligned} \tag{3.35}$$

where $\Phi(\ )$ is the normalized Gaussian cumulative distribution function. In the special case with $P_S(1) = P_S(2) = 0.5$, this can be simplified as

$$P\left[correct\right] = \Phi\left(\frac{\mu_1 - \mu_2}{2\sigma}\right) \tag{3.36}$$

Now, if the mean values were actually in the reverse order, $\mu_1 < \mu_2$, the decisions would go in the opposite direction and we would just have to exchange μ_1 and μ_2 in the formulae.

3.6 Example: Independent Gaussian Features

In this section we assume that the classifier observes a long sequence of feature variables, not only one scalar variable. Sometimes it is justifiable to assume that all the features are *statistically independent* of each other. This example shows that classifier design becomes very simple in this case.

Problem: Let us design a two-category classifier to guess the source type with minimum error probability, by observing a sequence of K feature variables $\boldsymbol{X} = (X_1, \ldots, X_K)^T$. We have reasons to assume that X_k has conditional mean μ_{ik} and variance σ^2, given that the source category is $S = i$, either 1 or 2. The variance is caused only by white additive external noise which is independent of the source category. Therefore, it it is safe to assume that the variance of the feature variable is the same for both source types, and that all feature elements are statistically independent of each other. The a priori probabilities are assumed to be equal, $P_S(1) = P_S(2) = 1/2$.

Solution: We design the classifier in the usual five distinct steps, discussed in detail in the following sub-sections.

3.6.1 Decision Criterion

To obtain minimum error probability, when the a priori probabilities are equal, we use the *ML rule* as defined in Eq. (3.16).

3.6.2 Discriminant Functions

In this special example the conditional density functions of the k-th feature variable can be explicitly written as

$$f_{X_k|S}(x_k \mid i) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x_k-\mu_{ik})^2}{2\sigma^2}}, \quad i \in \{1,2\}, \quad k \in \{1,\ldots,K\} \tag{3.37}$$

Because of the statistical independence, by definition, the probability density for the complete observed feature vector is then

$$f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) = f_{X_1|S}(x_1 \mid i) \cdot f_{X_2|S}(x_2 \mid i) \cdots f_{X_K|S}(x_K \mid i) \tag{3.38}$$

To implement the ML decision rule of Eq. (3.16) we begin by defining two preliminary discriminant functions directly according to the general prescription in Eq. (3.18):

$$\begin{aligned} g_i'(\boldsymbol{x}) &= \ln f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) \\ &= K \ln \frac{1}{\sqrt{2\pi}\sigma} - \sum_{k=1}^{K} \frac{(x_k-\mu_{ik})^2}{2\sigma^2}, \quad i \in \{1,2\} \end{aligned} \tag{3.39}$$

3.6.3 Simplify Discriminant Functions

In a similar way as with the scalar feature in the previous example, we can immediately omit the first constant term in the two preliminary discriminant functions and get the same decisions. Define new discriminant functions as

$$g_i''(\boldsymbol{x}) = -\frac{1}{2\sigma^2} \sum_{k=1}^{K} (x_k^2 - 2x_k\mu_{ik} + \mu_{ik}^2) \tag{3.40}$$

The x_k^2 terms occur identically in both discriminant functions and can therefore be omitted. We also multiply both discriminant functions by the positive scale factor σ^2 and redefine them again as

$$g_i(\boldsymbol{x}) = \sum_{k=1}^{K} x_k\mu_{ik} - \mu_{ik}^2/2 \tag{3.41}$$

Now we see that the discriminant functions involve only a linear combination of feature-vector elements x_k. As there are only two categories, we simplify the design by defining a single discriminant function that transforms the feature vector into a single scalar decision variable

$$\begin{aligned} y = g(\boldsymbol{x}) &= g_1(\boldsymbol{x}) - g_2(\boldsymbol{x}) = \\ &= \sum_{k=1}^{K} (\mu_{1k} - \mu_{2k}) \left(x_k - \frac{\mu_{1k} + \mu_{2k}}{2} \right) \end{aligned} \tag{3.42}$$

3.6.4 Decision Function

The classifier can now be implemented with a simple threshold decision, using the scalar decision variable $y = g(\boldsymbol{x})$:

$$d(x) = \begin{cases} 1, & y = g(\boldsymbol{x}) > 0 \\ 2, & \text{otherwise} \end{cases} \tag{3.43}$$

3.6.5 Classifier Performance

To calculate the probability of correct classifications, we must regard any observed value of the decision variable y as an outcome of a scalar random variable Y defined as

$$Y = g(\boldsymbol{X}) = \sum_{k=1}^{K} (\mu_{1k} - \mu_{2k}) \left(X_k - \frac{\mu_{1k} + \mu_{2k}}{2} \right) \tag{3.44}$$

The random variable Y is a linear combination of the Gaussian random variables X_k, and is therefore also Gaussian. The conditional distribution of Y is then completely determined by conditional means $\mu_{Y,i}$ and a variance σ_Y^2 that must be determined:

$$\begin{aligned} E[Y \mid S = 1] &= \sum_{k=1}^{K} (\mu_{1k} - \mu_{2k}) \left(E[X_k \mid S = 1] - \frac{1}{2}(\mu_{1k} + \mu_{2k}) \right) = \\ &= \sum_{k=1}^{K} (\mu_{1k} - \mu_{2k})^2 / 2 = \\ &= \|\boldsymbol{d}\|^2 / 2 \\ E[Y \mid S = 2] &= -E[Y \mid S = 1] = -\|\boldsymbol{d}\|^2 / 2 \\ \text{var}[Y] &= \sum_{k=1}^{K} (\mu_{1k} - \mu_{2k})^2 \sigma^2 = \\ &= \|\boldsymbol{d}\|^2 \sigma^2 \end{aligned} \tag{3.45}$$

Here we have introduced a vector notation for the difference between means for the feature vector $\boldsymbol{X}$ as

$$\boldsymbol{d} = \begin{pmatrix} \mu_{11} - \mu_{21} \\ \vdots \\ \mu_{1K} - \mu_{2K} \end{pmatrix} \tag{3.46}$$

The calculation of classifier performance becomes slightly easier if we trans-

form the decision variable again into a new variable with variance 1, as

$$\begin{aligned} Z &= Y / \|\boldsymbol{d}\| \sigma \\ \mu_{Z,1} &= E[Z \mid S=1] = \|\boldsymbol{d}\| / 2\sigma \\ \mu_{Z,2} &= E[Z \mid S=2] = -\mu_{Z,1} \\ \text{var}[Z] &= 1 \end{aligned} \tag{3.47}$$

Then we can easily calculate

$$\begin{aligned} P[\mathit{correct}] &= P[Y>0 \mid S=1] P_S(1) + P[Y \le 0 \mid S=2] P_S(2) = \\ &= P[Z>0 \mid S=1]/2 + P[Z \le 0 \mid S=2]/2 = \\ &= \Phi(\|\boldsymbol{d}\|/2\sigma) \end{aligned} \tag{3.48}$$

3.7 General Gaussian Feature Vector

Similar methods as in the example of the previous section can be used in the general situation where the decision is based on a feature vector $\boldsymbol{X}$ with K elements

$$\boldsymbol{X} = \begin{pmatrix} X_1 \\ \vdots \\ X_K \end{pmatrix} \tag{3.49}$$

Let us assume that the feature vector has a multivariate Gaussian density (see Råde and Westergren, 1995, section 17.2) with exactly known conditional mean vectors and covariance matrices

$$\begin{aligned} \boldsymbol{\mu}_i &= E[\boldsymbol{X} \mid S=i] \\ \boldsymbol{C}_i &= \text{cov}[\boldsymbol{X} \mid S=i] = E\left[(\boldsymbol{X}-\boldsymbol{\mu}_i)(\boldsymbol{X}-\boldsymbol{\mu}_i)^T \;\middle|\; S=i\right] \end{aligned} \tag{3.50}$$

The conditional density functions of the feature vector can then be written explicitly as

$$f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) = \frac{1}{(2\pi)^{K/2}\sqrt{\det \boldsymbol{C}_i}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i)^T \boldsymbol{C}_i^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_i)} \tag{3.51}$$

If the classifier is designed to guess the source category with minimum error probability, we must use the MAP decision rule. We define a preliminary set of discriminant functions

$$\begin{aligned} g_i(\boldsymbol{x}) &= \ln f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) P_S(i) = \\ &= -\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i)^T \boldsymbol{C}_i^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_i) - \frac{K}{2}\ln 2\pi - \frac{1}{2}\ln\det \boldsymbol{C}_i + \ln P_S(i) \end{aligned} \tag{3.52}$$

Here the first quadratic term is usually called the *Mahalanobis distance* from $\boldsymbol{x}$ to $\boldsymbol{\mu}_i$. There is only one common term which can be immediately omitted to simplify these general discriminant functions. However, in many practical situations the discriminant functions can be greatly simplified, as discussed in the following subsections.

3.7.1 Special case: Equal diagonal covariance matrices

If the variability of the elements in the feature vector $\boldsymbol{X}$ is caused only by external additive noise, and not by the source, the covariance does not depend on the source category. Then all covariance matrices are identical. If, furthermore, the random variations in different features are uncorrelated and have equal variance σ^2, all covariance matrices are equal and very simple:

$$\boldsymbol{C}_i = \boldsymbol{C} = \begin{pmatrix} \sigma^2 & 0 & \cdots & 0 \\ 0 & \sigma^2 & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & \sigma^2 \end{pmatrix} = \sigma^2 \boldsymbol{I}, \quad \text{all } i \tag{3.53}$$

Here $\boldsymbol{I}$ denotes the K-by-K identity matrix. The inverse covariance matrix is then

$$\boldsymbol{C}^{-1} = \frac{1}{\sigma^2}\boldsymbol{I} \tag{3.54}$$

Omitting identical terms from the general discriminant functions in Eq. (3.52) we redefine them as

$$\begin{aligned} g_i'(\boldsymbol{x}) &= -\frac{(\boldsymbol{x}-\boldsymbol{\mu}_i)^T(\boldsymbol{x}-\boldsymbol{\mu}_i)}{2\sigma^2} + \ln P_S(i) = \\ &= -\frac{\|\boldsymbol{x}-\boldsymbol{\mu}_i\|^2}{2\sigma^2} + \ln P_S(i) = \\ &= -\frac{\boldsymbol{x}^T\boldsymbol{x} - 2\boldsymbol{\mu}_i^T\boldsymbol{x} + \boldsymbol{\mu}_i^T\boldsymbol{\mu}_i}{2\sigma^2} + \ln P_S(i) \end{aligned} \tag{3.55}$$

These discriminant functions are quadratic in $\boldsymbol{x}$, as they calculate a Euclidean distance in K-dimensional feature space. If all $P_S(i)$ are equal, the classifier should always select the class with the mean nearest to the observed feature vector. Therefore, the classifier is sometimes called a *minimum-distance* classifier.

However, it is not actually necessary to compute Euclidean distances in this case. The quadratic term $\boldsymbol{x}^T\boldsymbol{x}$ is the same in all discriminant functions and can therefore be omitted. We can also rescale the discriminant functions. Thus, we design an optimal classifier with only the following simple *linear discriminant functions*:

$$g_i(\boldsymbol{x}) = \boldsymbol{\mu}_i^T\boldsymbol{x} - \boldsymbol{\mu}_i^T\boldsymbol{\mu}_i/2 + \sigma^2 \ln P_S(i) \tag{3.56}$$

3.7.2 Special case: Equal covariance matrices

Using a similar method as in the previous section it is rather easy to show that *linear discriminant functions* are always sufficient for optimal classification, if all the covariance matrices are equal, $\boldsymbol{C}_i = \boldsymbol{C}$, for all source categories i. Different feature elements need not be uncorrelated and have

identical variances, as in the previous special case. The proof is left as an exercise for the reader.

If all the variability of the feature vector is caused by *additive* Gaussian noise, independent of the source category, the covariance matrices must be equal for all source types. Thus, the linear discriminant functions are indeed theoretically optimal, if the added noise is Gaussian, white or non-white. Linear classifiers can be optimal also in many other cases where the features do not have a Gaussian distribution. This is exemplified in some of the exercise problems.

3.8 Discrete Feature Distribution

Until now it was assumed that the observable features are continuous-valued, i.e., they can have any real value from $-\infty$ to ∞. This is realistic whenever the features are derived from measurements in the physical world, for example by analysing a sound received by a microphone.

However, in some applications the features can have only discrete values. A simple example was discussed in chapter 2, where the observed feature was just the outcome of a tossed coin, which obviously has only two possible values. A more realistic example is the electrical activity in a nerve cell. This activity consists of discrete pulses which all have the same amplitude. If the activity is recorded during a fixed time interval, the observed feature is the total number of observed pulses, which can only have an integer value.

In such cases the elements of the feature vector $\boldsymbol{X}$ can only take a value from an enumerable set $\{\boldsymbol{x}_k,\ k = 1 \ldots N_x\}$, and we might simply use the integer index k to denote which outcome was observed. Then the probability distributions for $\boldsymbol{X}$ cannot be defined by continuous density functions $f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j)$. Instead we use discrete probability-mass functions $P_{\boldsymbol{X}|S}(\boldsymbol{x}_k \mid j)$ to describe the statistical behaviour of the features.

In principle, the features are discrete also whenever a physical signal is digitized by an A/D converter before it is analyzed by the classifier. If the A/D conversion has a high resolution, the mathematical models for continuous-valued features can still be used with good accuracy. However, when the features only can assume very few discrete values it is necessary to use a discrete distribution explicitly.

The good news is that the classification methods and decision rules described in previous sections are still valid and can be used with only a small notational change. We just have to use the discrete distributions instead of the continuous density functions. The fundamental formula is still *Bayes*

rule in the following form:

$$P_{S|\boldsymbol{X}}(j \mid \boldsymbol{x}_k) = \frac{P_{\boldsymbol{X}|S}(\boldsymbol{x}_k \mid j) P_S(j)}{P_{\boldsymbol{X}}(\boldsymbol{x}_k)} \qquad \text{where } P_{\boldsymbol{X}}(\boldsymbol{x}_k) = \sum_{j=1}^{N_s} P_{\boldsymbol{X}|S}(\boldsymbol{x}_k \mid j) P_S(j) \tag{3.57}$$

This looks almost the same as before. Therefore, all the decision rules derived from Bayes rule can be used exactly as before. We just have to remember that the feature vector is now discrete. For example, all probability calculations in feature space, as discussed in section 3.4, are now performed as sums instead of integrals. When the classifier is designed to guess the category of the source, the probability of a correct decision is now expressed as

$$P\left[\mathit{correct}\right] = \sum_{i=1}^{N_s} P\left[\boldsymbol{X} \in \Omega_i \cap S = i\right] = \sum_{i=1}^{N_s} P_S(i) \sum_{\boldsymbol{x} \in \Omega_i} P_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) \tag{3.58}$$

because the decision regions Ω_i now consist of enumerable sets of discrete points in feature space.

Summary

This chapter introduced the *general Bayesian classifier*. Main points of the section:

Source category: A signal source has a random category $S \in \{1, \ldots, N_s\}$. A priori probabilities $P_S(j)$ are assumed to be known exactly.

Decision: The classifier selects one action with index number $d \in \{1, \ldots, N_d\}$.

Observation: A random feature vector $\boldsymbol{X}$ with known conditional probability density functions $f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j)$ for all source categories j.

Loss Matrix: Matrix with elements L_{ij} defining the loss caused by decision i, when the source type is j.

Decision Function: A deterministic function $d(\boldsymbol{x}) \in \{1, \ldots, N_d\}$, specifying the decision for any actual observation $\boldsymbol{x}$.

Decision Region: A set of $\boldsymbol{x}$-values causing the same decision.

Discriminant Functions: A set of functions $g_i(\boldsymbol{x})$, $i \in \{1, \ldots, N_d\}$, such that the classifier can decide

$$d(\boldsymbol{x}) = \underset{i}{\operatorname{argmax}}\, g_i(\boldsymbol{x})$$

Bayes Minimum-risk Decision Rule:

$$d(\boldsymbol{x}) = \underset{i}{\operatorname{argmin}}\, R(i \mid \boldsymbol{x}), \quad R(i \mid \boldsymbol{x}) = \sum_{j=1}^{N_s} L_{ij} P_{S|\boldsymbol{X}}(j \mid \boldsymbol{x})$$

Special cases of the Minimum-risk Decision Rule:

$$\text{MAP rule:} \quad d(\boldsymbol{x}) = \underset{i}{\operatorname{argmax}}\, P_{S|\boldsymbol{X}}(i \mid \boldsymbol{x})$$

$$\text{ML rule:} \quad d(\boldsymbol{x}) = \underset{i}{\operatorname{argmax}}\, f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i)$$

Remaining Questions

We have still not answered the following important question:
How do we find the conditional probability density functions $f_{\boldsymbol{X}|S}(\ )$ from training data, if they are not known in advance?

Problems

3.1 You observe a set of two five-digit sequences $\boldsymbol{x} = (12345; 24153)$ and somebody tells you that one sequence was produced by a random-number generator and the other was written down by a human person. Which one was most probably generated by the human being?

We have two possibilities for the "state" S of the "signal source": if $S = 1$ the order of generators is *human*; *computer*, and if $S = 2$ the order is the reverse. Both alternatives have equal a priori probability. Suggest reasonable estimates of the conditional probabilities $P_{S|\boldsymbol{X}}(1 \mid \boldsymbol{x})$ and $P_{S|\boldsymbol{X}}(2 \mid \boldsymbol{x})$, given the particular observation $\boldsymbol{x}$. What is the optimal MAP decision about the source, using your probability estimates?

3.2 A signal generator randomly selects a state S which can be either of two signal types with equal probabilities $P_S(1) = P_S(2) = 0.5$. A two-category signal classifier observes a scalar feature variable X which is $N(\mu_i, \sigma)$ whenever the signal category is $S = i$.

3.2.a Derive a decision rule $d(x)$ as a simple threshold for x, giving minimum error probability.

3.2.b Calculate the minimum probability of error for $\mu_1 = 2$, $\mu_2 = -2$, and $\sigma = 1$.

3.3 A signal source randomly selects a category S from the set $\{1 \dots N\}$. All prior probabilities $P_S(j)$ and conditional probability density functions $f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j)$ for the feature vector $\boldsymbol{X}$ are known for all $j = 1 \dots N$. A classifier is designed to primarily determine the category of the source, i.e., $d(\boldsymbol{x}) \in \{1 \dots N\}$, on the basis of an observed feature vector $\boldsymbol{x}$. In addition, a "reject" decision "cannot decide" is allowed, which is coded as $d(\boldsymbol{x}) = N+1$. This rejection alternative is chosen whenever the observation does not give a sufficiently clear indication about the most probable signal category. The cost for any erroneous decision is c and the cost for a rejection is r, i.e., the cost function is

$$L(d(\boldsymbol{x}) = i \mid S = j) = \begin{cases} 0, & i = j = 1 \dots N \\ r, & i = N+1,\ j = 1 \dots N \\ c, & \text{otherwise} \end{cases}$$

3.3.a Show that minimum expected cost is achieved if the classifier decides

$$d(\boldsymbol{x}) = \begin{cases} i = \operatorname{argmax}_j P_{S|\boldsymbol{X}}(j \mid \boldsymbol{x}), & \text{if } P_{S|\boldsymbol{X}}(i \mid \boldsymbol{x}) \geq 1 - r/c \\ N+1, & \text{otherwise} \end{cases}$$

3.3.b What happens if $r = 0$ or $c \to \infty$?

3.3.c What happens if $r > c$?

3.3.d Show that the following discriminant functions give minimum expected cost:

$$g_i(\boldsymbol{x}) = f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) P_S(i), \qquad i = 1 \ldots N$$
$$g_{N+1}(\boldsymbol{x}) = \left(1 - \frac{r}{c}\right) \sum_{j=1}^{N} g_j(\boldsymbol{x})$$

3.3.e Sketch these discriminant functions for the case $N = 2$, equal a priori probabilities and a scalar feature X, which is Gaussian $N(1, 1)$ when the signal category is $S = 1$ and $N(-1, 1)$ when the signal category is $S = 2$, and setting $r = 1$ and $c = 4$.

3.3.f What is the total probability of "reject" decisions in the situation defined just above? For which range of cost relations r/c will the "reject" decision never be chosen?

3.4 A two-category pattern classifier is designed to make optimal decisions on the basis of a two-element random feature vector

$$\boldsymbol{X} = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix}$$

The distribution of the feature vector is determined by the random category S of the signal source, which can be either 0 or 1. The features are contaminated by additive noise: $\boldsymbol{X} = \boldsymbol{a}_i + \boldsymbol{W}$, with known

$$\boldsymbol{a} = \begin{pmatrix} a_{i1} \\ a_{i2} \end{pmatrix}$$

for each category i. The added noise vector

$$\boldsymbol{W} = \begin{pmatrix} W_1 \\ W_2 \end{pmatrix}$$

is zero-mean and Gaussian and independent of the signal category. The variances of the noise components and the correlation between them are

$$\begin{aligned} \text{var}\,[W_1] &= \sigma_1^2 \\ \text{var}\,[W_2] &= \sigma_2^2 \\ \text{cov}\,[W_1, W_2] &= E(W_1 W_2) = \rho \sigma_1 \sigma_2 \end{aligned}$$

where ρ is a correlation coefficient between 0 and 1.

3.4.a Express the conditional probability density functions $f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i)$ in matrix form.

3.4.b Sketch the two probability density functions in the form of iso-density curves in an x_1, x_2 diagram, using parameter values $\sigma_1 = \sigma_2 = 10$, and $\rho = 0.5$, and

$$\boldsymbol{a}_1 = \begin{pmatrix} 5 \\ 0 \end{pmatrix}; \quad \boldsymbol{a}_2 = \begin{pmatrix} 0 \\ 5 \end{pmatrix}$$

3.4.c Show that a linear discriminant function $Y = g(\boldsymbol{X}) = \boldsymbol{q}^T \boldsymbol{X}$ gives a scalar decision variable Y which allows optimal MAP classification. Choose the vector $\boldsymbol{q}$ for optimal performance.

3.5 A signal source randomly selects a state S out of two categories, 1 or 2, with equal a priori probabilities $P_S(1) = P_S(2) = 1/2$. A two-category classifier uses a K-dimensional feature vector $\boldsymbol{X}$ which has Gaussian conditional density distributions with means depending on the signal category and a constant diagonal covariance matrix with constant equal variances along the diagonal:

$$S = i \quad \Rightarrow \quad \boldsymbol{X} \text{ is } N(\boldsymbol{a}_i, \sigma^2 \boldsymbol{I})$$

3.5.a Define decision rules for minimum probability of error, using a scalar decision variable and a simple threshold mechanism.

3.5.b Show that the minimum probability of error is

$$P_e = 1 - \Phi(d'/2), \text{ where}$$

$$d' = \|\boldsymbol{a}_2 - \boldsymbol{a}_1\| / \sigma, \quad \Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}z^2} dz$$

3.6 (Coin game) Your friend has two coins numbered $S = 0$ and $S = 1$. They appear identical to you, but your friend can tell them apart using a secret mark. However, only coin#0 is fair, i.e., when you toss it, the sides come up with equal probability:

$$P\left[head \mid S = 0\right] = P\left[tail \mid S = 0\right] = 1/2$$

The other coin is heavier on the tail side so that

$$P\left[head \mid S = 1\right] = p > 1/2, \quad P\left[tail \mid S = 1\right] = 1 - p$$

You both know the value of p. Your friend randomly selects one of the coins and tosses it K times. You observe the resulting sequence $\underline{x}$ of heads and tails and must guess which of the two coins was used. The observed sequence contains k heads.

3.6.a Define a discriminant function $g(\underline{x})$ such that you can guess with minimum probability of error which coin was used, using the simple threshold rule:
guess coin#1, if $g(\underline{x}) > 0$, otherwise guess coin#0.

3.6.b Determine your average probability of error for $K = 5$ and $p = 0.6$.

3.7 A signal source randomly selects a state S among categories $\{1 \dots N_s\}$ with known probabilities $P_S(j)$. The generated signal is analyzed by a feature-extraction system producing a vector with only binary-valued features

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_K \end{pmatrix}$$

where every feature x_k is either 0 or 1. The feature extraction is noisy. Therefore, any particular observation $\boldsymbol{x}$ is regarded as an outcome of a random vector $\boldsymbol{X}$, characterized by the known probabilities

$$p_{kj} = P\left[X_k = 1 \mid S = j\right]$$

All feature elements are statistically independent from each other. Determine a set of linear discriminant functions on the form

$$g_j(\boldsymbol{x}) = \boldsymbol{w}_j^T \boldsymbol{x} + w_{0j}$$

such that minimum probability of error is achieved by the usual decision rule

$$d(\boldsymbol{x}) = \underset{j}{\operatorname{argmax}}\, g_j(\boldsymbol{x})$$

3.8 A signal source randomly selects and generates one out of two exactly known signal segments, either $u_0(n)$ or $u_1(n)$, for $n = 0 \dots L-1$. The generated signal is contaminated by additive noise. At time $n = L-1$ a binary classifier has received the finite signal sample sequence

$$\boldsymbol{X} = \boldsymbol{u}_S + \boldsymbol{W}, \quad \boldsymbol{u}_S = \begin{pmatrix} u_S(0) \\ \vdots \\ u_S(L-1) \end{pmatrix}, \quad \boldsymbol{W} = \begin{pmatrix} W_0 \\ \vdots \\ W_{L-1} \end{pmatrix}$$

for a source category $S = j \in \{0, 1\}$. All noise samples are zero-mean and Gaussian, and they may be correlated, but are independent of the source category.

3.8.a Show that a linear discriminant function $Y = g(\boldsymbol{X}) = \boldsymbol{q}^T \boldsymbol{X}$ allows optimal MAP classification using only the scalar variable Y.

3.8.b Determine the optimal weight vector $\boldsymbol{q}$, expressed in terms of the known signal vectors $\boldsymbol{u}_0$ and $\boldsymbol{u}_1$ and the covariance matrix C_W for the vector $\boldsymbol{W}$ containing the noise samples.

3.9 (2I2AFC) The Two-Interval Two-Alternative Forced Choice method (2I2AFC) is a popular psychoacoustic test procedure. A test sound is presented during either the first or the second of two well-defined time intervals. After each pair of intervals, the listener is required to indicate which interval he believes contained the test sound. The response "I don't know" is not allowed. Thus, the listener always has 50% chance of guessing correctly, even when he cannot hear anything.

Psychoacoustic test data are often modelled by assuming that the listener's responses are determined by an internal sensory random variable X which is $N(0,1)$ whenever the test sound is not presented, and $N(d',1)$, if the sound is actually presented. The "detectability index" d' is a measure of the true audibility of the test sound. In a 2I2AFC test the listener's response is determined by a random sensory feature vector $\boldsymbol{X}$ with two elements, one for each of the two presentation intervals:

$$\boldsymbol{X} = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix}$$

The two elements are assumed to be statistically independent and the variance is assumed to be constant =1. The means of the two-dimensional distribution depends on whether the test sound is presented in interval 1 or 2, and these two possible alternatives are modelled as a discrete random variable S. Both presentation types have equal probability. There are two possible conditional mean vectors for the observed vector:

$$E[\boldsymbol{X} \mid S=1] = \begin{pmatrix} d' \\ 0 \end{pmatrix}, \quad E[\boldsymbol{X} \mid S=2] = \begin{pmatrix} 0 \\ d' \end{pmatrix}$$

3.9.a Determine an optimal decision rule to minimize the total error probability.

3.9.b What is the probability of correct response as a function of d', assuming that the listener's brain always uses the optimal decision rule?

3.10 (3I3AFC) The Three-Interval Three-Alternative Forced Choice psychoacoustic test procedure is a natural extension from the 2I2AFC procedure. In this procedure there are three time intervals in each trial, and exactly one of the intervals contains the test signal to be detected. The listener's response is based on a three-element sensory feature vector

$$\boldsymbol{X} = \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} \tag{3.59}$$

with one element for each of the three presentation intervals. Other assumptions in the model are exactly the same as in problem 3.9.

3.10.a The 3I3AFC data analysis assumes the listener has adopted a decision rule to guess "Interval j" whenever the feature variable X_j happens to be greater than the the other two feature variables. Show that this is indeed the optimal decision rule to minimize the error probability.

3.10.b What is the probability of correct response as a function of d', assuming that the listener's brain always uses the optimal decision rule. Hint: There is no simple closed-form solution. The result must be given as an integral. Suppose the test sound was actually presented in Interval 1 and that $X_1 = x$. Calculate the condtional probability of correct response, given this situation, i.e.

$$P\left[\mathit{correct} \mid (X_1 = x) \cap (S_1 = 1)\right] = P\left[(X_2 < x) \cap (X_3 < x) \mid S = 1\right]$$

Then calculate the average over all possible values of x.

3.10.c Generalize the result to determine the probability of correct response as a function of d' in an M-interval-M-alternative Forced-Choice test procedure.

3.11 (Noise intensity discrimination) A signal source randomly selects a state $S = 1$ or $S = 2$ with equal probability. In both states the signal source generates stationary *Gaussian zero-mean noise* $X(n)$ with *uncorrelated* elements. In state 1 the variance of every noise element is known to be σ_1^2 and in state 2 the variance is σ_2^2.

A two-category classifier receives and stores L consecutive noise samples $(X(0) \ldots X(L-1))$ and then guesses the state of the signal source. Design this classifier for minimum error probability, assuming that σ_1 and σ_2 are exactly known.

Hint: Design a (possibly non-linear) discriminant function to produce a single decision variable Y, such that optimal classification can be obtained by a simple threshold mechanism with this feature variable as input.

3.12 (2I2AFC intensity discrimination) A signal source randomly selects a source state $S = 1$ or $S = 2$ with equal probability. The source state controls the choice of signals presented in two well-defined time intervals with equal duration. The signals in both intervals are stationary *Gaussian zero-mean white noises*, but the noise variance is different in the two intervals. The noise variance can be either σ_l^2 or $\sigma_h^2 = \alpha\sigma_l^2$, with $\alpha > 1$. If the state was randomly chosen as $S = 1$, the noise with the higher variance is presented in the first time interval, and if $S = 2$, the higher variance is in the

second interval. A two-category classifier receives and stores $2L$ consecutive noise samples, with L samples from each of the two time intervals, and then guesses the state of the signal source, i.e. whether the higher intensity was presented in the first or second interval. Thus the observed feature vector is

$$\begin{aligned} \boldsymbol{X} &= (X_1 \dots X_L, X_{L+1} \dots X_{2L}), \text{ where} \\ E[X_k] &= 0 \\ \text{var}[X_k] &= \begin{cases} \sigma_h^2, & 1 \le k \le L \\ \sigma_l^2, & L+1 \le k \le 2L \end{cases}, \quad \text{if } S = 1 \\ & \text{vice versa, if } S = 2 \end{aligned}$$

3.12.a Design this classifier for minimum error probability, using a single (possibly non-linear) discriminant function to produce a single decision variable Y, such that optimal classification can be obtained by a simple threshold mechanism with this variable as input.

3.12.b Show that the decision threshold can be selected optimally without knowing σ_l^2 or σ_h^2.

3.12.c Show that the variance of the decision variable, σ_Y^2, is independent of the source state, i.e. the order of presentations.

3.13 (Error of doing hard decisions too early) The input data to a classifier is a sequence $\underline{X} = (X_1, \dots, X_T)$ with real-valued scalar data samples. You know that the data has been generated from one of two possible sources, either $S = 0$ or $S = 1$. The choice of source is random, with equal probability for both source categories. The sources have slightly different statistical characteristics. For any t we know that X_t is Gaussian with

$$\begin{aligned} E[X_t] &= \begin{cases} -1, & S = 0 \\ +1, & S = 1 \end{cases} \\ \text{var}[X_t] &= 4, \quad \text{for both source categories} \end{aligned}$$

All samples in the observed sequence come from the same source S, and all samples in the sequence are statistically independent of each other. You will now compare the performance of two ways to design a classifier to guess the source category. The observed sequence length is $T = 9$ samples. (Note: complete solution given in the Answers section.)

3.13.a What is the smallest possible probablity of decision error for the *ideal* classifier in this case?

3.13.b Your amateur friend (who did not take the Pattern Recognition course) interprets each sample X_t as a discrete "vote" for $S = 1$ if $X_t > 0$, and a vote for $S = 0$ otherwise. He has designed a classifier that simply counts Z = the number of observed samples where $X_t > 0$, and then guesses that $S = 0$, iff $z < T/2$; otherwise the classifier guesses that $S = 1$. What is the error probability for this classifier?

Chapter 4

Classification in Practical Applications

Bayes-optimal classification, as presented in the previous chapter, requires perfect knowledge of the statistical properties of the different classes. In other words, the class probabilities $P_S(j)$ and class-conditional feature distributions $f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j)$ or $P_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j)$ have to be known without error. This is typically not the case in real-world applications. Instead, these distributions have to be approximated somehow. It is up to us to create these approximations, based on what we know about the problem and about classification in general.

In a nutshell, the standard approach to practical classification has a number of key steps, and the purpose of this chapter is to discuss these:

1. Define the problem by deciding on a specific quantitative performance measure to be optimized, e.g., misclassification rate.
2. Gather relevant data. (Covered in Section 4.1.)
3. Design a feature extractor. (Section 4.2.)
4. Propose an approximate probabilistic description of the features, a *model*. (Section 4.3.)
5. Use sample data from the problem to "fill in the blanks" in the model, by estimating the model parameters. (Section 4.4.)
6. Estimate the performance of the resulting classifier. (Section 4.5.)
7. Go back and tweak the design of the classifier, until its performance is good enough. (Sections 4.6 through 4.8 discuss some common problems.)
8. Apply the final solution to the original problem.

Generally speaking, designing a good classifier involves equal parts art and science, but experience and common sense can help a lot. While not always an easy task, the design process is often a fun challenge in creativity and engineering. The course project in Appendix A is intended to provide hands-on experience of classifier design. The short paper by Domingos (2012) provides an excellent overview of the issues in applied machine learning, essential to any practitioner.

4.1 Data Collection

Classification is the task of predicting classes from input data attributes. To perform classification, we thus need to gather data. Specifically, we require a set of examples from the different classes and their associated attributes. We will write $\widetilde{\mathcal{D}}$ to denote a set[1] of N pairs of class labels s and corresponding raw input attribute vectors $\widetilde{\boldsymbol{x}}$, as in

$$\widetilde{\mathcal{D}} = \{(\widetilde{\boldsymbol{x}}_1, s_1), \ldots, (\widetilde{\boldsymbol{x}}_N, s_N)\}. \tag{4.1}$$

(We only consider labelled examples where all attributes are available. If only a few or none of the examples have labels, we have what is known as a *semi-supervised* or *unsupervised learning* problem, which will not be treated here. The same goes for *censored data*, where some attribute values are missing.)

The data in $\widetilde{\mathcal{D}}$ is used to investigate the properties of the problem we are studying, to subsequently build a suitable classifier, and then to estimate classification performance.

For classification to be possible, there must be some kind of structure to the problem, captured by the data in $\widetilde{\mathcal{D}}$, so that the input attributes are in some way correlated with the class. If the attributes have nothing to do with the class labels, informed classification is impossible. By the same token, $\widetilde{\mathcal{D}}$ needs to be representative of the situation where we want to apply our finished classifier. If the data comes from a different problem than the one we want to solve, classification accuracy may suffer, as discussed in Section 4.8.3.

When collecting data, there is often a trade-off between expensive, high-quality examples, which may only be available in small amounts, and cheap, lower-quality data available in bulk. Usually, data that requires human intervention, such as recording a sound database or annotating text, is time-consuming and expensive to collect. In contrast, data that is generated and processed automatically, or available from the web, can frequently be obtained in large amounts at low cost. The price we pay is that such data may not match our particular problem well, and may contain disturbances

[1] Strictly speaking, $\widetilde{\mathcal{D}}$ is actually a *multiset*, since the same attribute-label pair may appear more than once.

or examples that are irrelevant or even misleading for our purposes. These considerations apply both to which particular raw input attributes we can collect and include as elements in the vector $\widetilde{\boldsymbol{x}}$, as well as the intrinsic quality of the numbers we decide to include, e.g., how much noise or other errors the attributes and labels contain.

Suitability to the application at hand determines what counts as "high-quality" data. If we are designing a speech recognizer to be used in a specific noisy environment, it is often better to train on examples recorded with the exact same noise condition, rather than on clean (noise-free) data, which is less representative of the intended application.

4.2 Feature Extraction

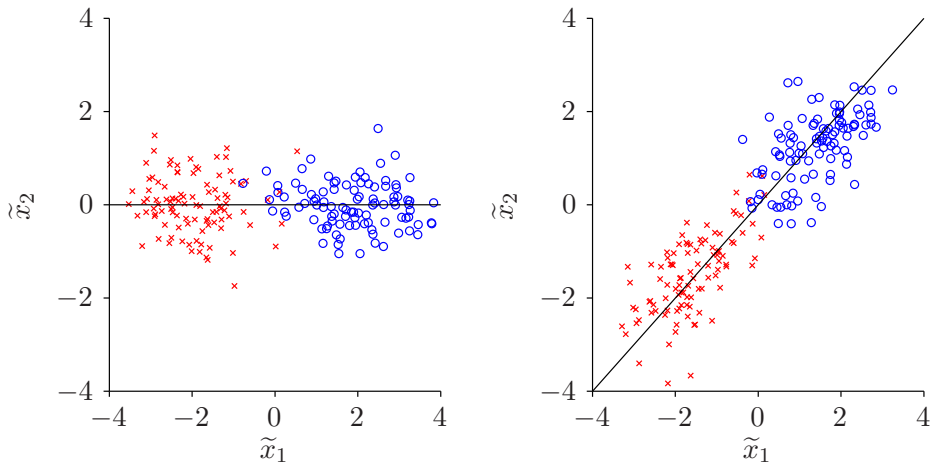
The role of the feature extractor in a classification system is to transform the data to expose the structure of the problem in such a way that the classifier can differentiate between the different classes by looking at the transformed data. In other words, we want to take the raw inputs $\widetilde{\boldsymbol{x}}$ and process them into feature vectors $\boldsymbol{x}$ that are as useful as possible for classification. How to do this depends a lot on the application. Feature engineering, along with data collection, therefore tend to be the most labour-intensive steps when building practical classifiers—once we have processed the input into a usable form, we can simply feed those features into one of the many ready-made machine-learning models available.

It may be helpful to think about feature extraction as a way to take the information in the raw data relevant to classification, and organize and concentrate it. In practice, if the raw data is high-dimensional, this is often accomplished through dimensionality-reduction techniques such as PCA in Eq. (1.24) or its close relative factor analysis; see Figure 4.1.

Independent aspects of the data should preferably be separated into different components of the feature vector. *Fourier analysis*, for instance, is an orthonormal transform often applied in audio classification problems to separate the different frequencies in a sound. This ensures that similar sounds have similar feature vectors, as illustrated in Figure 4.2. PCA and factor analysis are also orthogonal transforms, and have a similar ability to separate linearly independent aspects of the features, but in a data-adaptive manner. While these methods are not guaranteed to return the dimensions of the data that are most useful for classification, they typically provide a good heuristic.

If there is unimportant information in the raw data which does not help classification, it is often best discarded. If, for example, some features are random noise, independent of the class (like dimensions $\widetilde{x}_2$ in Figure 4.1a and $\widetilde{x}_2 - \widetilde{x}_1$ in Figure 4.1b), these should not be included.[2] However, one

[2]Alternatively, if we know how noisy the different features are, there are techniques,



(a) Raw input: simple case **(b)** Raw input: more advanced case

Figure 4.1: Toy examples of dimensionality reduction. In the left scenario, only the input $\widetilde{x}_1$ is informative about the class label, and should be kept by the feature extractor. In the right scenario, the combined feature $x = \widetilde{x}_1 + \widetilde{x}_2$ (oblique axis) contains nearly all information relevant for classification, and suffices to build a good classifier. The essence of PCA is to rotate the coordinate system to find dimensions with a lot of variation, since these often contain most of the information.

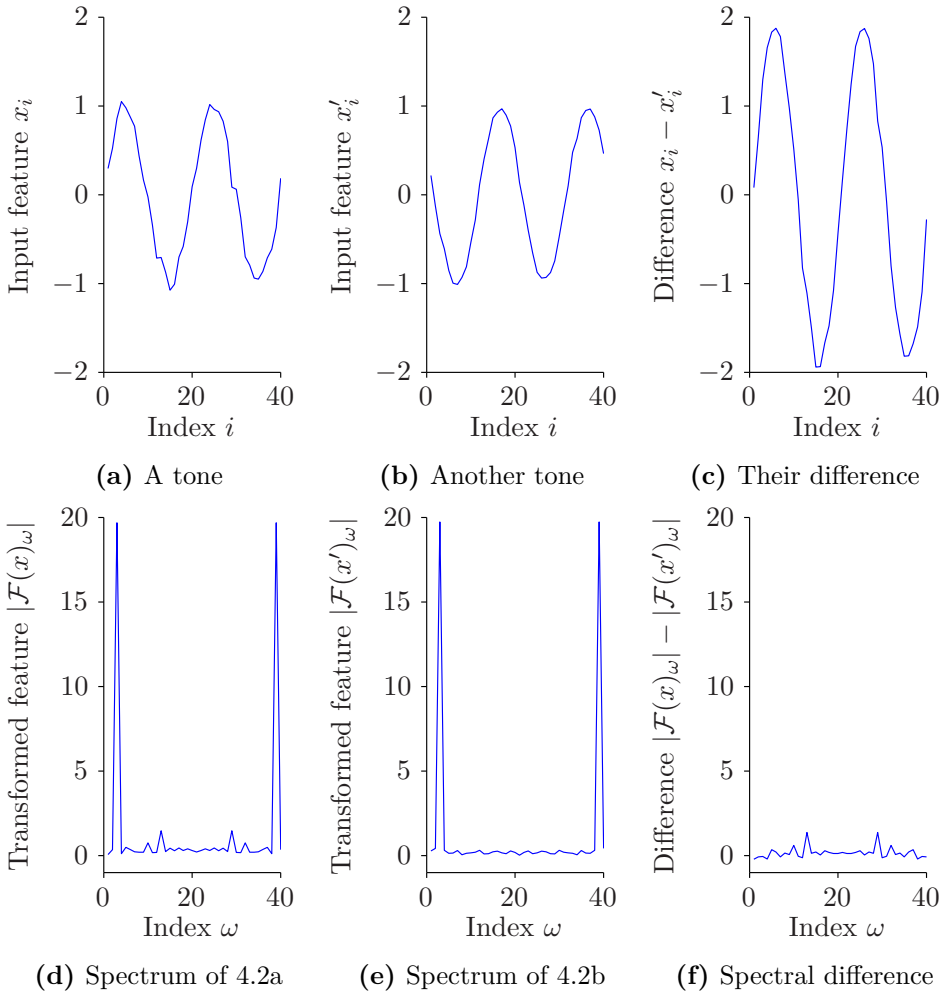
should be careful not to make the feature extractor "too clever:" feature extraction is just a mechanical process to make it easier for the classifier to do its job, and our probabilistic approaches to classification are often quite good at making sense of noisy and uncertain information.

Nature is a great problem solver, and physical or biological considerations can be used to guide feature extractor design in many applications. The standard features used in commercial speech recognition systems (the MFCCs detailed in Section A.2.2), are for instance inspired by the properties of the human hearing system. The idea is to keep only the information in the signal that we humans use for classification.[3]

In a practical situation, it is often straightforward to use what we know about the problem, apply a little bit of thought, and come up with features that extract only the most relevant pieces of information from the raw input. This typically works much better than simply throwing data at the classifier without thinking first.

known as *uncertainty propagation*, to let the classifier take this information into account.

[3]On the other hand, there could be useful clues in the audio that humans are unable to hear. This is one reason for the curious mismatch between the problems that machines are good at solving, and the problems that humans are good at solving, known as *Moravec's paradox*.



(a) A tone **(b)** Another tone **(c)** Their difference

(d) Spectrum of 4.2a **(e)** Spectrum of 4.2b **(f)** Spectral difference

Figure 4.2: Illustration of information extraction and separation. The first row shows sampled tones (top left and middle) in noise with different phase but similar frequency. Though these features are visually similar, the difference $\boldsymbol{x} - \boldsymbol{x}'$ is large (top right). The second row shows another data representation, computed as the absolute value of the discrete Fourier transform. The transformation extracts tonal energy information in the data, leading to features that are nearly indistinguishable, both to the eye and mathematically. The spectral difference (bottom right) is very small. The two largest peaks that remain in the spectral difference correspond to a small high-frequency tone in the first signal, which is hard to see with the naked eye, but shows up more clearly in figures 4.2d and 4.2f. Fourier analysis thus extracts both meaningful similarities and differences between the two example signals here.

4.3 Statistical Modelling

Once we have decided on the features, the next task is to build a *model* of the situation. There are several kinds of models used in classification: *gener-*

ative models, which describe the joint distribution $P[S, \boldsymbol{X}]$ of labels and features, and *discriminative models*, which only describe the label distribution $P[S \mid \boldsymbol{X}]$ given the features. The latter is virtually the same as describing the discriminant function(s) $g_j(\boldsymbol{x})$ directly. There are also non-probabilistic discriminative approaches, which determine discriminant functions based on distances in feature space rather than class probabilities. These are known as *geometric* methods.

Statistical models have an advantage over geometric approaches, in that they also model the uncertainty in the classification, making it easier to estimate the error rate. For many geometric methods, there exists an equivalent probabilistic formulation which produces the same class boundaries. The Mahalanobis distance in equation (3.52) is an example of a distance measure arising from a particular probabilistic model. Additionally, generative models describe the behaviour of the features themselves, which makes the model easier to interpret and provides further insight into the structure of the problem. Generative models can also be used for tasks other than classification, such as generating new sound features in a speech synthesizer. In this book, we focus on generative, probabilistic approaches.

4.3.1 Parametric Models

As seen in Chapter 3, generative models can be broken into a class distribution $P[S]$ and a class-conditional feature distribution $P[\boldsymbol{X} \mid S]$. When building a statistical model, we make the assumption that these generating distributions have a particular mathematical form

$$P[S = j] = \widehat{p}_j \tag{4.2}$$

$$f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j) = \widehat{f}_j(\boldsymbol{x} \mid \boldsymbol{\theta}_j), \tag{4.3}$$

for all $j \in \{1, \ldots, N_s\}$, where $\boldsymbol{\theta}_j$ is a vector of (possibly unknown) parameters. The key aspect of statistical modelling is to choose an appropriate mathematical form for the distributions $\widehat{f}_j$; a *model structure*. It is possible that different classes can have the same expression for $\widehat{f}_j$; the associated feature distributions $\widehat{f}_j(\boldsymbol{x} \mid \boldsymbol{\theta}_j)$ can still be different if their parameter vectors $\boldsymbol{\theta}_j$ do not have the same value.

Typically, we want the specific mathematical expression $\widehat{f}_j$ we propose to be similar to the true distribution $P[\boldsymbol{X} \mid S = j]$ of the class features. To come up with an educated guess that well matches the actual data distribution, we use our prior knowledge about the problem and how we expect that the classes may behave, so-called *domain knowledge*, along with the mathematical tools we have learned. The following example should make this process more concrete.

Example 4.1 (Doping Testing): In sports competitions, athletes are subjected to numerous tests to check for the presence of a forbidden doping

substance. An athlete can either pass or fail each test, but the tests are not always 100% reliable. How can we build a statistical model to well separate clean athletes from cheaters?

Solution:

Let the two classes be *clean* and *cheat*, and let $\boldsymbol{x}$ be a feature vector containing the binary test responses, with elements 0 indicating a pass and 1 indicating a failed test. We can model these tests by Bernoulli random variables (coin flips). For a clean athlete, each test $k \in \{1, \ldots, K\}$ has a certain, preferably low, probability $p_{k|clean}$ of producing a false positive (a 1). Different tests may have different accuracies. For a cheater, test k has a probability $p_{k|cheat}$ of flagging the athlete; hopefully, this is close to one.

Because of the test inaccuracies, the elements of $\boldsymbol{x}$ need not all have the same value. The main question is then how to combine the information from all the tests (the ones and zeroes in $\boldsymbol{x}$), to enable a more accurate decision on whether or not the current athlete is a cheater. As a first guess, we make the simplifying assumption that all tests are conditionally independent, given the doping status of the athlete. This leads to class-conditional feature distributions of the form

$$\widehat{p}_{clean}(\boldsymbol{x} \mid \boldsymbol{p}_{clean}) = \prod_{k=1}^{K} (1 - p_{k|clean})^{1-x_k} p_{k|clean}^{x_k} \tag{4.4}$$

$$\widehat{p}_{cheat}(\boldsymbol{x} \mid \boldsymbol{p}_{cheat}) = \prod_{k=1}^{K} (1 - p_{k|cheat})^{1-x_k} p_{k|cheat}^{x_k}, \tag{4.5}$$

since the probabilities of independent events multiply. We see that the accuracy characteristics $p_{k|clean}$ and $p_{k|cheat}$ for the K tests come in naturally as parameters of the model. In practice, these numbers may be specified by the firms that manufacture the different tests, but they can otherwise be estimated from a data bank of samples from clean sportsmen and confirmed cheaters.

We would like to point out that our assumption that the tests are independent probably is *false*. In all likelihood, some tests are chemically similar and will respond in a similar manner to various athletes and conditions. However, the simple model is probably a good starting point, and we can refine it later if it does not work well enough in preliminary tests.

In practice, there may be more than one possible doping substance, each of which has a different signature on our tests. A simple refinement is then to use a mixture model for the *cheat* class. Also, because of natural biological variability, not all athletes may respond in the same way to the tests. This suggests a model where both classes are modelled as mixtures.

Example 4.1

4.3.2 The Importance of Simple Models

In an ideal world, $\widehat{p}_j$ and $\widehat{f}_j$ perfectly match the distribution $P[S, \boldsymbol{X}]$ that actually generated the data. In practice, however, they are only approximations, e.g., $\widehat{f}_j(\boldsymbol{x} \mid \theta_j) \approx f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j)$. For this reason, we may not get theoretically optimal classifiers in the end.

Surprisingly, even a relatively crude guess $\widehat{f}_j$ can often give good classification performance in practice. This is because the classifier need not describe all aspects of the data well to perform well. What matters is that the model captures the feature properties that are most relevant to classification, that is, those distinguishing differences that set the classes apart. The classifiers used in the course project in Appendix A, for example, typically give good results even though they are far from perfect descriptions of the data.

In practice, it is often best to start by assuming a relatively a simple and easy-to-understand structure $\widehat{f}_j$. This model can later be refined, depending on the results in tests and initial applications. If we start from an overly complicated model and get poor results, it can be difficult to figure out which aspect of the model that should be simplified to improve performance.

4.3.3 Some Common Feature Distributions

For continuous or fine-grained discrete features, it is very common to take $\widehat{f}_j$ to be a Gaussian distribution. This is appropriate whenever it can be argued that the feature consists of a large number of small, independent effects—by the central limit theorem, the sum of many independent variables is approximately Gaussian distributed.

If a class consists of many different subclasses, a mixture model is often a good choice, as mentioned in the doping example. Gaussian mixture models can be shown to be able to approximate *any* distribution well, in a certain mathematical sense, assuming the number of components is sufficiently large. Gaussian mixtures are therefore a popular fall-back model when we do not have a clear impression of how the features will behave, as GMMs in theory can approximate anything.

In the doping example earlier, we used the Bernoulli distribution, which is the standard choice for binary features. Another common type of discrete feature is nonnegative integer data which represent some kind of count. If the counted events are conditionally independent given the class label j, and there is no bound on the maximum number of events, the *Poisson distribution* is often a good approximation. This distribution follows

$$P[X = k \mid S = j, \lambda_j] = \mathrm{Po}(k \mid \lambda_j) \equiv \frac{\lambda_j^k e^{-\lambda_j}}{k!}, \tag{4.6}$$

where X takes values k on the nonnegative integers. $\lambda_j > 0$ is known as the *rate parameter*.

Examples of approximately Poisson-distributed data include the number of photons that hit a light sensor, the number of neurons that respond to a stimulus, the number of customers in a store in one day, and similar. Since the rate λ_j is tied to the expected value of X, such that $E[X \mid S = j, \lambda_j] = \lambda_j$, we can think of the parameter as corresponding to the intensity of the light source, the intensity of the neural response, and so on. Note that if the rate is high, we have a large number of mutually independent events, and the Poisson distribution becomes very similar to a normal distribution by virtue of the central limit theorem.

4.4 Parameter Estimation

There are often important properties of the class probabilities that we do not know beforehand. For the model in Eq. (4.3), these aspects are represented by unknown parameters $\boldsymbol{\theta}_j$. To determine these parameters, we apply statistical *parameter estimation* techniques on sample data from the problem. In machine learning, this is known as model *training* or *learning*, and the data as *training data*. Usually, we must solve an optimization problem of some kind to find parameters that fit with the data.

Once we have some estimate $\widehat{\boldsymbol{\theta}}_j$ of the parameters, we can plug this into the model structure in Eq. (4.3) to construct approximate feature distributions

$$P[\boldsymbol{X} = \boldsymbol{x} \mid S = j] \approx \widehat{f}_j(\boldsymbol{x} \mid \widehat{\boldsymbol{\theta}}_j). \tag{4.7}$$

These distributions can then be used to define classifiers according to the various criteria described in Section 3.2. If the class probabilities are unknown, we may treat these as additional parameters to be estimated. With a good choice of model structure and adequate training data, we hope to end up with a trained model that describes the data well, and may come close to the Bayes optimal classification performance, given the features we have.

The act of estimation is very similar to classification: both involve choosing a single element from a set of possible candidates to optimize some criterion, with the difference that parameter estimates are selected from a continuous space, whereas the set of classes is discrete. Parameter estimation therefore has many connections to the decision theory in Chapter 3, and principles such as maximum likelihood, maximum a posteriori, or minimum risk can be applied to estimation as well.

The training data used for parameter estimation, obtained by applying feature extraction to the raw attributes in $\widetilde{\mathcal{D}}$, will be written $\mathcal{D}$. It consists of N feature-label pairs,

$$\mathcal{D} = \{(\boldsymbol{x}_1, s_1), \ldots, (\boldsymbol{x}_N, s_N)\}. \tag{4.8}$$

The standard parameter estimation approach for generative models is to take all training examples corresponding to class j (where $s_n = j$), and use these to estimate the parameters $\boldsymbol{\theta}_j$ of that class. We will write

$$\mathcal{D}_j = (\boldsymbol{x}_{j,1}, \ldots, \boldsymbol{x}_{j,N_j}) \tag{4.9}$$

for this subset of the training data feature vectors. In this case, each class-conditional feature distribution $\widehat{f}_j$ is trained independently of all the others, and if we change $\mathcal{D}_j$ only the estimate $\widehat{\boldsymbol{\theta}}_j$ is affected.

4.4.1 Maximum Likelihood Parameter Estimation

There are many ways to compute an estimate $\widehat{\boldsymbol{\theta}}_j$ based on the training data. A very common approach, which should be familiar from introductory statistics courses, is to use *maximum likelihood parameter estimation* (ML estimation or MLE). Similar to the maximum likelihood decision rule in Equation (3.16), the idea is to find the parameter set that maximizes the likelihood, which is the probability of the given data as a function of the parameters,

$$P\left[\mathcal{D} \;\middle|\; \{\boldsymbol{\theta}_j, p_j\}_{j=1}^{N_s}\right] = \prod_{n=1}^{N} \widehat{f}_{s_n}(\boldsymbol{x}_n \mid \boldsymbol{\theta}_{s_n}) p_{s_n} \tag{4.10}$$

$$= \prod_{j=1}^{N_s} p_j^{N_j} P\left[\mathcal{D}_j \mid \boldsymbol{\theta}_j\right] \tag{4.11}$$

$$P\left[\mathcal{D}_j \mid \boldsymbol{\theta}_j\right] = \prod_{n=1}^{N_j} \widehat{f}_j(\boldsymbol{x}_{j,n} \mid \boldsymbol{\theta}_j). \tag{4.12}$$

(We have made the standard assumption that the training data examples are mutually independent and identically distributed within each class.)

We see that the total likelihood in Eq. (4.11) separates across the different classes into functions (Eq. (4.12)) that can be optimized independently of each other. The maximum likelihood estimate $\widehat{\boldsymbol{\theta}}_{\mathrm{ML}}$ for class j is the parameter value for which the corresponding observed data $\mathcal{D}_j$ has the highest probability.

From a mathematical perspective, it is often more convenient to maximize the logarithm of the likelihood. Since the logarithm $\ln(x)$ is a monotonically increasing function of x, this log-likelihood has maxima at the same points where the likelihood in Eq. (4.12) is maximized. We will write the log-likelihood as

$$\mathcal{L}(\boldsymbol{\theta} \mid \mathcal{D}_j) = \sum_{n=1}^{N_j} \ln \widehat{f}_j(\boldsymbol{x}_{j,n} \mid \boldsymbol{\theta}). \tag{4.13}$$

We can then define the maximum likelihood parameter estimate $\widehat{\boldsymbol{\theta}}_{\mathrm{ML}}$ as

$$\widehat{\boldsymbol{\theta}}_{\mathrm{ML}}(\mathcal{D}_j) = \operatorname*{argmax}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta} \mid \mathcal{D}_j). \tag{4.14}$$

Maximum likelihood estimation has a number of appealing properties that make it a popular choice in applications. For instance it is asymptotically consistent, unbiased, and efficient. Roughly speaking, this means that—assuming the data was actually generated by a model of the form $\widehat{f}_j$—the ML estimate will, given enough samples, converge towards the true parameter value, with an expected error that is not bigger than that of *any* other reasonable estimation procedure. It is also invariant under reparameterization, so estimating the parameter $\boldsymbol{\theta}$ or a transformed $\boldsymbol{\theta}' = \boldsymbol{g}(\boldsymbol{\theta})$ gives mutually consistent results.

We will now review a few standard examples of maximum likelihood estimation:

Example 4.2 (MLE for a Gaussian Distribution): Assume that the data for class j consists of independent samples from a scalar normal distribution with unknown parameters μ and $\sigma > 0$. Given sample data

$$\mathcal{D}_j = (x_{j,1}, \ldots, x_{j,n}, \ldots, x_{j,N_j}), \tag{4.15}$$

find the maximum likelihood estimates of μ and σ. Also draw a graph of the log-likelihood as a function of the parameter σ for the example dataset $\mathcal{D}_j = (-1,\, 0,\, 1)$ and $\mu = 0$.

Solution:

Following Eq. (4.13), the log-likelihood has the form

$$\mathcal{L}(\mu, \sigma \mid \mathcal{D}_j) = \sum_{n=1}^{N_j} \ln \widehat{f}_j(x_{j,n} \mid \mu, \sigma) = \tag{4.16}$$

$$= -\frac{N_j}{2} \ln(2\pi) - N_j \ln \sigma - \frac{1}{2\sigma^2} \sum_{n=1}^{N_j} (x_{j,n} - \mu)^2. \tag{4.17}$$

This function reaches its maximum value where the gradient is zero. We thus need find $\widehat{\boldsymbol{\theta}}_{\mathrm{ML}} = (\widehat{\mu}_{\mathrm{ML}}, \widehat{\sigma}_{\mathrm{ML}})^T$ that satisfies

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta} \mid \mathcal{D}_j)\big|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}_{\mathrm{ML}}} = \mathbf{0}. \tag{4.18}$$

The gradient of (4.17) with respect to μ is simply

$$\frac{\partial \mathcal{L}}{\partial \mu} = -\frac{1}{\sigma^2} \sum_{n=1}^{N_j} (x_{j,n} - \mu) = \tag{4.19}$$

$$= \frac{1}{\sigma^2} \left(N_j \mu - \sum_{n=1}^{N_j} x_{j,n} \right). \tag{4.20}$$

It is easy to see that this only is zero for

$$\widehat{\mu}_{\mathrm{ML}} = \frac{1}{N_j} \sum_{n=1}^{N_j} x_{j,n}. \tag{4.21}$$

This is the familiar *sample mean*, often written $\overline{x}$.

For the standard deviation σ, the relevant partial derivative of (4.17) is

$$\frac{\partial \mathcal{L}}{\partial \sigma} = -\frac{N_j}{\sigma} + \frac{1}{\sigma^3} \sum_{n=1}^{N_j} (x_{j,n} - \mu)^2. \tag{4.22}$$

By inserting the maximum likelihood estimate $\widehat{\mu}_{\mathrm{ML}}$ of μ from Equation (4.21) and multiplying by σ^3, we see that the gradient is zero for $\widehat{\sigma}_{\mathrm{ML}}$ satisfying

$$N_j \widehat{\sigma}^2_{\mathrm{ML}} = \sum_{n=1}^{N_j} (x_{j,n} - \overline{x})^2. \tag{4.23}$$

The maximum likelihood parameter estimates for the normal distribution are therefore the sample mean and the square root of the sample variance,

$$\widehat{\boldsymbol{\theta}}_{\mathrm{ML}} = (\widehat{\mu}_{\mathrm{ML}}, \widehat{\sigma}_{\mathrm{ML}})^T = \left(\overline{x}, \sqrt{\frac{1}{N_j} \sum_{n=1}^{N_j} (x_{j,n} - \overline{x})^2} \right)^T. \tag{4.24}$$

Strictly speaking, one also needs to show that this extreme point is a (global) maximum, and that there are no maxima on the boundary $\sigma = 0$, but we will not do that here.

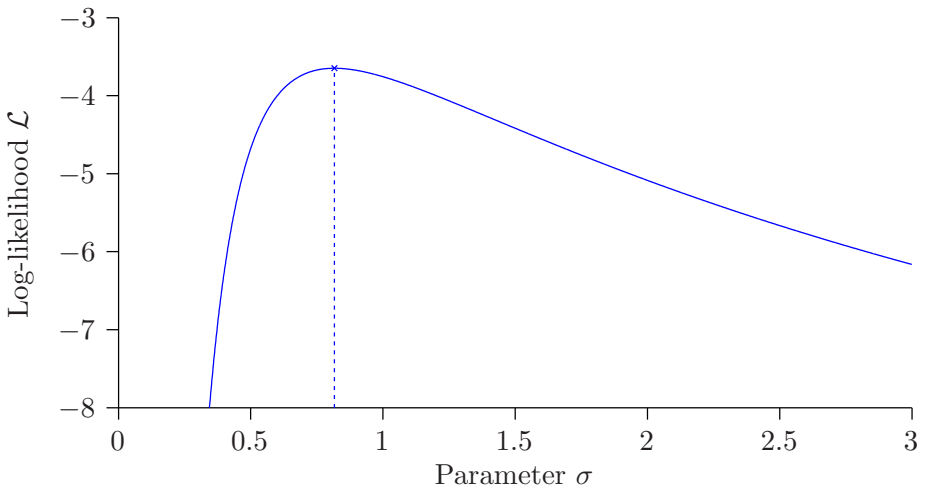


Figure 4.3: Example Gaussian log-likelihood as a function of the parameter σ.

A graph of the log-likelihood $\mathcal{L}(\mu = \widehat{\mu} = 0, \sigma \mid \mathcal{D}_j)$ of the toy data $\mathcal{D}_j = (-1, 0, 1)$ for different σ is provided in Figure 4.3. The optimum value $\widehat{\sigma}_{\mathrm{ML}} = \sqrt{2/3} \approx 0.82$ is indicated by a dotted line. We see that, from a likelihood perspective, underestimating σ gives a much worse fit than overestimating the standard deviation. This is because the MLE objective

wants to assign a high probability to all training data points simultaneously, and therefore must ensure that $\widehat{f}_j$ covers the entire span (support) of the training data.

Example 4.2

Example 4.3 (MLE for a Categorical Distribution): The class labels s_n in the training data $\mathcal{D}$ are drawn from a discrete distribution with probabilities $P[S_n = j] = p_j$ satisfying $\sum_{j=1}^{N_s} p_j = 1$ and $p_j \geq 0$ for all j. This general discrete distribution is often called a *categorical distribution*. What is the maximum likelihood estimate of the class probabilities $\boldsymbol{p} = (p_1, \ldots, p_{N_s})^T$, based on the data $\mathcal{D}$?
Solution:
In this case, the log of the likelihood function (4.13) becomes

$$\mathcal{L}(\boldsymbol{p} \mid \mathcal{D}) = \sum_{n=1}^{N} \ln p_{s_n}, \tag{4.25}$$

For every example $s_n = j$ that belongs to class j, we have $\ln p_{s_n} = \ln p_j$. Since the number of examples of class j in the training database in N_j, we can write the log-likelihood as

$$\mathcal{L}(\boldsymbol{p} \mid \mathcal{D}) = \sum_{j=1}^{N_s} N_j \ln p_j. \tag{4.26}$$

We want to maximize Eq. (4.26) under the condition $\sum_{j=1}^{N_s} p_j = 1$. There are several ways to do this. Here, we will use the method of *Lagrange multipliers*, which should be familiar from courses on constrained optimization. We introduce a Lagrange multiplier $\lambda \in \mathbb{R}$ for the constraint

$$\left(1 - \sum_{j=1}^{N_s} p_j\right) = 0, \tag{4.27}$$

and seek the points $(\widehat{\boldsymbol{p}}, \lambda)$ where the Lagrangian

$$L(\boldsymbol{p}, \lambda) = \mathcal{L}(\boldsymbol{p} \mid \mathcal{D}) + \lambda \left(1 - \sum_{j=1}^{N_s} p_j\right) \tag{4.28}$$

has zero gradient, i.e.,

$$\nabla_{\boldsymbol{p}} L = \mathbf{0} \tag{4.29}$$

$$\frac{\partial L}{\partial \lambda} = 0. \tag{4.30}$$

Note that, even though the data here is discrete, the parameters are defined on a continuous space, and we can take derivatives as in the previous

example. We will ignore the nonnegativity constraint $p_j \geq 0$ as the final solution will anyhow satisfy nonnegativity; this is not always the case for more advanced problems.

Setting the gradient in Eq. (4.29) to zero gives

$$\frac{\partial L}{\partial p_j}(\widehat{\boldsymbol{p}}, \lambda) = \frac{N_j}{\widehat{p}_j} - \lambda = 0 \tag{4.31}$$

$$\widehat{p}_j = \frac{N_j}{\lambda} \tag{4.32}$$

for all $j \in \{1, \ldots, N_s\}$. Condition (4.30), meanwhile, equates to

$$\frac{\partial L}{\partial \lambda}(\widehat{\boldsymbol{p}}, \lambda) = 1 - \sum_{j=1}^{N_s} \widehat{p}_j = 0. \tag{4.33}$$

Slotting in the values for $\widehat{p}_j$ from Eq. (4.32), we find that

$$1 - \frac{1}{\lambda} \sum_{j=1}^{N_s} N_j = 0 \tag{4.34}$$

$$\lambda = \sum_{j=1}^{N_s} N_j = N. \tag{4.35}$$

Inserting this λ back into Eq. (4.32), we get the maximum likelihood parameter estimates

$$\widehat{p}_{\mathrm{ML}\,j} = \frac{N_j}{N}. \tag{4.36}$$

This is simply the *relative frequency* of class j, the fraction of training examples belonging to j. (We ignore the steps necessary to show that this is the unique global maximum of the likelihood.)

Example 4.3

4.4.2 Other Parameter Estimation Methods

The main alternative to maximum likelihood estimation is to use so-called *Bayesian learning*. This is a very different paradigm from maximum likelihood. In Bayesian approaches, we also treat the parameters as random variables. There is then not a single guess $\widehat{\boldsymbol{\theta}}$ (point estimate) for the parameter, but rather a whole distribution $\widehat{f}_{\boldsymbol{\Theta}|\mathcal{D}}(\boldsymbol{\theta} \mid \mathcal{D})$ of possible parameter values given the data. This way, one can represent not only what value(s) of $\boldsymbol{\theta}$ that are probable, but also how certain we are about this guess or estimate. If we want to select a single point estimate of the parameter values, we can pick the most probable parameter vector conditioned on the data. This is the so-called *maximum a posteriori estimate* $\widehat{\boldsymbol{\theta}}_{\mathrm{MAP}}$, defined by

$$\widehat{\boldsymbol{\theta}}_{\mathrm{MAP}}(\mathcal{D}_j) = \operatorname*{argmax}_{\boldsymbol{\theta}} \widehat{f}_{\boldsymbol{\Theta}|\mathcal{D}}(\boldsymbol{\theta} \mid \mathcal{D}_j), \tag{4.37}$$

in analogy to the MAP decision rule in Equation (3.14).

Many of the problems discussed later in this chapter can be avoided by using Bayesian approaches. On the other hand, Bayesian methods often involve more complicated mathematics, and are based on somewhat controversial assumptions. Specifically, the distribution $\widehat{f}_{\boldsymbol{\Theta}}(\boldsymbol{\theta})$ we specify for the parameter values prior to training can have a large impact on the end result.

We reserve Chapter 8 for an in-depth discussion of Bayesian methods. In Chapters 5–7, we will stick with traditional maximum likelihood estimation. Some other estimation techniques, which can work better than MLE on difficult practical problems, are however outlined in Section 4.7.

4.5 Evaluating Classifier Performance

Once parameters have been estimated, we have an unambiguously defined, probabilistic model of our classification problem. This can be used to classify new data, following the various criteria in Section 3.2. But, before we start applying the classifier in practice, we probably want to know if it is any good.

4.5.1 Cross-Validation

In order to get an impression of how the classifier performs, we can check the amount of misclassifications over the training data. Unfortunately, this figure is often misleading, since the model essentially was optimized to make this number as small as possible. Even very bad classifiers can show good results on the training data. (By the same token, the raw log-likelihood of the training data should never be used as a performance indicator either.) To see how well the model generalizes, it is much more informative to check the performance on new, previously unseen *test data* instead.

In case new test data cannot be acquired or generated on demand, one can apply a procedure called *cross-validation* to gauge classification performance. In cross-validation, we partition the available data $\mathcal{D}$ into a *training set* $\mathcal{D}^{(\text{train})}$ and a *test set* or *validation set* $\mathcal{D}^{(\text{test})}$. The split is often random, but other choices are possible. During parameter estimation, only the data in the training set is used. After training, the resulting classifier is applied to the examples in test set, which it has not seen before. The classification accuracy on this test data is often a good indicator of how well the classifier will perform in later applications. The log-probability that the trained model assigns to the test dataset is another common performance indicator. If we have several models to choose from, we typically prefer the one that has the best performance on the test data.

Various forms of cross-validation are the gold standard in evaluating classifiers for use in practical applications today. A common extension is to repeat the cross-validation procedure several times, each time partitioning

the data in different ways, and compute the average performance over the different experiments. This average value is typically less susceptible to random variation, and thus more reliable.

4.5.2 Improving the Classifier

Now that we have an indication of how well the classifier performs on unseen data, we can start tweaking the design, and see if the validation-set performance improves. Can we go back and change the features, or the model structure, or the parameter estimation procedure, to get a better end result? This kind of iterative improvement procedure is often necessary to devise a high-quality classifier.

During iterative refinement of classifiers, it is common to have not one, but *two* separate validation sets: one so-called *development set*, and one actual test set, which is saved for last. After estimating parameters on the training set, we estimate classification accuracy using the development set, and use the result to go back and tune our approach until we get good development-set performance. Only when we have decided on the final system to apply in real life do we compute how well we classify the real test set, to get a final, unbiased estimate of system performance.

Occasionally, it is informative to not only consider the number of test-set misclassifications during classifier design, but also look at *which* particular examples that were classified incorrectly. This might tell us where and why the model comes up short. in practice, this is most useful when working with simple and interpretable models, since it is hard to attribute blame in a large and complicated model.

Compared to discriminative models, probabilistic, generative models can allow even further insights into model behaviour. This is because generative approaches are capable of *synthesis*: We can create "fake" new data by sampling from the trained feature distribution $\widehat{f}_j(\boldsymbol{x} \mid \widehat{\theta}_j(\mathcal{D}_j))$. By comparing these samples against the training data $\mathcal{D}_j$, we can draw conclusions on what the model has learned, what aspects of the data it has *not* learned, and possibly get additional clues on how we could improve the model. Ideally, the samples and the training examples should behave in much the same way.

4.5.3 Ensemble Methods

If we cannot get sufficient performance by tweaking a single classifier, a final resort is to create many different classifiers, and try to combine their output to get an even better decision in the end. Methods that do this are known as *ensemble methods*. The hope is that the different component models may complement each others' strengths and weaknesses. As a result, ensembles tend to work particularly well if the component models are highly diverse.

Some prominent ensemble methods, in order of increasing sophistication, are *bagging*, *boosting*, and *stacking*. Among these, boosting is famous for having very impressive theoretical guarantees on the asymptotic performance that can be achieved. Stacking is the idea of feeding the outputs of a set of learners into another "higher-level" classifier, like in artificial neural networks, which are composed of stacked layers of small, simple learners called perceptrons.

Ensemble methods have proven to be very successful in prediction contests like the famous *Netflix prize*, where the two best entries were stacked ensembles containing over 100 different component models.

4.6 Feature Extraction Problems

As we have seen, there is a lot of guesswork and approximations involved in building practical classifiers. The adequacy and accuracy of these guesses and approximations naturally affect the accuracy of the associated classifier. There are three principal things that can go wrong:

1. The features may not adequately extract and present the information available for classification.

2. The probabilistic model may not match the true behaviour of the data, i.e., $f_{\boldsymbol{X}|S} \neq \widehat{f}_{\boldsymbol{X}|S}$.

3. The data $\widetilde{\mathcal{D}}$ may be of poor quality, or not representative of the situation in which the classifier will be applied.

Some knowledge about these issues is invaluable in getting the most out of your classifier and in troubleshooting classifier performance problems. In this section and the ones that follow, we comment on problems with the feature extractor, the model, and the data, in turn.

4.6.1 The Curse of Dimensionality

Since feature extraction is quite domain-specific, it is not easy to describe everything that can go wrong, but a few tips on effective feature extraction were shared in Section 4.2. In this section, we concentrate on the balance between removing too much information and removing too little.

Features which remove too much information will have almost no information left on which classification can be based, leading to poor performance. Large feature vectors, where little information is removed, are not necessarily good either. This may be appear surprising, since they in theory may contain all the information we need to perform accurate classification.

As an example, consider a situation with 1 useful feature along with 999 features which are pure noise, completely independent of the true class.

In practice, it is likely that a few of the noisy features will correlate well with the true class on the data we have observed, due to chance alone. Some might even appear to correlate better with the class label than the first feature, depending on how noisy that first feature is. It may thus be possible to construct a classifier based only on the 999 noisy features which looks good on paper, and classifies most historical examples correctly. However, it will not be able to predict new samples at all—after all, the input is just uninformative noise.

Unfortunately, even if we instead have just 20 features that all are informative to some degree, problems persist. The reason is that the geometry of high-dimensional spaces is very different from the low-dimensional spaces we are used to thinking about; for instance, high dimensions contain more space than one may expect. If the datapoints are binary features distributed over the corners of a 20-dimensional hypercube, we need more than a million samples to have an example from each corner! Even the task of estimating probability distributions in high dimensions is therefore fundamentally hard.

Another fact that can be seen from the hypercube example is the that the number of nearest neighbours increases with dimensionality. Even worse, it turns out that, in high dimensions, nearly all of the points on the hypercube (and in space in general) are approximately the same distance from each other. The consequence is that many methods based on distances, especially highly flexible ones such as k-nearest neighbour approaches, tend to break down in high-dimensional spaces. The many problems that arise when working with high-dimensional data are frequently collected under the umbrella term *the curse of dimensionality*.

Fortunately, things are seldom as bad as the previous examples suggest. If there is some structure to the data, the datapoints will lie on a low-dimensional surface (manifold or subspace) in the high-dimensional space. Dimensionality reduction and other techniques can then extract or approximate this surface, enabling efficient probability estimation and classification on the low-dimensional space where the data resides.

4.7 Model Problems

Even if the features do a good job at removing and structuring information in the raw input, we must use a suitable model that can pick up on the structure and behaviour of the data. The situation where the assumed probabilistic model does not match the true distribution of the feature data,

$$P\left[\boldsymbol{X} = \boldsymbol{x} \mid S = j\right] \neq \widehat{f}_j(\boldsymbol{x} \mid \boldsymbol{\theta}_j) \tag{4.38}$$

for all values of θ_j, is known as *model misspecification*.

One obvious solution to the problem is to try to create an improved model $\widehat{f}'_j$ that better matches the properties of the data. A common issue

can be that the class j may be composed of several different sub-classes with quite different feature behaviour. In a speech recognizer, certain sounds may be pronounced very differently depending on the accent of the speaker, and in a handwriting recognition problem, letters can be written in more than one way, e.g., "A" and "a." In this situation, mixture models or ensemble methods can be helpful, as discussed earlier.

4.7.1 Discriminative Training

Even when we may not know how to change the model to get a better description, or are required to use a specific form for $\widehat{f}_j$, there are other techniques that can be used to improve performance. Recall that Bayes optimal classification requires us to know exactly how the data is distributed. Standard maximum likelihood estimation will asymptotically converge on the correct model, *if* the assumed model structure $\widehat{f}_j$ matches the true distribution of the feature data. But when the model $\widehat{f}_j$ is incorrect there are other approaches that can perform better than the standard method. The key idea is to choose parameters that *distinguish* the different classes optimally, rather than describe them. This is known as *discriminative training*.

A nice way to think about discriminative training is that standard probabilistic models try to describe all the data, which makes regions with many data points especially important, but the only part that matters for classification is where the classification boundary between the different classes is located. Discriminatively trained (generative) models are thus similar in spirit to discriminative models, which learn discriminant functions directly. On the other hand, discriminatively-trained models should not be used to generate data, and can be difficult to interpret, since they have not been trained to actually describe the data material, but only to the represent the differences between classes.

In discriminative training, class-conditional parameters are not estimated in isolation, but the parameter estimates $\widehat{\boldsymbol{\theta}}_j$ for class j depend on *all* the training data $\mathcal{D}$, not just the subset $\mathcal{D}_j$, and likely on the parameter estimates for the other classes as well. Discriminative parameter estimation procedures are therefore often more complicated and computationally demanding than MLE.

4.7.2 Minimum Classification Error Training

One appealing idea for discriminative training is to try to choose parameters that classify the training data as well as possible. In other words, one wants to minimize the amount of misclassifications over $\mathcal{D}$ or $\mathcal{D}^{\text{train}}$. This is known as *Minimum Classification Error* (MCE). Mathematically, this amounts to

finding the parameter set which maximizes

$$F_{\mathrm{MCE}}(\{\boldsymbol{\theta}_j, p_j\}_{j=1}^{N_s} \mid \mathcal{D}) = \\ = \frac{1}{N}\sum_{n=1}^{N} I\left(\max_{\substack{j'\in\{1,\ldots,N_s\}\\ j'\neq s_n}} \widehat{f}_{j'}(\boldsymbol{x}_n \,\middle|\, \boldsymbol{\theta}_{j'})p_{j'} \geq \widehat{f}_{s_n}(\boldsymbol{x}_n \,\middle|\, \boldsymbol{\theta}_{s_n})p_{s_n}\right), \quad (4.39)$$

where $I(C)$ is an indicator variable which is 1 if and only if the condition C is true, and 0 otherwise.

In practice, MCE can be a difficult criterion to work with. An obvious complication is that the number of training errors is an integer-valued, piecewise constant function of the parameters. If we change $\widehat{\boldsymbol{\theta}}_j$ just a little, the number of training errors probably won't change at all. If, however, the classification boundary passes over one of the training examples, there will be a step change in the training error.

To make MCE easier to handle, one can approximate the discontinuous objective function in Eq. (4.39) by a continuous and differentiable function which allows gradient-based optimization. The step function $I(C)$ can be replaced by a smooth, S-shaped "sigmoid" function, for instance.

As the end result of MCE, we can expect a classifier that attempts to place the classification boundary as much "in between" the classes as possible, maximizing the separation between them. This is very similar to the philosophy behind *maximum margin methods* like *support vector machines* (SVMs), which have been one of the most successful geometric classification methods of the last decades. MCE thus provides classifiers similar to SVMs in flavour, but based on probabilistic principles. This probabilistic nature can be helpful in building and interpreting the models, since it makes the fundamental assumptions made by the models more apparent.

Compared to maximum likelihood estimation, MCE has drawbacks in that the objective function typically must be approximated, and that training can be much slower than the methods used for MLE.

4.7.3 Maximum Mutual Information Training

Another discriminative training approach, perhaps more common in practice than MCE, is to seek a classifier that tries to assign a high posterior probability to the correct class of the different examples. Specifically, one chooses the parameter set that maximizes the conditional likelihood

$$F_{\mathrm{MMI}}(\{\boldsymbol{\theta}_j, p_j\}_{j=1}^{N_s} \mid \mathcal{D}) = \frac{1}{N}\sum_{n=1}^{N} \ln \widehat{P}\left[S = s_n \,\middle|\, \boldsymbol{X} = \boldsymbol{x}_n, \{\boldsymbol{\theta}_j, p_j\}_{j=1}^{N_s}\right] \quad (4.40)$$

$$= \frac{1}{N}\sum_{n=1}^{N}\left(\ln \widehat{f}_{s_n}(\boldsymbol{x}_n \,\middle|\, \boldsymbol{\theta}_{s_n}) + \ln p_{s_n} - \ln \sum_{j=1}^{N_s} \widehat{f}_j(\boldsymbol{x}_n \,\middle|\, \boldsymbol{\theta}_j)p_j\right), \quad (4.41)$$

where the $\ln \widehat{P}$-term represents the posterior probability of the correct class for example n, according to the classifier. This approach is, somewhat obscurely, called *Maximum Mutual Information training* (MMI), as the resulting class-label estimates $d(\boldsymbol{x}_n)$ are as informative as possible about the true class s_n in the sense of a mathematical framework known as information theory.

Unlike MCE, maximum mutual information training maximizes a continuous function of the parameters, so it is not necessary to introduce any tuning parameters or additional approximations. Unfortunately, the criterion in (4.41) is still slow to optimize in practice, since we cannot easily apply efficient training techniques like those described in Chapter 7.

To summarize and complement the discussion, Figure 4.4 shows an example of how many different classification techniques behave on a nontrivial classification dataset.

4.8 Training Data Problems

Sometimes, the problem is not the model $\widehat{f}$, but the training data $\mathcal{D}$, which may be of poor quality or not representative of the situations where the classifier will be applied. There can be a variety of reasons for this.

4.8.1 Scarce Data and Overfitting

The simplest data problem is if the training set is small, so that we have too little data for one or more of the classes. It is easy to realize that a small dataset might not contain examples of everything that can happen in practice, and what the data looks like will depend a lot on chance.

The solution to this problem is either to gather more data—then the law of large numbers becomes applicable, and the data increasingly looks like what one might expect, as illustrated in Figure 4.5—or to use simple models with only a few unknown parameters. Simple models typically require fewer samples in order to estimate the parameters accurately. The size of the training dataset therefore limits how complicated models we can use.

If we use complicated models with many parameters, we often end up in a situation where the model looks very good on the training data, but performs much worse when classifying new examples. This problem is known as *overfitting* or sometimes *overtraining*, and is a well-known issue in applied statistics. (It is also related to the curse of dimensionality discussed in Section 4.6.1.) Because a complicated model has many degrees of freedom, it can fit itself to the random variations (noise) in the data, instead of seeing the simple, underlying trends, as illustrated in Figure 4.6. Cross-validation is frequently a good way to check for overfitting.

To avoid overfitting, it is actually often preferable to choose an oversimplified model, rather than a complex model structure with many parameters—

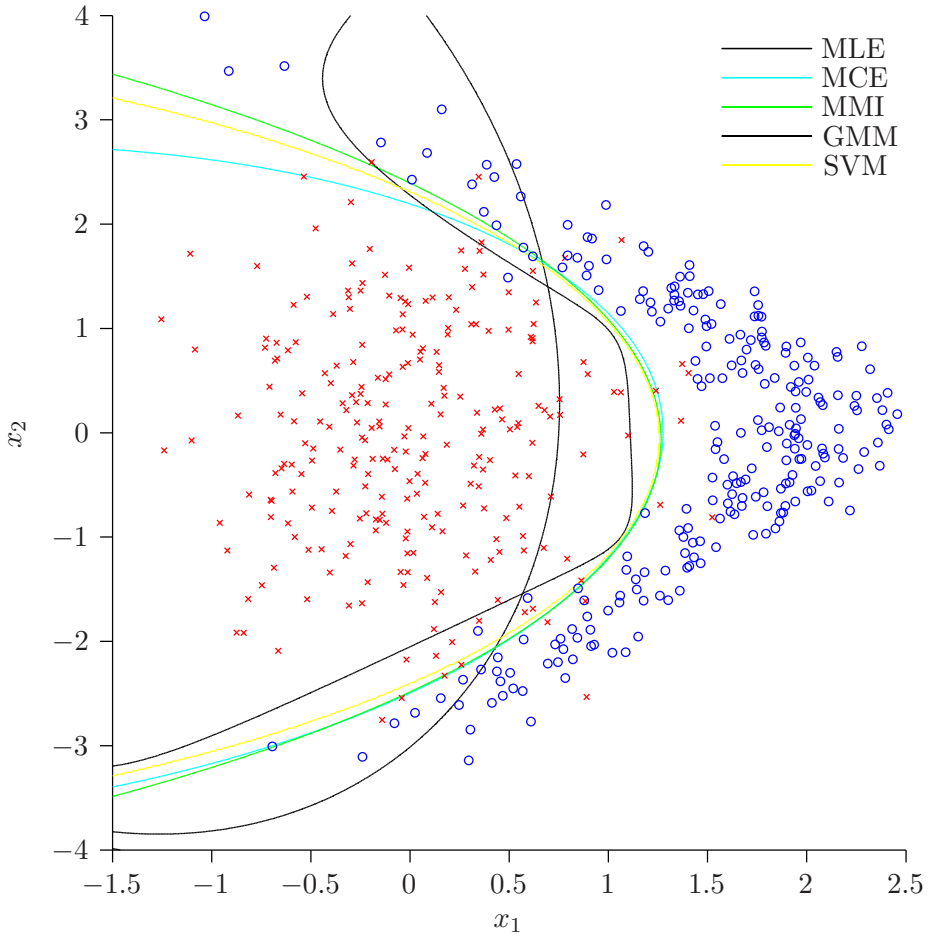


Figure 4.4: Comparison of classification techniques, showing decision region boundaries. The dataset consists of 250 random samples from each of two classes, where the circled class has a non-Gaussian distribution. If we assume $\widehat{f}_1$ and $\widehat{f}_2$ to be Gaussian—a misspecified model—standard MLE gives a sub-optimal classifier (dark, mostly vertical curve). A number of other classifiers all yield significantly improved performance. The common theme is that these have similar class boundaries along the lines where the two point clouds meet. The other classifiers are: discriminative training (MCE and MMI) for the misspecified model; MLE with a different, more appropriate model (dark zigzag curve) having a 3-component GMM approximating the distribution of the circled class; and an SVM (a non-probabilistic, discriminative method) using quadratic features $(1, x_1, x_2, x_1^2, x_2^2, x_1x_2)$, which enables similar decision region shapes as for the non-GMM classifiers.

Classifier	MLE	MCE	MMI	GMM	SVM
Training set misclassifications	48	17	22	32	20
Validation set error rate (percent)	10.6	4.4	4.4	5.5	4.2

Table 4.1: Performance evaluation of the classifiers in Figure 4.4. The validation set consisted of 4000 examples from each class.

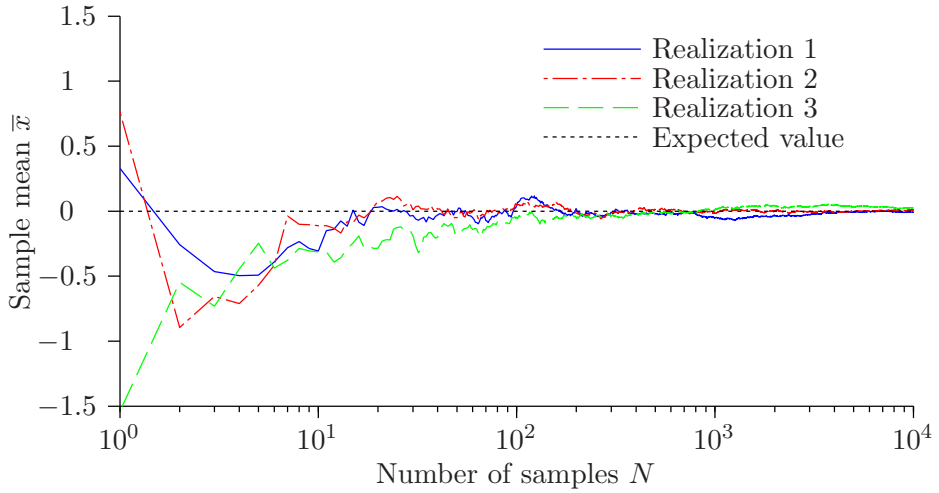


Figure 4.5: The importance of sufficient data. For each curve, 1000 independent samples X_n were drawn from the standard normal distribution. The curves show how the sample mean of $(x_1, \ldots x_N)$ changes as we increase N to consider longer segments of the data. As N grows large, all realizations eventually converge on the expected value.

even if the latter in theory can describe the data perfectly. The great statistician George E. P. Box once said that "all models are wrong, but some are useful." When designing classifiers, our task is to find useful models, rather than models that are somehow "right."

To give preference to simple models, we can introduce a regularization term into the log-likelihood objective function which penalizes complex models. Then we are biased towards choosing a simple model unless there is strong evidence to the contrary.[4] A common penalty term to discourage overfitting is Akaike's information criterion, or AIC.

Another kind of regularizer is the prior used in Bayesian learning methods. These methods have the additional advantage that they can be given a probabilistic interpretation. Effectively, Bayesian approaches contain an entire distribution over different parameter values that are probable given the training data. If we have too little data, the distribution will typically be broad and uncertain, reflecting the fact that the parameters cannot be determined accurately. Situations with too little data are thus easy to spot.

Moreover, optimal prediction or classification using Bayesian approaches

[4]This is often connected to a philosophical argument known as *Occam's razor*, which states that among several competing hypotheses that are consistent with the data, one should prefer the simplest explanation. This is not always for performance reasons—ensembles, which by nature are highly complex, often give improved accuracy—but simple models may also be preferable in their own right, for instance as they are easier to interpret and to work with.

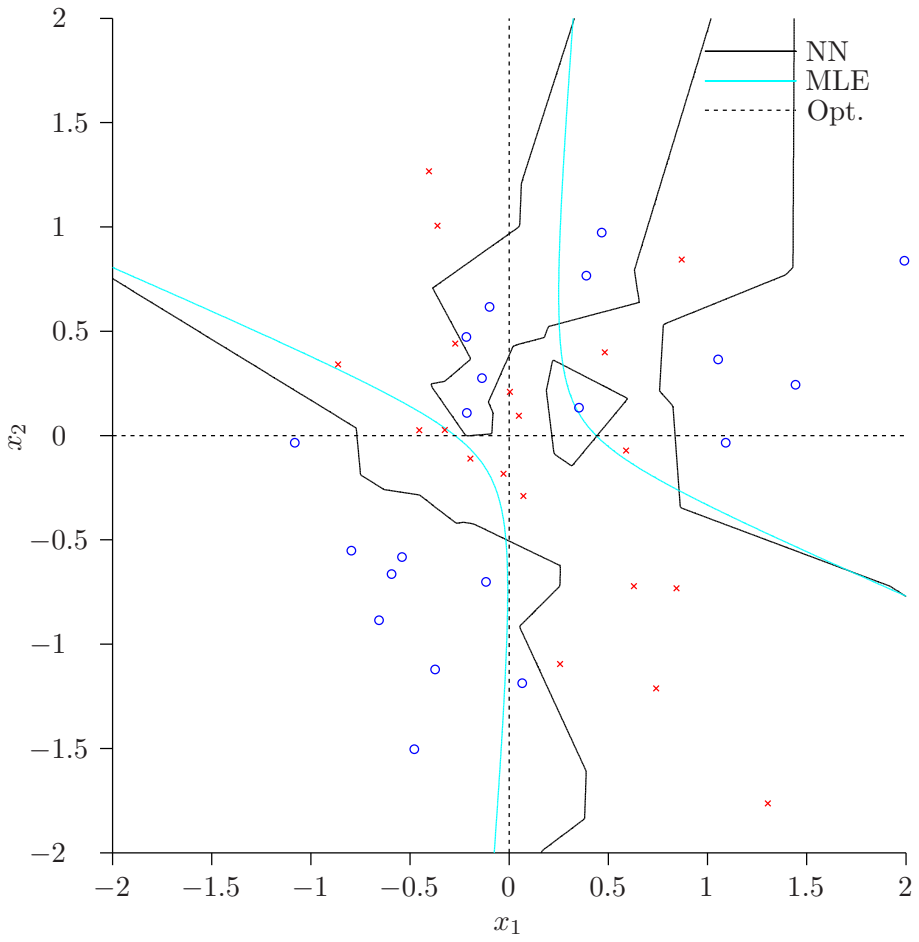


Figure 4.6: Example of overfitting. The dataset consist of 20 samples from each of two classes. Both classes follow two-dimensional Gaussian distributions with zero mean, but with different covariance matrices having mutually orthogonal leading eigenvectors. The optimal classification boundaries (dotted) follow the x_1 and x_2 axes; samples in the first and third quadrant should be classified as the circled class, and the rest as crosses. The figure also shows the decision boundaries of two different classifiers applied to the data: The jagged contour with enclaves and exclaves comes from a nearest neighbour (NN) approach, while the smooth hyperbolas represent class boundaries of a standard MLE approach with Gaussian-distributed classes. Even though the highly flexible NN contour classifies all the training examples correctly, it has a 35.5% error rate on an independent validation set with 4000 examples of each class. The simpler MLE approach misclassifies several training examples, but generalizes better, with a significantly lower error rate of 31.5% on the validation data. The NN classifier thus shows evidence of overfitting. The optimal classifier, for reference, had a 28.9% error rate on the validation set.

essentially perform a weighted average over the entire distribution of possible models (parameter values). This is reminiscent of the ensemble methods from Section 4.5.3, and usually gives better results than picking a single model which can be wildly wrong.[5] When we get more data, so that we become more certain about the correct parameter settings, the Bayesian procedure automatically becomes more similar to the MLE approach.

Mathematically, the choice between simple or complex models can be expressed as a *bias-variance trade-off*. Simple models probably cannot describe the data perfectly, and will have a residual *asymptotic bias* (systematic error) in their predictions. On the other hand, parameters can be estimated accurately from a relatively small dataset. A complex model can often fit the data better in the asymptotic case, yielding low bias. The many parameters of the model are hard to estimate simultaneously, however, and the models will show high variance around the expected value (i.e., overfitting), resulting in poor performance unless the training dataset is large. There is a limit, related to the Cramér-Rao lower bound in statistics, on how small the joint sum of the bias and variance terms of the error can be made, given the available data.

4.8.2 Large Datasets

In rare cases, we may run into the problem of having too much data for our model to handle. Even though large datasets allow for accurate parameter estimation, in practice these estimates may take an infeasibly long time to compute. This tends to be especially problematic for complicated models with many parameters.

As datasets become very large, there is commonly a point at which one begins to reduce model complexity and starts using simpler models again. This is particularly relevant in the emerging "big data" regime, where computation generally is the bottleneck. Alternatively, one could make parameter estimation faster by only using a subset of the data for training, though this feels like a somewhat wasteful approach.

4.8.3 Mismatched Data

Even if we have a good amount of data, there could still be trouble. There are many situations where the data does may not give an accurate picture of the real situation where the classifier is applied. It is possible that some kind of data examples are cheaper and easier to obtain than others, making expensive data under-represented or absent in the database. In statistics, this is a classic problem often referred to as *biased data*. In machine learning,

[5] Bayesian probability is a quite restricted kind of ensemble method, however, as all models share the same structure and therefore aren't very complementary. A more diverse ensemble may improve classification accuracy further.

the problem of mismatches between the training data distribution and later application data is sometimes known as *dataset shift*.

Generally speaking, there is simply no way the training database can cover *all* the variation that will be seen in a practical application, and we have to be aware of this fact when we design classifiers. If we think that mismatch between the training data (the so-called *source domain*) and the intended application (the *target domain*) is going to be a problem, we can use *domain adaptation* techniques to try to adjust ourselves to the situation.

A simple domain adaptation method is to weight the training examples differently, depending on how well they match the usage scenario. This requires that we have at least a few examples in the training data from conditions that match those in the source domain, and we have to re-do training every time the domain changes, which may be slow.

Other approaches involve transforming the feature data from the application to look more like the training data features, or transforming the parameters of the trained model to match the new domain. In the following example, we look at how these techniques are used in speech recognition.

Example 4.4 (Speaker Adaptation): For speech recognition, it is just not feasible to create a training database that contains every possible voice and noise condition the classifier might face in the real world. This is true even though training databases typically are very large, e.g., hundreds of hours of recordings. The method of weighting the training examples differently is therefore not considered a good idea, but transformation-based adaptation approaches are quite common.

A known source of variability between speakers is the vocal tract length. The longer the vocal tract, the lower the resonant frequencies that determine what the voice sounds like; males typically have longer vocal tracts than females. The length of the vocal tract has a predictable effect on the features, and we can thus account for it by introducing an additional parameter λ in the model.

The vocal-tract length parameter is constant for each speaker, but is allowed to vary between speakers, and is estimated separately for each of them. When the system encounters a new speaker in an application, it can gather a little data from this speaker (one sentence, say), and apply the same procedure to produce an estimate $\widehat{\lambda}$ for the new voice. This is known as *vocal-tract length normalization* (VTLN).

When we have more data, we may also adapt to how a particular speaker pronounces various speech sounds, by transforming the model parameters corresponding to each individual sound based on data from the speaker. There are many methods to do this, for instance so-called maximum likelihood linear regression (MLLR) or eigenvoices.

There may be other voice properties and sources of variation (e.g., the microphone, or background noise) whose influence is difficult to anticipate

or model well mathematically. In these cases, it is common to merely try to lessen the unhelpful feature variability that the different conditions introduce.

Typically, the effect of noise is to move the datapoints around randomly in feature space. This can easily make the data end up in a region where there hasn't been any training material to teach the classifier what's right and what's wrong. As a result, the classification accuracy is poor on noisy data. A somewhat *ad hoc* method that can help in practice is then to translate each example to a common region of feature space, for instance by making each training data feature sequence (a sentence or similar) have a mean of zero. In speech recognition, the standard features are called cepstra, so this is known as *cepstral mean normalization* (CMN).

In addition to setting the mean to zero, the standard deviation is often normalized to a common value as well. Among other things, this can prevent numerical problems which sometimes occur if different features differ wildly in size. New data sequences are given the same treatment. The effect is to relocate all data to a common region in feature space where the learned classifier is applicable, a kind of feature transformation.

For all the above methods, one typically needs to first gather a bit of data when exposed to a new condition, before one can compute new parameters or perform any kind of transformation. In both cases it is important to keep the set of numbers that need to be re-estimated small, otherwise a lot of data will be needed to adapt. If we need to perform classification before we have had time to adapt, generic "population average" parameters can be used.

Example 4.4

4.8.4 Novel Events

Another problem with the large variability seen in practical situations is that maximum likelihood estimation may underestimate the breadth of possible things that can happen. This can have problematic consequences for discrete feature data.

In a sense, MLE is a very aggressive method that, if it can, will try to assign zero probability to *any* event not seen in the database. This can be seen, e.g., in the solution (4.36) to example 4.3. If there is a novel occurrence during application, for instance if we run across a word in a text classification task that was not in the training data, all models will say that this event—and thus the entire text—has zero probability. This causes the classifier to fail.

Somewhat surprisingly, it is actually very common to encounter, e.g., uncommon words in a text, so this can be a significant problem in practice. This is because the number of uncommon words in a language typically is extremely large. There are reasons to believe that about half of the words

and spellings in written English have only ever been used once in history! Such words, which occur only once in a source text, are known as *hapax legomena*, which is Greek for "said once."

Issues with identically-zero probabilities can often be overcome by using Bayesian learning approaches to widen the estimated distributions and account for unseen possibilities, or by post-processing the parameters to avoid probabilities that are identically zero, so-called *smoothing*.

4.8.5 Errors in the Data

Sometimes, the data is not merely bad, but downright wrong. The data may be tainted by *outliers* (points that behave markedly different from the rest of the data) due to incorrect measurements, equipment malfunctions, or other problems. Alternatively, the labels s_n in the training data could occasionally be incorrect, a situation known as *label noise*. This can happen in situations where the true label s_n is just too difficult or costly to find out, like in medical diagnostics or in espionage, or be due to simple mistakes when creating the database.

In the case of outliers or label noise, we need models that try to explain most of the data well, but that are allowed to "give up" on some datapoints if they just cannot be made to fit in. Such methods are an example of *robust approaches*, which is an umbrella term for techniques that are designed to work well, or at least degrade gracefully, also in highly averse conditions. Robust methods are also commonly used in dataset-shift situations, such as noisy speech recognition, or any time where the model and the distribution of the assumptions do not match. The price for robustness is that these approaches tend to be quite conservative, and often perform a little worse than optimal when the dataset is unproblematic.

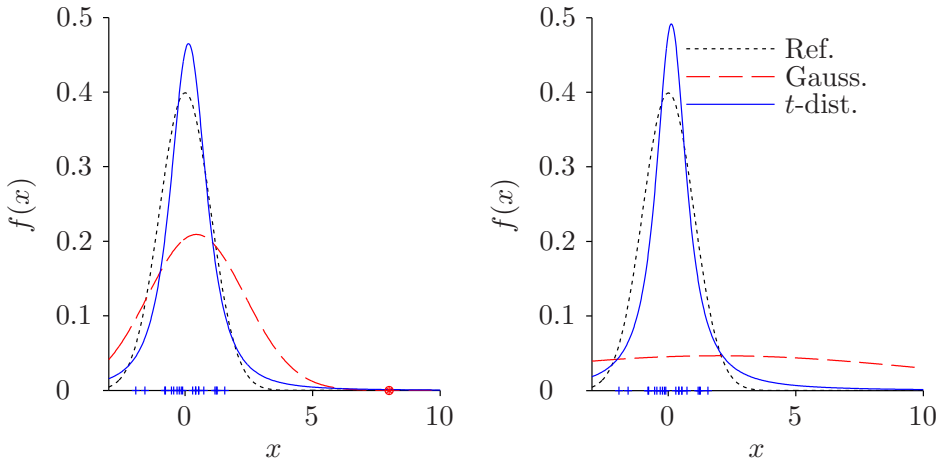
There are many different "robust" methods, which may be robust against different kinds of problems. CMN, for instance, is a technique that is robust against certain acoustic variability. One example of a robust approach that is appropriate for outliers and label noise is to assume that features follow a *Student's t-distribution*,

$$f(t \mid \mu, \sigma, \nu) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\,\Gamma(\frac{\nu}{2})\sigma}\left(1 + \frac{(t-\mu)^2}{\nu\sigma^2}\right)^{-\frac{\nu+1}{2}}, \tag{4.42}$$

instead of a Gaussian distribution as we normally would. The Student's t-distribution has heavier tails than the normal distribution, and is therefore much less affected by datapoints which are far away from the center. Even if we move one datapoint extremely far away, the Student's t-distribution, as estimated by maximum likelihood, will not change substantially, as shown in Figure 4.7.

Interestingly, the Student's t-distribution arises naturally in Bayesian learning for the Gaussian distribution with unknown mean and standard

deviation. The uncertainty about which Gaussian parameters that are correct blurs the picture, and leads to a combined, predicted distribution that is Student's t-distributed.



(a) Outlier at $x = 8$ **(b)** Outlier at $x = 40$

Figure 4.7: Robust distribution estimation. The plots show 20 samples (lines on x-axis) from the standard normal distribution (dotted reference), corrupted by one additional outlier. This outlier exerts a very large influence on a maximum-likelihood estimated Gaussian distribution (dashed line). As the outlier moves away from the data, this Gaussian becomes smeared out and looks nothing like the reference. In contract, a maximum-likelihood fitted Student's t-distribution (solid line) is hardly affected at all by this single corrupt datapoint, and agrees much better with the reference distribution.

Summary

In real-world classification problems, we do not know exactly how the data is generated. Instead we must make assumptions. We base our assumptions on what we know about the problem and on the *training data* $\mathcal{D}$ at our disposal.

Creating a practical classifier involves several steps, especially:

1. **Feature extraction:** We convert the raw input into features $\boldsymbol{x}$, which concentrate the input information relevant to classification.
2. **Model building:** We assume a mathematical form $\widehat{f}_j(\boldsymbol{x} \mid \boldsymbol{\theta}_j)$ for each of the N_s class-conditional feature data distributions.
3. **Training/parameter estimation:** We use the training data $\mathcal{D}$ to estimate the unknown *parameters* $\{\boldsymbol{\theta}_j\}_{j=1}^{N_s}$ of the model.

Classification is then performed using the criteria from Chapter 3.

When the classifier is complete we can use *cross-validation* to estimate its performance on new data. Validation data can also help in further improving the classifier, but we should always save a separate test set for last.

Some Important Concepts in Applied Classification:

Maximum likelihood estimation: The standard parameter estimation method. If our model building assumptions are accurate, this often gives high-quality classifiers.

Discriminative training: Parameter estimation methods specific to classification, which can work better than MLE when the model assumptions are inaccurate. Discriminative training concentrates on parameters that lead to a good decision function $d(\boldsymbol{x})$, rather than on estimating the feature distributions $f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid j)$ accurately.

Robust approaches: Methods that work especially well under difficult conditions where model assumptions and data behaviour do not match, for instance with noisy data.

Bayesian learning: An alternative to maximum likelihood, where more than one possible parameter value is considered. This can avoid many problems with the standard MLE approach. Bayesian learning is described in Chapter 8.

Problems

4.1 (MLE for the Exponential Distribution) A common problem in engineering is to assess the reliability of a system, given statistical data on how long the different components tend to last before they fail. It is then important to determine when and how often each component can be expected to break down.

Let the data

$$\mathcal{D} = (x_1, \ldots, x_N)$$

be a set of measured life-lengths for different examples of a certain component. This data is by nature nonnegative and continuous-valued. A very simple model of such life-length data is the *exponential distribution*

$$f_X(x \mid \mu) = \frac{1}{\mu} \exp(-\frac{x}{\mu})$$

for $x \geq 0$, where $\mu > 0$ is a life-length parameter.

The exponential distribution has the unique property that it is *memoryless*. This means that new and old components have the same probability of failing with in the same time span, and there is no good way to guess when a part will fail based on its age. This is not always a good representation of reality, but can be a reasonable first guess for light bulbs and other components that break suddenly and unpredictably.

4.1.a Assume that all samples are independent. Write down the probability of the dataset $\mathcal{D}$ for a given parameter value μ.

4.1.b Find the maximum-likelihood parameter estimate $\widehat{\mu}_{\mathrm{ML}}$ as a function of the data.

4.1.c A certain type of light bulb has an exponentially distributed lifetime. Your other light bulbs of this kind have lasted 35 050, 15 450, and 15 800 hours. What is the maximum-likelihood parameter estimate for this data?

4.2 (MLE for the Possion Distribution) The Possion distribution was presented in Section 4.3.3. It is defined by the probability mass function

$$P\left[X = k \mid \lambda\right] = \mathrm{Po}(k \mid \lambda) \equiv \frac{\lambda^k e^{-\lambda}}{k!}$$

on nonnegative integers k.

4.2.a Compute the maximum-likelihood estimate $\widehat{\lambda}_{\mathrm{ML}}$ of the rate parameter for a sequence $(x_1, \ldots, x_N)$ of independent samples from the Poisson distribution.

4.2.b The fictitious dataset (3, 6, 3, 4) represents the annual number of customers in the first four years for the makers of a certain, extremely expensive luxury yacht. Assuming that the number of customers is Poisson distributed and that the years are mutually independent, what is the maximum likelihood estimate of the rate parameter?

4.2.c If the yacht gets less than two customers in an entire year, the company behind it will fold. What is the probability that the yacht will have less than two buyers in a given fiscal year, given the rate-parameter estimate from the previous subproblem?

4.3 (Doping Testing) In this problem we will look a bit deeper at the doping test scenario in example 4.1 on page 62. We will also look at what happens when the model is wrong. Read the example first, then answer the following questions:

4.3.a Assume that the a-priori probability that a tested individual is under the influence of a certain forbidden substance is $p_{cheat} = 0.9$. Also assume that the individual tests give the right indication 90% of the time, so that $p_{k|clean} = 0.1$ and $p_{k|cheat} = 0.9$, and that $k = 2$ tests are made. What is the probability that a tested athlete is influenced by the substance if both tests are positive? What if only one test is positive? What about the case when both test come up clean?

4.3.b In practice, many tests are correlated, and tend to make mistakes on the same examples. Drug tests to indicate heroin use, for instance, typically also react to other, innocent substances, such as poppy seeds commonly used in baking bread. To get an impression of how such correlations can affect the conclusions, we consider a second model, in which the two tests are totally correlated, so that the second test always gives the same result as the first. The first test is still assumed to be accurate 90% of the time.

What is the probability in this new model that an athlete is influenced by the substance if both tests are positive? What if both tests are clean? (Note that we cannot get only one positive result in this case, since the tests never disagree.)

4.3.c Compare your answer to the previous two problems. What would be the practical consequences of using the wrong model, acting as if the tests are independent as in the first case, when in reality they are not? How can we notice or avoid these issues in practice?

Chapter 5

Hidden Markov Models for Sequence Classification

In previous chapters, we have discussed classification rules for situations where we need to make exactly *one decision* using exactly *one observed feature vector*. Many practical situations are more complicated: Often, we need to make intelligent guesses about a long sequence of patterns, not just one single pattern. If consecutive patterns are statistically independent, then each decision can also be made independently of previous decisions, exactly as in the previous chapter. However, the patterns in the sequence are usually correlated, and optimal decisions cannot be made independently of each other.

For example, consider designing an automatic speech-recognition system. The input data is a sound signal recorded by a microphone. The primary task is to classify this signal into a sequence of discrete phonetic labels, such as /mmaammmmaaa/,which can later be interpreted as a word or a sentence in a known language. The speech signal has highly variable characteristics. Those variations convey the message to the listener. The time scale of the variations is determined by the speed of the movements of the speaker's jaw, lips, tongue, etc. During vowel segments the signal characteristics can be relatively stable for 100–200 ms or so.

In order to decrease the amount of data to process, a speech recognizer usually segments the signal into a sequence of short time frames, usually 10–20 ms each. The signal is relatively stable within each frame. It is reasonable to assume that the signal source was in some constant discrete state S_t defining the speech sound that was pronounced by the speaker at time t, while this particular frame was recorded. (Here the index t is an integer defining the frame number, not the continous real time.) In the classifier, each signal frame is processed by a set of feature extractors and each frame is described by a feature vector $\boldsymbol{x}_t$ containing about 10–30 feature values.

It would now seem completely straightforward to classify each frame indepenently by choosing a state index $d(\boldsymbol{x}_t) = i_t$, indicating the particular class of speech sound that the speaker was most probably pronouncing at time t. The a priori probability $P_S(j)$ of various speech sounds can be measured for this particular language, and the conditional density functions for the feature vector can also be estimated. The simple MAP decision rule would then guarantee minimum probability of classification error. However, this design is too simple, and the classifier would perform very poorly.

The problem here is the fact that the state S_t of the signal source is not statistically independent of previous and following states. Some sound sequences like /mama/ may occur in the speaker's natural language, whereas sequences like /snrpt/ are probably impossible in all languages. The speech recognizer will perform much better if it is allowed to make use of this kind of knowledge.

Theoretically, we have already solved this problem. We only need to introduce a small notational change: We now assume that the signal source selects a random discrete *state sequence*

$$\underline{S} = (S_1 \dots S_t \dots S_T) \tag{5.1}$$

where each state S_t in the sequence can assume a value from a finite set of N possible states. The classifier observes a corresponding *feature-vector sequence*

$$\underline{\boldsymbol{x}} = (\boldsymbol{x}_1 \dots \boldsymbol{x}_t \dots \boldsymbol{x}_T) = \begin{pmatrix} x_{11} & \dots & x_{1t} & \dots & x_{1T} \\ \vdots & & \vdots & & \vdots \\ x_{K1} & \dots & x_{Kt} & \dots & x_{KT} \end{pmatrix}. \tag{5.2}$$

The classifier selects a decision sequence

$$\underline{d}(\underline{\boldsymbol{x}}) = (d_1 \dots d_t \dots d_T), \tag{5.3}$$

where each decision index $d_t \in \{1 \dots N\}$ indicates one of the possible source states.

Mathematically this slight change in notation introduces nothing new. As each state S_t and each decision index d_t can have only a finite number of integer values, the total number of possible state sequences, and the total number of decision sequences, is still finite. Thus, in principle, Fig. 3.1 might still be used to describe this situation. The only new thing is that the total number of possible source state sequences, $N_s = N^T$, can now be enormously large, although finite.

Minimum probability of error is still guaranteed by the well-known MAP decision rule:

$$\underline{d}(\underline{\boldsymbol{x}}) = \underset{\underline{i}}{\operatorname{argmax}}\, f_{\underline{\boldsymbol{X}}|\underline{S}}(\underline{\boldsymbol{x}} \mid \underline{i}) P_{\underline{S}}(\underline{i}) \tag{5.4}$$

The only difference from the previous form of the MAP rule in Eq. (3.15) is the vector notation for $\underline{d}$ and $\underline{i}$, used to emphasize that we are now considering sequences instead of single values. This decision rule states that the decision mechanism must search among all possible state sequences $\underline{i} = (i_1 \dots i_t \dots i_T)$.

In practice however, this simple formula implies an enormous amount of computation. If there are N possible states at each time t, there are N^T possible state sequences to consider. Even for a very short utterance with a duration of, say, $T = 100$ frames, it would be completely impossible for any computer to search among all the possible state sequences.

Fortunately, there are methods to simplify this problem into something that can be solved in practice. The key to a solution is to assume a simple model of the dependence between subsequent source states. The so-called *Hidden Markov Models (HMM)* make it much easier to solve the problem. Hidden Markov models are important for three main reasons:

1. HMMs have a very simple mathematical structure.
2. The HMM structure is still rich enough to describe also very complicated feature-vector sequences.
3. HMMs can be automatically adapted to given training data.

Hidden Markov models are used to characterise signals in widely different areas: Bach music, chaotic sea-water waves, bird song-styles, speech, handwriting, and gestures. Good introductions to Hidden Markov models are given by Rabiner and Juang (1986) and Rabiner (1989), and somewhat deeper reviews by Ephraim and Merhav (2002) and Ephraim and Roberts (2005).

5.1 Hidden Markov Model – Definition

When a random pattern sequence $\underline{\boldsymbol{X}} = (\boldsymbol{X}_1 \dots \boldsymbol{X}_t \dots \boldsymbol{X}_T)$ is described by a hidden Markov model, we imagine that the sequence has been generated as shown in Fig. 5.1. The internal state S_t controls the switch to determine which of the N internal sub-sources is connected to the output at time t. In a speech-recognition application, for example, sub-source #1 may generate /a/-like speech sounds, sub-source #2 /s/-like sounds, etc.

The model is called "hidden Markov", because the state sequence $\underline{S} = (S_1 \dots S_t \dots S_T)$ is a Markov chain[1], and this sequence is hidden inside the generator. Only the output sequence $\underline{\boldsymbol{X}}$ can be observed. All sub-sources

[1] A (first-order) *Markov chain* is a sequence of random variables, such that each variable in the sequence is statistically dependent only on its nearest predecessor (Råde and Westergren, 1995, Sec. 17.3).

can be noisy, i.e., they generate continuous-valued random vectors which must be characterised by probability density functions. There may also be noise sources which are common to all sub-sources.

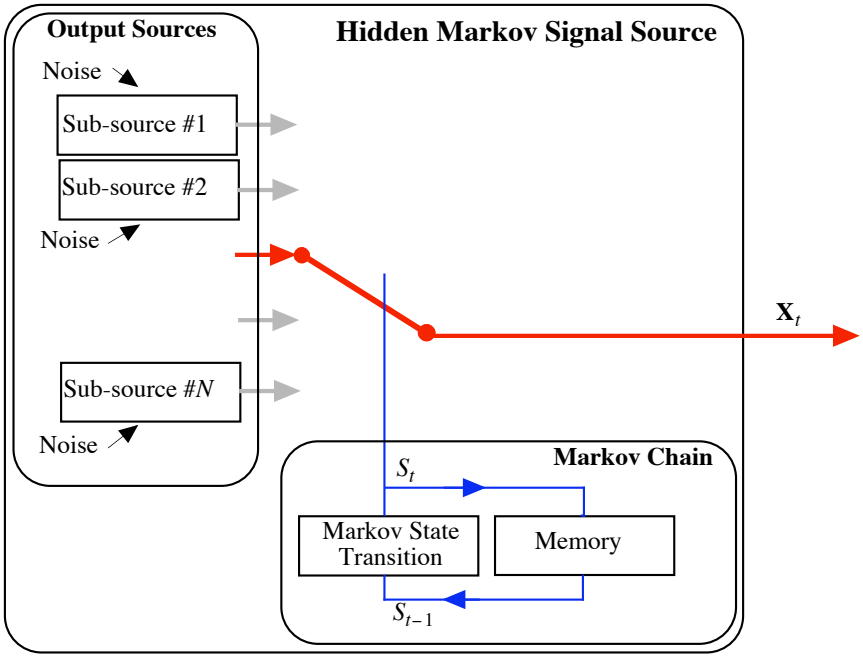


Figure 5.1: A generator of random vector sequences, described as a hidden Markov model (HMM). The HMM consists of two objects: a discrete Markov chain, and an array of output sub-sources, one for each Markov-chain state. Each sub-source is characterized by its probability distribution. If the Markov-chain state is $S_t = j$, the current feature vector $\boldsymbol{X}_t$ is generated from the j-th sub-source. The state S_t is generated as a first-order Markov chain, i.e., it depends only on the previous state S_{t-1}. The effects of any transduction and feature-extraction processing are included in the sub-source characteristics and are not explicitly shown in the figure.

Three key factors make the HMM very simple to apply:

1. All sub-sources are stationary.
2. Sub-sources do not influence each other, i.e., any correlation over time is caused only by the hidden state sequence.
3. The state sequence is a time-invariant (also called homogeneous) Markov chain, i.e., the probability distribution of state S_t depends only on the previous state S_{t-1}, and this dependence is time-invariant.

A HMM source can generate either a *stationary* or a *non-stationary* random process, because the hidden state sequence can be stationary or non-stationary, although all parameters defining the HMM are *time-invariant*.

A HMM can even be defined to generate strictly *deterministic* sequences, periodic or aperiodic, although the most useful application is for random sequences. These are just a few examples of the modelling power of the hidden-Markov structure.

The crucial assumptions behind the hidden Markov model are sometimes only approximately valid. For example, if a speech signal is recorded in a room with reverberation (echoes), the sound in each time frame may be contaminated by echoes from previous frames. Then the signal correlation over time is caused not only by the hidden state sequence, as required by the model. Nevertheless, the HMM may still work acceptably well as an engineering approximation. Often, the mathematical simplicity speaks in favor of using the HMM, even when we know it is not strictly valid.

As illustrated in Fig. 5.1, any HMM can be defined by a set λ containing two time-invariant objects, a *Markov Chain* (MC), and an array of *Output Probability Distributions* (B) with one distribution for each possible value of the discrete Markov-chain state. The Markov chain is defined by two quantities, an initial state probability distribution q, and a transition probability matrix A.

$$\boxed{\lambda = \{MC, B\} = \{\{q, A\}, B\}}$$

For simplicity, the HMM parameter set is often written as

$$\boxed{\lambda = \{q, A, B\}}$$

The Markov-chain parameters $\{q, A\}$ define the characteristics of the state sequence, and B defines the characteristics of each of the sub-sources:

Initial state probability distribution: A vector q, with N elements

$$q_j = P\left[S_1 = j\right] \tag{5.5}$$

defines the probability-mass distribution for the first state in the sequence.

Transition probability matrix: A matrix A, with N rows, and elements

$$a_{ij} = P\left[S_{t+1} = j \mid S_t = i\right] \tag{5.6}$$

defines the conditional probabilities for transitions from a state i to another state j. This matrix defines the dependence between states over time, but the matrix itself is constant and independent of t.

Output probability distributions: B is an array of N conditional probability density functions, defining the probability density for observing a feature-vector $\boldsymbol{X}_t = \boldsymbol{x}$, for any given state $S_t = j$,

$$b_j(\boldsymbol{x}) = f_{\boldsymbol{X}_t|S_t}(\boldsymbol{x} \mid j). \tag{5.7}$$

These functions define the characteristics of the sub-sources at any time t, but the functions themselves are fixed and invariant with t.

5.1.1 Continuous-valued observations

The mathematical form of the observation probability density functions $b_j(\boldsymbol{x})$ can be chosen in many ways, depending on the application. As any density function can be approximated by a weighted sum of Gaussian densities, a very general and commonly used form is the *Gaussian Mixture Density* (GMM). If the feature vectors have K elements, the Gaussian mixture can be expressed as a weighted sum of M different K-dimensional Gaussian density functions:

$$b_j(\boldsymbol{x}) = \sum_{m=1}^{M} w_{jm} \frac{1}{(2\pi)^{K/2}\sqrt{\det C_{jm}}} e^{-\frac{1}{2}(\boldsymbol{x}-\mu_{jm})^T C_{jm}^{-1}(\boldsymbol{x}-\mu_{jm})} \tag{5.8}$$

The weight factors must be chosen such that

$$\sum_{m=1}^{M} w_{jm} = 1 \tag{5.9}$$

A GMM can be used either as a stand-alone model of any feature-vector distribution, or associated with an HMM state. In either case, efficient methods are available to adapt GMM:s to observed training data, as discussed in Sec. 7.5 and 7.6.5.

The general GMM state distributions in Eq. 5.8 include NM Gaussian component functions, which require a large number of model parameters to be specified. Therefore, a model with *tied observation densities* is sometimes used. Then, the Gaussian component functions are the same for all states, and only the weight factors are state-specific. This HMM variant is sometimes called *semi-discrete*. The tied state-conditional density functions can be expressed as

$$b_j(\boldsymbol{x}) = \sum_{m=1}^{M} w_{jm} \frac{1}{(2\pi)^{K/2}\sqrt{\det C_m}} e^{-\frac{1}{2}(\boldsymbol{x}-\mu_m)^T C_m^{-1}(\boldsymbol{x}-\mu_m)} \tag{5.10}$$

Here, only M different Gaussian components need to be specified. Typically, M in Eq. (5.10) is greater than in the general GMM of Eq. (5.8), but the total number of parameters can, nevertheless, be smaller.

5.1.2 Discrete-valued (quantized) observations

In some applications it is convenient to use a discrete approximation to the observation probability density functions. The space of possible output vectors is then divided into distinct regions V_m, and the probability for the output vector to fall into each region is simply tabulated in a matrix. In this case we have a *discrete HMM*, where the observation probabilities are defined by an

Observation probability matrix: The probability mass distribution of the discrete output values is defined by a matrix B with elements

$$b_{jm} = P\left[\boldsymbol{X}_t \in V_m \mid S_t = j\right] \tag{5.11}$$

The regions V_m are often defined as the so-called Voronoi regions of a *Vector Quantizer (VQ)*. A vector quantizer is defined by a *codebook* defining a set of points in K-dimensional space. The VQ encodes any vector $\boldsymbol{x}$ by the integer index of the nearest codebook vector found in the codebook.

A Voronoi region V_m is located closely around one codebook vector and contains all points $\boldsymbol{x}$ that are closer to the m-th codebook vector than to any other vector in the codebook. Thus, the Vector Quantizer encodes each continous-valued random output vector $\boldsymbol{X}_t$ into a a corresponding random integer Z_t:

$$\boldsymbol{X}_t \in V_m \quad \Leftrightarrow \quad Z_t = m \tag{5.12}$$

There are simple algorithms to adapt a codebook $\boldsymbol{C}$ to any distribution of random vectors $\boldsymbol{X}$, to miminize the average loss of precision caused by the encoding (Gersho and Gray, 1992; Linde et al., 1980). If a VQ is used we can regard the integer sequence $\underline{Z} = (Z_1 \ldots Z_T)$ as the HMM output sequence. Then, using Eqs. (5.11) and (5.12), the elements of the observation probability matrix can be defined as

$$b_{jm} = P\left[Z_t = m \mid S_t = j\right] \tag{5.13}$$

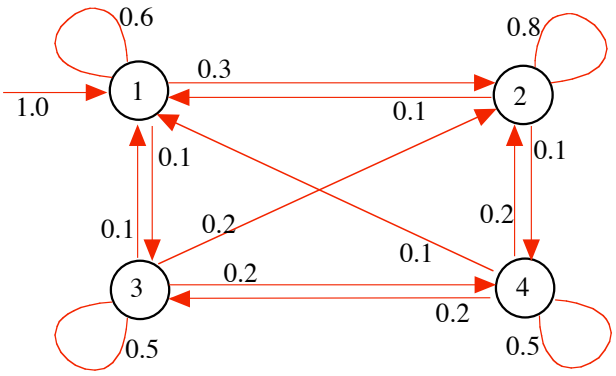


Figure 5.2: A state graph describing an example of a Markov chain with four possible states, with initial probability vector and transition matrix as specified in Eq. (5.14), according to the definition in Eq. (5.6). The arrows in the graph represent transitions with non-zero probabilities.

5.2 HMM Representations

The Markov chain characterising the state sequence can be visualised in at least three different graph types, a state graph (Fig. 5.2), a state-time graph (Fig. 5.3), or a Bayesian network graph (Fig. 5.4). All these graph examples illustrate a Markov state sequence defined by q and A:

$$q = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.6 & 0.3 & 0.1 & 0 \\ 0.1 & 0.8 & 0 & 0.1 \\ 0.1 & 0.2 & 0.5 & 0.2 \\ 0.1 & 0.2 & 0.2 & 0.5 \end{pmatrix} \tag{5.14}$$

Note that the sum of every row in the transition matrix must equal 1, because each row defines a conditional probability-mass distribution for the next state.

A *state graph* shows the possible states and the transitions between them without any time reference. A *state-time graph* illustrates all possible state values as a function of time, and the possible transitions between states. A *Bayesian network* graph gives a more abstract view of the statistical dependencies between elements in the state sequence and the observation sequence.

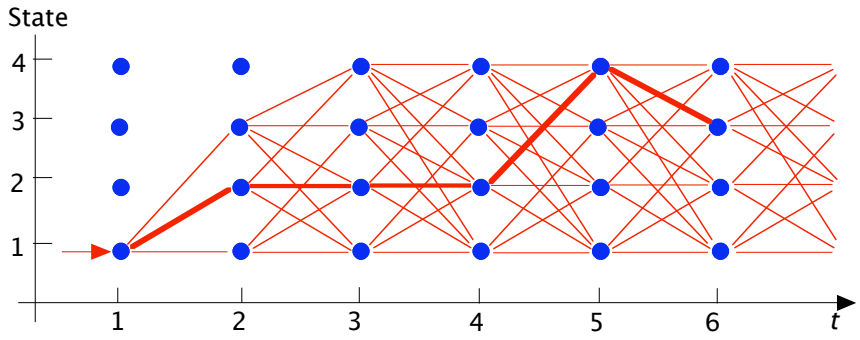


Figure 5.3: A state-time graph, describing a Markov chain with four possible states at each time sample, with initial probability vector and transition matrix defined in Eq. (5.14). The thicker line illustrates the first six elements in a particular state sequence $\underline{S} = (1, 2, 2, 2, 4, 3, \ldots)$. The graph includes only transitions with non-zero probabilities.

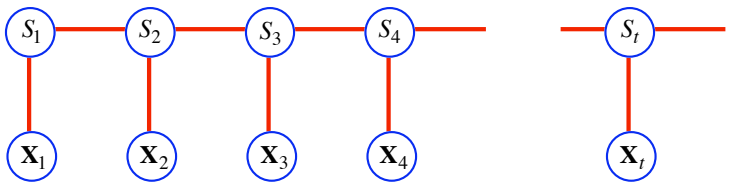


Figure 5.4: Statistical dependence between the random variables in a Hidden Markov Model, illustrated as an undirected graph. Here each node represents a random variable, but the possible outcomes are not explicitly shown. The lines between nodes illustrate direct statistical dependence. For example, there is an indirect statistical dependence between $\boldsymbol{X}_4$ and S_3, but this dependence is mediated completely by S_4. Given a known S_4, the conditional probability distribution of $\boldsymbol{X}_4$ is completely determined, and knowing also S_3 contributes no additional information about the distribution of $\boldsymbol{X}_4$. Thus, for example, $P\left[\boldsymbol{X}_4 \mid S_3, S_4\right] = P\left[\boldsymbol{X}_4 \mid S_4\right]$, and $P\left[S_3 \mid S_1, S_2, S_4, \boldsymbol{X}_3, \boldsymbol{X}_4\right] = P\left[S_3 \mid S_2, S_4, \boldsymbol{X}_3\right]$

5.3 Hidden Markov Model Structures

Hidden Markov models can describe very different types of signals, stationary as well as non-stationary, with finite or infinite duration.

In one application, an HMM might be used to characterize a particular type of sound environment, such as "traffic noise". The traffic noise can in principle go on for ever, and the random modulation pattern of the noise has no specific start or end, although any recording of the sound must of course have a finite duration. This kind of signal would be modelled by an HMM with *infinite duration*, discussed in Sec. 5.3.1.

Another HMM can describe the sequence of sound patterns in a recording of a particular word, for example /faiv/. This recording probably starts with a silent segment before the /f/ sound, then comes the /a/ sound, etc., in a specific order. The recording probably ends with another silent segment after the /v/ sound, and then the word comes to an end. Such a word is best described by an HMM with *finite duration* (Sec. 5.3.2).

These two types of HMM have rather different structures. The "traffic noise" model might have a state graph like the one in Fig. 5.2. This type of HMM has *infinite* duration and also an *ergodic* structure (Sec. 5.3.5).

The state graph for the word /faiv/ has severe restrictions on the possible transitions, as illustrated in Fig. 5.7. This HMM has *finite* duration and a *left-right* structure (Sec. 5.3.3).

Finite-duration left-right models and infinite-duration ergodic models are most common in practical applications. The relation between the different

types is illustrated schematically in Fig. 5.5.

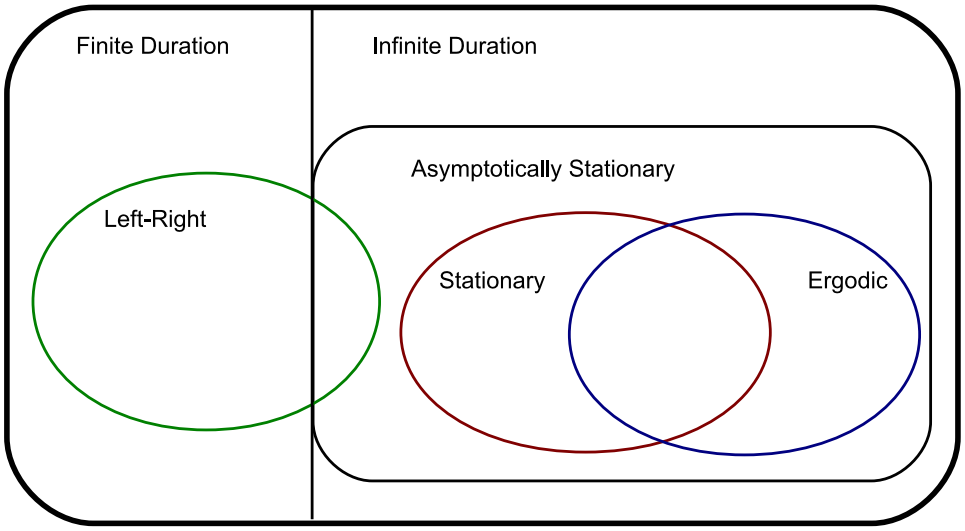


Figure 5.5: Subset relations among variants of Markov chains.

5.3.1 Infinite Duration

In an HMM with *infinite* duration the internal Markov state sequence never stops. At every state transition $t \to t+1$, the next state is chosen at random among the available N states, as $S_{t+1} \in \{1 \dots N\}$. All possible state transitions are indicated by the values in the state transition probability matrix. The transition probability matrix A is square, and

$$\sum_{j=1}^{N} a_{ij} = 1, \quad \text{for any } i \tag{5.15}$$

The following example specifies an infinite-duration Markov chain with only two states. This chain will continue for ever, alternating between states 1 and 2.

$$q = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.7 & 0.3 \\ 0.1 & 0.9 \end{pmatrix} \tag{5.16}$$

5.3.2 Finite Duration

To model sequences with *finite* duration, we must introduce a special *exit* state. As soon as the internal Markov chain reaches the exit condition, no observable output is generated, and the process simply stops. If the HMM has N states it is convenient to represent the exit condition as an additional "state" identified as number $N+1$. At every state transition $t \to t+1$,

the next state can now be chosen at random as $S_{t+1} \in \{1 \ldots N, N+1\}$. If $S_{T+1} = N + 1$ the process stops, and the last generated observable output from the HMM is $\boldsymbol{x}_T$.

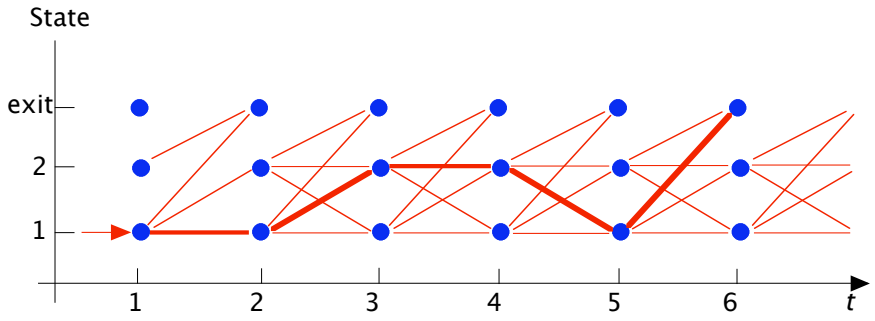


Figure 5.6: A state-time graph, describing a *finite-duration* Markov chain with two possible states at each time sample. The lines illustrate transitions with non-zero probabilities. The thicker line illustrates a particular finite state sequence $\underline{S} = (1, 1, 2, 2, 1)$ with 5 elements. At time $t = 6$, the process reached the exit state and stopped without producing any output.

The following example, and Fig. 5.6, specifies a finite-duration HMM with two states. It resembles the previous example, but there is now a finite non-zero probability at every transition, that the process will go to the special exit state.

$$q = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.7 & 0.25 & 0.05 \\ 0.1 & 0.8 & 0.1 \end{pmatrix} \tag{5.17}$$

5.3.3 Left-right HMM

Definition 5.1 (Left-right HMM): *A left-right HMM implements special restrictions on allowable state transitions, that can be described by a state graph, such as in Fig. 5.7, allowing transitions to a state only from itself or from states to the left of itself.* □

State numbers are irrelevant for this definition, but the states can always be re-numbered so that the state number *never decreases* at any allowed transition. The transition probability matrix A is then upper triangular. If, in addition, no state in the chain can be by-passed, the model is known as a *no-skip* left-right HMM. In these models, only the matrix elements on and immediately above the diagonal of A can be nonzero.

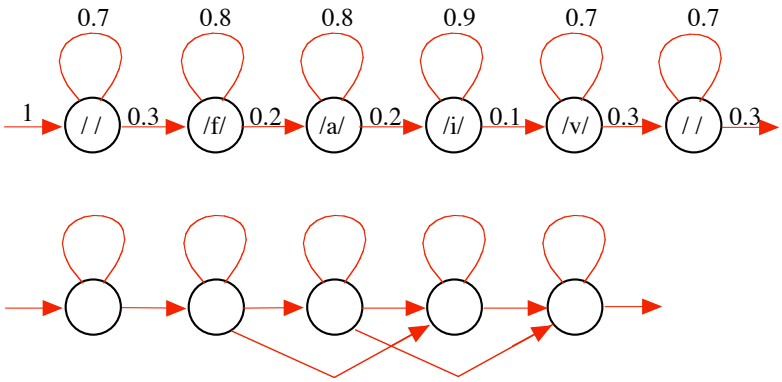


Figure 5.7: Two examples of state graphs for finite-duration *left-right* hidden Markov models. The rightmost transition arrow represents the *exit* condition. The top graph represents a word as a sequence of six sounds that must occur in a specified order. The lower left-right graph allows some states to be by-passed.

A left-right HMM can have either finite or infinite duration. If the duration is infinite, the process must inevitably converge to the rightmost state and then remains in this state for ever. This state is called a *final* or *absorbing* state. In most practical applications where a left-right HMM is appropriate, the model is designed to have *finite* duration.

The state graph in the upper part of Fig. 5.7 illustrates a left-right HMM with 6 states, defined as

$$q = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.7 & 0.3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.8 & 0.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.8 & 0.2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.9 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.7 & 0.3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.7 & 0.3 \end{pmatrix} \tag{5.18}$$

5.3.4 Stationary HMM

The state probabilities at time $t+1$ are determined by the state probabilities at t, in combination with the transition probabilities, as

$$\begin{aligned} P\left[S_{t+1} = j\right] &= \sum_{i=1}^{N} P\left[S_{t+1} = j \cap S_t = i\right] = \\ &= \sum_{i=1}^{N} \underbrace{P\left[S_{t+1} = j \mid S_t = i\right]}_{a_{ij}} P\left[S_t = i\right] \end{aligned} \tag{5.19}$$

Let us denote the state probabilities at time t by a (column) vector $\boldsymbol{p}_t$ with elements $p_{t,j} = P[S_t = j]$. Then the state probability transitions can be expressed in a compact matrix notation as

$$\boldsymbol{p}_{t+1}^T = \boldsymbol{p}_t^T A \quad \Leftrightarrow \quad \boldsymbol{p}_{t+1} = A^T \boldsymbol{p}_t \tag{5.20}$$

The state probabilities after n steps can then be written simply as

$$\boldsymbol{p}_{t+n}^T = \boldsymbol{p}_t^T A^n \tag{5.21}$$

Definition 5.2 (Stationary State Distribution): *A stationary state probability distribution $\boldsymbol{p}$ with elements $p_j = P[S_t = j]$ does not change with time t, i.e., the stationary state probabilities must satisfy the equation*

$$\boldsymbol{p}^T = \boldsymbol{p}^T A \quad \Leftrightarrow \quad \boldsymbol{p} = A^T \boldsymbol{p} \tag{5.22}$$

□

In other words, any *eigenvector* of the transposed transition probability matrix A^T, with eigenvalue 1, is a possible stationary probability vector. If several eigenvalues are 1, any linear combination of the corresponding eigenvectors can be a stationary probability vector.

5.3.5 Ergodic HMM

Many natural signals can be modelled as *ergodic* random processes. This means that every single realisation of the process will sooner or later display any possible pattern of the process, if only the realisation is studied long enough. For example, it may be reasonable to model "traffic noise at Valhallavägen" or "human babble noise" as ergodic processes. A time average of some statistical parameter obtained from a single realisation will converge asymptotically toward the corresponding true expected value, with increasing duration of the observed realisation.

A *non-ergodic* random process has a tendency to get locked up in one of several different "modes". Once a realisation of the process has entered one of these modes, it will stay there and cannot change into another mode. This means that we cannot obtain information about the general statistical characteristics of the process by studying just one realisation of the process, even if we have a very long recording of this particular realisation. For example, "human speech" may be modelled as a non-ergodic process, because we cannot observe all possible speech characteristics by recording the sound of just one single speaker. If this speaker happened to be a woman, we would never be able to measure the characteristics of male speech.

Definition 5.3 (Ergodic): *A Markov chain is called ergodic, if the probability of getting from state j to any state k in n steps converges asymptotically to the same non-zero value for any j, i.e.,*

$$\lim_{n\to\infty} P[S_{t+n} = k \mid S_t = j] = u_k > 0, \quad \textit{independent of } j \tag{5.23}$$

□

This means that the transition probability matrix for n steps, A^n, must approach a matrix with all rows equal and with all elements non-zero, i.e.,

$$\lim_{n\to\infty} [A^n]_{jk} = u_k > 0, \quad \text{independent of } j \tag{5.24}$$

It can be shown that the probability vector $\boldsymbol{u}$, with elements u_k, is a stationary probability vector, and this is the only stationary probability vector.

In an ergodic HMM the state probabilites converge asymptotically towards *one unique* state probability distribution, regardless of the initial state distribution[2]. An ergodic HMM must, of course, be designed for *infinite* duration. Non-ergodic infinite-duration hidden Markov models may converge to one of several different stationary state distributions, depending on the actual initial state.

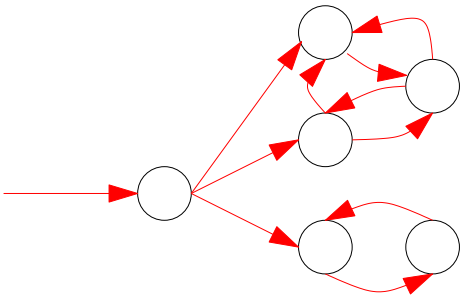


Figure 5.8: State transition graph for an example of a *reducible* Markov chain. This chain is not ergodic. Is it stationary?

Definition 5.4 (Irreducible): *A Markov chain is called irreducible, if it is possible to go from any state to any other state with non-zero probability in a finite number of steps.* □

All states are said to be *communicating* in an irreducible Markov chain. A *reducible* Markov chain has some disjoint subsets of states and allows transitions only within each subset but not from one subset to another (see example in Figs. 5.8 and 5.9). Such an HMM will obviously get locked up for ever once it entered one of these state subsets.

Definition 5.5 (Period): *A Markov-chain state has period d, if it is possible to return to the same state with non-zero probability only in some multiple of d steps, and d is the largest integer with this property.* □

[2] If the initial state distribution differs from the asymptotic final distribution, the HMM is *not stationary* from the start, only asymptotically stationary after many transitions.

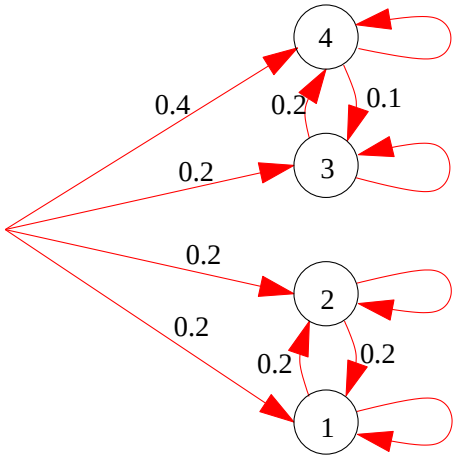


Figure 5.9: Another example of a *reducible*, non-ergodic, Markov chain, similar to the one in Fig. 5.8. It is stationary. Some other initial state distributions can also make it stationary.

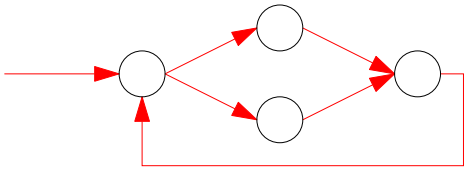


Figure 5.10: State transition graph for an example of a *periodic* Markov chain with period 3. This chain is not ergodic.

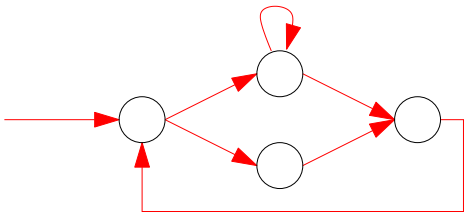


Figure 5.11: In this example, a single additional transition path with non-zero probability prevents the periodicity of the Markov chain in Fig. 5.10. This Markov chain is *aperiodic* and *irreducible*, and, therefore, *ergodic*.

For example, if it is possible to return to the state only in an even number of steps, the state has period $d = 2$. Another example is shown in Fig. 5.10. A *periodic* Markov chain has some disjoint subsets of states and follows a periodic sequence of transitions between these subsets. It may be in one

subset at time t, in a different subset at $t+1$, and can return to the first subset only after a multiple of d steps, where $d > 1$ is the period. For such a chain the initial state determines for ever in which phase of the periodic sequence the chain will be at any time t.

Definition 5.6 (Aperiodic): *A Markov chain is called* aperiodic *if all its states have period* $d = 1$. □

An aperiodic Markov chain can return to the same state with non-zero probability in n steps, for any value of n greater than some finite lower limit.

Theorem 5.1: *It can be shown that an* irreducible *and* aperiodic *Markov chain with* finite *number of states is guaranteed to be* ergodic. □

Although the HMM structure implies that the model is "started" at a particular time index $t = 1$, a stationary and ergodic random process can be observed from any starting time. The absolute time index t is not relevant. We assume that the process has the same characteristics at any time $-\infty < t < \infty$. Such a process can be modeled by an ergodic HMM. If we simply define the "initial" probability vector q as the stationary state distribution, we can always pretend that the HMM has been running from time $t = -\infty$, although we can of course only observe output sequences with finite duration.

The initial probability vector q for an ergodic stationary HMM is completely determined by the transition probability matrix A, and it is not a free design parameter of the HMM.

5.4 Probability of an Observed Sequence – the Forward Algorithm

5.4.1 Problem Introduction

Assume that we have a set of different hidden Markov source models $\lambda_m = \{q_m, A_m, B_m\}$, $m = 1 \ldots M_s$, which represent M_s different classes of pattern sequences, for example different spoken words. We want to design a word recognition system, i.e., a classifier that can observe a pattern sequence $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1 \ldots \boldsymbol{x}_t \ldots \boldsymbol{x}_T)$, with duration T, and guess which of the M_s words was most probably pronounced. We must then regard the observed sequence as one realisation of a random process $\underline{\boldsymbol{X}}$ defined by one of the available hidden Markov models. If all words are a priori equally probable, the classifier must use the ML decision rule and decide

$$d(\underline{\boldsymbol{x}}) = \underset{m}{\operatorname{argmax}}\, P\left[\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}} \mid \lambda_m\right] = \underset{m}{\operatorname{argmax}} \log P\left[\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}} \mid \lambda_m\right] \qquad (5.25)$$

If we are using continuous-valued HMMs, $P[\,]$ here actually represents a probability *density*, and should properly have been written as $f_{\underline{\boldsymbol{X}}|W}(\underline{\boldsymbol{x}} \mid \lambda_m)$, where W denotes the random word that might have been pronounced. However, in the following we are discussing either discrete or continuous HMMs and will therefore use the short-hand notation introduced here, as the context will make it clear what is meant.

We must obviously design the classifier to calculate the conditional probability density of the observed sequence, given each of the possible hidden Markov sources. The problem is, of course, similar for each source:

Problem, infinite: Given an *infinite-duration* HMM λ and a particular observed data sequence $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1 \dots \boldsymbol{x}_T)$, calculate the probability density $P[\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}} \mid \lambda]$.

Problem, finite: Given a *finite-duration* HMM λ and a particular observed data sequence $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1 \dots \boldsymbol{x}_T)$, we assume that the source came to a stop after the observed sequence, i.e., the source reached the special exit state at time $t = T + 1$. Therefore, we should calculate the probability density $P[\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}} \cap S_{T+1} = exit \mid \lambda]$.

If we try to solve this problem mechanically, we must evaluate a nested sum over all the possible state sequences $(i_1 \dots i_t \dots i_T)$ that may have caused the observed sequence. Each state in the sequence can have N different values, and the nested sum can be explicitly written as

$$P[\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}} \mid \lambda] = \\ = \sum_{i_1=1}^{N} q_{i_1} b_{i_1}(\boldsymbol{x}_1) \sum_{i_2=1}^{N} a_{i_1 i_2} b_{i_2}(\boldsymbol{x}_2) \cdots \sum_{i_T=1}^{N} a_{i_{T-1} i_T} b_{i_T}(\boldsymbol{x}_T) \quad (5.26)$$

Fortunately, we will never need to perform this summation of N^T terms. Instead, the *Forward Algorithm* gives us exactly the same result using only about $N^2 T$ operations.

From now on, the notation for the event $\boldsymbol{X}_t = \boldsymbol{x}_t$ is often simplified as just $\boldsymbol{x}_t$, when the meaning is evident anyway. Similarly, we use a shorthand notation for the probability of sequences of events, as

$$P[\boldsymbol{X}_1 = \boldsymbol{x}_1 \cap \boldsymbol{X}_2 = \boldsymbol{x}_2 \cdots] = P[\boldsymbol{x}_1, \boldsymbol{x}_2 \cdots] = P[\boldsymbol{x}_1 \boldsymbol{x}_2 \cdots]$$

To find $P[\boldsymbol{x}_1 \cdots \boldsymbol{x}_T \mid \lambda]$ we first observe, using Bayes rule repeatedly, that

$$P[\boldsymbol{x}_1 \cdots \boldsymbol{x}_t \mid \lambda] = P[\boldsymbol{x}_1 \mid \lambda]\, P[\boldsymbol{x}_2 \mid \boldsymbol{x}_1, \lambda] \cdots P[\boldsymbol{x}_t \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda] \quad (5.27)$$

As all the probabilities in this product are < 1, the resulting quantity gets smaller and smaller with increasing t. For t-values like 100 or

1000 it may not be possible to store the total probability using the computer's standard numeric representation. Therefore, it is better to calculate $P\left[\boldsymbol{x}_t \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda\right]$ for each t in a vector, and finally calculate

$$\ln P\left[\boldsymbol{x}_1 \cdots \boldsymbol{x}_T \mid \lambda\right] = \ln P\left[\boldsymbol{x}_1 \mid \lambda\right] + \sum_{t=2}^{T} \ln P\left[\boldsymbol{x}_t \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda\right] \tag{5.28}$$

In other applications, for example sound source classification in real time, we need a current updated estimate of the state probabilities of the HMM source, given the present and all previous observations. We must then calculate, recursively for each t, a vector $\hat{\boldsymbol{\alpha}}_t$ with elements

$$\hat{\alpha}_{j,t} = P\left[S_t = j \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_t, \lambda\right] \tag{5.29}$$

As we shall soon see, the Forward Algorithm produces both quantities, $P\left[\boldsymbol{x}_1 \cdots \boldsymbol{x}_T \mid \lambda\right]$ and $\hat{\boldsymbol{\alpha}}_t$, in one very efficient calculation procedure. The key idea behind this algorithm is to organise the computation as illustrated in Fig. 5.12.

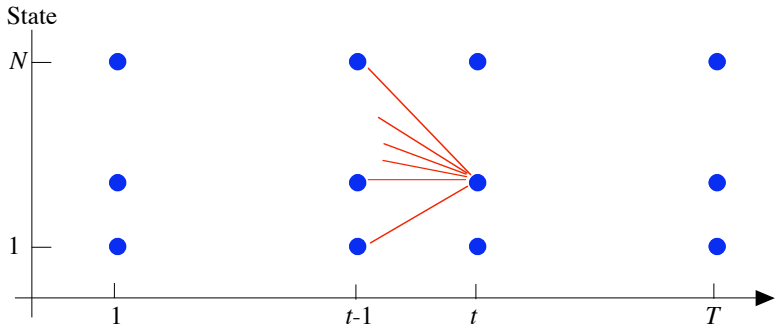


Figure 5.12: A partial state-time graph, illustrating one recursive step of the forward algorithm, from $t-1$ to t. Knowing the state probabilities $\hat{\alpha}_{i,t-1}$ for all states i at time $t-1$, it is easy to calculate the probability of being in any particular state j at time t.

5.4.2 Forward Variables

Definition 5.7 (Forward Variable): *The forward variable is a vector*

$$\boldsymbol{\alpha}_t = \begin{pmatrix} \alpha_{1,t} \\ \vdots \\ \alpha_{j,t} \\ \vdots \\ \alpha_{N,t} \end{pmatrix}, \quad \text{where } \alpha_{j,t} = P\left[\boldsymbol{X}_1 = \boldsymbol{x}_1 \cdots \boldsymbol{X}_t = \boldsymbol{x}_t, S_t = j \mid \lambda\right] \tag{5.30}$$

□

This variable is used for theoretical purposes, but it is not well suited for practical computation, because it gets smaller and smaller with increasing t. Therefore, we use a scaled version of the forward variable.

Definition 5.8 (Scaled Forward Variable): *The scaled forward variable is a vector $\hat{\boldsymbol{\alpha}}_t$, with elements*

$$\hat{\alpha}_{j,t} = P\left[S_t = j \mid \boldsymbol{X}_1 = \boldsymbol{x}_1 \cdots \boldsymbol{X}_t = \boldsymbol{x}_t, \lambda\right] \tag{5.31}$$

□

The scaled forward variable $\hat{\boldsymbol{\alpha}}_t$ can be seen simply as a normalized version of $\boldsymbol{\alpha}_t$, such that

$$\sum_{j=1}^{N} \hat{\alpha}_{j,t} = 1 \tag{5.32}$$

as required for any probability mass vector.

Definition 5.9 (Forward Scale Factors): *The forward scale factors form a sequence $(c_1, \ldots, c_T)$ or $(c_1, \ldots, c_T, c_{T+1})$ with elements*

$$c_t = P\left[\boldsymbol{x}_t \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda\right], \quad 1 \leq t \leq T \tag{5.33}$$

For a finite-duration *HMM, we also calculate the exit probability as*

$$c_{T+1} = P\left[S_{T+1} = N + 1 \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_T, \lambda\right] \tag{5.34}$$

□

The final scale factor c_{T+1} is used only for a finite-duration HMM and is introduced to allow the probability of the observed finite sequence to be calculated simply as

$$P\left[\boldsymbol{x}_1 \cdots \boldsymbol{x}_T \cap S_{T+1} = N + 1 \mid \lambda\right] = c_1 \cdots c_T c_{T+1} \tag{5.35}$$

The definition for both infinite and finite duration implies that

$$P\left[\boldsymbol{x}_1 \cdots \boldsymbol{x}_t \mid \lambda\right] = c_1 \cdots c_t \tag{5.36}$$

$$\ln P\left[\boldsymbol{x}_1 \cdots \boldsymbol{x}_t \mid \lambda\right] = \sum_{k=1}^{t} \ln c_k \tag{5.37}$$

The scaling is formally obtained directly from Bayes rule, as

$$\begin{aligned}\hat{\alpha}_{j,t} = P\left[S_t = j \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_t, \lambda\right] = \frac{P\left[\boldsymbol{x}_1 \cdots \boldsymbol{x}_t, S_t = j \mid \lambda\right]}{P\left[\boldsymbol{x}_1 \cdots \boldsymbol{x}_t \mid \lambda\right]} = \\ = \frac{\alpha_{j,t}}{c_1 \cdots c_t}\end{aligned} \tag{5.38}$$

To facilitate the computation it is also convenient to use the following help variable:

Definition 5.10 (Temporary forward variable): *The temporary forward variable is a vector α_t^{temp}, with elements*

$$\alpha_{j,t}^{temp} = P\left[\boldsymbol{X}_t = \boldsymbol{x}_t, S_t = j \mid \boldsymbol{x}_1 \ldots \boldsymbol{x}_{t-1}, \lambda\right] \tag{5.39}$$

□

The relation between the temporary and the scaled forward variables is easily obtained from Bayes rule, as

$$\begin{aligned}\hat{\alpha}_{j,t} = P\left[S_t = j \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \boldsymbol{x}_t, \lambda\right] = \frac{P\left[\boldsymbol{x}_t, S_t = j \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda\right]}{P\left[\boldsymbol{x}_t \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda\right]} = \\ = \frac{\alpha_{j,t}^{temp}}{\sum_{k=1}^{N} \alpha_{k,t}^{temp}}\end{aligned} \tag{5.40}$$

It should also be noted that

$$\begin{aligned}c_t = P\left[\boldsymbol{x}_t \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda\right] = \sum_{k=1}^{N} P\left[\boldsymbol{x}_t, S_t = k \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda\right] = \\ = \sum_{k=1}^{N} \alpha_{k,t}^{temp}, \quad 1 \leq t \leq T\end{aligned} \tag{5.41}$$

5.4.3 Forward Calculation Procedure

The forward algorithm is a simple iterative procedure involving only the scaled forward variable $\hat{\boldsymbol{\alpha}}_t$, the temporary forward variable $\boldsymbol{\alpha}_t^{temp}$, and the forward scale factors c_t. At each step, the algorithm uses parameters from the known hidden Markov model $\lambda = \{q, A, B\}$:

Initialization: At $t = 1$ we easily obtain

$$\alpha_{j,1}^{temp} = P\left[\boldsymbol{X}_1 = \boldsymbol{x}_1, S_1 = j \mid \lambda\right] = q_j b_j(\boldsymbol{x}_1), \quad j = 1 \ldots N \tag{5.42}$$

$$c_1 = \sum_{k=1}^{N} \alpha_{k,1}^{temp} \tag{5.43}$$

$$\hat{\alpha}_{j,1} = \alpha_{j,1}^{temp} / c_1, \quad j = 1 \ldots N \tag{5.44}$$

Forward step: At $t = 2$ we obtain, using Bayes rule:

$$\begin{aligned}\alpha_{j,2}^{temp} &= P\left[\boldsymbol{X}_2 = \boldsymbol{x}_2, S_2 = j \mid \boldsymbol{x}_1, \lambda\right] \\ &= P\left[\boldsymbol{X}_2 = \boldsymbol{x}_2 \mid S_2 = j, \boldsymbol{x}_1, \lambda\right] P\left[S_2 = j \mid \boldsymbol{x}_1, \lambda\right]\end{aligned} \tag{5.45}$$

By HMM definition, $\boldsymbol{X}_2$ is conditionally independent of $\boldsymbol{X}_1$, given S_2. Therefore, the first factor in the above expression is simply

$$P\left[\boldsymbol{X}_2 = \boldsymbol{x}_2 \mid S_2 = j, \boldsymbol{x}_1, \lambda\right] = P\left[\boldsymbol{X}_2 = \boldsymbol{x}_2 \mid S_2 = j, \lambda\right] = b_j(\boldsymbol{x}_2) \tag{5.46}$$

The second factor is

$$\begin{aligned}P\left[S_2 = j \mid \boldsymbol{x}_1, \lambda\right] &= \sum_{i=1}^{N} P\left[S_2 = j, S_1 = i \mid \boldsymbol{x}_1, \lambda\right] \\ &= \sum_{i=1}^{N} P\left[S_2 = j \mid S_1 = i, \boldsymbol{x}_1, \lambda\right] \underbrace{P\left[S_1 = i \mid \boldsymbol{x}_1, \lambda\right]}_{\hat{\alpha}_{i,1}}\end{aligned} \tag{5.47}$$

As S_2 depends only on S_1 by HMM definition, the first factor in the sum is simply

$$P\left[S_2 = j \mid S_1 = i, \boldsymbol{x}_1, \lambda\right] = P\left[S_2 = j \mid S_1 = i, \lambda\right] = a_{ij} \tag{5.48}$$

Thus, the temporary forward variable can be obtained as

$$\alpha_{j,2}^{temp} = b_j(\boldsymbol{x}_2) \left(\sum_{i=1}^{N} \hat{\alpha}_{i,1} a_{ij} \right) \tag{5.49}$$

The step from $t = 1$ to $t = 2$ is equivalent to a step from any $t-1$ to t, and the forward step can be immediately generalized, for $t = 2, \ldots, T$, as

$$\alpha_{j,t}^{temp} = b_j(\boldsymbol{x}_t) \left(\sum_{i=1}^{N} \hat{\alpha}_{i,t-1} a_{ij} \right), \quad j = 1 \ldots N \tag{5.50}$$

$$c_t = \sum_{k=1}^{N} \alpha_{k,t}^{temp} \tag{5.51}$$

$$\hat{\alpha}_{j,t} = \alpha_{j,t}^{temp} / c_t, \quad j = 1 \ldots N \tag{5.52}$$

Termination: For the special case of a *finite-duration* HMM, we need to include also the special exit condition $S_{T+1} = N + 1$ for the finite

observed sequence:

$$\begin{aligned} c_{T+1} &= P\left[S_{T+1} = N+1 \mid \boldsymbol{x}_1 \ldots \boldsymbol{x}_T, \lambda\right] \\ &= \sum_{k=1}^{N} P\left[S_T = k \cap S_{T+1} = N+1 \mid \boldsymbol{x}_1 \ldots \boldsymbol{x}_T, \lambda\right] \\ &= \sum_{k=1}^{N} \hat{\alpha}_{k,T} a_{k,N+1} \end{aligned} \tag{5.53}$$

Now the calculated sequence of scale factors can be used to obtain two different measures of the total observation probability,

either as

$$\ln P\left[\boldsymbol{x}_1 \ldots \boldsymbol{x}_T \mid \lambda\right] = \sum_{t=1}^{T} \ln c_t \tag{5.54}$$

or as

$$\ln P\left[\boldsymbol{x}_1 \ldots \boldsymbol{x}_T, S_{T+1} = N+1 \mid \lambda\right] = \sum_{t=1}^{T+1} \ln c_t \tag{5.55}$$

Eq. (5.54) is most appropriate for an *infinite-duration* HMM, but can in principle also be used for a finite-duration HMM, if the observed segment does not include the complete output sequence. Eq. (5.55) is the most appropriate, if we know that the observed sequence is the complete output of a *finite-duration* HMM.

If the total probability density of the complete observed sequence is all we need, as determined by Eq. (5.54) or (5.55), we would not actually need to save the scaled forward variable as a function of t. However, in the HMM training procedure (section 6.1) we will also need the scaled forward variables for all $t = 1 \ldots T$. For this purpose, it is convenient to let the forward algorithm store and return all the $\hat{\alpha}_{j,t}$ values in a matrix with N rows and T columns, and all the scale factors c_t in a row vector of length T or $T+1$. However, the temporary forward variable is not needed outside the forward algorithm and should therefore be hidden as a local variable inside the calculation procedure.

5.5 The Backward Algorithm

5.5.1 Problem Introduction

The previous sections assumed that the hidden Markov model λ was already available for calculations. But where did the model come from? The next

chapter will discuss how a HMM can be trained, i.e., automatically adapted to conform with given training data. The training data includes pre-recorded observed sequences like $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1 \dots \boldsymbol{x}_t \dots \boldsymbol{x}_T)$, known to be generated by the source we want to characterize by an HMM.

In the training process we will need to estimate probabilities of the hidden states, such as $P[S_t = j \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_t \dots \boldsymbol{x}_T, \lambda]$ for any t in the range $1 \le t \le T$. Note that this probability is conditional on the complete observation sequence, not only the initial part from time 1 to t. The forward algorithm, described in section 5.4, calculates the conditional probabilities $\hat{\alpha}_{j,t} = P[S_t = j \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_t, \lambda]$. However, when estimating the probability of a particular state at time $t < T$ we should also make use of the information provided by later observations $(\boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T)$ in the known training sequence. This is important, because generally

$$P[S_t = j \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_t, \lambda] \neq P[S_t = j \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_T, \lambda], \quad \text{for } t < T \tag{5.56}$$

It is quite possible that the first part of the observed data indicates that a particular state j has very high probability at time t, but later observations may force us to realise that this state had actually *zero* probability. Therefore, we need to combine the information obtained from the Forward Algorithm by the information provided by later observations. We define new computational variables $\beta_{j,t}$ and $\gamma_{j,t}$ for this special purpose.

For a *finite-duration* HMM, we include the exit condition $S_{T+1} = N+1$, and define

$$\begin{aligned}
&\gamma_{j,t} = \\
&= P[S_t = j \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_T, S_{T+1} = N+1, \lambda] \\
&= \frac{P[S_t = j, \boldsymbol{x}_1 \dots \boldsymbol{x}_t \boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T, S_{T+1} = N+1 \mid \lambda]}{P[\boldsymbol{x}_1 \dots \boldsymbol{x}_t \boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T, S_{T+1} = N+1 \mid \lambda]} \\
&= \frac{P[\boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T, S_{T+1} = N+1 \mid S_t = j, \boldsymbol{x}_1 \dots \boldsymbol{x}_t \lambda]}{c_1 \cdots c_T c_{T+1}} \underbrace{P[S_t = j, \boldsymbol{x}_1 \dots \boldsymbol{x}_t \mid \lambda]}_{\alpha_{j,t}} \\
&= \underbrace{P[\boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T, S_{T+1} = N+1 \mid S_t = j, \lambda]}_{\beta_{j,t}} \alpha_{j,t} / (c_1 \cdots c_T c_{T+1}) \\
&= \alpha_{j,t} \beta_{j,t} / (c_1 \cdots c_T c_{T+1})
\end{aligned} \tag{5.57}$$

For a *infinite-duration* HMM, a very similar relation is used, except that the exit condition $S_{T+1} = N+1$ and the extra scale factor c_{T+1} are omitted. The simplification from line 3 to line 4 of this derivation is valid because all observations $\boldsymbol{X}_{t+1} \dots$ are conditionally independent of earlier observations $\dots \boldsymbol{X}_t$, given S_t, as illustrated in Fig. 5.4.

The new variables $\beta_{j,t}$ that we introduced in Eq. (5.57), in addition to the Forward variables $\alpha_{j,t}$, are obtained using the *Backward Algorithm.* This

algorithm works backwards in time, from $t = T$ down to $t = 1$. The simple idea utilized by the backward algorithm is the following application of Bayes rule, also illustrated in Fig. 5.13:

$$P\left[\boldsymbol{X}_{t+1} = \boldsymbol{x}_{t+1} \mid S_t = i, \lambda\right] = \sum_{j=1}^{N} P\left[S_{t+1} = j, \boldsymbol{X}_{t+1} = \boldsymbol{x}_{t+1} \mid S_t = i, \lambda\right] =$$

$$= \sum_{j=1}^{N} \underbrace{P\left[S_{t+1} = j \mid S_t = i, \lambda\right]}_{a_{ij}} \underbrace{P\left[\boldsymbol{X}_{t+1} = \boldsymbol{x}_{t+1} \mid S_{t+1} = j, \lambda\right]}_{b_j(\boldsymbol{x}_{t+1})} \quad (5.58)$$

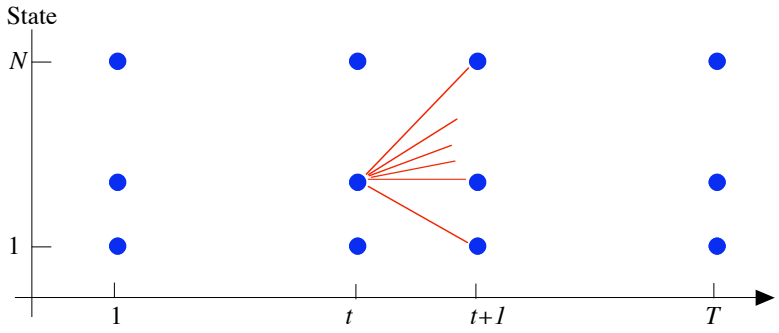


Figure 5.13: A partial state-time graph illustrating one recursive step of the backward algorithm, from $t + 1$ to t.

5.5.2 Backward Variables

Definition 5.11 (Backward Variable): *The backward variable is a vector β_t with elements defining the probability of later observations, for each state $S_t = i$ at time t:*
For an infinite-duration HMM,

$$\beta_{i,t} = P\left[\boldsymbol{X}_{t+1} = \boldsymbol{x}_{t+1} \ldots \boldsymbol{X}_T = \boldsymbol{x}_T \mid S_t = i, \lambda\right] \quad (5.59)$$

For a finite-duration HMM,

$$\beta_{i,t} = P\left[\boldsymbol{X}_{t+1} = \boldsymbol{x}_{t+1} \ldots \boldsymbol{X}_T = \boldsymbol{x}_T, S_{T+1} = N + 1 \mid S_t = i, \lambda\right] \quad (5.60)$$

□

Note that the backward variable is *not symmetric* with the forward variable. For computational purposes we also need a scaled backward variable.

Definition 5.12 (Scaled Backward Variable): *The scaled backward variable is a vector $\hat{\beta}_t$, with the following elements:*
For an infinite-duration HMM,

$$\hat{\beta}_{i,t} = \beta_{i,t}/(c_t \cdots c_T) \tag{5.61}$$

For a finite-duration HMM,

$$\hat{\beta}_{i,t} = \beta_{i,t}/(c_t \cdots c_T c_{T+1}) \tag{5.62}$$

Scale factors c_t are obtained from the Forward Algorithm. □

The definitions of the scaled forward and backward variables show that the total conditonal state probability, given the complete observed sequence, as defined in Eq. (5.57), can also be expressed directly in terms of the scaled variables, as

$$\begin{aligned}\gamma_{j,t} &= \hat{\alpha}_{j,t}\hat{\beta}_{j,t}c_t = \\ &= \begin{cases} P\left[S_t = j \mid \boldsymbol{x}_1 \ldots \boldsymbol{x}_T, \lambda\right], & \text{(infinite duration)} \\ P\left[S_t = j \mid \boldsymbol{x}_1 \ldots \boldsymbol{x}_T, S_{T+1} = N+1, \lambda\right], & \text{(finite duration)} \end{cases}\end{aligned} \tag{5.63}$$

5.5.3 Backward Calculation Procedure

Initialization: At $t = T$ we define, for reasons to be evident later,

for an infinite-duration HMM:

$$\beta_{i,T} = 1; \qquad \hat{\beta}_{i,T} = 1/c_T \tag{5.64}$$

for a finite-duration HMM λ:

$$\beta_{i,T} = a_{i,N+1}; \quad \hat{\beta}_{i,T} = \beta_{i,T}/(c_T c_{T+1}) \tag{5.65}$$

Backward step: The actual calculations use only the scaled backward variable, recursively updated as

$$\hat{\beta}_{i,t} = \frac{1}{c_t}\sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{x}_{t+1})\hat{\beta}_{j,t+1}, \quad t = T-1, T-2, \ldots, 1 \tag{5.66}$$

To see why this update algorithm is valid, we first observe, at $t = T-1$, for a finite-duration HMM,

$$\begin{aligned}
\beta_{i,T-1} &= P\left[\boldsymbol{X}_T = \boldsymbol{x}_T \cap S_{T+1} = N+1 \mid S_{T-1} = i, \lambda\right] \\
&= \sum_{j=1}^{N} P\left[\boldsymbol{X}_T = \boldsymbol{x}_T \cap S_T = j \cap S_{T+1} = N+1 \mid S_{T-1} = i, \lambda\right] \\
&= \sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{x}_T) a_{j,N+1} \\
&= \sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{x}_T) \beta_{j,T}
\end{aligned} \tag{5.67}$$

Expressed in terms of the scaled backward variables, this is equivalent to

$$\hat{\beta}_{i,T-1} = \frac{1}{c_{T-1}} \sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{x}_T) \hat{\beta}_{j,T} \tag{5.68}$$

Generalizing to any $t < T$, we use Bayes rule repatedly to obtain

$$\begin{aligned}
\beta_{i,t} &= P\left[\boldsymbol{x}_{t+1}\boldsymbol{x}_{t+2}\ldots\boldsymbol{x}_T, S_{T+1} = N+1 \mid S_t = i, \lambda\right] \\
&= \sum_{j=1}^{N} P\left[\boldsymbol{x}_{t+1}\boldsymbol{x}_{t+2}\ldots\boldsymbol{x}_T, S_{t+1} = j, S_{T+1} = N+1 \mid S_t = i, \lambda\right] \\
&= \sum_{j=1}^{N} \underbrace{P\left[S_{t+1} = j \mid S_t = i, \lambda\right]}_{a_{ij}} \cdot \\
&\qquad \cdot P\left[\boldsymbol{x}_{t+1}\boldsymbol{x}_{t+2}\ldots\boldsymbol{x}_T, S_{T+1} = N+1 \mid S_{t+1} = j, S_t = i, \lambda\right] \\
&= \sum_{j=1}^{N} a_{ij} P\left[\boldsymbol{x}_{t+1}\boldsymbol{x}_{t+2}\ldots\boldsymbol{x}_T, S_{T+1} = N+1 \mid S_{t+1} = j, \lambda\right] \\
&= \sum_{j=1}^{N} a_{ij} P\left[\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_{t+2}\ldots\boldsymbol{x}_T, S_{t+1} = j, S_{T+1} = N+1, \lambda\right] \cdot \\
&\qquad \cdot \underbrace{P\left[\boldsymbol{x}_{t+2}\ldots\boldsymbol{x}_T, S_{T+1} = N+1 \mid S_{t+1} = j, \lambda\right]}_{\beta_{j,t+1}} \\
&= \sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{x}_{t+1}) \beta_{j,t+1}
\end{aligned} \tag{5.69}$$

The equality between line 3 and 4 in this derivation is valid because, $\boldsymbol{X}_{t+1}$ is statistically independent of S_t, given S_{t+1}. Similarly, in line 5, $\boldsymbol{X}_{t+1}$ is statistically independent of $\boldsymbol{X}_{t+2}\ldots\boldsymbol{X}_T$ and S_{T+1}, given S_{t+1}. The derivation is exactly the same for an infinite-duration HMM, except that the special

exit condition $S_{T+1} = N + 1$ is not included. Expressed in terms of the scaled backward variables, the update equation is equivalent to

$$\begin{aligned}\hat{\beta}_{i,t} &= \beta_{i,t}/(c_t \cdots c_T c_{T+1}) \\ &= \frac{1}{c_t} \sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{x}_{t+1}) \hat{\beta}_{j,t+1}\end{aligned} \tag{5.70}$$

5.6 Most Probable State Sequence – the Viterbi Algorithm

5.6.1 Problem Formulation

In automatic recognition of continuous speech the HMM states can represent different speech sounds like /a/, /s/, etc., or combinations of speech sounds, or smaller segments of speech sounds. The observed feature-vector sequence represents an unknown spoken sentence with one or more words. To determine the spoken word sequence the recognizer first needs to determine the most probable sequence of speech sounds, given the observed data.

Problem, infinite: Given an *infinite-duration* HMM λ and an observed feature-vector sequence $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1 \dots \boldsymbol{x}_T)$, find the most probable underlying state sequence, using the MAP criterion:

$$\hat{\underline{i}} = \widehat{(i_1 \dots i_T)} = \underset{(i_1 \dots i_T)}{\operatorname{argmax}} P\left[S_1 = i_1, \dots, S_T = i_T \mid \boldsymbol{x}_1, \dots, \boldsymbol{x}_T, \lambda\right] \tag{5.71}$$

Problem, finite: Given a *finite-duration* HMM λ and an observed feature-vector sequence $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1 \dots \boldsymbol{x}_T)$, we assume that the source reached the special *exit* state at time $t = T + 1$. Therefore, we define the most probable underlying state sequence as:

$$\begin{aligned}\hat{\underline{i}} &= \widehat{(i_1 \dots i_T)} = \\ &= \underset{(i_1 \dots i_T)}{\operatorname{argmax}} P\left[S_1 = i_1, \dots, S_T = i_T, S_{T+1} = exit \mid \boldsymbol{x}_1, \dots, \boldsymbol{x}_T, \lambda\right]\end{aligned} \tag{5.72}$$

The MAP criterion can be formulated in several different ways, giving the same result, and we are free to choose the version that is easiest to

compute. For example, in the infinite-duration case:

$$\begin{aligned}\hat{\underline{i}} = \widehat{(i_1 \dots i_T)} &= \underset{(i_1 \dots i_T)}{\operatorname{argmax}} P\left[S_1 = i_1, \dots, S_T = i_T \mid \boldsymbol{x}_1, \dots, \boldsymbol{x}_T, \lambda\right] \\ &= \underset{(i_1 \dots i_T)}{\operatorname{argmax}} \frac{P\left[S_1 = i_1, \dots, S_T = i_T, \boldsymbol{x}_1, \dots, \boldsymbol{x}_T \mid \lambda\right]}{P\left[\boldsymbol{x}_1, \dots, \boldsymbol{x}_T \mid \lambda\right]} \\ &= \underset{(i_1 \dots i_T)}{\operatorname{argmax}} P\left[S_1 = i_1, \dots, S_T = i_T, \boldsymbol{x}_1, \dots, \boldsymbol{x}_T \mid \lambda\right] \\ &= \underset{(i_1 \dots i_T)}{\operatorname{argmax}} \log P\left[S_1 = i_1, \dots, S_T = i_T, \boldsymbol{x}_1, \dots, \boldsymbol{x}_T \mid \lambda\right]\end{aligned} \tag{5.73}$$

If the HMM has N states, and a sequence of duration T has been observed, there are N^T possible state sequences that might have generated the observed feature-vector sequence. The argmax expression implies that we must search among all these N^T state sequences. Fortunately, the computation can be greatly reduced by using the *Viterbi algorithm*, which is conceptually similar to the forward algorithm. The algorithm is a variant of dynamic programming.

The key idea of the algorithm, illustrated in fig. 5.14, is to organize the computation recursively for $t = 1 \dots T$. At any time t, there are only N possible state values $i_t \in \{1 \dots N\}$, and the optimal state sequence must pass through exactly one of these states. Therefore, at any time t there can be only N candidates for the initial part $\widehat{(i_1 \dots i_t)}$ of the optimal sequence, one candidate for each possible state $i_t \in \{1 \dots N\}$. To be able to track any of these candidate sequences backwards, we need to store, for each state, a pointer to the previous state for this particular candidate. The final decision about the complete optimal state sequence must be delayed until time $t = T$.

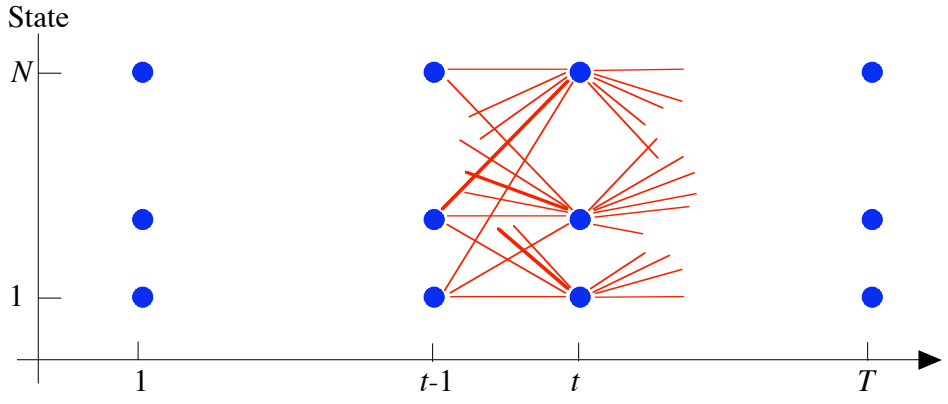


Figure 5.14: A partial state-time graph illustrating the central idea of the Viterbi algorithm. There are N^t possible partial state sequences $(i_1 \dots i_t)$ ending at time t, but only the best single one for each state need to be considered further. These N candidates are indicated by thicker lines in the diagram.

5.6.2 Viterbi Calculation Variables

To see formally why it is possible to maximize separately over partial sequences ending at time t, we can use Bayes rule to express

$$\begin{aligned} & P\left[i_1, \ldots, i_t, \ldots, i_T, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t, \ldots, \boldsymbol{x}_T \mid \lambda\right] = \\ & = P\left[i_1, \ldots, i_t, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t \mid \lambda\right] \cdot \\ & \quad \cdot P\left[i_{t+1}, \ldots, i_T, \boldsymbol{x}_{t+1}, \ldots, \boldsymbol{x}_T \mid i_1, \ldots, i_t, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t, \lambda\right] \\ & = P\left[i_1, \ldots, i_t, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t \mid \lambda\right] P\left[i_{t+1}, \ldots, i_T, \boldsymbol{x}_{t+1}, \ldots, \boldsymbol{x}_T \mid i_t, \lambda\right] \end{aligned} \tag{5.74}$$

Here, the second equality follows from the hidden-Markov property that $S_{t+1} = i_{t+1}$ and all other later state events are conditionally independent of $S_1 \ldots S_t$ and $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t)$, given $S_t = i_t$. Therefore, the maximization of the total probability can be separated as

$$\begin{aligned} & \max_{(i_1 \ldots i_T)} P\left[i_1, \ldots, i_t, \ldots, i_T, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t, \ldots, \boldsymbol{x}_T \mid \lambda\right] \\ & \quad = \max_{i_t} \underbrace{\left(\max_{(i_1 \ldots i_{t-1})} P\left[i_1, \ldots, i_t, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t \mid \lambda\right] \right)}_{\chi_{i_t,t}} \cdot \\ & \quad \cdot \left(\max_{(i_{t+1} \ldots i_T)} P\left[i_{t+1}, \ldots, i_T, \boldsymbol{x}_{t+1}, \ldots, \boldsymbol{x}_T \mid i_t, \lambda\right] \right) \end{aligned} \tag{5.75}$$

Note that the expressions in both the large parentheses are functions only of i_t, after the maximizations over all possible sequences $(i_1 \ldots i_{t-1})$ and $(i_{t+1} \ldots i_T)$ have been done. This separation shows that only one candidate value for each state i_t needs to be remembered for further consideration after time t. The central idea of the Viterbi algorithm is to calculate $\chi_{i_t,t}$ recursively, for $t = 1 \ldots T$. At time $t = T$ the second big parenthesis in the above expression is empty, and the maximization problem is solved. To implement the calculation procedure we need only one vector and one matrix that are recursively updated:

Definition 5.13 (Viterbi Probability Vector): *The partial-sequence probability vector χ_t contains one element $\chi_{j,t}$ for each possible state, and indicates the probability of the best partial candidate state sequence that ends in state $S_t = j$ at time t:*

$$\chi_{j,t} = \max_{(i_1 \ldots i_{t-1})} P\left[S_1 = i_1, \ldots, S_{t-1} = i_{t-1}, S_t = j, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t \mid \lambda\right] \tag{5.76}$$

□

Definition 5.14 (Viterbi Backpointer Matrix): *The Viterbi backpointer is a matrix ζ with elements $\zeta_{j,t}$ indicating, for each possible state $S_t = j$, the previous state S_{t-1} of the best partial candidate sequence that ends in state $S_t = j$ at time t:*

$$\zeta_{j,t} = \operatorname*{argmax}_i \chi_{i,t-1} a_{ij} \tag{5.77}$$

□

5.6.3 Viterbi Calculation Procedure

The Viterbi is a simple procedure involving only the two calculation variables defined in the previous sub-section. At each step, the algorithm uses parameters from the known hidden Markov model $\lambda = \{q, A, B\}$:

Initialization: At $t = 1$ we immediately obtain, without maximisation,

$$\chi_{j,1} = P\left[S_1 = j, \boldsymbol{x}_1\right] = q_j b_j(x_1) \tag{5.78}$$

The backpointers $\zeta_{j,1}$ are irrelevant here, because there is no previous state.

Viterbi Forward step: At $t = 2$ we must search among all previous states to obtain, using Bayes rule,

$$\begin{aligned}
\chi_{j,2} &= \max_i P\left[S_1 = i, S_2 = j, \boldsymbol{x}_1, \boldsymbol{x}_2 \mid \lambda\right] \\
&= \max_i P\left[S_1 = i, \boldsymbol{x}_1 \mid \lambda\right] P\left[S_2 = j, \boldsymbol{x}_2 \mid S_1 = i, \boldsymbol{x}_1, \lambda\right] \\
&= \max_i P\left[S_1 = i, \boldsymbol{x}_1 \mid \lambda\right] P\left[S_2 = j \mid S_1 = i, \lambda\right] P\left[\boldsymbol{x}_2 \mid S_2 = j, \lambda\right] \\
&= \max_i \chi_{i,1} a_{ij} b_j(\boldsymbol{x}_2)
\end{aligned}$$

and

$$\zeta_{j,2} = \operatorname*{argmax}_i \chi_{i,1} a_{ij} \tag{5.79}$$

The step from $t = 1$ to $t = 2$ is equivalent to a step from any $t-1$ to t, and the forward step can be immediately generalized, for $t = 2, \ldots, T$, as

$$\begin{cases} \chi_{j,t} = b_j(\boldsymbol{x}_t) \max_i \chi_{i,t-1} a_{ij} \\ \zeta_{j,t} = \operatorname*{argmax}_i \chi_{i,t-1} a_{ij} \end{cases} \tag{5.80}$$

Note that both the max and argmax results can be obtained in one single search[3] over i, as the argument is the same in both operations.

Termination: If the HMM has *finite duration*, the final transition to the special *exit* state must be included. Then, the final step in the algorithm must be slightly modified as

$$\begin{cases} \chi_{j,T} = b_j(\boldsymbol{x}_T) a_{j,N+1} \max_i \chi_{i,T-1} a_{ij} \\ \zeta_{j,T} = \operatorname*{argmax}_i \chi_{i,T-1} a_{ij} \quad \text{(unchanged)} \end{cases} \tag{5.81}$$

[3] In MatLab, the `max` function can return, in one single function call, both the maximum value and the index to the vector element where the maximum was found.

Backtracking: At time $t = T$, the desired state sequence is easily found as

$$\begin{aligned} \hat{i}_T &= \operatorname*{argmax}_i \chi_{i,T} \\ \hat{i}_t &= \zeta_{\hat{i}_{t+1},t+1}, \quad t = T-1, T-2, \ldots, 1 \end{aligned} \tag{5.82}$$

To avoid numerical problems and to improve computation speed further, the Viterbi algorithm can also be implemented using log-probabilities, defining instead

$$\chi_{j,t} = \max_{(i_1 \ldots i_{t-1})} \log P\left[S_1 = i_1, \ldots, S_{t-1} = i_{t-1}, S_t = j, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t \mid \lambda\right] \tag{5.83}$$

The necessary small modifications to the Viterbi update expressions are left as an exercise for the reader. Multiplications will be simply replaced by additions of the corresponding logarithmic quantities in the Viterbi calculations. The required logarithms of HMM parameters can be conveniently pre-calculated once and for all. This is actually the preferred version of the Viterbi algorithm, because it requires no multiplications and works very fast.

Summary

This chapter introduced the Hidden Markov Model (HMM) for sequence classification. A sequence of random feature vectors $\boldsymbol{X}_1 \dots \boldsymbol{X}_T$ is generated by a signal source, controlled by a hidden sequence of internal states $S_1 \dots S_T$, where each $S_t \in \{1 \dots N\}$. The state sequence forms a first-order Markov chain. A *finite-duration* HMM stops at the special exit condition $S_{T+1} = N + 1$.

HMM Definition: $\lambda = \{\{q, A\}, B\}$, where

$$\begin{aligned} q_j &= P\left[S_1 = j\right] \\ a_{ij} &= P\left[S_{t+1} = j \mid S_t = i\right] \\ b_j(\boldsymbol{x}) &= f_{\boldsymbol{X}_t|S_t}(\boldsymbol{x} \mid j) \end{aligned}$$

Forward Algorithm: A procedure to calculate the variables

$$\begin{aligned} \alpha_{j,t} &= P\left[\boldsymbol{x}_1 \dots \boldsymbol{x}_t, S_t = j \mid \lambda\right] \\ \hat{\alpha}_{j,t} &= P\left[S_t = j \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_t, \lambda\right] = \alpha_{j,t}/(c_1 \cdots c_t) \\ c_t &= P\left[\boldsymbol{x}_t \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda\right], \quad 1 \le t \le T \\ c_{T+1} &= P\left[S_{T+1} = N + 1 \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_T, \lambda\right], \text{ (finite duration)} \end{aligned}$$

Backward Algorithm: A procedure to calculate the variables

$$\begin{aligned} \beta_{j,t} &= P\left[\boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T \mid S_t = j, \lambda\right] \\ \hat{\beta}_{j,t} &= \beta j, t/(c_t \cdots c_T), \quad \text{(infinite duration)} \end{aligned}$$

$$\begin{aligned} \beta_{j,t} &= P\left[\boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T, S_{T+1} = N + 1 \mid S_t = j, \lambda\right] \\ \hat{\beta}_{j,t} &= \beta j, t/(c_t \cdots c_T c_{T+1}), \quad \text{(finite duration)} \end{aligned}$$

Conditional State Probability:

$$\begin{aligned} \gamma_{j,t} &= \hat{\alpha}_{j,t}\hat{\beta}_{j,t}c_t \\ &= \begin{cases} P\left[S_t = j \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_T, \lambda\right], & \text{(infinite duration)} \\ P\left[S_t = j \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_T, S_{T+1} = N + 1, \lambda\right], & \text{(finite duration)} \end{cases} \end{aligned}$$

Viterbi Algorithm: A procedure to calculate the Maximum A Posteriori Probability (MAP) state sequence

$$\hat{\underline{i}} = \widehat{(i_1 \dots i_T)} = \underset{(i_1 \dots i_T)}{\operatorname{argmax}} P\left[S_1 = i_1 \dots S_T = i_T \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_T, \lambda\right]$$

Problems

5.1 A single-word recognizer uses an HMM to represent each word. The speech signal is described at each frame interval by an acoustic feature vector, which is vector-quantized to four discrete values, ranging from 1 through 4. One particular word is modeled by a discrete HMM $\lambda = \{q, A, B\}$ with three states:

$$q = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.3 & 0.7 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{pmatrix}; \quad B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0.4 & 0.1 \\ 0.1 & 0.1 & 0.2 & 0.6 \end{pmatrix}$$

In one test run the observed sequence was $\underline{z} = (1, 2, 4, 4, 1)$.

5.1.a Is this particular word model a left-right or an ergodic HMM? Why?

5.1.b Use the forward algorithm to calculate the probability that the given word model produced the given observation sequence. Show intermediate computation results in a table with one row for each state and a column for each time sample.

5.1.c Determine the total number of possible state sequences of length $T = 5$ for this particular HMM.

5.1.d Determine the most probable state sequence for the given observed sequence using the most efficient method you know.

5.2 (991026:1) A sound-environment recognizer uses hidden Markov models to represent two different types of sound environments. The acoustic signal is picked up by a microphone, analyzed and described at each frame interval by an acoustic feature vector, which is vector-quantized to four discrete values, ranging from 1 through 4. The recognizer is designed to guess which of the two environments generated the observation sequence, using a decision criterion which gives minimum error probability. The a priori probabilities for being in either environment are equal. The sound environments are modeled by discrete HMMs with two states each. The transition matrices for the two environments are

$$A_1 = \begin{pmatrix} 0.2 & 0.8 \\ 0.8 & 0.2 \end{pmatrix}; \quad A_2 = \begin{pmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{pmatrix}$$

The initial state probability vectors and probability mass functions for the observed features are exactly the same in both environments:

$$q = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}; \quad B = \begin{pmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.4 & 0.3 & 0.2 & 0.1 \end{pmatrix}$$

In one test run the observed acoustic signal sequence is $\underline{z} = (3, 1, 4, 2)$.

5.2.a What is the recognizer's guess for this particular observed sequence?

5.2.b Determine the conditional probability that the recognizer made a correct decision, given this particular observation.

5.3 (001027:3) You have observed the output sequence $\underline{x} = (x_1 \dots x_{100})$ from a discrete hidden-Markov source, but you do not know the corresponding internal state sequence $\underline{S} = (S_1 \dots S_{100})$ in the source. The source HMM $\lambda = (q, A, B)$ is known as

$$q = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}; \quad A = \begin{pmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{pmatrix}; \quad B = \begin{pmatrix} 0.4 & 0.3 & 0.2 & 0.1 \\ 0.1 & 0.2 & 0.3 & 0.4 \end{pmatrix}$$

You have observed all elements in the output sequence, and a few output samples are

$$x_1 = 2; \ \dots \ x_{18} = 1; \quad x_{19} = 3; \quad x_{20} = 4$$

Previous calculations, using the complete sequence, have shown that

$$P(S_{18} = i | x_1 x_2 \dots x_{18}, \lambda) = \begin{cases} 0.3, & i = 1 \\ 0.7, & i = 2 \end{cases}$$

5.3.a Calculate $P(S_{19} = j | x_1 x_2 \dots x_{18} x_{19}, \lambda)$ for j =1 and 2.

5.3.b Calculate $P(S_{19} = j | x_1 x_2 \dots x_{18} x_{19} x_{20}, \lambda)$ for j=1 and 2. (Note the time indices!)

5.4 A Markov chain is defined by the state-flow diagram in Fig. 5.15.

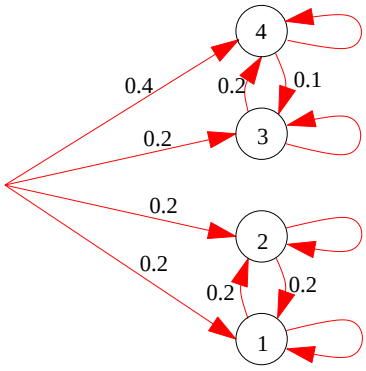


Figure 5.15: State-flow diagram for a Markov chain, with given initial and transition probabilities.

5.4.a Is this Markov chain stationary? If so, what is the stationary state probability vector?

5.4.b Is this Markov chain ergodic?

5.4.c Disregard the initial state probabilites given in the figure, and find another initial state probability vector, that would give a stationary state distribution. Show by qualitative reasoning, using only the state-flow diagram, that infinitely many stationary state distributions are possible, with a suitable choice of initial probabilities, and give a general expression defining all these stationary state probability vectors.

5.4.d Use the Matlab `eig` function to calculate eigenvectors and eigenvalues of the transposed transition probability matrix. Show that the result agrees with your previous qualitative conclusions.

5.5 For a discrete hidden-Markov source $\lambda = (q, A, B)$ the initial probability vector q and the transition probability matrix A are known as

$$q = \begin{pmatrix} 0.6 \\ 0.1 \\ 0.3 \end{pmatrix}; \quad A = \begin{pmatrix} 0.9 & 0 & 0.1 \\ 0 & 1 & 0 \\ 0.2 & 0 & 0.8 \end{pmatrix}$$

5.5.a Is the duration of this HMM finite or infinite?

5.5.b Determine eigenvalues and corresponding eigenvectors for the transposed transition matrix A^T.

5.5.c Is this HMM source stationary? If so, what are the stationary state probabilities?

5.5.d Is this HMM source ergodic?

5.6 A hidden Markov model has transition probability matrix A with elements $a_{ij} = P(S_{t+1} = j | S_t = i)$. We define the discrete-time *state duration* D_i as the number of consecutive time instants that the state remains as $S_t = i$ once this state has been entered. In other words, $S_{t-1} \neq i$ and $S_t = i, \ldots S_{t+D_i-1} = i$ and $S_{t+D_i} \neq i$. As state transitions are random events, the state duration D_i is a discrete random variable.

5.6.a Determine the probability-mass distribution for D_i, i.e.,

$$f_{D_i}(d) = P(D_i = d)$$

expressed as a function of a_{ii}.

5.6.b Determine the expected value $E[D_i]$ of the state duration. Compare the result to the most likely value of D_i.

5.7 A general *finite-duration hidden Markov model* with N states has an initial probability vector q and transition matrix A with elements $a_{ij} = P(S_{t+1} = j|S_t = i)$ for $1 \leq i, j \leq N$ and $a_{i,N+1} = P(S_{t+1} = N+1|S_t = i)$. The HMM is *not necessarily left-right*, i.e., transitions from any state to any other state may occur with non-zero probability.

We define the discrete-time *total HMM duration* D as the total number of consecutive time instants where the model has not yet reached the special END state $N+1$ and stopped. In other words, $S_t \in \{1, \ldots, N\}$ for $1 \leq t \leq D$ and $S_{D+1} = N+1$. As state transitions are random events, the total HMM duration D is a discrete random variable.

5.7.a Determine the probability-mass distribution for D, i.e.,

$$f_D(d) = P(D = d)$$

Hint: Calculate $P(D = d) = P(D > d-1) - P(D > d)$.

5.7.b Determine the expected value $E[D]$ of the total HMM duration.

5.8 In an attempt to design a *speaker-independent* word recognition classifier, you have trained $N \times M$ different hidden Markov models (HMM); one separate model λ_{nm} for word type $W = n \in \{1, \ldots, N\}$ and speaker $S = m \in \{1, \ldots M\}$. Several training examples were used for each word type and each speaker.

You will now design a classifier to identify sequences of J words, $\underline{W} = (W_1, \ldots, W_J)$, where all words in the sequence are pronounced by the same speaker. The jth recorded test word is represented as usual by a stream of feature vectors, denoted as

$$\underline{\boldsymbol{x}}_j = (\boldsymbol{x}_{j1}, \ldots, \boldsymbol{x}_{jT_j})$$

You already have a procedure that calculates the log-probability of any single recorded test word for any of the known models, i.e.,

$$L_{jnm} = \ln P(\underline{\boldsymbol{x}}_j|\lambda_{nm}), \quad \text{for } j = 1, \ldots, J; \quad n = 1, \ldots, N; \quad m = 1, \ldots, M.$$

Any of the speakers is engaged at random with equal probabilities $1/M$, and each of the possible word types occurs with equal probabilities $1/N$, independently of which speaker is engaged, and independently of the other words in the sequence. The feature distributions are assumed to be conditionally independent across all words in a sequence, given the word types and the speaker. For optimal performance, the classifier should utilize the knowledge that all observed test words were recorded from the *same* speaker, although it is not known who among the possible speakers actually pronounced the test words.

5.8.a Construct a decision rule to identify the most probable *word sequence* $\underline{\hat{w}} = (\widehat{w_1, \ldots, w_J})$ using your calculated log-probabilities L_{jnm}. The classifier will need to search across all N^J possible word sequences.

5.8.b Now design a decision rule to find, instead, the most probable *single word* $\hat{w}_j$, at any position j in the word sequence. Note that this is a different criterion than in the previous sub-problem, because now the classifier needs to search only among the N possible words at each word position. Calculate also the probability of correct decision, for each word.

Chapter 6

Hidden Markov Model Training

In the preceding chapter we assumed that the hidden Markov model $\lambda = \{q, A, B\}$ was known. Where did it come from?

In practice we do not have a model to start with. Usually, only a set of observed pattern sequences are available. The data were recorded from some more or less well-defined physical source, and we want to find an HMM that characterizes the source as well as possible. For example, to design a word recognizer we may have several recordings from different people pronouncing the word /faiv/ and we want to design one particular HMM to represent this type of word. We also have other similar sets of recordings to design other HMM:s to represent those other words.

The available observation sequences are called *"training data"*, and the source model is said to be *"trained"* on the training data to represent these observations. Some people might say that the HMM "learns" from the training data. One of the best things about hidden Markov models is the fact that automatic procedures are available for this training process. The procedure is guaranteed to converge to a solution that is optimal in some sense. This chapter presents the *Baum-Welch algorithm* for HMM training. This algorithm is derived as an application of the *Expectation Maximisation (EM)* algorithm, discussed in chapter 7. The EM approach is very powerful and can be used in many situations, when it is necessary to make estimations regarding random variables that cannot be directly observed, such as the hidden internal state sequence in the HMM.

6.1 The Baum-Welch Algorithm

Given a set of observed training data $\underline{\boldsymbol{x}}$, the Baum-Welch algorithm searches for the best possible hidden Markov model $\lambda_{ML} = \{\{q, A\}, B\}$, in the Maximum Likelihood sense, i.e., the model with the highest probability of gen-

erating the given training data:

$$\lambda_{ML} = \underset{\lambda}{\operatorname{argmax}}\, P\left[\underline{\boldsymbol{x}} \mid \lambda\right] \tag{6.1}$$

However, the Baum-Welch algorithm cannot be guaranteed to find the globally optimal solution. Like most optimization methods the Baum-Welch algorithm can only find a *local maximum*, given an initial estimate of the desired HMM.

The Baum-Welch training procedure is mainly iterative:

Initialize: Design manually an initial HMM $\lambda^0 = \{\{q, A\}, B\}$ with a structure that is appropriate for the given application.

Repeat: For $n = 1, 2, \ldots$, find an improved HMM λ^n based on the previous model λ^{n-1}, such that

$$\log P\left[\underline{\boldsymbol{x}} \mid \lambda^n\right] > \log P\left[\underline{\boldsymbol{x}} \;\middle|\; \lambda^{n-1}\right] \tag{6.2}$$

using *all* available training data $\underline{\boldsymbol{x}}$ in each improvement step.

Terminate: Keep repeating the improvement step at least a few times and stop the iteration, either after a fixed number of iterations, or when the improvement becomes smaller than some predetermined threshold Δ_{min}, i.e.

$$\log P\left[\underline{\boldsymbol{x}} \mid \lambda^n\right] - \log P\left[\underline{\boldsymbol{x}} \;\middle|\; \lambda^{n-1}\right] < \Delta_{min} \tag{6.3}$$

Trim: In some cases it may be necessary to adjust the trained model manually, after the automatic training procedure.

The crucial point of the algorithm is obviously how to choose the modified model in the improvement step. This section will only give intuitive arguments for the choice of new HMM parameters. A formal proof is presented in Sec. 7.6. First, a minor notational problem must be discussed.

If we want to create a *finite-duration* HMM, for example to model a particular spoken word, we usually have several, say R, recorded examples of the word. The examples may have slightly different durations. We must use all those recordings together, in each step of the Baum-Welch algorithm. There are many ways to store R separate observation sequences with different lengths. Regardless of the storage method, we must always keep track of the start and end of each spoken example, because we want the HMM to model sequences with special start and end characteristics.

In other situations, where we have reasons to assume that the HMM source is *ergodic* with *infinite duration*, the start and end of each recorded training data has no particular relevance, as we assume the source has the same statistical characteristics both before and after our recording. Nevertheless, we may want to include several separate recordings in the training

data. For example, if we want a single HMM to represent "traffic noise" in general, we would probably need to record real traffic noise from several different streets. We must use all the recordings in each step of the Baum-Welch training procedure. No matter how we store the training data, we always need a way to keep track of the separate recordings, because the trained HMM should not attempt to model the sudden change from one recording to the next.

Depending on the programming language, various data structures can be used to store R separate observation sequences with different lengths. Mathematically, the training data can be written as one long sequence in the following way:

$$\underline{\boldsymbol{x}} = \left((\boldsymbol{x}_1^1 \dots \boldsymbol{x}_{T_1}^1), (\boldsymbol{x}_1^2 \dots \boldsymbol{x}_{T_2}^2), \dots, (\boldsymbol{x}_1^R \dots \boldsymbol{x}_{T_R}^R)\right) \tag{6.4}$$

It is also convenient to introduce a corresponding notation for the hidden state sequence in the HMM that is assumed to have generated the training data:

$$\underline{S} = \left((S_1^1 \dots S_{T_1}^1), (S_1^2 \dots S_{T_2}^2), \dots, (S_1^R \dots S_{T_R}^R)\right) \tag{6.5}$$

Here, the complete training data set consists of R separate sub-sequences, indexed $1, 2, \dots, R$. Each sub-sequence can have a different length, denoted as $T_1, T_2, \dots, T_R$. As each single observation $\boldsymbol{x}_t^r$ is a column vector, the complete data set $\underline{\boldsymbol{x}}$ can be seen as a matrix with $T_1 + T_2 + \dots + T_R$ columns.

In each iterative step of the Baum-Welch algorithm we use a given λ, obtained from the previous step, to obtain an improved model λ^{new}. The proof of the Baum-Welch algorithm shows that the three components of the new model can be calculated separately and independently of each other:

$$\begin{aligned} \{\{q, A\}, B\} &\rightarrow q^{new} \\ \{\{q, A\}, B\} &\rightarrow A^{new} \\ \{\{q, A\}, B\} &\rightarrow B^{new} \end{aligned} \tag{6.6}$$

Each of these update steps makes use of the Scaled Forward and Backward variables, derived in chapter 5. The forward and backward algorithms are defined only for a single sub-sequence, and must first be applied sequentially to each of the R sub-sequences in the training data.

6.1.1 Updating the Initial Probability Vector

The initial probability vector defines $q_j = P[S_1 = j]$. Now, when we must create an estimate based only on observed data, it seems very reasonable to choose the updated version as the average of conditional state probabilities at the start of each training sequence, given the training data:

$$q_j^{new} = \frac{1}{R} \sum_{r=1}^{R} P\left[S_1^r = j \mid \boldsymbol{x}_1^r \dots, \boldsymbol{x}_{T_r}^r, \lambda\right] \tag{6.7}$$

where R is the number of concatenated training sub-sequences. The required conditional probabilities in the sum are easily calculated, for each sub-sequence no. r, with a combination of the forward and backward variables, using the help function γ defined in section 5.5:

$$P\left[S_1^r = j \mid \boldsymbol{x}_1^r \ldots, \boldsymbol{x}_{T_r}^r \lambda\right] = \gamma_{j,1}^r = \hat{\alpha}_{j,1}^r \hat{\beta}_{j,1}^r c_1^r \tag{6.8}$$

For the desired average across R sub-sequences, we sum all the initial state probability vectors and then normalize the resulting vector to get the new initial probability distribution as

$$\overline{\gamma}_j = \sum_{r=1}^{R} \gamma_{j,1}^r, \quad \text{all } j \tag{6.9}$$

$$q_j^{new} = \frac{\overline{\gamma}_j}{\sum_{k=1}^{N} \overline{\gamma}_k} \tag{6.10}$$

If we are training a *finite-duration* HMM, the update equation must also include the special exit condition $S_{T+1} = N + 1$, but this is then already included in the calculated $\hat{\beta}_{j,t}$.

6.1.2 Updating the Transition Probability Matrix

The transition matrix A has elements defined as

$$a_{ij} = P\left[S_{t+1} = j \mid S_t = i\right] = \frac{P\left[S_t = i \cap S_{t+1} = j\right]}{\sum_k P\left[S_t = i \cap S_{t+1} = k\right]} \tag{6.11}$$

We do not know these true probabilities, but we can estimate them from available training data. It seems intuitively correct to estimate transition probabilities from the corresponding time-averaged conditional probability of state combinations $S_t = i \cap S_{t+1} = j$, given the training data and the old model, as

$$\overline{\xi}_{ij} = \sum_{r=1}^{R} \sum_{t=1}^{T_r - 1} \underbrace{P\left[S_t^r = i \cap S_{t+1}^r = j \mid \boldsymbol{x}_1^r \ldots \boldsymbol{x}_{T_r}^r, \lambda\right]}_{\xi_{ij,t}^r} \tag{6.12}$$

We then calculate the updated estimate A^{new} of the transition probability matrix with elements

$$a_{ij}^{new} = \frac{\overline{\xi}_{ij}}{\sum_k \overline{\xi}_{ik}} \tag{6.13}$$

by normalizing each row in the matrix $\overline{\xi}$ as required to ensure that the sum of each row in A equals 1.

If the training data contains several sub-sequences, we must first calculate the desired state-combination probabilities separately for each sub-sequence. The calculation is slightly different for finite-duration and infinite-duration HMMs.

For an HMM with *infinite duration* we know nothing about state probabilities outside the observed time interval of each sub-sequence. We then calculate, for each sub-sequence r, a three-dimensional matrix ξ^r with size $N \times N \times (T_r - 1)$ and with elements

$$\xi^r_{ij,t} = P\left[S^r_t = i, S^r_{t+1} = j \mid \boldsymbol{x}^r_1 \ldots, \boldsymbol{x}^r_{T_r}, \lambda\right] \tag{6.14}$$

and then accumulate the total sum in two steps as

$$\overline{\xi}^r_{ij} = \sum_{t=1}^{T_r-1} \xi^r_{ij,t} \tag{6.15}$$

$$\overline{\xi}_{ij} = \sum_{r=1}^{R} \overline{\xi}^r_{ij} \tag{6.16}$$

For an HMM with *finite duration* we must also include the known special *exit* condition $S_{T_r+1} = N + 1$ at the end of each observed sub-sequence. In this case we must define, for $1 \leq t \leq T_r - 1$,

$$\xi^r_{ij,t} = P\left[S^r_t = i, S^r_{t+1} = j \mid \boldsymbol{x}^r_1 \ldots, \boldsymbol{x}^r_{T_r}, S^r_{T_r+1} = N + 1, \lambda\right] \tag{6.17}$$

This is no complication, as we use the previously calculated $\hat{\beta}_{j,t}$ which already includes the exit condition, as defined in Sec. 5.5.2.

However, now we should also include the known non-zero transition probabilities to the exit state at the end of each sub-sequence. We therefore supplement the $\overline{\xi}^r$ matrix with one additional column indicating the transition probabilities from each possible state $S^r_{T_r} = i$ at the end of the r-th sub-sequence to the exit state $S^r_{T_r+1} = N + 1$:

$$\begin{aligned} \overline{\xi}^r_{i,N+1} &= P\left[S^r_{T_r} = i \mid \boldsymbol{x}^r_1 \ldots, \boldsymbol{x}^r_{T_r}, S^r_{T_r+1} = N + 1, \lambda\right] \\ &= \gamma_{i,T_r} = \hat{\alpha}^r_{i,T_r} \hat{\beta}^r_{i,T_r} c^r_{T_r} \end{aligned} \tag{6.18}$$

The required probabilities in Eq. (6.14) or (6.17) are obtained using a combination of the forward and backward variables, calculated for each subsequence of training data. The following derivation applies Bayes' rule repeatedly, and also uses the special Markov dependence characteristics of the HMM, as illustrated in Fig.5.4. For any state combination, $i = 1, \ldots, N; j =$

$1, \ldots, N$, and all but the last time sample, $t = 1, \ldots, T_r - 1$, we obtain

$$\begin{aligned}
\xi^r_{ij,t} &= P\left[S^r_t = i, S^r_{t+1} = j \mid \boldsymbol{x}^r_1 \ldots \boldsymbol{x}^r_{T_r}, \lambda\right] = \\
&= \frac{P\left[S^r_t = i, \boldsymbol{x}^r_1 \ldots \boldsymbol{x}^r_t, S^r_{t+1} = j, \boldsymbol{x}^r_{t+1} \ldots \boldsymbol{x}^r_{T_r} \;\middle|\; \lambda\right]}{P\left[\boldsymbol{x}^r_1 \ldots \boldsymbol{x}^r_{T_r} \;\middle|\; \lambda\right]} = \\
&= \underbrace{P\left[S^r_t = i, \boldsymbol{x}^r_1 \ldots \boldsymbol{x}^r_t \mid \lambda\right]}_{\alpha^r_{i,t}} \cdot \\
&\quad \cdot \frac{P\left[S^r_{t+1} = j, \boldsymbol{x}^r_{t+1} \ldots \boldsymbol{x}^r_{T_r} \;\middle|\; S^r_t = i, \boldsymbol{x}^r_1 \ldots \boldsymbol{x}^r_t, \lambda\right]}{c^r_1 \cdots c^r_{T_r}} = \\
&= \alpha^r_{i,t} \frac{P\left[S^r_{t+1} = j, \boldsymbol{x}^r_{t+1} \ldots \boldsymbol{x}^r_{T_r} \;\middle|\; S^r_t = i, \boldsymbol{x}^r_1 \ldots \boldsymbol{x}^r_t, \lambda\right]}{c^r_1 \cdots c^r_{T_r}} = \\
&= \alpha^r_{i,t} \frac{P\left[S^r_{t+1} = j, \boldsymbol{x}^r_{t+1} \ldots \boldsymbol{x}^r_{T_r} \;\middle|\; S^r_t = i, \lambda\right]}{c^r_1 \cdots c^r_{T_r}} = \\
&= \alpha^r_{i,t} \underbrace{P\left[S^r_{t+1} = j \mid S^r_t = i, \lambda\right]}_{a_{ij}} \frac{P\left[\boldsymbol{x}^r_{t+1} \ldots \boldsymbol{x}^r_{T_r} \;\middle|\; S^r_{t+1} = j, \lambda\right]}{c^r_1 \cdots c^r_{T_r}} = \\
&= \alpha^r_{i,t} a_{ij} \frac{P\left[\boldsymbol{x}^r_{t+1} \mid S^r_{t+1} = j, \lambda\right] P\left[\boldsymbol{x}^r_{t+2} \ldots \boldsymbol{x}^r_{T_r} \;\middle|\; S^r_{t+1} = j, \boldsymbol{x}^r_{t+1} \lambda\right]}{c^r_1 \cdots c^r_{T_r}} = \\
&= \frac{\alpha^r_{i,t} a_{ij} b_j(\boldsymbol{x}^r_{t+1}) \beta^r_{j,t+1}}{c^r_1 \cdots c^r_{T_r}} = \\
&= \hat{\alpha}^r_{i,t} a_{ij} b_j(\boldsymbol{x}^r_{t+1}) \hat{\beta}^r_{j,t+1}
\end{aligned} \tag{6.19}$$

The derivation is very similar for a finite-duration HMM, where the exit condition is included. The result looks formally exactly the same, as the exit condition is already included in the $\hat{\beta}_{j,t}$ values calculated for the finite-duration model. This derivation shows clearly why it was a good idea to define the scaled Forward and Backward variables in the way it was done in the previous chapter.

6.1.3 Updating the Observation Probability Matrix

The observation probabilities are defined differently for continuous and discrete HMM:s, as discussed in section 5.1. A discrete HMM uses quantized observations Z_t, such that $\boldsymbol{X}_t \in V_m \Leftrightarrow Z_t = m$. The observation probability matrix B has elements defined as

$$b_{jm} = P\left[Z_t = m \mid S_t = j\right] = \frac{P\left[S_t = j \cap Z_t = m\right]}{\sum_{k=1}^{M} P\left[S_t = j \cap Z_t = k\right]} \tag{6.20}$$

Therefore, it seems plausible to calculate an updated version from the corresponding conditional probabilities, given the observed training data, as

$$\begin{aligned}\overline{b}_{jm} &= \sum_{r=1}^{R}\sum_{t=1}^{T_r} P\left[S_t^r = j \cap Z_t = m \mid z_1^r \dots z_{T_r}^r\right] = \\ &= \sum_{r=1}^{R}\sum_{t=1}^{T_r} P\left[S_t^r = j \mid z_1^r \dots z_{T_r}^r\right] \delta_{z_t^r,m} = \\ &= \sum_{r=1}^{R}\sum_{t=1}^{T_r} \gamma_{j,t}^r \delta_{z_t^r,m}\end{aligned} \tag{6.21}$$

Here, Kronecker's δ is used to indicate that the sum includes only terms where $z_t^r = m$. Then each matrix row is normalized as usual to give the new desired conditional observation probabilities

$$b_{jm}^{new} = \frac{\overline{b}_{jm}}{\sum_{k=1}^{M} \overline{b}_{jk}} \tag{6.22}$$

Continous probability density functions, for example Gaussian mixtures, are updated using similar principles derived in Sec. 7.6.5 (Rabiner and Juang, 1986; Rabiner, 1989; Young et al., 2002).

6.2 HMM Training: Practical aspects

A few steps must be taken before the automatic training procedure can be started. The following factors are the most important to consider:

- Choice of features and feature-extraction method
- Choice of model size
- How to initialize the HMM before the automatic training

6.2.1 Feature Extraction

The input to a HMM classifier is a sequence of observed data vectors (or scalars). The data should of course be as simple as possible to allow fast computation, but the features must be sufficiently detailed to indicate the differences between patterns. When a HMM is used for the classification, the HMM can represent rather complex "high-level" patterns in the data. Therefore, features should represent the data only at the most basic level of detail. There is usually no advantage in trying to build any advanced intelligence into the feature extraction mechanism. It is usually better to let the HMM structure provide all the needed intelligence.

An important decision is to choose either *absolute* or *differential* features. For example, a *musical-instrument* classifier might use a pitch estimator

to obtain an absolute fundamental frequency for each sound segment, as instruments may differ in the range of absolute frequencies that they can generate. On the other hand, a *musical-melody* classifier might record only the relative steps in fundamental frequency in order to recognize a melody regardless of the key in which it is played. Differential features are obviously more robust if we want the classifier to look mainly for variational patterns in the data, and disregard absolute feature values. A combination of absolute and differential features can also be used.

The choice of features depends entirely on the classification problem we are facing. Practical experimentation may be necessary to reveal if the chosen features are sufficient.

6.2.2 Model Size

Before the HMM training can start we must decide on the size of the HMM. How many states do we need? How complex output probability distributions do we need?

The number of states must be estimated by considering the complexity of the various patterns that the HMMs will be used to distinguish. For example, to represent various graphical written patterns, we must look at the possible patterns and estimate how many distinguishable segments the pattern contains.

It may be a good idea to use different numbers of states in the different HMM:s used to represent separate classes of patterns. For example, to represent a spoken word like /faiv/ that has 4 distinct sound segments, we may need only 4 states, plus possibly 2 more to allow for the initial and final silent segments.

A general problem to consider is the amount of training data that can be obtained. A complex HMM with many states includes many separate parameters that must be determined by training. Therefore, we need more training data for a complex HMM. As a rule of thumb, the number of samples in the training data set should be at least on the order of $10 - 100$ times greater than the number of free HMM parameters to be trained by the data.

6.2.3 Initializing a Left-Right HMM

Before starting the iterative Baum-Welch algorithm we must assign initial values to all parameters in the HMM. There is only one general requirement on this assignment: The initial model must indicate, somehow, what we want the different model states to represent. This requirement has different consequences, depending on the type of HMM.

A *left-right HMM* always starts in one particular state, that can be arbitrarily designated as state number 1. For each new time sample, the state can jump back to itself, or only to the nearest following state. Therefore,

the initial probability vector q and transition matrix A should be initialized as

$$q = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}; \qquad A = \begin{pmatrix} a_{11} & 1-a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & 1-a_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_{NN} & 1-a_{NN} \end{pmatrix} \tag{6.23}$$

The diagonal elements a_{ii} of the transition matrix can be chosen to indicate approximately the average state durations, as shown in problem 6.1.

This is sufficient to indicate to the automatic training procedure that state 1 is intended to represent the first part of the training data, state 2 the next part, etc. Therefore, all the output probability distributions for different states can be initialized with the *same parameters for all states*. The first step in the Baum-Welch iteration will then use the training data to calculate more correct output probability parameters for each state.

For example, in a discrete HMM, all elements of the output probability matrix B can be uniformly initialized as

$$b_{jm} = 1/M, \quad j = 1 \ldots N, \quad m = 1 \ldots M \tag{6.24}$$

6.2.4 Initializing an Ergodic HMM

An infinite-duration *ergodic HMM* may start in any state. For each new time sample, the state can in principle jump to any other state. Therefore, the elements of the initial probability vector q and transition matrix A should be initialized with non-zero values as

$$q_j = 1/N, \quad j = 1 \ldots N; \qquad a_{ij} = \frac{1-a_{ii}}{N-1}, \quad j \neq i \tag{6.25}$$

The diagonal elements a_{ii} of the transition matrix can be chosen to indicate approximately the average state durations, as shown in problem 6.1.

In this case the initialization of the output probability distributions is more critical than for a left-right HMM. The output distributions must be initialized *differently for each state*, because this is the only way we have to indicate to the training procedure what the N various states are supposed to represent. This initialization can be performed either manually or automatically. One way to do it automatically is to cluster the training data into N separate categories and initialize the output probability distributions using one cluster of training data for each state.

6.2.5 Termination of HMM Training

The EM (Baum-Welch) training algorithm is very efficient. Often a good model is reached already after 5–10 iterations. If the HMM is to be used for signal classification, it is usually pointless to continue modifying the model in many steps that give very small improvements. This may even degrade classifier performance, because the HMM can become *overtrained* on the specific set of available training data. The trained model must be general enough to correctly represent also a new test sequence that never occurred during training.

6.2.6 Adjustment after Training

It is usually very difficult to record sufficient amounts of training data. Therefore, some observations may *never* occur in the limited set of training data, although we may know that they might have occurred with some small probability. If a discrete HMM is trained on such data, the Baum-Welch will assign *zero* observation probabilities at some elements of the observation probability matrix, because this is the correct ML estimate of the probability, given the training data. In such a case, we may want to assign a very small non-zero value instead, and then re-normalize the rows of the matrix as required.

A similar problem can occur with the *transition probability* matrix. For a *left-right* HMM we have intentionally defined many elements of the transition probability matrix to be exactly zero, These elements will still have zero values after the Baum-Welch training, and should of course remain zero.

However, for an ergodic HMM we may have reason to believe that no elements of the transition probability matrix should be exactly zero, even if some state transitions never occurred in the limited training data. Then it may be a good idea to assign a very small non-zero value to these elements, and then re-normalize all rows as required.

For an ergodic HMM we may also want the *initial probability* vector to represent *stationary* state probabilities at any arbitrary starting time, instead of the particular starting point of the available training data. Then we should calculate the stationary state probability vector from the trained transition probability matrix, as shown in Eq. (5.22), and then re-assign the initial probability vector with these values.

Summary

This chapter introduced training methods to optimize hidden Markov Models to conform with *training data*, i.e. a set of concatenated observed feature-vector sequences, organized as

$$\underline{\boldsymbol{x}} = ((\boldsymbol{x}_1^1 \dots \boldsymbol{x}_{T_1}^1), (\boldsymbol{x}_1^2 \dots \boldsymbol{x}_{T_2}^2), \dots, (\boldsymbol{x}_1^R \dots \boldsymbol{x}_{T_R}^R))$$

Baum-Welch Algorithm: Iteratively optimizes a HMM. Each iterative step of the algorithm updates the model $\lambda = \{\{q, A\}, B\}$ obtained from the previous step to an improved model $\lambda^{new} = \{\{q^{new}, A^{new}\}, B^{new}\}$, using all available training data. Each of the HMM components is calculated separately, based on the previous model:

$$\{\{q, A\}, B\} \rightarrow q^{new}$$
$$\{\{q, A\}, B\} \rightarrow A^{new}$$
$$\{\{q, A\}, B\} \rightarrow B^{new}$$

Problems

6.1 (Exam 991026: 5) You are designing an HMM single-word recognizer. You are planning to train each word model with the Baum-Welch algorithm. You just need to initialize the word models first. For one particular word you intend to use a left-right HMM with 5 states: a silence state (before the word actually starts), three different phoneme states, and finally another state of silence. The average duration of the word (not including the surrounding silence segments) is 600 ms, which corresponds to 30 observations, each with a frame duration of 20 ms. You estimate that the average duration of the initial and final silence states as 100 ms. Estimate an initial version of the transition probability matrix A, such that it correctly describes the average durations in each state. We assume initially that the three phoneme states have the same average duration.

Chapter 7

Expectation Maximization

The Baum-Welch algorithm used for HMM training can be derived from the *Expectation Maximization (EM)* algorithm. The EM algorithm is extremely useful as a general method to obtain estimates of parameters of a probability distribution, when some of the involved random variables cannot be observed directly, or when observations are missing for any other reason. The EM method was first presented by Dempster et al. (1977). The convergence properties have been studied by e.g. Wu (1983) and Ma et al. (2000).

The EM algorithm can be used to estimate parameters for any probabilistic model, not only the special parameters in a hidden Markov model. The only requirement is that the model somehow defines the joint probability distribution of a random vector $\boldsymbol{S}$ and another random vector $\boldsymbol{X}$. These vectors need not have the same size. The joint probability distribution of $\boldsymbol{S}$ and $\boldsymbol{X}$ is completely specified by some parameter set λ.

A particular outcome $\boldsymbol{X} = \boldsymbol{x}$ has been observed, but $\boldsymbol{S}$ cannot be observed. We use the observed data to estimate numerical values of the parameters in the set λ, such that the model agrees with the observed data as well as possible. The fundamental "trick" in the EM algorithm is a method to derive information about the hidden sequence from the available observations.

Of course, in the HMM application it is the hidden Markov state sequence $\underline{S} = (S_1, \ldots, S_T)$ that cannot be directly observed, and the parameter set λ is the usual set of HMM parameters that we must calculate using the observed training data.

This chapter will first give some intuitive arguments for the EM algorithm, then prove formally that it works, and finally present some applications. Section 7.6 will prove that the EM procedure leads exactly to the HMM update formulas of the Baum-Welch procedure presented intuitively in Chapter 6.

7.1 Intuitive Introduction

The EM procedure is iterative. In each step of the procedure we have a previous estimate of the parameter set λ, and we want to calculate a better estimate λ^{new}, using all the observed training data[1] $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, so that the new model parameters are guaranteed to fit the observations better than the previous model, in the Maximum Likelihood sense, i.e.

$$P\left[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T \mid \lambda^{new}\right] > P\left[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T \mid \lambda\right] \tag{7.1}$$

We can perhaps rather easily calculate $P\left[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T \mid \lambda\right]$, but it is usually very difficult to maximize this quantity analytically as a function of λ. The EM algorithm solves this problem.

As an introduction to the EM procedure, let us now assume for a moment that we had actually observed a specific outcome $\underline{S} = (i_1 \ldots i_T)$ of the hidden sequence, and not only the outcome $\underline{\boldsymbol{X}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$. Then the estimation problem would be much easier. We would simply use all the available observations to determine the new model parameters by conventional maximization of the ML criterion function. Considering any possible parameter values λ', we could easily define a criterion function as

$$y(\lambda') = \log P\left[\underline{S} = (i_1 \ldots i_T) \cap \underline{\boldsymbol{X}} = \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T \mid \lambda'\right] \tag{7.2}$$

and then choose

$$\lambda^{new} = \underset{\lambda'}{\operatorname{argmax}}\, y(\lambda') \tag{7.3}$$

In reality we do not know $(i_1 \ldots i_T)$. However, we know a great deal about all the possible outcomes of $\underline{S}$. We can in fact calculate the conditional probability for any sequence $\underline{S} = (i_1 \ldots i_T)$, given the previous parameter set λ and the known observations:

$$P\left[\underline{S} = (i_1 \ldots i_T) \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T, \lambda\right] \tag{7.4}$$

To account for all possible state sequences that may have caused the observed data sequence, let us now define a scalar random variable $Y(\lambda')$, for any parameter values λ', as

$$Y(\lambda') = \log P\left[\underline{S} \cap \underline{\boldsymbol{X}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T) \mid \lambda'\right] \tag{7.5}$$

One particular outcome of this random variable is the value $y(\lambda')$ in Eq. (7.2). The distribution is discrete, because the outcome of $Y(\lambda')$ is completely determined by the outcome of $\underline{S}$. Therefore, Eq. (7.4) already defined the conditional probability for any outcome of $Y(\lambda')$, given the observations.

[1] As this discussion applies not only to HMM training, we express the observed training data simply as one long sequence. We leave out the special notation to keep track of separate sub-sequences, introduced in Eq. (6.4) and (6.5).

To find λ^{new} we might want to maximize $Y(\lambda')$, but that is not possible, because it is a random variable, not a deterministic function.

The fundamental and very simple idea behind the EM algorithm is this: You cannot maximize a random variable, but you can always maximize its *Expected Value*!

The expected value of $Y(\lambda')$, given the observed data, is a deterministic function of both the previous parameter set λ and the hypothetical new set λ'. We denote this function as

$$\begin{aligned} Q(\lambda', \lambda) &= E\left[Y(\lambda') \mid (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T), \lambda\right] = \\ &= E\left[\log P\left[\underline{S} \cap \underline{\boldsymbol{X}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T) \mid \lambda'\right] \mid (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T), \lambda\right] = \\ &= \sum_{i_1=1}^{N} \ldots \sum_{i_T=1}^{N} P\left[\underline{S} = (i_1 \ldots i_T) \mid (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T), \lambda\right] \cdot \\ &\qquad \cdot \log P\left[\underline{S} = (i_1 \ldots i_T) \cap \underline{\boldsymbol{X}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T) \mid \lambda'\right] \end{aligned} \tag{7.6}$$

Note that we take the expected value of something that depends on the free hypothetical new parameters λ', but the probability values we use as weight factors in the averaging sum depend on the previously known parameter set λ. This is the central trick in the EM algorithm that often allows us to maximize $Q(\lambda', \lambda)$ analytically, by varying λ' and keeping λ fixed.

In the iterative search for the ML parameter estimate, we might accept any new parameter set λ' such that $Q(\lambda', \lambda) > Q(\lambda, \lambda)$, because the expected value has increased.[2] However, the maximization procedure will be most efficient if the improvement is made as large as possible in each iterative step. Therefore, whenever possible we choose the updated parameter set in each iterative step as

$$\lambda^{new} = \underset{\lambda'}{\operatorname{argmax}}\, Q(\lambda', \lambda) \tag{7.7}$$

It turns out that this maximization problem can be solved analytically in many important practical applications. Then we set $\lambda = \lambda^{new}$ and repeat the same procedure in the next step.

7.2 Algorithm Steps

The EM algorithm can now be formally summarized in these simple steps:

Help function: For the particular problem, formulate the EM help function defined as

$$Q(\lambda', \lambda) = E_{\underline{S}}\left[\log P\left[\underline{S}, \underline{\boldsymbol{x}} \mid \lambda'\right] \mid \underline{\boldsymbol{x}}, \lambda\right] \tag{7.8}$$

where $\underline{S}$ represents the hidden random variables assumed in the model, and $\underline{\boldsymbol{x}}$ is the complete sequence of all observed training data.

[2] This criterion is sometimes called the Generalised EM algorithm (GEM) (Dempster et al., 1977).

Initialize: Assign manually some reasonable starting values for the parameter set λ^0.

Repeat: For $n = 1, 2, \ldots$; Find an improved parameter vector λ^n, such that

$$\lambda^n = \underset{\lambda'}{\operatorname{argmax}}\, Q(\lambda', \lambda^{n-1}) \tag{7.9}$$

Terminate: Stop the iteration after a fixed number of iterations, or when the improvement is smaller than a predetermined threshold Δ_{min}, i.e.

$$Q(\lambda^n, \lambda^{n-1}) - Q(\lambda^{n-1}, \lambda^{n-1}) < \Delta_{min} \tag{7.10}$$

7.3 EM Algorithm Proof

So far we have only argued intuitively, and the EM "trick" mentioned above probably made the critical reader suspicious. Fortunately, the following theorem guarantees that the "trick" is valid: Increasing the value of the help function $Q(\ )$ always improves the parameter set in the desired ML sense. Note that this theorem is quite general and makes no reference to the special application in HMM training.

Theorem 7.1: *Assume that a parameter set λ specifies the joint probability distribution of a discrete random sequence $\underline{S}$ and a continous or discrete random sequence $\underline{\boldsymbol{X}}$. A particular outcome sequence $\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}$ has been observed, but $\underline{S}$ cannot be observed. Define, for any two parameter sets λ and λ^{new},*

$$Q(\lambda^{new}, \lambda) = E\left[\log P\left[\underline{S} \cap (\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}) \mid \lambda^{new}\right] \mid \underline{\boldsymbol{x}}, \lambda\right] \tag{7.11}$$

Then,

$$\begin{aligned} &\log P\left[\underline{\boldsymbol{x}} \mid \lambda^{new}\right] - \log P\left[\underline{\boldsymbol{x}} \mid \lambda\right] \geq Q(\lambda^{new}, \lambda) - Q(\lambda, \lambda), \ \textit{i.e.} \\ &\qquad Q(\lambda^{new}, \lambda) > Q(\lambda, \lambda) \ \Rightarrow \ P\left[\underline{\boldsymbol{x}} \mid \lambda^{new}\right] > P\left[\underline{\boldsymbol{x}} \mid \lambda\right] \end{aligned} \tag{7.12}$$

□

Proof:

$$\log P\left[\underline{\boldsymbol{x}} \mid \lambda^{new}\right] - \log P\left[\underline{\boldsymbol{x}} \mid \lambda\right] = \log \frac{P\left[\underline{\boldsymbol{x}} \mid \lambda^{new}\right]}{P\left[\underline{\boldsymbol{x}} \mid \lambda\right]} =$$

$$= \log \sum_{(i_1 \ldots i_T)} \frac{P\left[\underline{S} = (i_1 \ldots i_T), \underline{\boldsymbol{x}} \mid \lambda^{new}\right]}{P\left[\underline{\boldsymbol{x}} \mid \lambda\right]} =$$

$$= \log \sum_{(i_1 \ldots i_T)} \underbrace{\frac{P\left[\underline{S} = (i_1 \ldots i_T) \mid \underline{\boldsymbol{x}}, \lambda\right]}{P\left[\underline{S} = (i_1 \ldots i_T) \mid \underline{\boldsymbol{x}}, \lambda\right]}}_{1} \cdot \frac{P\left[\underline{S} = (i_1 \ldots i_T), \underline{\boldsymbol{x}} \mid \lambda^{new}\right]}{P\left[\underline{\boldsymbol{x}} \mid \lambda\right]} =$$

$$= \log \sum_{(i_1 \ldots i_T)} P\left[\underline{S} = (i_1 \ldots i_T) \mid \underline{\boldsymbol{x}}, \lambda\right] \frac{P\left[\underline{S} = (i_1 \ldots i_T), \underline{\boldsymbol{x}} \mid \lambda^{new}\right]}{P\left[\underline{S} = (i_1 \ldots i_T), \underline{\boldsymbol{x}} \mid \lambda\right]} =$$

$$= \log E\left[\frac{P\left[\underline{S}, \underline{\boldsymbol{x}} \mid \lambda^{new}\right]}{P\left[\underline{S}, \underline{\boldsymbol{x}} \mid \lambda\right]} \,\middle|\, \underline{\boldsymbol{x}}, \lambda\right] \geq \text{/Jensen inequality/}$$

$$\geq E\left[\log \frac{P\left[\underline{S}, \underline{\boldsymbol{x}} \mid \lambda^{new}\right]}{P\left[\underline{S}, \underline{\boldsymbol{x}} \mid \lambda\right]} \,\middle|\, \underline{\boldsymbol{x}}, \lambda\right] =$$

$$= E\left[\log P\left[\underline{S}, \underline{\boldsymbol{x}} \mid \lambda^{new}\right] \mid \underline{\boldsymbol{x}}, \lambda\right] - E\left[\log P\left[\underline{S}, \underline{\boldsymbol{x}} \mid \lambda\right] \mid \underline{\boldsymbol{x}}, \lambda\right] =$$

$$= Q(\lambda^{new}, \lambda) - Q(\lambda, \lambda)$$

■

The crucial point in the proof is the $\geq$ relation between lines 5 and 6, which follows from the general Jensen Inequality (see e.g. Råde and Westergren, 1995, section 17.1), as the logarithm is a concave function. Actually, the hidden sequence $\underline{S}$ need not be discrete. It can also be a continuous-valued random sequence. It is left to the reader to slightly modify the proof for this case.

7.4 EM Examples with Solutions

Example 7.1 (Missing Data): You want to describe the statistical distribution of body weight in the population of 25-year-old people living in Stockholm. You select a sample of 100 anonymous test persons completely at random, and measure their weights. When you have collected all the data you realize that it would actually have been a good idea to record for each test person if it was a man or a woman, because the body-weight distributions may be systematically different among women and men. When you plot a histogram of the data it actually looks like the distribution has two separate peaks.

However, it is now too late to correct this mistake. You do not know for any measured weight value if it belongs to a man or a woman. You can not even know exactly how many women there were in your test group, but as the participants were selected completely at random, there were probably about 50% women and 50% men in the sample.

It seems reasonable to describe the total probability density function as a GMM, i.e., as a weighted sum of two Gaussian functions, each defined by its own mean and variance. How can you estimate means and variances for those two separate distributions?

Solution:
Fortunately, you know the EM algorithm. Of course, the EM algorithm can never tell you which test persons were actually men or women. But we can still formulate a general double-peak density function, modeled as a Gaussian Mixture Model with two components. Then the EM algorithm can help us estimate all parameters to make the model fit the observed data.

We assume that the probability density function for the weight X of any randomly selected 25-year-old person in Stockholm can be modelled as an equally weighted sum of two Gaussian density functions:

$$f_X(x) = 0.5\frac{1}{\sigma_1\sqrt{2\pi}}e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} + 0.5\frac{1}{\sigma_2\sqrt{2\pi}}e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} \tag{7.13}$$

This distribution is completely specified by a set of four parameters, $\theta = (\mu_1, \sigma_1, \mu_2, \sigma_2)$, but we do not yet know the values of these parameters.

Let us denote the observed data sequence $\underline{x} = (x_1, \ldots, x_T)$, where x_t is the measured weight of test person no. t. We regard the observed values x_t as outcomes of a set of independent and identically distributed random variables X_t, all characterized by the density function in Eq. (7.13). We must now use all observations $\underline{x}$ to derive a good estimate $\hat{\theta}$ of the parameter vector θ. As we have no a priori knowledge of the parameters we use the ML estimate

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, P\left[\underline{X} = \underline{x} \mid \theta\right] \tag{7.14}$$

To express this problem in the EM framework, we assume that each X_t depends on a hidden discrete "state" variable S_t that we have not observed. This hidden "state" represents the fact that test person no. t may have been either a man or a woman. The distribution of the random variable X_t can be seen as the result of a two-step random procedure:

- First, the hidden random variable S_t is chosen as either $S_t = 1$ (woman) or $S_t = 2$ (man), with equal probability.
- Then, depending on the value of $S_t = i$, a value of X_t is drawn from the simple Gaussian *conditional* distribution, expressed with the shorthand notation $g(\ )$, as

$$f_{X_t|S_t}(x \mid i) = g(x, \mu_i, \sigma_i) = \frac{1}{\sigma_i\sqrt{2\pi}}e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} \tag{7.15}$$

This two-step procedure will obviously guarantee that each X_t has the density function of Eq. (7.13).

To apply the EM algorithm for this particular problem, we now only need to define the EM help function for this problem, as

$$Q(\theta', \theta) = E\left[\ln P\left[\underline{S}, \underline{x} \mid \theta'\right] \mid \underline{x}, \theta\right] \tag{7.16}$$

In this slightly simplified notation, θ represents the *old* parameter estimate with known fixed values resulting from the previous iteration step, and θ' is the corresponding parameter set that we are free to manipulate to maximize the $Q(\ )$ function. In the formal algorithm description in Sec. 7.2 the old parameter set was called λ^{n-1}, and the result of the maximization in step no. n was denoted λ^n.

As we assumed that each measurement sample is statistically independent of the other samples, all elements in $(S_1, \ldots, S_T)$ are statistically independent[3], and elements in $(X_1, \ldots, X_T)$ are also independent. Therefore, for any particular state sequence $\underline{S} = (i_1 \ldots i_T)$ we can easily express the random variable for which the EM function takes the expectation value, using the short-hand notation introduced in Eq. (7.15):

$$\begin{aligned} \ln P[\underline{S} = (i_1 \ldots i_T) \cap (x_1 \ldots x_T) \mid \theta'] &= \ln \prod_{t=1}^{T} 0.5 g(x_t, \mu'_{i_t}, \sigma'_{i_t}) \\ &= \sum_{t=1}^{T} \left(\ln 0.5 + \ln g(x_t, \mu'_{i_t}, \sigma'_{i_t})\right) \end{aligned} \tag{7.17}$$

Again because of the statistical independence, we can obtain the conditional probability of any state sequence as

$$\begin{aligned} P[\underline{S} = (i_1 \ldots i_T) \mid (x_1 \ldots x_T), \theta] &= /\text{all } S_t \text{ stat. indep.}/ \\ &= \prod_{t=1}^{T} P[S_t = i_t \mid (x_1 \ldots x_T), \theta] \\ &= /\text{all } X_t \text{ stat. indep}/ \\ &= \prod_{t=1}^{T} \underbrace{P[S_t = i_t \mid x_t, \theta]}_{\gamma_{i_t,t}} \end{aligned} \tag{7.18}$$

where, using Bayes' law as usual, we have

$$\begin{aligned} \gamma_{1,t} &= P[S_t = 1 \mid x_t, \theta] = \frac{g(x_t, \mu_1, \sigma_1)}{g(x_t, \mu_1, \sigma_1) + g(x_t, \mu_2, \sigma_2)} \\ \gamma_{2,t} &= 1 - \gamma_{1,t} \end{aligned} \tag{7.19}$$

[3] In other applications, for example in HMM training, the hidden state variables are *not* independent.

Using the definition in Eq. (7.16) together with Eq. (7.17) we can immediately express the EM help function for this problem, as

$$\begin{aligned} &Q(\theta', \theta) = \\ &= \sum_{i_1=1}^{2} \sum_{i_2=1}^{2} \dots \sum_{i_T=1}^{2} P[\underline{S} = (i_1 \dots i_T) \mid \underline{x}, \theta] \sum_{t=1}^{T} \left(\ln 0.5 + \ln g(x_t, \mu'_{i_t}, \sigma'_{i_t})\right) \end{aligned} \tag{7.20}$$

We first rearrange the order of the nested sum, as

$$\begin{aligned} &Q(\theta', \theta) = \\ &= \sum_{t=1}^{T} \sum_{i_1=1}^{2} \sum_{i_2=1}^{2} \dots \sum_{i_T=1}^{2} P[\underline{S} = (i_1 \dots i_T) \mid \underline{x}, \theta] \left(\ln 0.5 + \ln g(x_t, \mu'_{i_t}, \sigma'_{i_t})\right) \end{aligned} \tag{7.21}$$

We then apply Bayes' rule as[4]

$$P[(i_1 \dots i_T) \mid \underline{x}, \theta] = P[(i_1 \dots i_{t-1} i_{t+1} \dots i_T) \mid S_t = i_t, \underline{x}, \theta] P[S_t = i_t \mid x_t, \theta] \tag{7.22}$$

Then, we rearrange the order of nested sums again to separate the sum over i_t, and simplify the Q function as

$$\begin{aligned} Q(\theta', \theta) = &\sum_{t=1}^{T} \sum_{i_t=1}^{2} \underbrace{P[S_t = i_t \mid x_t, \theta]}_{\gamma_{i_t,t} \text{ from Eq. (7.19)}} \left(\ln 0.5 + \ln g(x_t, \mu'_{i_t}, \sigma'_{i_t})\right) \cdot \\ &\cdot \underbrace{\sum_{i_1=1}^{2} \dots \sum_{i_{t-1}=1}^{2} \sum_{i_{t+1}=1}^{2} \dots \sum_{i_T=1}^{2} P[(i_1 \dots i_{t-1} i_{t+1} \dots i_T) \mid S_t = i_t, \underline{x}, \theta]}_{=1} \\ = &\sum_{t=1}^{T} \sum_{i=1}^{2} \gamma_{i,t} \left(\ln 0.5 + \ln g(x_t, \mu'_i, \sigma'_i)\right) = /\text{Eq. (7.15)}/ \\ = &\sum_{t=1}^{T} \sum_{i=1}^{2} \gamma_{i,t} \left(\ln 0.5 - \frac{1}{2} \ln 2\pi - \ln \sigma'_i - \frac{(x_t - \mu'_i)^2}{2\sigma'^2_i}\right) = \\ = &\sum_{t=1}^{T} \gamma_{1,t} \left(\ln 0.5 - \frac{1}{2} \ln 2\pi - \ln \sigma'_1 - \frac{(x_t - \mu'_1)^2}{2\sigma'^2_1}\right) + \\ &+ \gamma_{2,t} \left(\ln 0.5 - \frac{1}{2} \ln 2\pi - \ln \sigma'_2 - \frac{(x_t - \mu'_2)^2}{2\sigma'^2_2}\right) \end{aligned} \tag{7.23}$$

The complicated nested sum was first simplified using Bayes' rule. Then, we observed that the sum with 2^{T-1} terms in the second line is a sum of

[4] We use the shorthand notation $(i_1 \dots i_T)$ for the event $\underline{S} = (i_1 \dots i_T)$.

the probabilities of all possible state sequences $(i_1 \dots i_{t-1} i_{t+1} \dots i_T)$, and is therefore simply 1. As only one state-index summation remains, the summation index was also re-named as simply i instead of i_t.

Note again, that the EM help function requires us to take the expectation value of a quantity that depends on the free *new* parameter set θ', whereas the conditional state probabilities $\gamma_{i,t}$ are calculated using the *old* parameter estimate with known fixed values resulting from the previous iteration step.

To maximize the EM help function we must satisfy four equations

$$\begin{aligned} 0 = \frac{\partial q}{\partial \mu_i'} &= \sum_{t=1}^{T} \gamma_{i,t} \frac{(x_t - \mu_i')}{\sigma_i'^2}, \quad i \in \{1,2\} \\ 0 = \frac{\partial q}{\partial \sigma_i'} &= \sum_{t=1}^{T} \gamma_{i,t} \left(-\frac{1}{\sigma_i'} + \frac{(x_t - \mu_i')^2}{\sigma_i'^3} \right), \quad i \in \{1,2\} \end{aligned} \tag{7.24}$$

with solutions

$$\begin{aligned} \mu_i^{new} = \mu_i' &= \frac{\sum_{t=1}^{T} \gamma_{i,t} x_t}{\sum_{t=1}^{T} \gamma_{i,t}}, \quad i \in \{1,2\} \\ \sigma_i^{new} = \sigma_i' &= \sqrt{\frac{\sum_{t=1}^{T} \gamma_{i,t} (x_t - \mu_i^{new})^2}{\sum_{t=1}^{T} \gamma_{i,t}}}, \quad i \in \{1,2\} \end{aligned} \tag{7.25}$$

These update equations seem intuitively very natural: We estimate μ_i^{new} as a weighted average of all observed data, and $(\sigma_i^{new})^2$ is a weighted average of square deviations from the new estimated mean. The weight factor $\gamma_{i,t}$ is simply the conditional probability that the observation x_t was actually caused by an event where $S_t = i$.

Example 7.1

Example 7.2 (Nonlinear Signal Processing):
You observe a discrete-time random signal sequence $\underline{x} = (x_1 \dots x_T)$, which is known to be generated in a partly random filtering process as

$$\begin{aligned} x_0 &= 0 \\ x_t &= S_t a x_{t-1} + W_t, \quad t = 1, \dots T \end{aligned}$$

Here a is a constant but unknown parameter, $\underline{S} = (S_1 \dots S_T)$ is a discrete state sequence which is randomly chosen for each t as either $S_t = +1$ or $S_t = -1$, with equal probability, and $\underline{W} = (W_1 \dots W_T)$ is a white Gaussian noise sequence where each sample has zero mean and known variance σ^2. You can not observe the hidden $\underline{S}$ and $\underline{W}$ sequences, but you know that all samples in these sequences are statistically independent of each other. Determine an update rule to improve a previous estimate of a, using the Expectation Maximization (EM) approach.

Solution:

We denote an outcome of the hidden random state sequence as $\underline{S} = (i_1 \dots i_T)$, where each i_t can be either $+1$ or -1. The probability density for X_t depends only on the state S_t at the same time index, and on the previous observation x_{t-1}. Thus,

$$P\left[\underline{S} = (i_1 \dots i_T), \underline{x} \mid a\right] = \prod_{t=1}^{T} P\left[S_t = i_t, x_t \mid x_{t-1}, a\right]$$

where

$$\begin{aligned}
P\left[S_t = +1, x_t \mid x_{t-1}, a\right] &= \frac{1}{2}\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-(+1)ax_{t-1})^2}{2\sigma^2}} \\
P\left[S_t = -1, x_t \mid x_{t-1}, a\right] &= \frac{1}{2}\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-(-1)ax_{t-1})^2}{2\sigma^2}} \\
\gamma_{+,t} = P\left[S_t = +1 \mid x_t, x_{t-1}, a\right] &= \frac{e^{\frac{-(x_t - ax_{t-1})^2}{2\sigma^2}}}{e^{\frac{-(x_t - ax_{t-1})^2}{2\sigma^2}} + e^{\frac{-(x_t + ax_{t-1})^2}{2\sigma^2}}} \\
&= \frac{e^{\frac{ax_t x_{t-1}}{\sigma^2}}}{e^{\frac{ax_t x_{t-1}}{\sigma^2}} + e^{-\frac{ax_t x_{t-1}}{\sigma^2}}} \\
\gamma_{-,t} = 1 - \gamma_{+,t} &= \frac{e^{-\frac{ax_t x_{t-1}}{\sigma^2}}}{e^{\frac{ax_t x_{t-1}}{\sigma^2}} + e^{-\frac{ax_t x_{t-1}}{\sigma^2}}}
\end{aligned}$$

The EM help function can then be calculated as

$$\begin{aligned}
Q(a', a) &= \sum_{i_1} \cdots \sum_{i_T} P\left[\underline{S} = (i_1 \dots i_T) \mid \underline{x}, a\right] \ln P\left[\underline{S} = (i_1 \dots i_T), \underline{x} \mid a'\right] \\
&= \sum_{i_1} \cdots \sum_{i_T} P\left[\underline{S} = (i_1 \dots i_T) \mid \underline{x}, a\right] \sum_{t=1}^{T} \ln P\left[S_t = i_t, x_t \mid x_{t-1}, a'\right] \\
&= \sum_{t=1}^{T} \sum_{i} P\left[S_t = i \mid x_t, x_{t-1}, a\right] \ln P\left[S_t = i_t, x_t \mid x_{t-1}, a'\right] \\
&= \sum_{t=1}^{T} \gamma_{+,t} \frac{-(x_t - a'x_{t-1})^2}{2\sigma^2} + \gamma_{-,t} \frac{-(x_t + a'x_{t-1})^2}{2\sigma^2} + \dots
\end{aligned}$$

where $\dots$ stands for terms independent of a'. Now we can easily maximize

the expectation function as a function of a'. A necessary condition is

$$\begin{aligned} 0 = \frac{\partial Q(a', a)}{\partial a'} &= \sum_{t=1}^{T} \gamma_{+,t} \frac{(x_t - a' x_{t-1}) x_{t-1}}{\sigma^2} - \gamma_{-,t} \frac{(x_t + a' x_{t-1}) x_{t-1}}{\sigma^2} \\ &= \sum_{t=1}^{T} (\gamma_{+,t} - \gamma_{-,t}) \frac{x_t x_{t-1}}{\sigma^2} - \underbrace{(\gamma_{+,t} + \gamma_{-,t})}_{=1} \frac{a' x_{t-1}^2}{\sigma^2} \\ &= \sum_{t=2}^{T} (\gamma_{+,t} - \gamma_{-,t}) \frac{x_t x_{t-1}}{\sigma^2} - \frac{a' x_{t-1}^2}{\sigma^2} \end{aligned}$$

Noting that

$$\gamma_{+,t} - \gamma_{-,t} = \tanh \frac{a x_t x_{t-1}}{\sigma^2}$$

we can express the required EM update rule as

$$a^{(new)} = a' = \frac{\sum_{t=2}^{T} \tanh\left(\frac{a x_t x_{t-1}}{\sigma^2}\right) x_t x_{t-1}}{\sum_{t=2}^{T} x_{t-1}^2}$$

Example 7.2

7.5 Training a Gaussian Mixture Model

A weighted sum of Gaussian density functions can be used to approximate any complicated probability distribution. Therefore, the *Gaussian Mixture Model* (GMM) is commonly used in pattern classification. A GMM is just a generalisation of the model used in Eq. (7.13) in the previous section. The general GMM allows vector-valued observations, $\boldsymbol{x}$, and the mixing weight factors, w_m, are also free model parameters, not necessarily equal. The EM approach can be applied to mixtures of any types of density functions, not only Gaussian, as reviewed by Redner and Walker (1984).

In the GMM framework, we assume that any observed feature vector $\boldsymbol{x}$ from some particular source is an outcome of a random vector $\boldsymbol{X}$, with K elements, with a probability distribution formalized as a sum of M Gaussian components:

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \sum_{m=1}^{M} w_m \frac{1}{(2\pi)^{K/2} \sqrt{\det C_m}} e^{-\frac{1}{2}(\boldsymbol{x} - \mu_m)^T C_m^{-1} (\boldsymbol{x} - \mu_m)} \tag{7.26}$$

where μ_m is the mean vector and C_m is the covariance matrix for the m-th mixture component. The weight factors w_m must be normalised such that

$$\sum_{m=1}^{M} w_m = 1 \tag{7.27}$$

All the unknown parameters $\theta = \{w_m, \mu_m, C_m : m = 1 \dots M\}$ will be estimated to make the model agree as well as possible with a sequence of training observation vectors $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_T)$. Each observed vector $\boldsymbol{x}_t$ is regarded as an outcome of a random vector $\boldsymbol{X}_t$. All these random vectors are statistically independent for different t, and they are all characterized by the same density function (7.26).

To express this problem in the EM framework, we assume that each $\boldsymbol{X}_t$ depends on a hidden discrete "state" variable[5] U_t that we can not observe. The distribution of the random variable $\boldsymbol{X}_t$ is the result of a two-step random procedure:

- First, the hidden random variable U_t is chosen as $U_t = m \in \{1 \dots M\}$, with probability $P[U_t = m] = w_m$, independently of the outcome at any other t.
- Then, given the value of $U_t = m$, a value of $\boldsymbol{X}_t$ is drawn from the simple Gaussian conditional distribution

$$f_{\boldsymbol{X}_t|U_t}(\boldsymbol{x} \mid m) = g(\boldsymbol{x}, \mu_m, C_m) = \\ = \frac{1}{(2\pi)^{K/2}\sqrt{\det C_m}} e^{-\frac{1}{2}(\boldsymbol{x}-\mu_m)^T C_m^{-1}(\boldsymbol{x}-\mu_m)} \quad (7.28)$$

In a similar way as in Eq. (7.19) in the previous simple example, we calculate the conditional state probabilities, for each observed vector $\boldsymbol{x}_t$, using the old (non-primed) parameter values as

$$\gamma_{m,t} = P[U_t = m \mid \boldsymbol{x}_t, \theta] = \frac{w_m g(\boldsymbol{x}_t, \mu_m, C_m)}{\sum_{k=1}^{M} w_k g(\boldsymbol{x}_t, \mu_k, C_k)} \quad (7.29)$$

The EM help function is calculated in a similar way as in Eq. (7.23) in the previous section:

$$\begin{aligned} Q(\theta', \theta) &= \sum_{t=1}^{T}\sum_{m=1}^{M} \gamma_{m,t} \left(\ln w'_m + \ln g(\boldsymbol{x}_t, \mu'_m, C'_m)\right) = /\text{Eq. (7.28)}/ \\ &= -\frac{TMK}{2}\ln 2\pi + \underbrace{\sum_{t=1}^{T}\sum_{m=1}^{M} \gamma_{m,t} \ln w'_m}_{Q_1} + \\ &+ \underbrace{\sum_{t=1}^{T}\sum_{m=1}^{M} \gamma_{m,t} \left(-\frac{1}{2}\ln\det C'_m - \frac{1}{2}(\boldsymbol{x}_t - \mu'_m)^T {C'_m}^{-1}(\boldsymbol{x}_t - \mu'_m)\right)}_{Q_2} \\ &= const. + Q_1(\theta', \theta) + Q_2(\theta', \theta) \end{aligned} \quad (7.30)$$

[5] The states are arbitrarily called U_t here just to emphasise that this is *not* a HMM state. In later sections we must distinguish HMM states S_t from "sub-states" U_t.

Note that the Q function now includes one constant term and two separate sums. One sum, Q_1, can be maximised as a function of only the w'_m parameters, and the other, Q_2, depends only on the Gaussian parameters μ'_m and C'_m, for $m = 1 \ldots M$. Therefore, to mazimize the total Q function, we can maximize these two sums separately. The details are shown in the next two sections.

7.5.1 Updating the GMM Weight Factors

We will now maximize the function

$$\begin{aligned} Q_1(w'_1, \ldots, w'_M) &= \sum_{t=1}^{T} \sum_{m=1}^{M} \gamma_{m,t} \ln w'_m = \\ &= \sum_{t=1}^{T} (\gamma_{1,t} \ln w'_1 + \ldots + \gamma_{M,t} \ln w'_M) \end{aligned} \tag{7.31}$$

by varying the parameters $w'_1 \ldots w'_M$. However, we do not really have M free parameters, because of the extra constraint

$$w'_1 + w'_2 + \ldots + w'_M = 1. \tag{7.32}$$

To account for this constraint, we apply the technique with a Lagrange multiplier called ν here (see e.g. Råde and Westergren, 1995, section 10.5). We define a new criterion function including the constraint, as

$$F = \sum_{t=1}^{T} (\gamma_{1,t} \ln w'_1 + \ldots + \gamma_{M,t} \ln w'_M) + \nu \left(1 - (w'_1 + \ldots + w'_M)\right) \tag{7.33}$$

To maximize[6] Q_1, given the constraint, we must satisfy the following M equations:

$$0 = \frac{\partial F}{\partial w'_m} = \sum_{t=1}^{T} \frac{\gamma_{m,t}}{w'_m} - \nu, \quad m = 1 \ldots M \tag{7.34}$$

The equations have solutions

$$w'_m = \frac{1}{\nu} \sum_{t=1}^{T} \gamma_{m,t}, \quad m = 1 \ldots M \tag{7.35}$$

given any value of the Lagrange multiplier ν. However, to satisify also the given constraint (7.32), we must choose

$$\nu = \sum_{k=1}^{M} \sum_{t=1}^{T} \gamma_{k,t} \tag{7.36}$$

[6]The reader can easily verify that all the second partial derivatives of F are negative, so this is certainly a maximum and not a minimum.

Then, combining (7.35) and (7.36), the final update formula can be expressed as

$$w_m^{new} = w'_m = \frac{\sum_{t=1}^{T} \gamma_{m,t}}{\sum_{k=1}^{M} \sum_{t=1}^{T} \gamma_{k,t}}, \quad m = 1 \dots M \tag{7.37}$$

7.5.2 Updating the Gaussian Parameters

We must now also find values of all mean vectors μ'_m and covariance matrices C'_m to maximize the total EM help function of Eq. (7.30). To do this we maximize the second component of that help function, i.e., the function

$$Q_2 = \sum_{t=1}^{T} \sum_{m=1}^{M} \gamma_{m,t} \left(-\frac{1}{2} \ln \det C'_m - \frac{1}{2} (\boldsymbol{x}_t - \mu'_m)^T {C'_m}^{-1} (\boldsymbol{x}_t - \mu'_m) \right) \tag{7.38}$$

As usual, we must calculate all partial derivatives of Q_2 with respect to every element of the vectors μ'_m and matrices C'_m. To simplify these calculations, the following matrix notation is useful:

If $f(\boldsymbol{A})$ is a scalar function of a matrix $\boldsymbol{A}$ with elements a_{ij}, we define

$$\frac{\partial f(\boldsymbol{A})}{\partial \boldsymbol{A}} = \begin{pmatrix} \frac{\partial f(\boldsymbol{A})}{\partial a_{11}} & \frac{\partial f(\boldsymbol{A})}{\partial a_{12}} & \dots & \frac{\partial f(\boldsymbol{A})}{\partial a_{1j}} & \dots \\ \frac{\partial f(\boldsymbol{A})}{\partial a_{21}} & \ddots & \dots & \dots & \vdots \\ \vdots & \dots & & \dots & \\ \frac{\partial f(\boldsymbol{A})}{\partial a_{i1}} & \dots & \dots & \frac{\partial f(\boldsymbol{A})}{\partial a_{ij}} & \vdots \\ \vdots & \dots & \dots & \dots & \ddots \end{pmatrix} \tag{7.39}$$

This is a matrix with the same size as $\boldsymbol{A}$, containing all partial derivatives with respect to each element of $\boldsymbol{A}$.

Using this notation, the following relations can be shown for any square matrix $\boldsymbol{A}$ and column vector $\boldsymbol{x}$ (see problem 7.1)

$$\frac{\partial \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}}{\partial \boldsymbol{x}} = (\boldsymbol{A}^T + \boldsymbol{A}) \boldsymbol{x} \tag{7.40}$$

$$\frac{\partial \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}}{\partial \boldsymbol{A}} = \boldsymbol{x} \boldsymbol{x}^T \tag{7.41}$$

$$\frac{\partial \ln \det \boldsymbol{A}}{\partial \boldsymbol{A}^{-1}} = -\boldsymbol{A}^T \tag{7.42}$$

Noting that the covariance matrices C'_m are symmetric, we can now obtain partial derivatives of Q_2 with respect to each mean vector μ'_m and with respect to each inverse covariance matrix ${C'_m}^{-1}$. To maximize Q_2 we

must satisfy the following equations, for each $m = 1 \dots M$:

$$\begin{aligned} 0 = \frac{\partial Q_2}{\partial \mu'_m} &= \sum_{t=1}^{T} \gamma_{m,t} \left(-\frac{1}{2}\right) \frac{\partial}{\partial \mu'_m} (\boldsymbol{x}_t - \mu'_m)^T {C'_m}^{-1} (\boldsymbol{x}_t - \mu'_m) = \\ &= /\text{applying Eq. (7.40)}/ = \\ &= \sum_{t=1}^{T} \gamma_{m,t} {C'_m}^{-1} (\boldsymbol{x}_t - \mu'_m) \end{aligned} \tag{7.43}$$

$$\begin{aligned} 0 = \frac{\partial Q_2}{\partial {C'_m}^{-1}} &= -\frac{1}{2} \sum_{t=1}^{T} \gamma_{m,t} \frac{\partial \left(\ln \det C'_m + (\boldsymbol{x}_t - \mu'_m)^T {C'_m}^{-1} (\boldsymbol{x}_t - \mu'_m) \right)}{\partial {C'_m}^{-1}} \\ &= \frac{1}{2} \sum_{t=1}^{T} \gamma_{m,t} \left(C'_m - (\boldsymbol{x}_t - \mu'_m)(\boldsymbol{x}_t - \mu'_m)^T \right) \end{aligned} \tag{7.44}$$

The solutions are

$$\mu_m^{new} = \mu'_m = \frac{\sum_{t=1}^{T} \gamma_{m,t} \boldsymbol{x}_t}{\sum_{t=1}^{T} \gamma_{m,t}} \tag{7.45}$$

$$C_m^{new} = C'_m = \frac{\sum_{t=1}^{T} \gamma_{m,t} (\boldsymbol{x}_t - \mu_m^{new})(\boldsymbol{x}_t - \mu_m^{new})^T}{\sum_{t=1}^{T} \gamma_{m,t}} \tag{7.46}$$

These are the desired update equations to be applied in each step of the EM algorithm. These equations seem intuitively very natural: We estimate μ_m^{new} as a weighted average of all observed data, and C_m^{new} as a similarly weighted average of all observed covariances in the data. The weight $\gamma_{m,t}$, used in the averaging, is simply the conditional probability that the observation x_t was actually caused by an event where $U_t = m$.

7.6 HMM Training

This section will apply the EM approach to the problem of HMM training, and show that this approach leads to the Baum-Welch update equations already presented intuitively in Sec. 6.1.

A HMM is completely defined by the parameter set $\lambda = (q, A, B)$ as defined in Sec. 5.1. For each iterative step in the EM algorithm we maximise the EM help function $Q(\lambda', \lambda)$ by varying the hypothetical new parameter set $\lambda' = (q', A', B')$.

The training data may consist of R concatenated subsequences, as discussed in Sec. 6.1, which may be written as

$$\underline{\boldsymbol{x}} = ((\boldsymbol{x}_1^1 \dots \boldsymbol{x}_{T_1}^1), (\boldsymbol{x}_1^2 \dots \boldsymbol{x}_{T_2}^2), \dots, (\boldsymbol{x}_1^R \dots \boldsymbol{x}_{T_R}^R)) \tag{7.47}$$

However, for the EM approach it is easier to write the total set of observed training data simply as $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$.

If the HMM has N states, the EM help function, defined by Eq. (7.8), is a mean across all possible state sequences:

$$Q(\lambda', \lambda) = \sum_{i_1=1}^{N} \sum_{i_2=1}^{N} \cdots \sum_{i_T=1}^{N} P[\underline{S} = (i_1 \ldots i_T) \mid \underline{\boldsymbol{x}}, \lambda] \ln P[\underline{S} = (i_1 \ldots i_T) \cap \underline{\boldsymbol{x}} \mid \lambda'] \tag{7.48}$$

For any particular outcome of the state sequence, we have

$$P[\underline{S} = (i_1 \ldots i_T) \cap \underline{\boldsymbol{x}} \mid \lambda'] = P[\underline{S} = (i_1 \ldots i_T) \mid \lambda'] P[\underline{\boldsymbol{x}} \mid \underline{S} = (i_1 \ldots i_T), \lambda'] \tag{7.49}$$

which can be rewritten using

$$P[\underline{S} = (i_1 \ldots i_T) \mid \lambda'] = q'_{i_1} a'_{i_1 i_2} a'_{i_2 i_3} \cdots a'_{i_{T-1} i_T} \tag{7.50}$$

and

$$P[\underline{\boldsymbol{x}} \mid \underline{S} = (i_1 \ldots i_T), \lambda'] = b'_{i_1}(\boldsymbol{x}_1) b'_{i_2}(\boldsymbol{x}_2) \cdots b'_{i_T}(\boldsymbol{x}_T) \tag{7.51}$$

Using Eqs. (7.50) and (7.51) we can now simplify the EM help function in Eq. (7.48) as a sum of three separate components, by collecting the q', a', and b' logarithm terms in separate parts of the total sum:

$$\begin{aligned} Q(\lambda', \lambda) = & \underbrace{\sum_{i_1=1}^{N} \sum_{i_2=1}^{N} \cdots \sum_{i_T=1}^{N} P[\underline{S} = (i_1 \ldots i_T) \mid \underline{\boldsymbol{x}}, \lambda] \cdot \sum_{t \in \tau_1} \ln q'_{i_t}}_{Q_1(q', \lambda)} + \\ & + \underbrace{\sum_{i_1=1}^{N} \sum_{i_2=1}^{N} \cdots \sum_{i_T=1}^{N} P[\underline{S} = (i_1 \ldots i_T) \mid \underline{\boldsymbol{x}}, \lambda] \cdot \sum_{t \in \tau_-} \ln a'_{i_t i_{t+1}}}_{Q_2(A', \lambda)} + \\ & + \underbrace{\sum_{i_1=1}^{N} \sum_{i_2=1}^{N} \cdots \sum_{i_T=1}^{N} P[\underline{S} = (i_1 \ldots i_T) \mid \underline{\boldsymbol{x}}, \lambda] \cdot \sum_{t=1}^{T} \ln b'_{i_t}(\boldsymbol{x}_t)}_{Q_3(B', \lambda)} \\ & = Q_1(q', \lambda) + Q_2(A', \lambda) + Q_3(B', \lambda) \end{aligned} \tag{7.52}$$

Here, we have defined two subsets of observation indices just to keep track of the different sub-sequences in the training data $\underline{\boldsymbol{x}}$:

- τ_1 is a set containing all the t indices identifying the *first* observation in each sub-sequence.
- τ_- contains all t indices *except the last* in each sub-sequence.

To maximize the total Q function, we can maximize the three components Q_1, Q_2, and Q_3 separately, as Q_1 is a function of only the initial probability vector q', Q_2 depends only on the transition probabilites in the matrix A', and Q_3 depends only on the set of output distributions B'. Thus, for each step in the EM algorithm we must find three separate update functions, as already noted in Eq. (6.6) in Sec. 6.1,

$$\begin{aligned}\{\{q, A\}, B\} &\rightarrow q^{new} \\ \{\{q, A\}, B\} &\rightarrow A^{new} \\ \{\{q, A\}, B\} &\rightarrow B^{new}\end{aligned}$$

These three separate problems are solved in the following sub-sections.

7.6.1 Updating the Initial Probability Vector

In order to find an improved version of the HMM initial probability vector q', we must maximize the first component of the EM help function obtained in Eq. (7.52),

$$\begin{aligned}Q_1(q', \lambda) &= \sum_{i_1=1}^{N} \dots \sum_{i_T=1}^{N} P[\underline{S} = (i_1 \dots i_T) \mid \underline{\boldsymbol{x}}, \lambda] \cdot \sum_{t \in \tau_1} \ln q'_{i_t} = \\ &= \sum_{t \in \tau_1} \sum_{i_t=1}^{N} \underbrace{P[S_t = i_t \mid \underline{\boldsymbol{x}}, \lambda]}_{\gamma_{i_t,t}} \ln q'_{i_t} \cdot \\ &\underbrace{\sum_{i_1} \dots \sum_{i_{t-1}} \sum_{i_{t+1}} \dots \sum_{i_T} P[(i_1 \dots i_{t-1} i_{t+1} \dots i_T) \mid S_t = i_t, \underline{\boldsymbol{x}}, \lambda]}_{=1} = \\ &= \sum_{t \in \tau_1} \sum_{i=1}^{N} \gamma_{i,t} \ln q'_i\end{aligned} \tag{7.53}$$

Here, the complicated sum was rearranged and greatly simplified using Bayes' rule, noting that the third line is a sum of probabilities of all possible outcomes, and therefore equals 1. The weight factors $\gamma_{i,t}$ express the conditional probability that the hidden state was $S_t = i$ at time t, exactly as defined previously in Sec. 5.5. Note that all γ values are calculated using the *previous* (non-primed) model parameter set λ by applying the Forward and Backward algorithms on all available training sub-sequences.

When maximising the function $Q_1(q', \lambda)$ by varying the initial state probability vector q', we do not really have N free parameters, because of the constraint that

$$\sum_{i=1}^{N} q'_i = 1$$

To account for this constraint, we apply the technique with a Lagrange multiplier, exactly as in the very similar problem solved in Sec. 7.5.1. The solution is the final update equation

$$q_i^{new} = \frac{\sum_{t\in\tau_1}\gamma_{i,t}}{\sum_{k=1}^{N}\sum_{t\in\tau_1}\gamma_{k,t}}, \quad i = 1\dots N \tag{7.54}$$

Using the alternative notation with R separate training subsequences, we calculate conditional state probabilities $\gamma_{i,t}^r$ separately for each sub-sequence $(\boldsymbol{x}_1^r \dots \boldsymbol{x}_{T_r}^r)$. As the set τ_1 indicates we should use only the first element $\gamma_{i,1}^r$ in each subsequence, the update equation can be equivalently expressed as

$$q_i^{new} = \frac{\sum_{r=1}^{R}\gamma_{i,1}^r}{\sum_{k=1}^{N}\sum_{r=1}^{R}\gamma_{k,1}^r} = \frac{1}{R}\sum_{r=1}^{R}\gamma_{i,1}^r \tag{7.55}$$

The result is intuitively very natural. The new parameter q_i^{new} is simply the average conditional probability that the initial state was i, given all the training observations. This formal derivation also confirms that the intuitive approach in Sec. 6.1.1 was correct.

7.6.2 Updating the Transition Probability Matrix

In this section we find a transition probability matrix A', with elements a'_{ij}, that maximizes the second component of the EM help function defined in Eq. (7.52):

$$\begin{aligned} Q_2(A',\lambda) &= \sum_{i_1=1}^{N}\dots\sum_{i_T=1}^{N} P[\underline{S} = (i_1\dots i_T) \mid \underline{\boldsymbol{x}},\lambda]\cdot\sum_{t\in\tau_-}\ln a'_{i_t i_{t+1}} = \\ &= \sum_{t\in\tau_-}\sum_{i_t=1}^{N}\sum_{i_{t+1}=1}^{N}\underbrace{P[S_t = i_t \cap S_{t+1} = i_{t+1} \mid \underline{\boldsymbol{x}},\lambda]}_{\xi_{i_t i_{t+1},t}}\ln a'_{i_t i_{t+1}}\cdot \\ &\underbrace{\sum_{i_1}\dots\sum_{i_{t-1}}\sum_{i_{t+2}}\dots\sum_{i_T} P[(i_1\dots i_{t-1}i_{t+2}\dots i_T) \mid i_t, i_{t+1}, \underline{\boldsymbol{x}},\lambda]}_{=1} = \\ &= \sum_{t\in\tau_-}\sum_{i=1}^{N}\sum_{j=1}^{N}\xi_{ij,t}\ln a'_{ij} \end{aligned} \tag{7.56}$$

Here, the complicated sum was rearranged and greatly simplified using Bayes' rule, noting that the third line is a sum of probabilities of all possible outcomes, and therefore equals 1. The factors $\xi_{ij,t}$ express the conditional probability, given all observations, that the hidden state was $S_t = i \cap S_{t+1} = j$, exactly as defined previously in Sec. 6.1.2. Note that $\xi_{ij,t}$ is calculated using the *previous* (non-primed) model parameter set λ.

When maximising the function $Q_2(A', \lambda)$ by varying all elements of the transition probability matrix A', we also have N extra constraints, one for each row in the matrix:

$$\sum_{j=1}^{N} a'_{ij} = 1, \quad i = 1 \ldots N$$

To account for these constraints, we apply one separate Lagrange multiplier ν_i for each constraint (Råde and Westergren, 1995, section 10.5). Necessary conditions for a maximum[7] are then given by the set of N^2 equations

$$\begin{aligned} 0 &= \frac{\partial}{\partial a'_{ij}} \Big(Q_2(A', \lambda) + \nu_i (1 - \sum_{k=1}^{N} a'_{ik}) \Big) \\ &= \sum_{t \in \tau_-} \frac{\xi_{ij,t}}{a'_{ij}} - \nu_i \end{aligned} \tag{7.57}$$

with solutions, including the normalizing constraint,

$$a_{ij}^{new} = \frac{1}{\nu_i} \sum_{t \in \tau_-} \xi_{ij,t}; \quad \nu_i = \sum_{k=1}^{N} \sum_{t \in \tau_-} \xi_{ik,t} \tag{7.58}$$

Using the alternative notation with $\xi^r_{ij,t}$ calculated separately for each training sub-sequence $(\boldsymbol{x}^r_1 \ldots \boldsymbol{x}^r_{T_r})$, these update equations can be equivalently expressed as

$$\begin{aligned} \overline{\xi}_{ij} &= \sum_{r=1}^{R} \sum_{t=1}^{T_r - 1} \xi^r_{ij,t} \\ a_{ij}^{new} &= \frac{\overline{\xi}_{ij}}{\sum_{k=1}^{N} \overline{\xi}_{ik}} \end{aligned} \tag{7.59}$$

This result is intuitively natural. Element (i, j) of the matrix $\overline{\xi}$ is a sum across all t of the conditional probability of the event $S_t = i \cap S_{t+1} = j$, given all the training data and the previous model parameter set. The updated transition probability matrix A^{new} is then obtained by the required normalisation of each matrix row. This formal derivation confirms that the tentative approach in Sec. 6.1.2 was correct.

7.6.3 Updating the Output Probability Distributions

Finally, in this section we derive an improved version of the HMM output probability distributions by varying B' to maximize the third component of

[7] It is easy to verify that all the second partial derivatives are negative, so it is a maximum and not a minimum.

the EM help function defined in Eq. (7.52):

$$\begin{aligned} Q_3(B', \lambda) &= \sum_{i_1=1}^{N} \dots \sum_{i_T=1}^{N} P[\underline{S} = (i_1 \dots i_T) \mid \underline{\boldsymbol{x}}, \lambda] \cdot \sum_{t=1}^{T} \ln b'_{i_t}(\boldsymbol{x}_t) \\ &= \sum_{t=1}^{T} \sum_{i_t=1}^{N} \underbrace{P[S_t = i_t \mid \underline{\boldsymbol{x}}, \lambda]}_{\gamma_{i_t,t}} \ln b'_{i_t}(\boldsymbol{x}_t) \cdot \\ &\underbrace{\sum_{i_1} \dots \sum_{i_{t-1}} \sum_{i_{t+1}} \dots \sum_{i_T} P[(i_1 \dots i_{t-1} i_{t+1} \dots i_T) \mid S_t = i_t, \underline{\boldsymbol{x}}, \lambda]}_{=1} \\ &= \sum_{t=1}^{T} \sum_{i=1}^{N} \gamma_{i,t} \ln b'_i(\boldsymbol{x}_t) \end{aligned} \tag{7.60}$$

The further details depend of course on the mathematical form assumed for the output distributions $b'_i(\boldsymbol{x}_t)$.

7.6.4 Updating a Discrete Output Probability Distribution

If the HMM output is *discrete*, i.e. a sequence $\boldsymbol{z} = (z_1 \dots z_T)$ of integer values, the possible conditional output distributions are represented by an $N \times M$ matrix B' with elements $b'_{im} = P[Z_t = m \mid S_t = i, \lambda']$

In this case the EM function can be expressed as

$$Q_3(B', \lambda) = \sum_{t=1}^{T} \sum_{i=1}^{N} \gamma_{i,t} \ln b'_{i,z_t} \tag{7.61}$$

We must then account for the N constraints

$$\sum_{m=1}^{M} b'_{im} = 1 \tag{7.62}$$

by applying one separate Lagrange multiplier ν_i for each constraint. A necessary condition for maximum is the set of NM equations, one for each $i = 1, \dots, N$ and $m = 1, \dots, M$:

$$\begin{aligned} 0 &= \frac{\partial}{\partial b'_{im}} \Big(Q_3(B', \lambda) + \nu_i (1 - \sum_{m=1}^{M} b'_{im}) \Big) \\ &= \sum_{t=1}^{T} \frac{\gamma_{i,t}}{b'_{im}} \delta_{z_t,m} - \nu_i \end{aligned} \tag{7.63}$$

Here, Kronecker's δ is used to indicate that the partial derivative has non-zero contributions only for those t where $z_t = m$. The solutions are, including the normalizing constraint,

$$b^{new}_{im} = b'_{im} = \frac{1}{\nu_i} \sum_{t=1}^{T} \gamma_{i,t} \delta_{z_t,m}; \quad \nu_i = \sum_{k=1}^{M} \sum_{t=1}^{T} \gamma_{i,t} \delta_{z_t,k} \tag{7.64}$$

Using again the explicit notation with $\gamma_{i,t}^r$ calculated separately for each training sub-sequence, these update equations can be equivalently expressed as

$$\begin{aligned} \overline{b}_{im} &= \sum_{r=1}^{R} \sum_{t=1}^{T_r} \gamma_{i,t}^r \delta_{z_t^r,m} \\ b_{im}^{new} &= \frac{\overline{b}_{im}}{\sum_{k=1}^{M} \overline{b}_{ik}} \end{aligned} \tag{7.65}$$

The result is intuitively natural and confirms that the tentative approach in Sec. 6.1.3 was correct.

7.6.5 Updating GMM Output Distributions

As mentioned already in Sec. 5.1, when the HMM output sequence consists of continous-valued vectors of length K, Gaussian Mixture Models (GMM) are often used to approximate the probability density of the output vectors (Rabiner and Juang, 1986; Rabiner, 1989; Young et al., 2002). In this case each state-conditional output distribution is formalised as a GMM, just as described previously in Eq. (7.26):

$$\begin{aligned} f_{\boldsymbol{X}_t|S_t}(\boldsymbol{x}_t \mid i) = b_i(\boldsymbol{x}_t) &= \sum_{m=1}^{M} w_{im} g(\boldsymbol{x}_t, \mu_{im}, C_{im}), \qquad \text{where} \\ g(\boldsymbol{x}_t, \mu_{im}, C_{im}) &= \frac{1}{(2\pi)^{K/2}\sqrt{\det C_{im}}} e^{-\frac{1}{2}(\boldsymbol{x_t}-\mu_{im})^T C_{im}^{-1}(\boldsymbol{x_t}-\mu_{im})} \\ \sum_{m=1}^{M} w_{im} &= 1 \end{aligned} \tag{7.66}$$

Here, each of the NM probability density functions $g(\,)$ is a single Gaussian-vector density, specified by its mean vector μ_{im} with length K, and covariance matrix C_{im} with size $K \times K$. As described in Sec. 7.5 the output distributions can also be described by assuming that each observed output vector is created in a two-step random procedure:

- First, given that the HMM state was $S_t = i$, another discrete hidden "sub-state" variable U_t is chosen as $U_t = m \in \{1 \dots M\}$, with probability $P(U_t = m \mid S_t = i) = w_{im}$, independently of the outcome at any other t.

- Then, depending on the value of both S_t and U_t, a value of $\boldsymbol{X}_t$ is drawn from a simple Gaussian distribution with conditional density function

$$f_{\boldsymbol{X}_t|S_t,U_t}(\boldsymbol{x}_t \mid i, m) = g(\boldsymbol{x}_t, \mu_{im}, C_{im}) \tag{7.67}$$

In this model framework, the HMM output distribution set B is completely specified by the parameter set $\{w_{im}, \mu_{im}, C_{im}; i = 1 \ldots N; m = 1 \ldots M\}$. To improve all these $NM(1 + K + K^2)$ parameters[8] in each step of the EM algorithm, we must first slightly redefine the third component of the EM help function. We must apply the EM trick using *two* sets of hidden random variables: the ordinary HMM state sequence $\underline{S} = (S_1 \ldots S_T)$ and the additional sequence of sub-state variables $\boldsymbol{U} = (U_1 \ldots U_T)$. Therefore, the EM help function can be calculated in analogy with Eq. (7.60) as

$$\begin{aligned} Q_3(B', \lambda) &= \sum_{t=1}^{T} \sum_{i=1}^{N} \sum_{m=1}^{M} \underbrace{P[S_t = i \cap U_t = m \mid \underline{\boldsymbol{x}}, \lambda]}_{\gamma_{im,t}} \cdot \\ &\quad \cdot \ln P[U_t = m \cap \boldsymbol{X}_t = \boldsymbol{x}_t \mid S_t = i, \lambda'] \\ &= \sum_{t=1}^{T} \sum_{i=1}^{N} \sum_{m=1}^{M} \gamma_{im,t} (\ln w'_{im} + \ln g(\boldsymbol{x}_t, \mu'_{im}, C'_{im})) \end{aligned} \tag{7.68}$$

where we use the *previous* (non-primed) parameter set to calculate

$$\begin{aligned} \gamma_{im,t} &= P[S_t = i \cap U_t = m \mid \underline{\boldsymbol{x}}, \lambda] = \\ &= \underbrace{P[S_t = i \mid \underline{\boldsymbol{x}}, \lambda]}_{\gamma_{i,t}} P[U_t = m \mid S_t = i, \underline{\boldsymbol{x}}, \lambda] \\ &= \gamma_{i,t} \frac{w_{im} g(\boldsymbol{x}_t, \mu_{im}, C_{im})}{\sum_{k=1}^{M} w_{ik} g(\boldsymbol{x}_t, \mu_{ik}, C_{ik})} \end{aligned} \tag{7.69}$$

where $\gamma_{i,t} = P[S_t = i \mid \underline{\boldsymbol{x}}, \lambda]$ is calculated as usual with the Forward-Backward procedure. The detailed derivation of Eq. (7.69) is left as an exercise.

This problem is now very similar to the GMM training we solved in Sec. 7.5. The resulting update equations are

$$\begin{aligned} w_{im}^{new} &= \frac{\sum_{t=1}^{T} \gamma_{im,t}}{\sum_{k=1}^{M} \sum_{t=1}^{T} \gamma_{ik,t}} \\ \mu_{im}^{new} &= \frac{\sum_{t=1}^{T} \gamma_{im,t} \boldsymbol{x}_t}{\sum_{t=1}^{T} \gamma_{im,t}} \\ C_{im}^{new} &= \frac{\sum_{t=1}^{T} \gamma_{im,t} (\boldsymbol{x}_t - \mu_{im}^{new})(\boldsymbol{x}_t - \mu_{im}^{new})^T}{\sum_{t=1}^{T} \gamma_{im,t}} \end{aligned} \tag{7.70}$$

As usual, these update equations are intuitively very natural, considering the intended meaning of the parameters.

[8] It is rather common to reduce the number of parameters by forcing the covariance matrices C_{im} to be *diagonal*, because the mixture sum over m can still model any correlation between the K output vector elements.

Summary

This chapter formally derived training methods to optimize hidden Markov Models to conform with *training data*, i.e. a set of concatenated observed feature-vector sequences, organized as

$$\underline{\boldsymbol{x}} = ((\boldsymbol{x}_1^1 \dots \boldsymbol{x}_{T_1}^1), (\boldsymbol{x}_1^2 \dots \boldsymbol{x}_{T_2}^2), \dots, (\boldsymbol{x}_1^R \dots \boldsymbol{x}_{T_R}^R))$$

Expectation Maximization: For any probabilistic model (not only HMM) with parameters λ, assuming a hidden state sequence $\underline{S}$ and using an observed sequence $\underline{\boldsymbol{x}}$, find improved model parameters as

$$\lambda^{new} = \operatorname*{argmax}_{\lambda'} Q(\lambda', \lambda), \quad \text{where}$$
$$Q(\lambda', \lambda) = E_{\underline{S}} \left[\log P \left[\underline{S}, \underline{\boldsymbol{x}} \mid \lambda' \right] \mid \underline{\boldsymbol{x}}, \lambda \right]$$

Problems

7.1 To perform the Maximization step of the EM algorithm it is often necessary to calculate all the partial derivatives of some scalar function $f(\boldsymbol{A})$ with respect to each element a_{ij} of the argument matrix $\boldsymbol{A}$. Let us use the notation defined in Eq. (7.39), collecting all those partial derivatives in a matrix:

$$\frac{\partial f(\boldsymbol{A})}{\partial \boldsymbol{A}} = \begin{pmatrix} \frac{\partial f(\boldsymbol{A})}{\partial a_{11}} & \frac{\partial f(\boldsymbol{A})}{\partial a_{12}} & \cdots & \cdots \\ \frac{\partial f(\boldsymbol{A})}{\partial a_{21}} & \cdots & \cdots & \cdots \\ \vdots & \cdots & \frac{\partial f(\boldsymbol{A})}{\partial a_{ij}} & \cdots \\ \vdots & \cdots & \cdots & \cdots \end{pmatrix}$$

Calculate the following partial-derivative matrices and express the result in matrix form, assuming that $\boldsymbol{x}$ is a column vector and $\boldsymbol{A}$ is a square matrix.

7.1.a

$$\frac{\partial \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}}{\partial \boldsymbol{x}}$$

7.1.b

$$\frac{\partial \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}}{\partial \boldsymbol{A}}$$

7.1.c

$$\frac{\partial \det \boldsymbol{A}}{\partial \boldsymbol{A}}$$

Hint: Develop the determinant in terms of *cofactors* (Råde and Westergren, 1995, sec. 4.2).

7.1.d

$$\frac{\partial \ln \det \boldsymbol{A}}{\partial \boldsymbol{A}}$$

Hint: Note that $\det \boldsymbol{A}^{-1} = 1/\det \boldsymbol{A}$, and the inverse matrix can also be expressed in terms of *cofactors*.

7.1.e

$$\frac{\partial \ln \det \boldsymbol{A}}{\partial \boldsymbol{A}^{-1}}$$

7.2 (Exam001027:5) You have in your pocket two equal-looking coins, numbered as $S \in \{1, 2\}$. One of them is symmetrical, and the other one is asymmetrically balanced. You have coded the possible coin toss outcomes as $X = 1$, indicating a "head", and $X = 0$ meaning you obtained a "tail". You assume that the conditional observation probabilities are

$$P(X = 0|S = 1) = P(X = 1|S = 1) = 0.5, \text{ for the correct coin}$$
$$P(X = 1|S = 2) = \theta, \text{ for the asymmetrical coin}$$

but you do not yet know the exact value of θ. At the first trial you choose one of the coins at random, and then you systematically toss the two coins alternatingly ($ABABAB\ldots$). Thus, one coin is used in all odd-numbered time positions and the other coin at all even-numbered trials. In a sequence of 10 trials you have just obtained the outcome $\underline{x} = (0100110111)$, i.e. you see $x_t = 1$ at $K_o = 2$ times out of the $N_o = 5$ odd-numbered trials, and at $K_e = 4$ times out of the $N_e = 5$ even-numbered trials.

7.2.a Determine an iterative update rule to approach an estimate of the unknown parameter θ, using the Expectation Maximization (EM) algorithm. Derive a general expression as a function of K_o, K_e, N_o, N_e without assuming that $N_o = N_e$.
Hint: Given a starting value of θ, each step in the EM algorithm seeks a new parameter value θ' that maximizes

$$Q(\theta', \theta) = E\left[\ln P(\underline{S}, \underline{x}|\theta')|\underline{x}, \theta\right]$$

where $\boldsymbol{x}$ is the observed data sequence and $\underline{S}$ is the hidden state sequence.

7.2.b Apply the rule in just one step with the given data, starting with the initial guess that $\theta = 0.5$.

7.3 (031020:5) To obtain a good estimate of the brain's electrical response to a sound, a series of voltage measurements are made with electrodes placed on the head. The electrode voltages are measured in synchrony with repeated sound presentations. The results are listed as a sequence $\underline{x} = (x_1, \ldots, x_t, \ldots, x_T)$. Each sample in this list is assumed to be an outcome of a random variable X_t which has a normal (Gaussian) distribution $N(\mu, \sigma^2)$. Each sample is statistically independent of the other samples in the sequence.

The natural variance σ^2 of the voltage is known from previous similar measurements. The purpose of the present measurement is to estimate the unknown mean μ. Under ideal conditions, it is well known that a good unbiased estimate for the mean μ would have been simply the sample average

$$\hat{\mu}(\underline{x}) = \frac{1}{T}\sum_{t=1}^{T} x_t$$

However, the measurements are sometimes disturbed by spurious electromagnetic field transients from the power installation in the laboratory. These disturbances can occur with a known small probability d at any time during the measurement. If a disturbance happened to occur at measurement number t, the mean μ is not changed, but the variance of X_t is $\alpha^2\sigma^2$ instead of just σ^2, with a known value of $\alpha > 1$.

It seems reasonable that very large (positive or negative) sample values should now have a *reduced* weight in the estimate $\hat{\mu}$, as these sample values were probably influenced by the electrical disturbance. Fortunately, the Expectation-Maximization (EM) algorithm can be used to determine how each sample should be weighted in the average.

We represent the possibility of disturbances by a hidden random state variable sequence $\underline{S} = (S_1, \ldots, S_T)$, with $S_t = 1$ if a disturbance occurred at time t, and $S_t = 0$ otherwise.

Apply the EM algorithm to derive a formal update rule to improve a previous estimate $\hat{\mu}$ by searching for a new estimate $\hat{\mu}'$ that maximises the expected value

$$Q(\hat{\mu}', \hat{\mu}) = E_{\underline{S}}[\ln P(\underline{x}, \underline{S}|\hat{\mu}')|\underline{x}, \hat{\mu}]$$

Hint: A complete solution is given in the Answers section.

7.4 (040115:5) In physiological measurements of nerve activity, a very small electrode is inserted into the auditory nerve and the nerve impulses are counted during a fixed time interval D [s] from a single nerve cell. You have recorded such impulse counts from N different nerve cells, using identical sound stimuli, and stored the results in a data vector $\boldsymbol{x} = (x_1 \ldots x_n \ldots x_N)^T$. There are two different types of nerve cells. Therefore, each impulse count x_n may have been recorded either from a cell type denoted as $S_n = 1$ or cell type $S_n = 2$. Your task is now to estimate the impulse probabilities characterizing each cell type.

The relative proportion of cells of type 1 among all nerve cells is known to be p_1. Cells of type 1 have a rather high spontaneous activity and high sensitivity to external stimuli, and the other type has a lower spontaneous activity and lower sensitivity. For each type of cell the number of impulses counted during a time interval of duration D [s] can be regarded as a discrete random variable X_n with a Poisson distribution. This means that X_n has a conditional probability function

$$P(X_n = x_n|S_n = i) = \frac{e^{-c_i D}(c_i D)^{x_n}}{x_n!}, \qquad i \in \{1, 2\}$$

where the impulse rate c_i [counts /s] depends only on the cell type $S_n = i$, for the given external stimulus sound. The activity of each cell is statistically independent of all other cells.

During the measurement it is impossible to know from which type of cell the electrode is recording the activity. Only the observed impulse count gives a probabilistic indication of the cell type.

Use the Expectation Maximization (EM) algorithm to determine an update rule to improve previous estimates of c_1 and c_2, based on your observed vector $\boldsymbol{x}$ with impulse counts,

Hint: Given previously estimated model parameters $\lambda = (c_1, c_2)$, each step in the EM algorithm seeks a new parameter set λ' that maximizes the function

$$Q(\lambda', \lambda) = E_{\boldsymbol{S}}[\ln P(\boldsymbol{x}, \boldsymbol{S} \mid \lambda') \mid \boldsymbol{x}, \lambda]$$

where $\boldsymbol{x}$ is the observed vector of counts and $\boldsymbol{S}$ represents the corresponding vector of unknown cell types.

7.5 (041018:5) A signal source generates an observable random sequence $\underline{X} = (X_1, \ldots, X_t, \ldots)$ using hidden internal random variable sequences $\underline{U} = (U_1, \ldots, U_t, \ldots)$, $\underline{V} = (V_1, \ldots, V_t, \ldots)$, and $\underline{W} = (W_1, \ldots, W_t, \ldots)$ in a non-linear process, defined as:

$$\begin{aligned} U_0 &= 0 \\ U_t &= \begin{cases} +1, & \text{if } U_{t-1} + V_t \geq 0 \\ -1, & \text{if } U_{t-1} + V_t < 0 \end{cases} \\ X_t &= cU_t + W_t \end{aligned}$$

Here each element V_t is a random variable with zero-mean Gaussian distribution $N(0, \sigma_V)$, and all elements in the $\underline{V}$ sequence are statistically independent of each other. Similarly, each element W_t is $N(0, \sigma_W)$, and all elements in the $\underline{W}$ sequence are statistically independent of each other and of all elements in the $\underline{V}$ sequence. The scale factor c has a constant value somewhere around 1, but the exact value is not known.

You have observed an output sequence $\underline{x} = (x_1, \ldots, x_T)$ from this source. As the source can be characterised by a hidden Markov model λ with two states, you have already applied the well-known forward and backward algorithms to obtain a matrix γ with probability values

$$\begin{aligned} \gamma_{1,t} &= P(U_t = +1 | \underline{x}, \lambda, c) \\ \gamma_{2,t} &= P(U_t = -1 | \underline{x}, \lambda, c) \end{aligned}$$

for all $t = 1 \ldots T$, using a preliminary estimate $c = 1$ for the unknown constant.

Derive an update formula to obtain an improved estimate c' for the unknown constant, using the Expectation Maximization (EM) algorithm, assuming that all the $\gamma_{j,t}$ values have already been calculated.

Hint: The EM algorithm maximises the function

$$Q(c', c) = E\left[\ln P(\underline{U}, \underline{x} | \lambda, c') | \underline{x}, \lambda, c\right]$$

7.6 In a HMM with *tied observation densities* each state-conditional output density function is a GMM, but the Gaussian component functions are

identical for all states, and only the weight factors are state-specific. Here, the state-conditional output density functions can be written as

$$f_{\boldsymbol{X}_t|S_t}(\boldsymbol{x}|j) = b_j(\boldsymbol{x}) = \sum_{m=1}^{M} w_{jm} \frac{1}{(2\pi)^{K/2}\sqrt{\det C_m}} e^{-\frac{1}{2}(\boldsymbol{x}-\mu_m)^T C_m^{-1}(\boldsymbol{x}-\mu_m)}$$

In such an HMM the Markov initial and transition probabilities are trained as usual, as described in Sec. 7.6, but a new method must be used for the remaining parameters.

7.6.a Suggest a training procedure and intuitively reasonable update equations for the weight factors w_{jm} and the tied Gaussian parameters μ_m and C_m, in analogy with the GMM training procedure proven in Sec. 7.6.5.

7.6.b Use the formal EM approach to obtain update equations for parameters w_{jm}, μ_m, and C_m.

7.7 For an HMM with GMM:s as output distributions, prove the expression for the conditional combined state and mixture component probability in Eq. (7.69) on page 160:

$$\begin{aligned}\gamma_{im,t} &= P\left[S_t = i \cap U_t = m|\underline{\boldsymbol{x}}, \lambda\right] = \\ &= \gamma_{i,t} \frac{w_{im} g(\boldsymbol{x}_t, \mu_{im}, C_{im})}{\sum_{k=1}^{M} w_{ik} g(\boldsymbol{x}_t, \mu_{ik}, C_{ik})}\end{aligned}$$

where $\gamma_{i,t} = P[S_t = i|\underline{\boldsymbol{x}}, \lambda]$.

Chapter 8

Bayesian Learning

In the previous sections on GMM and HMM, we used the maximum likelihood (ML) approach from Chapter 4 to train the models. The ML approach can be summarized by the following steps, applied for each data category that the classifier must distinguish:

1. Collect training data, i.e., a sequence of feature vectors.
2. Choose a probabilistic *model structure* for the features, e.g., a GMM, specified by a set of *parameters*, to be determined by the training.
3. Find a *single parameter set* that fits the training data in the ML sense.
4. Use the *single trained model* to define a discriminant function.

This method often works very well, but it has some problems: It is always difficult to collect enough training data, and many slightly different model variants may fit the training data nearly equally well. Therefore, it is actually *sub-optimal* to use only one of these possible models in the final classifier. The model may also overfit, if we use a too complex model structure with too little training data. It is difficult to choose a good model size. *Bayesian Learning* avoids these difficulties. Bayesian learning has become increasingly popular in machine learning in recent decades.

8.1 Introduction

Bayesian Learning is a formal method with the following main advantages:

- After Bayesian training, the classifier uses not only a single set of model parameters, but instead an average across all model variants, weighted as indicated by the training data.
- The trained model includes a measure of its own reliability, for the given amount of training data.

- The model can easily be re-trained, adaptively, with new training data.
- It is possible to let the training procedure automatically determine the model complexity, i.e., the effective number of model parameters.

These advantages are achieved by a single crucial theoretical concept: In Bayesian Learning we assume that not only the feature vectors, but also all the *model parameters*, that define the feature-vector distributions, are *random variables*.[1] The goal of Bayesian Learning is to find conditional density functions for the model parameters, given the training data. We then design the classifier using a correctly weighted average of all possible model variants, that are allowed by the training data.

There are two main downsides of using Bayesian Learning: Firstly, computations usually become more complicated, and it is often not possible to derive an exact analytic solution. Secondly, we must choose an initial distribution of the model parameters. There is no formal, objective way to do this that works in all situations. This can introduce an element of subjectivity that bothers some scientists and philosophers.

A simple example can illustrate the Bayesian Learning approach and one of the main advantages.

Example 8.1 (Word Recognition): In a word-recognition test you present T test words, in background noise, and record all the listener's responses as a sequence $\underline{x} = (x_1, \ldots, x_T)$ with binary elements, where $x_t = 1$ means a correct response, and $x_t = 0$ means an incorrect response to word number t. We regard each observed result[2] as an outcome of a discrete random variable X_t, with a probability mass function $P[X_t = 1] = w$ and $P[X_t = 0] = 1 - w$.

- What is a reasonable estimate of the listener's probability for correct word recognition?
- What is the conditional probability of correct response in a *future* test, i.e., $P[X_{T+1} = 1 \mid x_1, \ldots, x_T]$, given all previous test results?

Tentative ML approach: The total number of correct responses, out of the T test words, is $z(\underline{\boldsymbol{x}}) = \sum_{t=1}^{T} x_t$, so the observed relative frequency of correct responses is obviously z/T. The ML point estimate of the probability is precisely the observed relative frequency: $\hat{w}_{ML} = z/T$, as discussed in Sec. 4.8.4 and Problem 8.1.

But what if all responses were correct, i.e., $z = T$? Then $\hat{w}_{ML} = 1$, exactly, but is this really a reasonable estimate of the probability? Perhaps the listener might have made his first error in test number $T + 1$, if we had

[1] In ML training, the model parameters were just some fixed but unknown numbers to be estimated from the training data.

[2] We assume here, for simplicity, that the probability of correct response is the same for every presented word. We also assume that all test results are statistically independent.

just included one more word in the test sequence. The ML-estimated model works as if this could never happen.

Solution:

The Bayesian Learning approach gives a more realistic result. We now assume that the unknown probability parameter is a random variable W that can have as outcome w any real value between 0 and 1 for different individuals in the human population. Before the test, we assume we know nothing about W. Therefore, it is reasonable to express this lack of knowledge by defining a neutral, uniform, *prior density function* for W as

$$f_W(w) = \begin{cases} 1, & 0 \leq w \leq 1 \\ 0, & \text{otherwise} \end{cases} \tag{8.1}$$

We now formulate the *observation probability model*: If we had known the exact probability of a correct response, for example as the outcome $W = w$, the probability of any observed test result would be

$$\begin{aligned} P\left[X_t = 1 \mid W = w\right] &= w \\ P\left[X_t = 0 \mid W = w\right] &= 1 - w \end{aligned} \tag{8.2}$$

Thus, assuming that all test results are statistically independent, we can calculate the conditional probability of the complete observed sequence as

$$\begin{aligned} P\left[\underline{X} = \underline{x} \mid W = w\right] &= \prod_{t=1}^{T} w^{x_t}(1-w)^{1-x_t} = \\ &= w^{z(\underline{x})}(1-w)^{T-z(\underline{x})}, \quad \text{with } z(\underline{x}) = \sum_{t=1}^{T} x_t \end{aligned} \tag{8.3}$$

Using Bayes' rule, we can also express the combined probability for the observed data sequence *and* any parameter outcome as

$$\begin{aligned} P\left[\underline{X} = \underline{x} \cap W = w\right] &= P\left[\underline{X} = \underline{x} \mid W = w\right] f_W(w) = \\ &= \begin{cases} w^{z(\underline{x})}(1-w)^{T-z(\underline{x})} \cdot 1, & 0 \leq w \leq 1 \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{8.4}$$

Here, we have already observed $\underline{x}$, and counted $z(\underline{x})$, so the expression in Eq. (8.4) is really just a function of w, although the shape of this function is influenced by the observed data sequence. This kind of function is usually called a *likelihood* function, because it is proportional to a probability, but not normalized as a proper probability density function. However, by applying Bayes' rule again, we can easily scale the likelihood function into a properly normalized conditional density function, given the observed data,

$$f_{W|\underline{X}}(w \mid \underline{x}) = \frac{1}{c} \begin{cases} w^{z(\underline{x})}(1-w)^{T-z(\underline{x})}, & 0 \leq w \leq 1 \\ 0, & \text{otherwise} \end{cases} \tag{8.5}$$

where

$$c = P\left[\underline{X} = \underline{x}\right] \tag{8.6}$$

This is now the *posterior probability density* for W, *after* observing the data. Fortunately, we do not need to calculate $P\left[\underline{X} = \underline{x}\right]$ directly. Instead we can simply find the value of c from the requirement that $f_{W|\underline{X}}(w \mid \underline{x})$ is normalized,

$$\int_0^1 f_{W|\underline{X}}(w \mid \underline{x})dw = \frac{1}{c}\int_0^1 w^z(1-w)^{T-z}dw = 1 \tag{8.7}$$

The integral is tabulated in Råde and Westergren (1995, sec. 7.5, no. 1). The normalizing constant is

$$c = \int_0^1 w^z(1-w)^{T-z}dw = \frac{\Gamma(z+1)\Gamma(T-z+1)}{\Gamma(T+2)} \tag{8.8}$$

where $\Gamma(\,\cdot\,)$ is the *gamma*[3] function. We could also identify Eq. (8.5) directly as the probability density function for a *beta distribution*, tabulated in Råde and Westergren (1995, Sec. 17.2), and also presented in Sec. 8.4.1. We see that Eq. (8.5) is precisely the Beta probability density function defined in Eq. (8.41), with parameters $a = z(\underline{x}) + 1; b = T - z(\underline{x}) + 1$.

The posterior density function for W expresses precisely all the knowledge we now have about the listener's word-recognition ability. A few examples of posterior density functions are shown in Fig. 8.1. A good single-value summary of the test results is the posterior expected value, which is found in the tabulated characteristics for the Beta distribution:

$$E\left[W \mid \underline{x}\right] = \int_0^1 w f_{W|\underline{X}}(w \mid \underline{x})dw = \frac{z(\underline{x}) + 1}{T + 2} \tag{8.9}$$

Note that this expected value will never be exactly 0 or 1, even in the extreme cases of $z = 0$, or $z = T$.

To find the conditional probability for a correct response in a future test, we must take an average across all possible posterior parameter values. Formally, we just apply Bayes' rule and integrate across all the possible events $W = w$, given the observed $\underline{x} = (x_1, \ldots, x_T)$:

$$\begin{aligned} P\left[X_{T+1} = 1 \mid \underline{x}\right] &= \int_0^1 P\left[X_{T+1} = 1 \mid W = w\right] f_{W|\underline{X}}(w \mid \underline{x})dw = \\ &= \int_0^1 w f_{W|\underline{X}}(w \mid \underline{x})dw = \\ &= E\left[W \mid \underline{x}\right] = \frac{z(\underline{x}) + 1}{T + 2} \end{aligned} \tag{8.10}$$

[3] In MatLab, $\Gamma(x)$ is calculated by the function `gamma(x)`, and $\ln\Gamma(x)$ by `gammaln(x)`.

In this sense, the conditional posterior expected value was obviously a very meaningful single-value summary of all the previous observations. The *maximum posterior probability* point estimate,

$$\hat{w}_{MAP} = \underset{w}{\operatorname{argmax}}\, f_{W|X}(w \mid \underline{x}) \tag{8.11}$$

is another common single-value approximation.

Example 8.1

In this introductory example, we used a *uniform prior density* $f_W(\cdot)$ in Eq. (8.1) to indicate our uniform willingness to accept any parameter value between 0 and 1. This simple prior was actually also an instance of the Beta distribution, as

$$f_W(w) = w^{a_0-1}(1-w)^{b_0-1} = 1 \tag{8.12}$$

when the parameters are chosen as $a_0 = 1, b_0 = 1$. Thus, we started with a prior Beta distribution for the parameter W, then applied the training-data observations, and finally arrived at another Beta distribution as the posterior density in Eq. (8.5), only with new refined parameter values, $a_T = z(\underline{x}) + a_0$, and $b_T = T - z(\underline{x}) + b_0$. If we would start with any other Beta distribution as the prior density, the posterior would still be a Beta distribution.

Because of this nice property of the Beta distribution, we can now take the trained posterior distribution as a starting point for further re-training of the model with new data. The posterior density function obtained in Eq. (8.5) is then simply re-used as a new prior distribution in Eq. (8.1), and the whole procedure is repeated in exactly the same way using the additional training data $(x_{T+1}, \ldots)$. The result will again be a new posterior density function, that is still another instance of the beta distribution.

In many applications of Bayesian Learning it is possible to assign a prior distribution such that another variant of the same type of distribution reappears as the posterior density, in a similar way as with the beta distribution above. Such a prior is called a *conjugate prior*.

8.2 Bayesian Learning Procedure

In this section we summarize the main steps of Bayesian Learning. The approach is the same as in the previous introductory example. This procedure is used to train a single distribution and can be used for each class of training data in a classification problem.

1. Collect a *training data* set $\mathcal{D} = \underline{\boldsymbol{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, containing a sequence of observed feature vectors $\boldsymbol{x}_t$. Each observed feature vector is regarded as an outcome of a random vector $\boldsymbol{X}_t$.

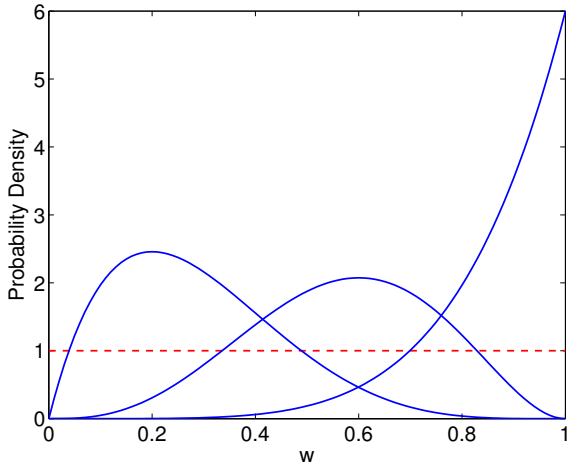


Figure 8.1: Examples of posterior density functions for the parameter W in the word-recognition Example 8.1, for $z = 1, 3, 5$ observed correct responses out of $T = 5$ test words. The dashed line is the uniform prior density, assumed for $T = 0$ and $z = 0$.

2. Formulate a *model* for the *conditional feature-vector density*,

$$f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w}) \tag{8.13}$$

given any parameter vector $\boldsymbol{w}$, regarded as an outcome of a random vector $\boldsymbol{W}$. (A probability *mass* function is used if the features have only discrete values, as in Example 8.1.) Here, we assume that the feature-vector distribution is the same for every t, and that all feature vectors in the sequence are statistically independent[4] of each other. Therefore, we can easily express the conditional density function for the complete training sequence, as

$$f_{\underline{\boldsymbol{X}}|\boldsymbol{W}}(\underline{\boldsymbol{x}} \mid \boldsymbol{w}) = \prod_{t=1}^{T} f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x}_t \mid \boldsymbol{w}) \tag{8.14}$$

3. Choose a form of the *prior density function* for the parameter vector,

$$f_{\boldsymbol{W}}(\boldsymbol{w}; \boldsymbol{a}^{(0)}) \tag{8.15}$$

In general, this density function is specified by some parameter vector $\boldsymbol{a}^{(0)}$. These parameters are usually called *hyper-parameters*[5]. As discussed in Sec. 8.3, it is often possible and desirable to assign hyper-parameter values manually to obtain a broad non-informative prior

[4] If consecutive feature vectors are statistically dependent, we can still model this dependence and get a generalized version of Eq. (8.14).

[5] The hyper-parameter vector can itself be regarded as a random variable and treated

density for $\boldsymbol{W}$. This allows the learning procedure to depend entirely on the observed training data.

4. Calculate the *Likelihood function* as the probability density of all the observed training data and any parameter-vector outcome,

$$f_{\underline{\boldsymbol{X}},\boldsymbol{W}}(\underline{\boldsymbol{x}}, \boldsymbol{w}) = f_{\underline{\boldsymbol{X}}|\boldsymbol{W}}(\underline{\boldsymbol{x}} \mid \boldsymbol{w}) f_{\boldsymbol{W}}(\boldsymbol{w}; \boldsymbol{a}^{(0)}) \tag{8.16}$$

5. Identify the conditional *posterior density function* for $\boldsymbol{W}$ as a normalized, scaled, version of the Likelihood function,

$$f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\boldsymbol{w} \mid \underline{\boldsymbol{x}}; \boldsymbol{a}^{(T)}) = c(\underline{\boldsymbol{x}}) f_{\underline{\boldsymbol{X}}|\boldsymbol{W}}(\underline{\boldsymbol{x}} \mid \boldsymbol{w}) f_{\boldsymbol{W}}(\boldsymbol{w}; \boldsymbol{a}^{(0)}) \tag{8.17}$$

   The posterior density is specified by some hyper-parameters $\boldsymbol{a}^{(T)}$. The hyper-parameters are often omitted to simplify the notation, but they are always there.

6. Calculate the *predictive density function*, i.e., the conditional feature-vector density function for any future observation, given all training data. To do this, we use the general form of the feature density defined in Eq. (8.13), integrated across all possible parameter vectors, with the weights defined by the posterior parameter density function in Eq. (8.17):

$$f_{\boldsymbol{X}|\underline{\boldsymbol{X}}}(\boldsymbol{x} \mid \underline{\boldsymbol{x}}; \boldsymbol{a}^{(T)}) = \int_{\text{all } \boldsymbol{w}} f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w}) f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\boldsymbol{w} \mid \underline{\boldsymbol{x}}; \boldsymbol{a}^{(T)}) d\boldsymbol{w} \tag{8.18}$$

   This is now the final density function to be used in a discriminant function in the classifier, trained by Bayesian Learning.

It is convenient to choose the prior parameter distribution such that the posterior parameter distribution, after training, will have the same mathematical form, only with modified hyper-parameter values. A parameter distribution that has this nice property is called a *conjugate distribution* for the given feature distribution.

8.3 The Prior Distribution

8.3.1 Subjective Informative Prior

Often we have some partial prior knowledge about the parameter to be estimated from observed data. The previous section showed how any such

in a Bayesian-learning approach, by introducing a prior distribution for the hyper-parameters, in turn specified by some hyper-hyper-parameters. In this way, *hierarchical Bayesian Learning* might in principle be extended to any level. Hyper-parameter values may also be assigned as a point estimate $\hat{\boldsymbol{a}}_{ML} = \text{argmax}_{\boldsymbol{a}} f_{\underline{\boldsymbol{X}}}(\underline{\boldsymbol{x}}; \boldsymbol{a})$, fitted to the training data by a so-called *Maximum Likelihood type 2* approximation (Bishop, 2006), but all this is outside the present scope.

prior knowledge can be formally used in the Bayesian learning process. We just need to express our prior knowledge in the form of a suitable prior probability density function $f_{\boldsymbol{W}}(\boldsymbol{w})$ for the unknown parameter vector $\boldsymbol{W}$. Then we use the observed data to derive an updated, posterior, density function $f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\,)$ for $\boldsymbol{W}$, usually with much smaller variance than the prior density. The reduction in the uncertainty of $\boldsymbol{W}$ is a measure of the increased knowledge that we gained from the observations. The resulting posterior density function for $\boldsymbol{W}$ can then be used again as a prior density for later experiments, because this density function now represents our present state of knowledge. This is one of the main advantages of Bayesian learning.

From this point of view, the probability density $f_{\boldsymbol{W}}(\boldsymbol{w})$ can be interpreted as a measure of the experimenter's *subjective belief* that the random parameter $\boldsymbol{W}$ has a value near $\boldsymbol{w}$. Examples of this use of the prior density for a scalar parameter W are shown in Fig. 8.2.

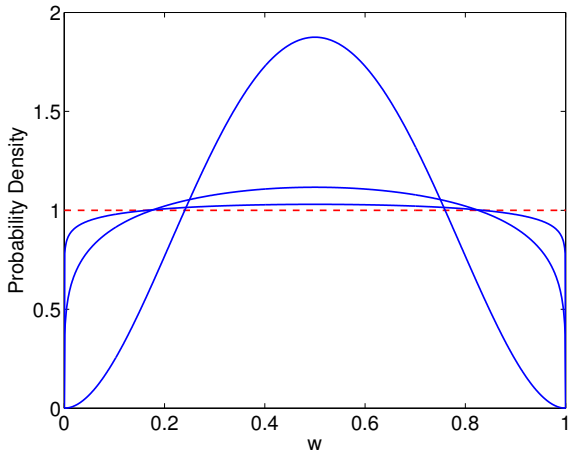


Figure 8.2: Three examples of subjective informative prior density functions that might be used for the parameter W in the word-recognition Example 8.1, all illustrating a prior belief that $w = 0$ and $w = 1$ are impossible parameter values. The dashed line is the uniform prior density.

8.3.2 Subjective Non-Informative Prior

In many situations it is considered scientifically very important to prevent any form of prejudice from having any influence on the interpretation of experimental observations. Instead, the experimenter wants to "let the data speak for itself". The intention is, of course, to ensure that the scientific results are *objective*, in the sense that other researchers should find similar results if they repeat the experiment.

In this case, we must choose a prior density function that has as little influence as possible on the posterior density for the parameters. In the

introductory example 8.1 the unknown single parameter W represented a probability for an event to happen. Then it seemed plausible to assign a *uniform density* for W to represent our prior subjective judgment that any probability value between 0 and 1 should be regarded as equally probable. This choice was made explicit in Eq. (8.1). As the Bayesian learning procedure is exactly defined, and the prior density is declared openly, the experiment and data analysis can easily be reproduced by other researchers. It is also easy to check objectively how the final result would change if we would choose another prior density function.

However, we must admit that this choice of prior was still a *subjective* decision. Other people might prefer to apply a different prior density function in a Bayesian analysis of the same set of observations. In this example, the parameter W was the unknown probability of a correct word response. Someone might argue that it is more "natural" to consider the ratio $W/(1-W)$ between probabilities for correct and incorrect responses. This ratio can then have any value between 0 and $+\infty$, if we do not have any prior knowledge. For symmetry reasons, the probability for the event $0.001 < W/(1-W) < 0.01$ should be equal to the probability for $100 < W/(1-W) < 1000$.

Therefore, the experimenter must formally express the prior belief that the transformed parameter $U = \ln W/(1-W)$ should approach a *uniform* probability density in the range from $-\infty$ to $+\infty$, i.e.,

$$f_U(u) \to \text{const} \tag{8.19}$$

Of course, this is not a proper␣density function, because it cannot be normalized. Such a prior is therefore called an *"improper prior"*. Nevertheless, such prior "density" functions can be used in practice, because the posterior density may be a proper density function with integral equal to 1. The improper prior can also be asymptotically approximated by a proper density function, for example, a Gaussian density with zero mean and a variance that approaches infinity.

In this example, it is rather easy to show that a uniform density for U is equivalent to assuming a probability density for W as[6]

$$f_W(w) \to \frac{1}{w(1-w)} \tag{8.20}$$

Density functions approaching this improper form are shown in Fig. 8.3.

Comparing Eqs. (8.1) and (8.20), it is evidently possible for two different researchers to argue plausibly in favor of quite different prior density functions for the same parameter, even when both researchers actually wanted

[6]Again, this asymptotic function is not a proper density, because it cannot be normalized, but it can be approximated, and the posterior density can be normalized anyway.

to express the same subjective belief, namely, a complete lack of prior knowledge about the parameter. If there is a large number of observations, the choice of prior density function typically has a very small influence on the posterior density, but the choice can be important if there are only a small number of actual observations.

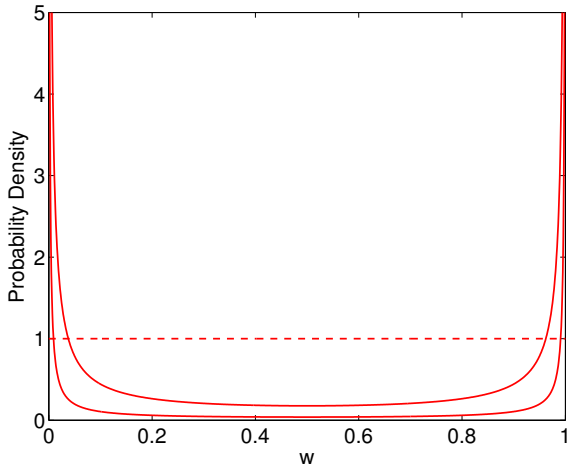


Figure 8.3: Two examples of proper prior densities approaching the improper prior of Eq. (8.20), shown here as beta density functions, with $a = b$ chosen as 0.1 and 0.02. Is the uniform (dashed) prior better, with $a = b = 1$?

8.3.3 Objective Non-Informative Prior

In order to resolve the problem with disagreement between subjective prior distributions, discussed in the previous section, it would be desirable to define a suitable prior density function based only on the objective experimental conditions and the probabilistic model for the possible observations in the experiment. The so-called *Jeffreys prior* (Jeffreys, 1946) is often used for this purpose. This approach will now be presented, first intuitively, and then formally.

Let us assume we have designed an experiment such that it is reasonable to assign a known form of the conditional density function $f_{X|W}(x \mid w)$ for the outcome X, given any value of the unknown scalar parameter W. We must consider, before the actual experiment, what kind of conclusions about W we might be able to make for any particular outcome $X = x$.

First consider for a moment a *maximum likelihood* (ML) point estimate for the unknown parameter, as

$$\hat{w}_{ML} = \underset{w}{\operatorname{argmax}} \ln f_{X|W}(x \mid w) \tag{8.21}$$

If the log-probability for the outcome x changes very sharply with changes in the parameter w, then the point estimate $\hat{w}_{ML}$ will have high precision. On the other hand, if the log-likelihood function $\ln f_{X|W}(x \mid w)$ has a broad and shallow shape as a function of w, the maximum point is less well defined, because the parameter w can be varied over a rather wide range with only small changes in the log-likelihood for the observation. Then the ML estimate is less reliable. This relation is formalized by the *Fisher Information*.

Definition 8.1 (Fisher Information): *If a random (vector or scalar) variable $\boldsymbol{X}$ has a density function $f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})$, specified by some parameter vector $\boldsymbol{w} = (w_1, \ldots, w_K)^T$, the Fisher Information $\boldsymbol{I}(\boldsymbol{w})$ is a matrix with elements*

$$I_{ij}(\boldsymbol{w}) = E_{\boldsymbol{X}}\left[\left(\frac{\partial \ln f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{X} \mid \boldsymbol{w})}{\partial w_i}\right)\left(\frac{\partial \ln f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{X} \mid \boldsymbol{w})}{\partial w_j}\right) \,\middle|\, \boldsymbol{w}\right] \tag{8.22}$$

□

Here, the expectation is taken over all possible outcomes of $\boldsymbol{X}$, for the given $\boldsymbol{w}$. For most probability distributions occurring in practice, the Fisher Information can also be obtained using the second derivatives, as[7]

$$I_{ij}(\boldsymbol{w}) = -E_{\boldsymbol{X}}\left[\frac{\partial^2 \ln f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{X} \mid \boldsymbol{w})}{\partial w_i \partial w_j} \,\middle|\, \boldsymbol{w}\right] \tag{8.23}$$

which is sometimes easier to calculate. If the density function depends only on a single parameter w, the definition obviously reduces to

$$I(w) = E_{\boldsymbol{X}}\left[\left(\frac{\partial \ln f_{\boldsymbol{X}|W}(\boldsymbol{X} \mid w)}{\partial w}\right)^2 \,\middle|\, w\right] = -E\left[\frac{\partial^2 \ln f_{\boldsymbol{X}|W}(\boldsymbol{X} \mid w)}{\partial w^2} \,\middle|\, w\right] \tag{8.24}$$

It can be shown, for any unbiased estimator $\hat{w}$, that the variance of the estimate is lower-bounded by the inverse Fisher information,

$$\text{var}\,[\hat{w}] \geq 1/I(\hat{w}) \tag{8.25}$$

This relation is called the *Cramér-Rao bound*.

Now let us consider the set of all possible experimental outcomes, for all possible parameter values. A simulated example is shown in Fig. 8.4.

In any region for the parameter value w, where the Fisher Information is large, an experiment will, on average, produce rather precise knowledge about the parameter. A smaller value for the Fisher information also gives a less precise estimate. For any set of outcomes X, caused by a true value $W = w$ of the parameter, we will be able to conclude that $W \in (\hat{w} \pm \Delta w/2)$

[7] The proof that (8.22) is equivalent to (8.23) is left as an exercise in Problem 8.7.

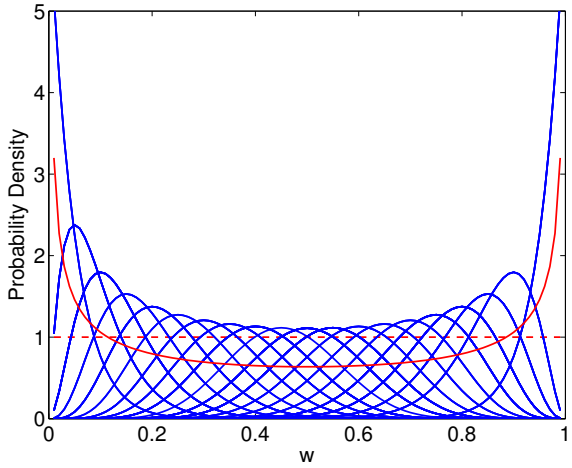


Figure 8.4: Posterior density functions for the parameter W in the word-recognition Example 8.1, obtained with simulated experiments with true parameter values $w = 0, 0.05, \ldots, 0.95, 1$, using a uniform prior density function (dashed). The peak heights of the posterior density functions are roughly proportional to $\sqrt{I(w)}$, i.e., to Jeffreys prior (the U-shaped curve).

with a fixed probability for any w, if we choose the interval width Δw as approximately proportional to the standard deviation of the estimator $\hat{w}$, i.e., approximately proportional to $1/\sqrt{I(w)}$. In other words, the probability density for W that can be obtained in any experiment is proportional to $\sqrt{I(w)}$, as illustrated in the simple example in Fig. 8.4.

This prediction is certain, even though we have not yet performed any of those experiments. In this sense, it seems reasonable to describe our lack of prior knowledge about W by using the *non-informative Jeffreys prior*:

Definition 8.2 (Jeffreys Prior): *If a random variable $\boldsymbol{X}$ has a density function specified by the outcome of a random parameter vector, $\boldsymbol{W} = \boldsymbol{w}$, the Jeffreys Prior density for the parameter vector is proportional to the square root of the determinant of the Fisher Information Matrix:*

$$f_{\boldsymbol{W}}(\boldsymbol{w}) \propto \sqrt{\det \boldsymbol{I}(\boldsymbol{w})} \tag{8.26}$$

□

If there is only a single scalar parameter W with Fisher information $I(w)$, the definition reduces to

$$f_W(w) \propto \sqrt{I(w)} \tag{8.27}$$

If we consider using a *transformed* parameter $U = g(W)$ instead of W in the definition of the conditional density for X, it can be shown that the method of Eq. (8.26) gives exactly equivalent results, regardless of the transformation g. In this sense, this method to choose a non-informative prior is objective

and unique and avoids the possible controversies about various subjective priors discussed in Sec. 8.3.2. Loosely speaking, even if different researchers prefer to choose different mathematical forms to express the same lack of prior knowledge, these different mathematical forms are equivalent, if they are based on the Jeffreys prior.

Of course, researchers must still take full responsibility for their choice of prior density. There is no guarantee that the Jeffreys prior is always the best choice. The definition of Jeffreys prior is purely mathematical and does not use any knowledge about physical constraints in the experiment.

Example 8.2: You plan to measure samples of a random variable X that is known to have a Gaussian distribution with zero mean but completely unknown standard deviation. Determine a non-informative Jeffreys prior density for the standard deviation parameter, here called W.
Solution:
The conditional distribution for the observed variable is

$$f_{X|W}(x \mid w) = \frac{1}{w\sqrt{2\pi}} e^{-\frac{(x-0)^2}{2w^2}} \tag{8.28}$$

The corresponding log-likelihood function is

$$\mathcal{L}(w \mid x) = \ln f_{X|W}(x \mid w) = -\ln w - \frac{x^2}{2w^2} + \text{const.} \tag{8.29}$$

The Fisher information is then

$$I(w) = -E_X\left[\frac{\partial^2 \ln f_{X|W}(X \mid w)}{\partial w^2}\right] = E_X\left[-\frac{1}{w^2} + \frac{3X^2}{w^4}\right] = \\ = -\frac{1}{w^2} + \frac{3E\left[X^2\right]}{w^4} = -\frac{1}{w^2} + \frac{3w^2}{w^4} = \frac{2}{w^2} \tag{8.30}$$

Thus, the Jeffreys prior for the standard deviation is

$$f_W(w) \propto \sqrt{I(w)} \propto \frac{1}{w} \tag{8.31}$$

This is not a proper density funtion, as it cannot be normalized, but it can still be used as an asymptotic limit. (A gamma density function with shape parameter $a \to 0$ and inverse scale parameter $b \to 0$ is a proper density that approaches the Jeffreys prior.)

Example 8.2

The non-informative prior density for an unknown *scale parameter*, such as the standard deviation for a Gaussian variable, always has the form of Eq. (8.31). This implies, for example, that the probability for an event $0.1 < W < 1$ is exactly the same as the probability that $100 < W < 1000$,

i.e., the *logarithm* of the parameter value has a *uniform* density. This is intuitively reasonable. If the parameter represents for example a physical voltage, it makes sense that the prior density has the same form, regardless of whether we measure the voltage in μV or kV.

Example 8.3: Now assume another researcher prefers to use the unknown *variance* $V = W^2$ as the unknown parameter, instead of the standard deviation W. What is the prior density for the transformed parameter V?
Solution:
We can use the Jeffreys prior, or we can consider the cumulative probability function for V, and then differentiate to get the density function:

$$F_V(v) = P\left[V \leq v\right] = P\left[0 \leq W \leq \sqrt{v}\right] = \int_0^{\sqrt{v}} f_W(u)du \tag{8.32}$$

$$f_V(v) = \frac{dF_V(v)}{dv} = f_W(\sqrt{v})\frac{d\sqrt{v}}{dv} \propto \frac{1}{\sqrt{v}}\frac{1}{2\sqrt{v}} \propto \frac{1}{v} \tag{8.33}$$

Example 8.3

Example 8.4: You plan to measure samples of a random variable X that is known to have a Gaussian distribution with unknown mean μ and standard deviation σ. Determine a non-informative Jeffreys prior density for the parameter vector $\boldsymbol{w} = (\mu, \sigma)$, and find a proper density function that asymptotically approaches Jeffreys prior.
Solution:
The conditional distribution for the observed variable is

$$f_{X|\boldsymbol{W}}(x \mid \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{8.34}$$

The corresponding log-likelihood function is

$$\mathcal{L}(\mu, \sigma \mid x) = \ln f_{X|W}(x \mid \mu, \sigma) = -\ln\sigma - \frac{(x-\mu)^2}{2\sigma^2} + \text{const.} \tag{8.35}$$

The Fisher information matrix $\boldsymbol{I}(\mu, \sigma)$ has elements

$$I_{11}(\mu, \sigma) = -E_X\left[\frac{\partial^2 \ln f_{X|W}(X \mid \mu, \sigma)}{\partial\mu^2}\right] = \frac{1}{\sigma^2} \tag{8.36}$$

$$I_{22}(\mu, \sigma) = -E_X\left[\frac{\partial^2 \ln f_{X|W}(X \mid \mu, \sigma)}{\partial\sigma^2}\right] = \frac{2}{\sigma^2} \tag{8.37}$$

$$I_{12}(\mu, \sigma) = I_{21}(\mu, \sigma) = -E_X\left[\frac{\partial^2 \ln f_{X|W}(X \mid \mu, \sigma)}{\partial\mu\partial\sigma}\right] = 0 \tag{8.38}$$

Thus, Jeffreys prior is

$$f_{\boldsymbol{W}}(\mu, \sigma) \propto \sqrt{\det \boldsymbol{I}(\mu, \sigma)} \propto \frac{1}{\sigma^2} \tag{8.39}$$

This is not a proper density funtion, as it cannot be normalized, but it can still be used as an asymptotic limit. A conditional Gaussian density for μ, given σ, with zero mean and variance σ^2/β, combined with a gamma density function for σ,

$$f_{\boldsymbol{W}}(\mu, \sigma) = g(\mu \mid \sigma)h(\sigma) = \frac{\sqrt{\beta}}{\sqrt{2\pi}\sigma} e^{-\frac{\mu^2\beta}{2\sigma^2}} \frac{(b\sigma)^a}{\Gamma(a)} \frac{1}{\sigma} e^{-b\sigma} \tag{8.40}$$

is proper, and approaches Jeffreys prior, when $\beta \to 0$, $a \to 0$, and $b \to 0$. A dimensionality check verifies that the density has the correct physical dimension, e.g., $1/\mathrm{V}^2$, if x, μ, and σ are all measured in V. This Gauss-gamma form is useful also because it is a *conjugate* density for the parameter pair (μ, σ). The non-informative prior density for an unknown *location parameter*, such as the mean μ for a Gaussian variable, is always *uniform*, as in the result (8.39).

Example 8.4

8.4 Useful Conjugate Probability Distributions

This section defines some probability distributions that are often needed in Bayesian learning, because they appear as conjugate distributions to some of the most common conditional distributions for the observed experimental variables.

8.4.1 Beta Distribution

The beta distribution can describe a scalar random variable that is restricted to values between 0 and 1. In Bayesian models, it is particularly useful for the prior and posterior distributions of any scalar parameter that represents the *probability* of an event.

If a random variable W has the *beta distribution*, with parameters (a, b),

$$f_W(w) = \begin{cases} \frac{1}{B(a,b)} w^{a-1}(1-w)^{b-1}, & 0 \le w \le 1 \\ 0, & \text{otherwise} \end{cases} \tag{8.41}$$

$$B(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \tag{8.42}$$

$$E[W] = \frac{a}{a+b} \tag{8.43}$$

$$\mathrm{var}[W] = \frac{ab}{(a+b)^2(a+b+1)} \tag{8.44}$$

$$E[\ln W] = \psi(a) - \psi(a+b) \tag{8.45}$$

$$\mathrm{mode}[W] = \operatorname*{argmax}_w f_W(w) = \frac{a-1}{a+b-2} \tag{8.46}$$

The beta distribution is defined for real parameter values $a > 0, b > 0$. The normalizing factor $B(a, b)$ is usually called the beta[8] function. The gamma and digamma functions[9] are defined as

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt; \quad \psi(x) = \frac{d \ln \Gamma(x)}{dx} \tag{8.47}$$

For positive integer values n, $\Gamma(n) = (n-1)!$.

8.4.2 Dirichlet Distribution

The Dirichlet distribution is a generalization of the Beta distribution. The Dirichlet distribution can describe a random vector where all elements are restricted to values between 0 and 1, and furthermore, the sum of all elements equals 1. In Bayesian models, it is useful for the prior and posterior distributions of a parameter vector that represents the set of probabilities for K *mutually exclusive events* such that only one of them can occur.

If a random vector $\boldsymbol{W} = (W_1, \ldots, W_K)^T$ has the *Dirichlet distribution*, with a parameter vector $\boldsymbol{a} = (a_1, \ldots, a_K)^T$,

$$f_{\boldsymbol{W}}(\boldsymbol{w}) = \begin{cases} \frac{1}{B(\boldsymbol{a})} \prod_{k=1}^K w_k^{a_k - 1}; & 0 \leq w_k \leq 1; \ \sum_{i=1}^K w_i = 1 \\ 0, & \text{otherwise} \end{cases} \tag{8.48}$$

$$B(\boldsymbol{a}) = \frac{\prod_{i=1}^K \Gamma(a_i)}{\Gamma(a_0)}; \quad a_0 = \sum_{i=1}^K a_i \tag{8.49}$$

$$E[W_k] = \frac{a_k}{a_0}; \quad E[\boldsymbol{W}] = \frac{\boldsymbol{a}}{a_0} \tag{8.50}$$

$$\text{var}[W_k] = \frac{a_k(a_0 - a_k)}{a_0^2(a_0 + 1)} \tag{8.51}$$

$$\text{cov}[W_i, W_j] = \frac{-a_i a_j}{a_0^2(a_0 + 1)}, \quad i \neq j \tag{8.52}$$

$$E[\ln W_k] = \psi(a_k) - \psi(a_0) \tag{8.53}$$

$$\text{mode}[\boldsymbol{W}] = \underset{\boldsymbol{w}}{\text{argmax}}\, f_{\boldsymbol{W}}(\boldsymbol{w}) = \frac{\boldsymbol{a} - 1}{a_0 - K}, \quad \text{all } a_k > 1 \tag{8.54}$$

The Dirichlet distribution is defined for real parameter vectors $\boldsymbol{a}$ with positive elements $a_k > 0$ for all k. For $K = 2$, the Dirichlet is equivalent to the beta distribution.

8.4.3 Gamma Distribution

The gamma distribution can describe a scalar random variable that is restricted to non-negative values. In Bayesian models, it is the conjugate prior

[8] The beta function is available as `beta(a,b)` in MatLab.

[9] In Matlab, the gamma function $\Gamma(a)$ is available as `gamma(a)`, and the digamma function $\psi(a)$ as `psi(a)`.

for several distributions, including the Gaussian with unknown variance.

If a random variable W has the *gamma distribution*, with shape parameter a, and scale parameter θ,

$$f_W(w) = \begin{cases} \frac{1}{\theta^a \Gamma(a)} w^{a-1} e^{-w/\theta}, & 0 \leq w \\ 0, & \text{otherwise} \end{cases} \tag{8.55}$$

or, equivalently, with inverse scale parameter b,

$$f_W(w) = \begin{cases} \frac{b^a}{\Gamma(a)} w^{a-1} e^{-bw}, & 0 \leq w \\ 0, & \text{otherwise} \end{cases} \tag{8.56}$$

$$E[W] = a\theta = a/b \tag{8.57}$$

$$\text{var}[W] = a\theta^2 = a/b^2 \tag{8.58}$$

$$\text{mode}[W] = \underset{w}{\text{argmax}}\, f_W(w) = \begin{cases} (a-1)\theta = (a-1)/b, & a > 1 \\ 0, & 0 < a < 1 \end{cases} \tag{8.59}$$

The Gamma distribution is defined for real parameter values $a > 0, \theta > 0$.

A dimensionality check verifies that the density has the correct physical dimension, e.g., 1/V, if w and the scale parameter θ are measured in V, or, equivalently, the inverse scale b is measured in units $1/V$. The shape parameter a is always just a dimensionless number with no physical unit.

Summary

This chapter introduced the fundamental principles of *Bayesian Learning*, which is built on a single crucial theoretical concept:

- In Bayesian learning we assume that not only the feature vectors $\boldsymbol{X}$, but also all the *model parameters* $\boldsymbol{W}$, that define the feature-vector distributions, are *random variables*.

Bayesian learning for classification includes the following main steps for each data category that the classifier should recognize:

- Collect a training data set $\underline{\boldsymbol{x}}$.
- Define a probabilistic model $f_{\underline{\boldsymbol{X}}|\boldsymbol{W}}(\underline{\boldsymbol{x}} \mid \boldsymbol{w})$ for the training data, given the model parameters.
- As part of model building, define a prior density $f_{\boldsymbol{W}}(\boldsymbol{w})$ for the parameters. Subjective or objective methods may be used to choose the prior.
- Calculate the posterior conditional probability density $f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\boldsymbol{w} \mid \underline{\boldsymbol{x}})$ for the model parameters, given the training data.
- Calculate the *predictive density* for any future feature observation $\boldsymbol{X}$ from the same class as the training data:

$$f_{\boldsymbol{X}|\underline{\boldsymbol{X}}}(\boldsymbol{x} \mid \underline{\boldsymbol{x}}) = \int_{\text{all } \boldsymbol{w}} f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w}) f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\boldsymbol{w} \mid \underline{\boldsymbol{x}}) d\boldsymbol{w}$$

  The predictive density is a weighted average over all possible parameter values. This is the main difference from Maximum Likelihood training, where the training procedure gives only a single point estimate of the model parameters.

Finally, the classifier is designed using the *predictive feature density* functions obtained for each of the data categories that the classifier should recognize.

Problems

8.1 In a word-recognition test you present T test words, in background noise, and record all the listener's responses as a sequence $\underline{x} = (x_1, \ldots, x_T)$ with binary elements, $x_t = 1$ indicating a correct response, and $x_t = 0$ for an incorrect response to word number t. The total number of correct responses is $z = \sum_{t=1}^{T} x_t$. We assume that all words are equally difficult to recognize, so that the probability w for a correct response is the same for every t. We also assume that all the T test results are statistically independent.

8.1.a Determine the Maximum Likelihood estimate $\hat{w}_{ML}$ for the probability of correct word recognition.
Hint: See discussion in the introductory Example 8.1.

8.1.b Determine Jeffreys prior density $f_W(w)$ for the unknown probability W of a correct response, in a Bayesian approach to the estimation.
Hint: It is sufficient to consider only a single binary response X in the definition of Jeffreys prior. This will give the same result as using a complete sequence $\underline{X} = (X_1, \ldots, X_T)$.

8.2 (Model Comparison) A word-recognition test with T test words is performed, just like in the previous problem, with two different listeners. The first listener gives z_1 correct responses, and the second listener answers correctly z_2 out of the T test words. We assume that all the T test words are equally difficult, that the results are statistically independent, and that the results for the two listeners are also independent. Now we compare the following two models (hypotheses) for interpreting the test results:

- Model H_0: Both listeners have the *same* probability $W = W_1 = W_2$ for correct recognition.
- Model H_1: The listeners may have *different* values W_1, W_2 for the probability of correct recognition.

The choice between these two models can be seen as a classification problem. We assume that both models are equally probable, *a priori*.

8.2.a Determine a general expression for the conditional probability of the models, $P[H_0 \mid z_1, z_2]$, and $P[H_1 \mid z_1, z_2] = 1 - P[H_0 \mid z_1, z_2]$. Use the Bayesian approach, treating the word-recognition probability parameters as random variables, with a conjugate prior distribution of the beta type

$$f(w \mid a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} w^{a-1}(1-w)^{b-1}$$

where the hyperparameters a and b are selected for a *uniform* prior parameter distribution.

8.2.b Determine $P[H_0 \mid z_1, z_2]$, and $P[H_1 \mid z_1, z_2]$ for the special case of $T = 10$ test words, and listener results $z_1 = 4$ and $z_2 = 6$ correct responses.

8.3 A sequence of random scalar values $\underline{X} = (X_1, \ldots, X_t, \ldots)$ is generated by the following first-order auto-regressive filter process:

$$X_0 = 0$$
$$X_t = aX_{t-1} + cW_t, \quad t \geq 1$$

Here, a and c are constant parameters, and W_t is for each t a Gaussian random variable with mean 0 and variance 1, and all W_t are statistically independent across different t. The constant c is exactly known, but the value of a is unknown. You have observed an outcome sequence $\underline{x} = (x_1, \ldots, x_t, \ldots, x_T)$ generated by this random source.

You will now apply *Bayesian Learning* to determine to what extent the value of a can be known, after using the observed sequence. In this approach we assume that the parameter a is an outcome of a random variable A, but it remains constant for all t. Before observing $\underline{x}$, we express our uncertainty about the parameter value by modeling A as a Gaussian random variable with some arbitrary mean μ_0 and a very large variance σ_0^2. Later we can let $\mu_0 \to 0$ and $\sigma_0 \to \infty$.

8.3.a Determine the *posterior* conditional probability density function for A,

$$f_{A|\underline{X}}(a \mid \underline{x})$$

given the observed $\underline{x} = (x_1, \ldots, x_T)$. Show that this density function has a Gaussian form, and determine its mean μ_T and variance σ_T^2, given the observed sequence.
Hint: For any given value of a and the observed previous value x_{t-1}, X_t is a Gaussian random variable with conditional mean ax_{t-1} and conditional variance c^2.

Determine the likelihood for the combined event $(\underline{X} = \underline{x} \cap A = a)$. Then identify this likelihood expression as a conditional density for A by regarding it as a function of a. Show that this posterior distribution for A is Gaussian, and determine its mean and variance.

8.3.b Express the posterior mean μ_T and variance σ_T^2 for the parameter A, given the observations, as a *recursive update* function, using only the previous values μ_{T-1} and variance σ_{T-1}^2 together with the most recent observations x_T and x_{T-1}.

8.3.c Determine the mean and variance of the next future sample X_{T+1}, given an observed sequence $(x_1, \ldots, x_T)$, including the effect of the remaining uncertainty about the parameter A.

8.4 Your friend seems to win your dice games too often. You suspect that he might be using an asymmetric die that shows symbol 6 with abnormally high probability. You have therefore counted the outcomes from a long series of consecutive trials, as shown in Table 8.1. We now analyze this

Table 8.1: Result of a series of dice throws.

Dice result k	1	2	3	4	5	6
Count N_k	8	12	8	7	10	15

experiment. Let us denote the series of trials as $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, where the outcome from trial no. t is recorded by a six-element binary vector $\boldsymbol{x}_t = (x_{1,t}, \ldots, x_{6,t})^T$ with a 1 in the position corresponding to the result, and 0 in all other positions. For example, if the die showed 4 in trial t, the result is noted as $\boldsymbol{x}_t = (0, 0, 0, 1, 0, 0)^T$. With this notation, the total recorded count, N_k, of results k is simply

$$N_k = \sum_{t=1}^{T} x_{k,t}$$

We regard each trial result $\boldsymbol{x}_t$ as an outcome of the corresponding random binary vector $\boldsymbol{X}_t$, with independent and identical distributions across all trials. The die is characterized by an (unknown) parameter vector $\boldsymbol{W} = (W_1, \ldots, W_6)^T$ with elements defining the probabilities of outcomes, as

$$P\left[X_{k,t} = 1 \mid W_k = w_k\right] = w_k \quad \Leftrightarrow \quad P\left[\boldsymbol{X}_t = \boldsymbol{x}_t \mid \boldsymbol{W} = \boldsymbol{w}\right] = \prod_{k=1}^{6} w_k^{x_{k,t}}$$

We will now apply Bayesian learning for the die probabilities, and therefore regard $\boldsymbol{W}$ as a random vector. As this random vector must have the restriction that all elements are limited as $0 \leq W_k \leq 1$ and also $\sum_k W_k = 1$, we assume that the prior probability density function for $\boldsymbol{W}$ has the *Dirichlet* form, which is designed to account for these restrictions, as descibed in Section 8.4.2.
Hint: Complete solutions are given in the Answers section.

8.4.a Show that the posterior probability density function for $\boldsymbol{W}$, given the observed sequence,

$$f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\boldsymbol{w} \mid \underline{\boldsymbol{x}})$$

also has the Dirichlet form, and determine the posterior hyper-parameter vector $\boldsymbol{a}^{(T)}$ that defines the posterior density after the observed sequence of results from T trials.

8.4.b What is the conditional probability that a future throw will come out with result 6,

$$P\left[X_{6,T+1} = 1 \mid \underline{\boldsymbol{x}}\right],$$

given the observed previous results from T trials?
The result should be expressed algebraically in terms of N_k and prior hyper-parameters $\boldsymbol{a}^{(0)} = (1, 1, 1, 1, 1, 1)^T$ chosen for a uniform prior parameter density. Use MatLab to compute a numerical result.

8.5 Based on extensive statistical measurements, your electricity distributor has guaranteed that the electrical power supply has a mean voltage of $\mu_0 = 225$ V, with a standard deviation $\sigma_0 = 5$ V, across all their customers. You want to measure the actual voltage in your house. Your voltmeter shows the true voltage $\pm$ a random error with standard deviation $\sigma = 3$ V when used in the range required for this measurement. You measure the voltage five times and obtain a sequence $\underline{x} = (x_1, \ldots, x_N) = \{215, 217, 216, 216, 214\}$ V. Assume that all random variables involved are Gaussian.

8.5.a Calculate a Bayesian estimate of the actual voltage using your a priori knowledge (i.e., the given guarantee) and only the first measured result.

8.5.b Calculate a Bayesian estimate of the actual voltage using the a priori knowledge and all the five measured values.

8.5.c Plot your increasing knowledge about the actual voltage as a sequence of probability density functions for the true voltage, using $n = 0$, 1, and 5 measured values.

8.6 (Hypothesis Test) *Dagens Nyheter*, 2010-07-27, wrote:

> "The hot weather scared away the ticks. Until July 26th, only 34 cases of tick-borne encephalitis (TBE) were reported, to be compared with 42 cases during the same period in 2009."

Should we believe that the risk of TBE infection was really lower in 2010 than in 2009, or is the observed reduction perhaps just a random fluctuation in the number of infected people?

Under some mild assumptions, the number X of independent random discrete events counted during a given time interval follows the well-known Poisson distribution with probability mass function

$$f_{X|W}(x \mid w) = P\left[X = x \mid W = w\right] = \frac{w^x e^{-w}}{x!}$$

specified by a single rate parameter w. This parameter defines both the expected value and the variance of the Poisson-distributed random count as

$$E\left[X \mid w\right] = \mathrm{var}\left[X \mid w\right] = w.$$

8.6.a For a Bayesian approach, we regard the unknown parameter w as an outcome of a random variable W. Show that the Gamma distribution (see section 8.4.3) for W is a *conjugate prior* for the Poisson distribution. In other words, given that W follows a Gamma distribution before any observation, show that the posterior distribution for W, given an observed outcome $X = x$, also is a Gamma distribution.

8.6.b Derive expressions for the Gamma hyper-parameters (a_1, θ_1) that specify the *posterior* distribution, given the observation $X = x$ and any assumed hyper-parameters (a_0, b_0) for the *prior* Gamma distribution for the parameter W.

8.6.c Show that Jeffreys prior for the Poisson distribution is

$$f_W(w) \propto \frac{1}{\sqrt{w}}$$

and express asymptotic limits of the hyperparameters for a gamma density that approaches this prior.

8.6.d Calculate the probability that the newspaper's conclusion was correct, i.e., the posterior probability $P\,[W_2 < W_1 \mid x_1, x_2]$ where W_1 is the Poisson rate of infections in 2009, with $x_1 = 42$ observed cases, and W_2 the corresponding rate in 2010, with $x_2 = 34$ cases.
Hint: Use non-informative hyper-parameters (a_0, θ_0) that approach the limiting values corresponding to the Jeffreys prior. The posterior gamma densities for W_1 and W_2 may be approximated numerically by Gaussian densities with corresponding means and variances.

8.7 (Fisher Information) A random variable $\boldsymbol{X}$ has probability density $f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})$, specified by a parameter vector $\boldsymbol{w}$. Prove the equivalence

$$E_{\boldsymbol{X}}\left[\left(\frac{\partial \ln f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{X} \mid \boldsymbol{w})}{\partial w_i}\right)\left(\frac{\partial \ln f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{X} \mid \boldsymbol{w})}{\partial w_j}\right)\right] = \\ = -E_{\boldsymbol{X}}\left[\frac{\partial^2 \ln f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{X} \mid \boldsymbol{w})}{\partial w_i \partial w_j}\right]$$

between the definitions in Eqs. (8.22) and (8.23). Note the conditions required for those two definitions to be equivalent.
Hint: Noting the normalization identity for the density function

$$\int_{\Omega_X} f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})\, d\boldsymbol{x} \equiv 1$$

we can develop the expected value of the first partial derivative as

$$E_{\boldsymbol{X}}\left[\frac{\partial \ln f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{X} \mid \boldsymbol{w})}{\partial w_i}\right] = \int_{\Omega_X} \frac{\partial \ln f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})}{\partial w_i} f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})\, d\boldsymbol{x} =$$
$$= \int_{\Omega_X} \frac{1}{f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})} \frac{\partial f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})}{\partial w_i} f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})\, d\boldsymbol{x} =$$
$$= \int_{\Omega_X} \frac{\partial f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})}{\partial w_i}\, d\boldsymbol{x} = \frac{\partial}{\partial w_i} \int_{\Omega_X} f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})\, d\boldsymbol{x} = 0$$

Then differentiate the integral on the first line again with respect to w_j.

Chapter 9

Approximate Bayesian Learning

Although the general approach to Bayesian learning is simple in principle, it can lead to computational difficulties when applied to complex probabilistic models. In Maximum-Likelihood learning of the parameters in hidden Markov models or Gaussian mixture models, we also encountered similar problems, and Ch. 7 showed that the Expectation Maximization algorithm provides an elegant solution.

Typically, computational difficulties in Bayesian learning arise when we want to derive posterior densities of model parameters in mixture models, where the complete model includes hidden variables that control the choice of mixture components as, for example, in a GMM or an HMM.

This chapter presents *Variational Inference* (VI) as an approach that can be used for Bayesian learning in those more complex situations. Bishop (2006) gives a detailed discussion of VI and other approximate methods for Bayesian learning, for example:

- *Numerical sampling.* Even if the exact posterior parameter distribution cannot be expressed in a closed form, it is possible to generate random samples that follow the exact distribution. The samples can be used, for example, to calculate predictive probabilities. The accuracy can be very good, at the cost of large amounts of computation. Because of the element of randomness and luck involved, these techniques are known as *Monte Carlo methods*; examples include Markov Chain Monte Carlo (MCMC) and Hamiltonian Monte Carlo (HMC). MacKay (2006) contains an excellent in-depth survey of different sampling methods.

- *Expectation Propagation* (EP). This is, like VI, an analytical method to approximate the posterior parameter distribution. However, while the VI approximation, in general, is most accurate near the peak of the

distribution, EP concentrates on describing the global properties of the distribution, such as its mean and covariance, but may conversely be less accurate near the peak. Which method that is preferable depends on the situation, but VI is often considered superior to EP in the case of mixture models.

Here we focus on *Variational Inference* mainly because it often gives explicit formulas that can be solved analytically, and because it is similar to EM. As we will see, VI can be regarded as a generalization of the "EM trick," introduced in Ch. 7. Like EM, we get an iterative optimization procedure that converges to a locally optimal solution. Unlike EM, however, we are able to stochastically model our uncertainty in the parameters and all other unknowns.

9.1 Variational Inference – Notation

For a very general solution to this kind of problem, in this section $\underline{\boldsymbol{Z}} = (\boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_M)$ denotes a combined set of groups of hidden variables and groups of other model parameters, all regarded as random variables. For the HMM, for example, we might let $\boldsymbol{Z}_1$ represent the sequence of discrete state variables that we used to denote as $\underline{S} = (S_1, \ldots, S_T)$, while $\boldsymbol{Z}_2$ might include all the elements in the transition probability matrix that we used to call A, and $\boldsymbol{Z}_3$ could include all the mean vectors of state-conditional Gaussian output density functions, etc. Thus, some variable groups can have discrete distributions, specified by probability mass functions, and other groups may have continuous distributions.

As usual, the training procedure uses a sequence $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ of observed feature vectors $\boldsymbol{x}_t$ that are regarded as samples of corresponding random vectors $\boldsymbol{X}_t$. The observed vectors in the sequence may be drawn from identical or different distributions, statistically independent or dependent of other variables in the sequence. All such model details are specified by some variables in $\underline{\boldsymbol{Z}}$. Some variable group $\boldsymbol{Z}_m$ may include exactly T elements, corresponding to the T observed vectors, whereas other parameter groups may have a fixed size, regardless of the number of observed feature vectors. One variable group $\boldsymbol{Z}_i$ may have a size that corresponds to the number of components in a mixture model, or the number of internal hidden states, and another variable group may control how many such mixture components, or hidden states, that are actually needed in the model. In short, $\underline{\boldsymbol{Z}}$ collects everything that is unknown in the current situation.

Just as in the standard Bayesian approach, we formulate an explicit conditional probability model $f_{\underline{\boldsymbol{X}}|\underline{\boldsymbol{Z}}}(\underline{\boldsymbol{x}} \mid \underline{\boldsymbol{z}})$ for the observations, given all the model parameters and hidden variables. We also need a prior model (density and/or mass) $f_{\underline{\boldsymbol{Z}}}(\underline{\boldsymbol{z}})$ for all the unknown variables and parameters.

The difficult step is to obtain a useful posterior distribution

$$f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\underline{\boldsymbol{z}} \mid \underline{\boldsymbol{x}}) \propto f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}}}(\underline{\boldsymbol{x}}, \underline{\boldsymbol{z}}) = f_{\underline{\boldsymbol{X}}|\underline{\boldsymbol{Z}}}(\underline{\boldsymbol{x}} \mid \underline{\boldsymbol{z}}) f_{\underline{\boldsymbol{Z}}}(\underline{\boldsymbol{z}}) \tag{9.1}$$

Although this expression still looks quite simple, the problem is that we would often like to obtain a separate posterior density for some of the parameter groups in $\underline{\boldsymbol{Z}} = (\boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_M)$, *independent* of other groups of hidden variables. For example, when training HMM parameters with the EM approach, the trained model λ should not depend explicitly on the specific hidden state sequence that might have generated the training data, because the model is to be applied to future test data, generated by other hidden state sequences. In the exact Bayesian posterior of Eq. (9.1), all the groups of parameters and hidden variables, gathered in $\underline{\boldsymbol{Z}}$, may still depend on each other in some complex way, and it may be computationally intractable to find the marginal distribution of, e.g., $\boldsymbol{Z}_1$, as

$$f_{\boldsymbol{Z}_1|\underline{\boldsymbol{X}}}(\boldsymbol{z}_1 \mid \underline{\boldsymbol{x}}) = \int_{\boldsymbol{z}_2} \ldots \int_{\boldsymbol{z}_M} f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_M \mid \underline{\boldsymbol{x}})\, d\boldsymbol{z}_2 \ldots d\boldsymbol{z}_M \tag{9.2}$$

by integrating out all the other unwanted variables.

9.2 Variational Inference – General Solution

The goal of the variational inference approach is to find the best possible approximation of the exact posterior distribution,

$$q(\underline{\boldsymbol{z}}) \approx f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\underline{\boldsymbol{z}} \mid \underline{\boldsymbol{x}}) \tag{9.3}$$

within the constraints imposed by the structure and mathematical form chosen for the approximate density (and/or mass) function q. The model designer is free to choose any suitable parametric mathematical form for this function.

In the following the shorthand notation $E_q\left[h(\underline{\boldsymbol{Z}})\right]$ denotes the expected value of the random (transformed) variable $h(\underline{\boldsymbol{Z}})$, calculated using the density function q, as

$$E_q\left[h(\underline{\boldsymbol{Z}})\right] = \int q(\underline{\boldsymbol{z}}) h(\underline{\boldsymbol{z}}) d\underline{\boldsymbol{z}} \tag{9.4}$$

We now derive an optimization criterion from the following expressions for the log-likelihood of the observed data:

$$\begin{aligned}
\ln f_{\underline{\boldsymbol{X}}}(\underline{\boldsymbol{x}}) &= E_q\left[\ln f_{\underline{\boldsymbol{X}}}(\underline{\boldsymbol{x}})\right] = \\
&= E_q\left[\ln \frac{f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\underline{\boldsymbol{Z}} \mid \underline{\boldsymbol{x}}) f_{\underline{\boldsymbol{X}}}(\underline{\boldsymbol{x}})}{f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\underline{\boldsymbol{Z}} \mid \underline{\boldsymbol{x}})}\right] = E_q\left[\ln \frac{f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\underline{\boldsymbol{Z}}, \underline{\boldsymbol{x}})}{f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\underline{\boldsymbol{Z}} \mid \underline{\boldsymbol{x}})}\right] = \\
&= \underbrace{E_q\left[\ln \frac{f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\underline{\boldsymbol{Z}}, \underline{\boldsymbol{x}})}{q(\underline{\boldsymbol{Z}})}\right]}_{\mathcal{L}(q)} + \underbrace{E_q\left[\ln \frac{q(\underline{\boldsymbol{Z}})}{f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\underline{\boldsymbol{Z}} \mid \underline{\boldsymbol{x}})}\right]}_{\mathrm{KL}\left(q \,\|\, f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}\right)}
\end{aligned} \tag{9.5}$$

Here, the equality on the first line is valid for any q, simply because the log-likelihood, $\ln f_{\underline{\boldsymbol{X}}}(\underline{\boldsymbol{x}})$, by definition does not depend on $\underline{\boldsymbol{Z}}$. The equality on the second line follows from Bayes' rule, and the expansion on the third line simply divides and multiplies by $q(\underline{\boldsymbol{Z}})$.

The Kullback-Leibler divergence $\mathrm{KL}\left(q \,\|\, f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}\right)$ is defined (see Sec. 9.2.1) to be a non-negative measure of the "distance" between q and $f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}$. It is zero only if $q(\underline{\boldsymbol{z}}) = f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\underline{\boldsymbol{z}} \mid \underline{\boldsymbol{x}})$ for all $\underline{\boldsymbol{z}}$. Thus, the remaining term $\mathcal{L}(q)$ on the third line of Eq. (9.5) is a lower bound to the log-likelihood $\ln f_{\underline{\boldsymbol{X}}}(\underline{\boldsymbol{x}})$. By adjusting the function q to find

$$\hat{q} = \underset{q}{\operatorname{argmax}}\, \mathcal{L}(q) = \underset{q}{\operatorname{argmax}}\, E_q\left[\ln \frac{f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\underline{\boldsymbol{Z}}, \underline{\boldsymbol{x}})}{q(\underline{\boldsymbol{Z}})}\right] \tag{9.6}$$

which maximizes the lower bound on the the log-likelihood, we reach an optimal approximation in the sense that the Kullback-Leibler divergence $\mathrm{KL}\left(q \,\|\, f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}\right)$ is minimal. As we already have an explicit expression for $f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\underline{\boldsymbol{z}}, \underline{\boldsymbol{x}}) = f_{\underline{\boldsymbol{X}}|\underline{\boldsymbol{Z}}}(\underline{\boldsymbol{x}} \mid \underline{\boldsymbol{z}}) f_{\underline{\boldsymbol{Z}}}(\underline{\boldsymbol{z}})$, and we are free to choose a mathematical form for q, the objective function $\mathcal{L}(q)$ can be expressed in a tractable form. The details of the optimization depend, of course, on the form chosen for q.

For example, if q is a density function defined by a set of hyper-parameters $\boldsymbol{\theta}$, the objective becomes a (non-linear) function $Q(\boldsymbol{\theta})$, and the optimization can be performed simply by varying those hyper-parameters using a suitable optimization algorithm. In this way, VI can be used as a technique to impose a simple approximate parametric form on complicated posterior distributions. As discussed in Sec. 9.3, we can sometimes use the expression $\mathcal{L}(q)$ to find a suitable mathematical form for q.

9.2.1 Kullback-Leibler Divergence

The Kullback-Leibler divergence, also called *relative entropy*, is a logarithmic measure of the difference between two probability distributions.

Definition 9.1 (KL divergence): *Given two probability density (or mass) functions q and p, both defined for random variables $\boldsymbol{Z}$ in the same space, the Kullback-Leibler divergence is defined as*

$$\mathrm{KL}\left(q \,\|\, p\right) = E_q\left[\ln \frac{q(\boldsymbol{Z})}{p(\boldsymbol{Z})}\right] \tag{9.7}$$

□

If the random variable is continuous-valued, the expectation is

$$\mathrm{KL}\left(q \,\|\, p\right) = E_q\left[\ln \frac{q(\boldsymbol{Z})}{p(\boldsymbol{Z})}\right] = \int_{\boldsymbol{z}} q(\boldsymbol{z}) \ln \frac{q(\boldsymbol{z})}{p(\boldsymbol{z})}\, d\boldsymbol{z} \tag{9.8}$$

If the distribution is discrete, q and p are probability mass functions, and the expectation is

$$\mathrm{KL}\left(q \,\|\, p\right) = E_q\left[\ln \frac{q(\boldsymbol{Z})}{p(\boldsymbol{Z})}\right] = \sum_{\boldsymbol{z}} q(\boldsymbol{z}) \ln \frac{q(\boldsymbol{z})}{p(\boldsymbol{z})} \tag{9.9}$$

The definition is asymmetric in the arguments, so $\mathrm{KL}\left(q \,\|\, p\right) \neq \mathrm{KL}\left(p \,\|\, q\right)$ in general.

Theorem 9.1: *The Kullback-Leibler divergence is non-negative,*

$$\mathrm{KL}\left(q \,\|\, p\right) \geq 0$$

with equality if and only if $q(\boldsymbol{z}) = p(\boldsymbol{z})$ *for all* $\boldsymbol{z}$. □

Proof: For any *convex* function $h(\,\cdot\,)$, *Jensen's inequality* guarantees that

$$E\left[h(g(\boldsymbol{Z}))\right] \geq h\left(E\left[g(\boldsymbol{Z}\right]\right)) \tag{9.10}$$

where $Y = g(\boldsymbol{Z})$ is a transformed scalar variable defined by some function g. As the function $h(\,\cdot\,) = -\ln(\,\cdot\,)$ is convex, Jensen's inequality can be applied to the Kullback-Leibler divergence as

$$\begin{aligned}\mathrm{KL}\left(q \,\|\, p\right) = E_q\left[-\ln \frac{p(\boldsymbol{Z})}{q(\boldsymbol{Z})}\right] \geq -\ln E_q\left[\frac{p(\boldsymbol{Z})}{q(\boldsymbol{Z})}\right] = \\ = -\ln \int_{\boldsymbol{z}} q(\boldsymbol{z}) \frac{p(\boldsymbol{z})}{q(\boldsymbol{z})}\, d\boldsymbol{z} = -\ln \int_{\boldsymbol{z}} p(\boldsymbol{z})\, d\boldsymbol{z} = 0\end{aligned} \tag{9.11}$$

The integral equals 1, simply because p is a normalized probability density function. If the distributions are discrete, the integral is replaced by a sum, and the result is the same. ■

9.3 Factorized Approximation

As already mentioned in Sec. 9.1, the complete set of hidden variables and model parameters, $\underline{\boldsymbol{Z}} = (\boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_M)$, might include separate groups of variables. In order to simplify the learning procedure, or for other reasons, it may be desirable to approximate the posterior density function as a factorized product, as

$$f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\underline{\boldsymbol{z}} \mid \underline{\boldsymbol{x}}) \approx q(\underline{\boldsymbol{z}}) = q_1(\boldsymbol{z}_1) \cdots q_M(\boldsymbol{z}_M) \tag{9.12}$$

This means that we model the different groups of variables in $(\boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_M)$ as statistically *independent*, although the exact posterior density in Eq. (9.1) may include some complex dependencies between the groups.

We will now show that the optimal density for each group can be found iteratively, by choosing

$$\ln q_i(\boldsymbol{z}_i) = E_{q_{j\neq i}}\left[\ln f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\boldsymbol{Z}_1, \ldots \boldsymbol{Z}_{i-1}, \boldsymbol{z}_i, \boldsymbol{Z}_{i+1}, \ldots, \boldsymbol{Z}_M, \underline{\boldsymbol{x}})\right] + c \quad (9.13)$$

Here, c is just a normalization constant, and $E_{q_{j\neq i}}[\,]$ means that the density functions q_j are kept fixed for all $j \neq i$, and these functions are used to calculate the expectation over all those *other* groups of variables $\boldsymbol{Z}_j$, *except* $\boldsymbol{Z}_i$. This is repeated for $i = 1, \ldots, M$ to improve each approximate density q_i at a time, while keeping all the other functions $q_{j\neq i}$ fixed. As the different density functions are actually coupled, the complete procedure must be iterated until the result approaches a stable solution.

The criterion function $\mathcal{L}(q)$ in Eq. (9.5) cannot decrease in any step of this procedure. Therefore, the complete procedure is guaranteed to converge towards a locally optimal solution to the problem formulated in Eq. (9.6). The stepwise improvement of $\mathcal{L}(q)$ is guaranteed by the following theorem:

Theorem 9.2: *Given a joint likelihood function $f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\boldsymbol{z}_1, \boldsymbol{z}_2, \underline{\boldsymbol{x}})$ for a set of variables $\underline{\boldsymbol{Z}} = (\boldsymbol{Z}_1, \boldsymbol{Z}_2)$ and an observation $\underline{\boldsymbol{x}}$, we may seek a factorized approximation $q(\boldsymbol{z}_1, \boldsymbol{z}_2) = q_1(\boldsymbol{z}_1)q_2(\boldsymbol{z}_2) \approx f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\boldsymbol{z}_1, \boldsymbol{z}_2 \mid \underline{\boldsymbol{x}})$. Then, for any fixed q_2, a density function q_1, obtained as*

$$\ln q_1(\boldsymbol{z}_1) = E_{q_2}\left[\ln f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\boldsymbol{z}_1, \boldsymbol{Z}_2, \underline{\boldsymbol{x}})\right] + c, \quad (9.14)$$

maximizes

$$\mathcal{L}(q) = E_q\left[\ln \frac{f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\underline{\boldsymbol{Z}}, \underline{\boldsymbol{x}})}{q(\underline{\boldsymbol{Z}})}\right] \quad (9.15)$$

□

Proof: After taking the expectation over $\boldsymbol{Z}_2$ in Eq. (9.14), the remaining expression is just a function of $\boldsymbol{z}_1$. With proper normalization, this function can be interpreted as the logarithm of a density function, called $\tilde{p}(\boldsymbol{z}_1)$ here.

The objective function $\mathcal{L}(q)$ can be expressed as

$$\begin{aligned}
\mathcal{L}(q) &= \int_{\boldsymbol{z}_1} q_1(\boldsymbol{z}_1) \underbrace{\int_{\boldsymbol{z}_2} q_2(\boldsymbol{z}_2) \ln f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\boldsymbol{z}_1,\boldsymbol{z}_2,\underline{\boldsymbol{x}})\, d\boldsymbol{z}_2}_{E_{q_2}\left[\ln f_{\underline{\boldsymbol{Z}},\underline{\boldsymbol{X}}}(\boldsymbol{z}_1,\boldsymbol{Z}_2,\underline{\boldsymbol{x}})\right]=\ln\tilde{p}(\boldsymbol{z}_1)+\text{const.}}\, d\boldsymbol{z}_1 \\
&\quad - \int_{\boldsymbol{z}_1}\int_{\boldsymbol{z}_2} q_1(\boldsymbol{z}_1) q_2(\boldsymbol{z}_2) \left(\ln q_1(\boldsymbol{z}_1) + \ln q_2(\boldsymbol{z}_2)\right) d\boldsymbol{z}_2\, d\boldsymbol{z}_1 = \\
&\quad = \int_{\boldsymbol{z}_1} q_1(\boldsymbol{z}_1) \ln\tilde{p}(\boldsymbol{z}_1)\, d\boldsymbol{z}_1 + \text{const.} \\
&\quad - \int_{\boldsymbol{z}_1} q_1(\boldsymbol{z}_1) \ln q_1(\boldsymbol{z}_1)\, d\boldsymbol{z}_1 \underbrace{\int_{\boldsymbol{z}_2} q_2(\boldsymbol{z}_2)\, d\boldsymbol{z}_2}_{=1} \\
&\quad - \underbrace{\int_{\boldsymbol{z}_1} q_1(\boldsymbol{z}_1)\, d\boldsymbol{z}_1}_{=1} \underbrace{\int_{\boldsymbol{z}_2} q_2(\boldsymbol{z}_2) \ln q_2(\boldsymbol{z}_2)\, d\boldsymbol{z}_2}_{=\text{const.}} = \\
&= \int_{\boldsymbol{z}_1} q_1(\boldsymbol{z}_1) \ln\tilde{p}(\boldsymbol{z}_1)\, d\boldsymbol{z}_1 - \int_{\boldsymbol{z}_1} q_1(\boldsymbol{z}_1) \ln q_1(\boldsymbol{z}_1)\, d\boldsymbol{z}_1 + \text{const.} = \\
&= \int_{\boldsymbol{z}_1} q_1(\boldsymbol{z}_1) \ln \frac{\tilde{p}(\boldsymbol{z}_1)}{q_1(\boldsymbol{z}_1)}\, d\boldsymbol{z}_1 + \text{const.} = \\
&= -\,\mathrm{KL}\left(q_1 \,\|\, \tilde{p}\right) + \text{const.} \qquad (9.16)
\end{aligned}$$

As the Kullback-Leibler divergence on the last line is minimized to zero, if $q_1(\boldsymbol{z}_1) = \tilde{p}(\boldsymbol{z}_1)$ for all $\boldsymbol{z}_1$, this choice maximizes $\mathcal{L}(q)$ as stated.

This proof also covers the general formulation in Eq. (9.13). We have simply renamed q_i and $q_{j\neq i}$ as q_1 and q_2 in the proof. ∎

9.4 VI Example with Solution

Example 9.1: Consider a sequence $\underline{x} = (x_1, \ldots, x_T)$ of scalar samples x_t, drawn from i.i.d. random variables X_t with the GMM density function[1]

$$f_{X_t|\Theta}(x_t \mid \theta) = 0.5\frac{1}{\sqrt{2\pi}} e^{-x_t^2/2} + 0.5\frac{1}{\sqrt{2\pi}} e^{-(x_t-\theta)^2/2}, \quad \text{for all } t \qquad (9.17)$$

The mean parameter θ of the second Gaussian component is unknown and modeled as an outcome of a random variable Θ. The prior distribution for Θ is assumed to be broad and uniform,

$$f_\Theta(\theta) \to \frac{1}{c}, \qquad c \to \infty \qquad (9.18)$$

[1] This example was proposed by Wasserman (2000, 2012) to point out an interesting difficulty with exact Bayesian learning in mixture models, also discussed in Problem 9.1. The solution presented here uses approximate Bayesian learning to avoid this difficulty.

Calculate the posterior density $f_{\Theta|\underline{X}}(\theta \mid \underline{x})$ for the parameter, given the observed data.

Solution:

The given mixture density is equivalent to assuming that each x_t is generated in a two-step random procedure: First, a binary sample z_t is drawn from the random variable Z_t, with $P[Z_t = 0] = P[Z_t = 1] = 0.5$, and then x_t is generated from the conditional distribution for X_t, given z_t. If $z_t = 0$, the Gaussian component with known zero mean is used, and if $z_t = 1$, the Gaussian component with unknown mean θ is used. Thus, the conditional density for X_t can be written as

$$f_{X_t|Z_t,\Theta}(x_t \mid z_t, \theta) = \left(\frac{1}{\sqrt{2\pi}} e^{-x_t^2/2}\right)^{1-z_t} \left(\frac{1}{\sqrt{2\pi}} e^{-(x_t-\theta)^2/2}\right)^{z_t} \tag{9.19}$$

The prior probability mass function for the hidden sequence $\underline{Z}$ can be written as

$$f_{\underline{Z}}(\underline{z}) = \prod_{t=1}^{T} f_{Z_t}(z_t) = \prod_{t=1}^{T} 0.5^{1-z_t} 0.5^{z_t} \tag{9.20}$$

Applying the improper constant prior $f_\Theta(\theta) = 1/c$, the joint probability density and mass for the complete data, including observations $\underline{X}$, hidden variables $\underline{Z}$, and the parameter Θ, is

$$\begin{aligned} f_{\underline{X},\underline{Z},\Theta}(\underline{x}, \underline{z}, \theta) &= f_\Theta(\theta) \prod_{t=1}^{T} f_{X_t|Z_t,\Theta}(x_t \mid z_t, \theta) f_{Z_t}(z_t) = \\ &= \frac{1}{c} \prod_{t=1}^{T} \left(\frac{0.5}{\sqrt{2\pi}} e^{-x_t^2/2}\right)^{1-z_t} \left(\frac{0.5}{\sqrt{2\pi}} e^{-(x_t-\theta)^2/2}\right)^{z_t} \end{aligned} \tag{9.21}$$

To find a posterior density of the desired form $f_{\Theta|\underline{X}}(\theta \mid \underline{x})$, we use the factorized approximation

$$f_{\Theta,\underline{Z}|\underline{X}}(\theta, \underline{z} \mid \underline{x}) \approx q_1(\theta) q_2(\underline{z}) \tag{9.22}$$

For this purpose, we must start from the log-likelihood

$$\ln f_{\underline{X},\underline{Z},\Theta}(\underline{x}, \underline{z}, \theta) = \sum_{t=1}^{T} -(1 - z_t)\frac{x_t^2}{2} - z_t \frac{(x_t - \theta)^2}{2} + \text{const.} \tag{9.23}$$

Here, the constant prior $1/c$, and all other constants, are collected in the "const." term. The solution in Eq. (9.13) gives two coupled equations:

$$\ln q_1(\theta) = \sum_{t=1}^{T} -E_{q_2}[Z_t] \frac{(x_t - \theta)^2}{2} + \text{const.} \tag{9.24}$$

$$\ln q_2(\underline{z}) = \sum_{t=1}^{T} -(1 - z_t)\frac{x_t^2}{2} - z_t \frac{E_{q_1}[(x_t - \Theta)^2]}{2} + \text{const.} \tag{9.25}$$

As the equations are coupled, they must be solved iteratively: When calculating $E_{q_2}[Z_t]$ in the first equation, we use the approximate q_2 obtained in the previous round. To calculate the expectation over Θ in the second equation, we use the approximate q_1 from the previous iteration.

As the log-density function $\ln q_1(\theta)$ in (9.24) is a second-degree polynomial in θ, the density must be Gaussian, i.e., it is specified by two hyperparameters, the mean μ and the variance σ^2, as

$$q_1(\theta) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(\theta-\mu)^2}{2\sigma^2}} \tag{9.26}$$

Using the shorthand notation $\gamma_t = E_{q_2}[Z_t]$ in (9.24), we get

$$\ln q_1(\theta) = -\frac{\theta^2 \sum_t \gamma_t - 2\theta \sum_t \gamma_t x_t + \sum_t \gamma_t x_t^2}{2} + \text{const.} \tag{9.27}$$

Here, the hyperparameters μ and σ are easily identified as

$$\frac{1}{\sigma^2} = \sum_{t=1}^{T} \gamma_t \tag{9.28}$$

$$\frac{\mu}{\sigma^2} = \sum_{t=1}^{T} \gamma_t x_t \tag{9.29}$$

$$\mu = \frac{\sum_t \gamma_t x_t}{\sum_t \gamma_t} \tag{9.30}$$

These results seem intuitively reasonable: the posterior mean μ for Θ is just a weighted average of observed values x_t, with each observation weighted by the currently estimated probability that it was actually generated by the Gaussian mixture component with the unknown mean Θ. The inverse posterior variance of Θ is $\sum_t \gamma_t$, which is the effective number of observed values assigned to this mixture component. The more observations, the lower the variance.

Similarly, in Eq. (9.25) we see that $\ln q_2(\underline{z}) = \sum_t \ln q_{2,t}(z_t)$, with

$$\begin{aligned} \ln q_{2,t}(z_t) &= -(1-z_t)\frac{x_t^2}{2} - z_t \frac{(x_t-\mu)^2 + E_{q_1}\left[(\mu-\Theta)^2\right]}{2} + \text{const.} = \\ &= (1-z_t)\left(-\frac{x_t^2}{2}\right) + z_t\left(-\frac{(x_t-\mu)^2+\sigma^2}{2}\right) + \text{const.} \end{aligned} \tag{9.31}$$

Thus, the posterior probability mass for Z_t has the form

$$q_{2,t}(z_t) \propto \left(e^{-\frac{x_t^2}{2}}\right)^{1-z_t} \left(e^{-\frac{(x_t-\mu)^2+\sigma^2}{2}}\right)^{z_t} \tag{9.32}$$

As the probability mass must be normalized so that $q_{2,t}(0) + q_{2,t}(1) = 1$, we can identify the posterior distribution as

$$q_{2,t}(z_t) = (1-\gamma_t)^{1-z_t} \gamma_t^{z_t} \tag{9.33}$$

where

$$\gamma_t = E_{q_2}[Z_t] = \frac{e^{-\frac{(x_t-\mu)^2+\sigma^2}{2}}}{e^{-\frac{x_t^2}{2}} + e^{-\frac{(x_t-\mu)^2+\sigma^2}{2}}} \tag{9.34}$$

Again, this result is intuitively reasonable: it is quite similar to the corresponding calculation in Sec. 7.4 for the weight factors in a Gaussian mixture, using the EM algorithm. The estimated weight factor $\gamma_t = P[Z_t = 1 \mid \underline{x}]$ is the currently estimated probability that the observed x_t was generated by the Gaussian component with the unknown mean Θ. The only difference from the EM solution is caused by the fact that the Bayesian approach does not assume a single point estimate $\Theta = \mu$, but also accounts for the remaining variance σ^2 of the parameter Θ.

To reach the final solution we apply Eqs. (9.28), (9.30), and (9.34) in sequence, and repeat this procedure until the result has converged. The only remaining issue is how to *initialize* the algorithm. The exact method for this is not necessarily critical, as the procedure is guaranteed to reach a local optimum anyway.

In this example, we know that about half of the observed samples are probably generated from each of the two mixture components. Therefore, we can initially make a hard assignment for half of the observations to the component centered at zero, by setting $\gamma_t = 0$ for those x_t values that are closest to zero, and $\gamma_t = 1$ for the other samples that are more distant from zero. When training a GMM with the EM approach, it is also common to initialize the component weight factors by a similar hard assignment of the observed samples to different mixture components.

Example 9.1

9.5 EM – Special Case of Variational Inference

This section derives the EM algorithm as a special case of the more general VI approach. We have an observed training-data sequence $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ which is regarded as an outcome of a corresponding random sequence $\underline{\boldsymbol{X}}$. We need to use a model which includes a set of hidden random variables $\underline{\boldsymbol{Z}}$, as well as a parameter vector $\boldsymbol{\theta}$, which is regarded as an outcome of a random vector $\boldsymbol{\Theta}$. For example, in the case of an HMM, the hidden variables $\underline{\boldsymbol{Z}}$ would be the state sequence, and the parameter vector $\boldsymbol{\theta}$ would include all the HMM parameters. The conditional distribution of the observations, given any outcome of the hidden variables and the parameters, is explicitly known as $f_{\underline{\boldsymbol{X}}|\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}} \mid \underline{\boldsymbol{z}}, \boldsymbol{\theta})$. Prior distributions are also known as $f_{\underline{\boldsymbol{Z}}|\boldsymbol{\Theta}}(\underline{\boldsymbol{z}} \mid \boldsymbol{\theta})$ for the hidden variables, and a (possibly non-informative, perhaps improper) density $f_{\boldsymbol{\Theta}}(\boldsymbol{\theta})$ for the unknown parameters. Thus, the

complete log-probability function can be written as

$$\ln f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{z}},\boldsymbol{\theta}) = \ln f_{\underline{\boldsymbol{X}}|\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}} \mid \underline{\boldsymbol{z}},\boldsymbol{\theta}) f_{\underline{\boldsymbol{Z}}|\boldsymbol{\Theta}}(\underline{\boldsymbol{z}} \mid \boldsymbol{\theta}) f_{\boldsymbol{\Theta}}(\boldsymbol{\theta}) \tag{9.35}$$

Let us now assume that the goal of the training procedure only is to find a *point estimate* $\hat{\boldsymbol{\theta}}$ for the parameters, in analogy with the EM approach, even though the Bayesian learning actually can produce a more sophisticated model in the form of a full posterior density for $\boldsymbol{\Theta}$. As the estimated parameter values will be applied to future observations, the estimate should also be independent of any particular outcome of the hidden variables $\underline{\boldsymbol{Z}}$, which are valid only for the training data set. Accepting both these restrictions, we apply a factorized approximation to the posterior distribution,

$$f_{\boldsymbol{\Theta},\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\boldsymbol{\theta},\underline{\boldsymbol{z}} \mid \underline{\boldsymbol{x}}) \approx q(\boldsymbol{\theta},\underline{\boldsymbol{z}}) = q_1(\boldsymbol{\theta} \mid \hat{\boldsymbol{\theta}}) q_2(\underline{\boldsymbol{z}}) \tag{9.36}$$

where we also force the posterior density q_1 to be very sharply peaked, such that nearly all the probability is concentrated at the single point $\hat{\boldsymbol{\theta}}$. Formally, this might be expressed as

$$q_1(\boldsymbol{\theta} \mid \hat{\boldsymbol{\theta}}) \rightarrow \prod_k \frac{1}{c_k} \delta\left(\frac{\theta_k - \hat{\theta}_k}{c_k}\right) \tag{9.37}$$

using a Dirac delta function for each element θ_k of the parameter vector, with some arbitrary scale hyper-parameters c_k having the same physical dimension as the corresponding θ_k. The main consequence of enforcing a very sharply peaked density is that the expected value of any transformation $g(\boldsymbol{\Theta})$ of the random variable is, asymptotically, just the value at the single point,

$$E_{q_1}\left[g(\boldsymbol{\Theta})\right] = \int q_1(\boldsymbol{\theta} \mid \hat{\boldsymbol{\theta}}) g(\boldsymbol{\theta})\, d\boldsymbol{\theta} \rightarrow g(\hat{\boldsymbol{\theta}}) \tag{9.38}$$

In each step of the VI iterative learning procedure, we first apply the general factorized solution in (9.13) to re-estimate the distribution of the hidden variables, using the fixed point estimate $\hat{\boldsymbol{\theta}}_{old}$ for $\boldsymbol{\Theta}$ that was obtained in the previous step:

$$\begin{aligned} \ln q_2(\underline{\boldsymbol{z}}) = E_{q_1}\left[\ln f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{z}},\boldsymbol{\Theta})\right] + \text{const.} = \\ = \ln f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{z}},\hat{\boldsymbol{\theta}}_{old}) + \text{const.} \end{aligned} \tag{9.39}$$

After proper normalization, we have

$$q_2(\underline{\boldsymbol{z}}) \propto f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{z}},\hat{\boldsymbol{\theta}}_{old}) \propto f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}},\boldsymbol{\Theta}}(\underline{\boldsymbol{z}} \mid \underline{\boldsymbol{x}},\hat{\boldsymbol{\theta}}_{old}) \tag{9.40}$$

To find a new point estimate for $\boldsymbol{\Theta}$, we use the general VI solution in Eq. (9.6), and define an objective function where we are free to choose any new

location point $\boldsymbol{\theta}'$ for $q_1(\boldsymbol{\theta} \mid \boldsymbol{\theta}')$:

$$\begin{aligned}\mathcal{L}(q) &= E_q\left[\ln \frac{f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta})}{q_1(\boldsymbol{\Theta} \mid \boldsymbol{\theta}')q_2(\underline{\boldsymbol{Z}})}\right] = \\ &= E_{q_2}\left[E_{q_1}\left[\ln f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}) - \ln q_1(\boldsymbol{\Theta} \mid \boldsymbol{\theta}')\right] - \ln q_2(\underline{\boldsymbol{Z}})\right] = \\ &= \underbrace{E_{q_2}\left[\ln f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{Z}},\boldsymbol{\theta}')\right]}_{Q(\boldsymbol{\theta}',\hat{\boldsymbol{\theta}}_{old})} \underbrace{-E_{q_1}\left[\ln q_1(\boldsymbol{\Theta} \mid \boldsymbol{\theta}')\right]}_{h(\boldsymbol{\Theta})} \underbrace{-E_{q_2}\left[\ln q_2(\underline{\boldsymbol{Z}})\right]}_{h(\underline{\boldsymbol{Z}})}\end{aligned} \tag{9.41}$$

Here, the entropy $h(\boldsymbol{\Theta}) = -E_{q_1}\left[\ln q_1(\boldsymbol{\Theta} \mid \boldsymbol{\theta}')\right]$ is constant, regardless of the location parameter $\boldsymbol{\theta}'$, because changing $\boldsymbol{\theta}'$ only translates the position of the peak of q_1 while its shape remains constant. The entropy $h(\underline{\boldsymbol{Z}}) = -E_{q_2}\left[\ln q_2(\underline{\boldsymbol{Z}})\right]$ does not involve $\boldsymbol{\theta}'$ at all. Thus, to maximize $\mathcal{L}(q)$ as a function of $\boldsymbol{\theta}'$, we only need to maximize the first term $Q(\boldsymbol{\theta}', \hat{\boldsymbol{\theta}}_{old})$. This objective is a function of the new point estimate $\boldsymbol{\theta}'$, which we are free to choose, and it is also a function of the previous fixed value $\hat{\boldsymbol{\theta}}_{old}$, because this value was used to obtain q_2, which was then used to calculate the expectation $E_{q_2}[\,\cdot\,]$. Thus, the optimal choice for the new point estimate is

$$\hat{\boldsymbol{\theta}}_{new} = \underset{\boldsymbol{\theta}'}{\operatorname{argmax}}\, Q\left(\boldsymbol{\theta}', \hat{\boldsymbol{\theta}}_{old}\right) = \underset{\boldsymbol{\theta}'}{\operatorname{argmax}}\, E_{q_2}\left[\ln f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{Z}},\boldsymbol{\theta}')\right] \tag{9.42}$$

Now recall that the EM update step was derived in Chapter 7 as

$$\hat{\boldsymbol{\theta}}^{ML}_{new} = \underset{\boldsymbol{\theta}'}{\operatorname{argmax}}\, Q(\boldsymbol{\theta}', \hat{\boldsymbol{\theta}}_{old}) = \underset{\boldsymbol{\theta}'}{\operatorname{argmax}}\, E_{\underline{\boldsymbol{Z}}}\left[\ln P\left[\underline{\boldsymbol{Z}}, \underline{\boldsymbol{x}} \mid \boldsymbol{\theta}'\right] \,\middle|\, \underline{\boldsymbol{x}}, \hat{\boldsymbol{\theta}}_{old}\right] \tag{9.43}$$

where $Q(\boldsymbol{\theta}', \hat{\boldsymbol{\theta}}_{old})$ is the "EM help function", and the notation $E_{\underline{\boldsymbol{Z}}}\left[\,\cdot \,\middle|\, \underline{\boldsymbol{x}}, \hat{\boldsymbol{\theta}}_{old}\right]$ was used to emphasize that we must use $P\left[\underline{\boldsymbol{Z}} \,\middle|\, \underline{\boldsymbol{x}}, \hat{\boldsymbol{\theta}}_{old}\right]$ to calculate the expected value.

In the VI update equation (9.42), we have

$$q_2(\underline{\boldsymbol{z}}) = f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}},\boldsymbol{\Theta}}(\underline{\boldsymbol{z}} \mid \underline{\boldsymbol{x}}, \hat{\boldsymbol{\theta}}_{old}) \tag{9.44}$$

and

$$\ln f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}},\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{Z}},\boldsymbol{\theta}') = \ln f_{\underline{\boldsymbol{X}},\underline{\boldsymbol{Z}}|\boldsymbol{\Theta}}(\underline{\boldsymbol{x}},\underline{\boldsymbol{Z}} \mid \boldsymbol{\theta}') f_{\boldsymbol{\Theta}}(\boldsymbol{\theta}') \tag{9.45}$$

Thus, the maximization step in (9.42) is exactly the same as the corresponding EM update step, if the prior parameter density is non-informative, i.e., if the density $f_{\boldsymbol{\Theta}}(\boldsymbol{\theta}')$ is constant.[2] Thus, the objective function $Q(\boldsymbol{\theta}', \hat{\boldsymbol{\theta}}_{old})$,

[2] Actually, a non-constant prior function $f_{\boldsymbol{\Theta}}(\boldsymbol{\theta}')$ could also have been applied in the EM procedure, as

$$\hat{\boldsymbol{\theta}}^{MAP}_{new} = \underset{\boldsymbol{\theta}'}{\operatorname{argmax}}\left(E_{\underline{\boldsymbol{Z}}}\left[\ln P\left[\underline{\boldsymbol{Z}}, \underline{\boldsymbol{x}} \,\middle|\, \boldsymbol{\theta}'\right] \,\middle|\, \underline{\boldsymbol{x}}, \hat{\boldsymbol{\theta}}_{old}\right] + \ln f_{\boldsymbol{\Theta}}(\boldsymbol{\theta}')\right)$$

resulting in MAP rather than Maximum-Likelihood estimates.

here derived from the VI approach, can be exactly identical to the EM help function defined in the EM procedure. It is interesting to see that both approaches can lead to exactly the same computational algorithm.

The difference here is, of course, that the Bayesian VI approach starts from a model where the parameters in $\boldsymbol{\Theta}$ are considered as random variables, whereas the EM approach only assumes that the parameters have some fixed unknown values. The VI approach is also much more general, as it can also handle several groups of parameters, and the "EM-like" point approximation may be used for all parameters, or only for a subset. Thus, the EM procedure can be seen as a special case of the VI approach.

Summary

This chapter introduced *Variational Inference* (VI) for approximate Bayesian learning. The VI approach is a general method for finding iterative algorithms to estimate an approximate posterior distribution for unknown parameters and hidden model variables $\underline{\boldsymbol{Z}} = (\boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_M)$, given a set of training observations $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$.

- The result of VI is a density function $q_{\underline{\boldsymbol{Z}}}$, which is a good approximation to the exact posterior density function $f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}$,

$$q_{\underline{\boldsymbol{Z}}}(\underline{\boldsymbol{z}}) \approx f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}(\underline{\boldsymbol{z}} \mid \underline{\boldsymbol{x}})$$

- The mathematical form of $q_{\underline{\boldsymbol{Z}}}$ can be chosen by the experimenter. Parametric and factorized forms

$$q_{\underline{\boldsymbol{Z}}}(\underline{\boldsymbol{z}}) = \prod_{m=1}^{M} q_m(\boldsymbol{z}_m; \boldsymbol{\theta}_m)$$

  are common.

- The approximation is guaranteed to be optimal in the sense that the Kullback-Leibler divergence $\mathrm{KL}\left(q_{\underline{\boldsymbol{Z}}} \,\|\, f_{\underline{\boldsymbol{Z}}|\underline{\boldsymbol{X}}}\right)$ is minimized, within the constraints imposed by the chosen mathematical form for the approximation.

- The VI approach can lead to computationally efficient methods even when the probabilistic model is highly complex.

- The *Expectation Maximization* (EM) algorithm can be seen as a special case of the much more general VI approach.

- Other approximate methods for Bayesian learning exist, including *sampling* methods and *Expectation Propagation* (EP).

Problems

9.1 Consider $\underline{x} = (x_1, \ldots, x_T)$ of scalar samples x_t as drawn from i.i.d. random variables X_t with the following GMM density function:[3]

$$f_{X_t|\Theta}(x_t \mid \theta) = 0.5\frac{1}{\sqrt{2\pi}}e^{-x_t^2/2} + 0.5\frac{1}{\sqrt{2\pi}}e^{-(x_t-\theta)^2/2}, \quad \text{all } t$$

The mean parameter θ of the second Gaussian component is unknown and modeled as an outcome of a random variable Θ. The prior distribution for Θ is assumed to be broad and uniform,

$$f_\Theta(\theta) \to \frac{1}{c}, \qquad c \to \infty$$

9.1.a Calculate the posterior density $f_{\Theta|\underline{X}}(\theta \mid \underline{x})$ for the parameter, given the observed data, using the standard approach for exact Bayesian learning, without introducing any hidden variables. Show that the posterior density becomes *improper*, if we start with the exactly uniform prior density $f_\Theta(\theta) = 1/c$, which is improper, i.e., it cannot be normalized. Show that this difficulty remains, even if the number of observed samples, T, becomes very large.

9.1.b Discuss possible reasons for this counter-intuitive result. In many cases the exact Bayesian learning approach gives a proper posterior parameter distribution even if the prior is improper. Why was there a problem in this case?

9.1.c Try with a Gaussian proper prior density with zero mean and variance σ_0^2,

$$f_\Theta(\theta) = \frac{1}{\sqrt{2\pi}\sigma_0}e^{-\frac{\theta^2}{2\sigma_0^2}}$$

and approach the uniform non-informative density asymptotically, by letting $\sigma_0 \to \infty$ in the final result. Approximate $\ln f_{\Theta,\underline{X}}(\theta, \underline{x})$ by a second-order Taylor series in θ, around the point μ, where the log-probability has a maximum, and interpret the result as a Gaussian posterior density.[4]

9.2 A sequence of random scalar values $\underline{X} = (X_1, \ldots, X_t, \ldots)$ is generated by a first-order auto-regressive filter process, as in Problem 8.3. However, now the generation process is occasionally severely disturbed by spurious

[3] This example was proposed by Wasserman (2000, 2012) to point out an interesting difficulty with exact Bayesian learning in mixture models. An alternative approach is discussed in Example 9.1 on page 197.

[4] This technique is known as the *Laplace approximation.*

additive noise samples with very high variance. The signal sequence is generated as follows:

$$\begin{aligned} X_0 &= 0 \\ X_t &= aX_{t-1} + cU_t + dZ_tV_t, \quad t \geq 1 \end{aligned}$$

Here, a and c are constant parameters for the regular AR process, where U_t is for each t a Gaussian random variable with mean 0 and variance 1. The additional disturbance is defined by another Gaussian random variable V_t with zero mean and variance 1, and the scale factor is $d \gg c$. The constants c and d are exactly known, but the value of a is unknown.

The event of a disturbance of sample t is modeled by a binary random variable Z_t, which can be either 0 or 1. The probability of a sample being disturbed is assumed to be constant for all t, $P[Z_t = 1 \mid w] = w$, but the value of w is unknown. All random variables U_t, V_t, Z_t are statistically independent across different t and independent of each other.

You have observed an outcome sequence $\underline{x} = (x_1, \ldots, x_t, \ldots, x_T)$ generated by this random source. You will now apply *Variational Inference* to determine to what extent the values of a and w can be determined, based on the observed sequence. We assume that the parameter a is an outcome of a random variable A, and that the parameter w is an outcome of a random variable W, and that both parameter values remain constant for all t. We express the prior uncertainty about the parameter value A by modeling it as a Gaussian random variable with zero mean and a large variance σ_0^2. Later we can let $\sigma_0 \to \infty$.

As W represents a probability, its prior density is modeled with a beta distribution

$$f_W(w) \propto w^{\alpha_0 - 1}(1-w)^{\beta_0 - 1}$$

with hyper-parameters $\alpha_0 = \beta_0 = 0.5$ as assigned by the Jeffreys prior.

9.2.a Determine the *posterior* conditional probability distributions for A and W, using a factorized approximation

$$q_{A,W,\underline{Z}}(a, w, \underline{z}) = q_{A,W}(a, w) q_{\underline{Z}}(\underline{z})$$

The posterior density for A and W may become factorized automatically as $q_{A,W}(a, w) = q_A(a) q_W(w)$, without any further approximation. If not, this factorization may have to be enforced as an approximation just to simplify the calculations.

Hint: Start by formulating the joint probability density of all observations, hidden variables, and parameters,

$$f_{\underline{X},\underline{Z},A,W}(\underline{x}, \underline{z}, a, w) = f_{\underline{X}|\underline{Z},A,W}(\underline{x} \mid \underline{z}, a, w) f_{\underline{Z}|W}(\underline{z} \mid w) f_W(w) f_A(a)$$

9.2.b Determine the *predictive* density for the next future sample X_{T+1}, given an observed sequence $(x_1, \ldots, x_T)$, including the effect of the remaining uncertainty about the parameters A and W.

Appendix A

Exercise Project: Pattern Recognition System

The goal of this exercise project is to let you design and build a pattern recognition system capable of recognizing spoken words, simple melodies, or written characters using hidden Markov models (HMMs). The project broadly follows the standard approach to practical pattern recognition laid out in Chapter 4. Within the project, you will develop and test a MatLab pattern recognition toolbox based on HMMs. This toolbox is quite general and can be used for many other purposes after the course.

The structure of a general hidden Markov model is presented in Chapter 5. There are three basic HMM problems of interest that must be solved for the model to be useful in real-world applications:

- Given an observed sequence of feature vectors and a model λ, how do we efficiently compute the probability that the given model would generate the observed sequence?
- How do we adjust the model parameters $\lambda = \{q, A, B\}$, given a training sequence $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1 \ldots \boldsymbol{x}_t \ldots \boldsymbol{x}_T)$, to maximize $P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}|\lambda)$?
- Given an observed sequence and a model λ, how do we determine which hidden state sequence best corresponds to the observations?

The exercise project consists of five assignment steps involving the first two problems. (The last problem type is not included.) The five steps are:

1. HMM signal source
2. Feature extraction
3. Algorithm implementation
4. Code verification and signal database
5. System demonstration

These assignments are presented in detail in the following sections.

A.1 HMM Signal Source

In this first project assignment you will discover how an HMM can generate a sequence with a kind of structured randomness that is typical of many real-life signals. You will do this by coding an HMM signal source in MatLab. You will also have to verify that your implementation is correct. You must submit your work before the deadline, as instructed on the course project web page.

Download the starting package `PattRecClasses` from the course web page, unpack it, and add the resulting directory `PattRecClasses` to your MatLab path. If you are not familiar with implementing classes and object-oriented programming in MatLab, you should consult the MatLab help system on this topic.

The `PattRecClasses` package contains definitions of a few classes that are intended to work together. Each class is defined in a separate sub-directory:

```
@DiscreteD %Implements discrete distribution
@GaussD %Implements a Gaussian scalar or vector distribution
@GaussMixD %Implements a GMM
@HMM %A general HMM class
@MarkovChain %Implements the state-generator part of an HMM
```

An `HMM` object consists of one object called `StateGen` of class `MarkovChain` and one `OutputDistr` object array of class `DiscreteD`, `GaussD`, or `GaussMixD`, to represent the state-conditional HMM output distributions. The purpose of this class structure is to allow extensions to any type of output probability distribution without changing the other classes. (You can easily define additional distribution types yourself, if needed.) Regardless of its type, the array of output distributions must include exactly one element for each of the possible `MarkovChain` states. It is very easy to define a simple HMM using the various class constructor methods:

```
%Example: Define and use a simple infinite-duration HMM
mc=MarkovChain([0.5;0.5], [0.9 0.1;0.05 0.95])%State generator
g1=GaussD('Mean',0,'StDev',1) %Distribution for state=1
g2=GaussD('Mean',2,'StDev',3) %Distribution for state=2
h=HMM(mc, [g1; g2]) %The HMM
x=rand(h, 100); %Generate an output sequence
```

Other examples are given in the sub-directory `testExamples`. Every MatLab class definition must include a *constructor* method that always has the same name as the class, e.g., `MarkovChain`. You will find that many of the class methods have already been implemented for you, although the code can probably be improved.

A.1.1 HMM Random Source

Your task is to code and verify MatLab methods to generate an output sequence of random real numbers $\underline{x} = (x_1 \dots x_t \dots x_T)$ from an HMM with scalar Gaussian output distributions. However, your code should be general enough to handle vector random variables as well.

An HMM output sequence is always the result of *two* separate random operations: First the hidden Markov chain must generate an integer state sequence $\underline{s} = (s_1 \dots s_T)$, and then, for each element s_t in the state sequence, the corresponding state-conditional output distribution generates the random observable output X_t. In the `PattRecClasses` code package three different functions are involved in the process: `@HMM/rand`, `@MarkovChain/rand`, and for example `@GaussD/rand`, if the output distribution is an instance of the `GaussD` class.

1. Open `@DiscreteD/rand` and finish the code precisely as specified by the predefined function interface. Save your function with the same name in the same directory. You may need to use the help system, or look into the `@DiscreteD/DiscreteD` class definition to see exactly how the class properties are stored internally.

2. Open `@MarkovChain/rand` and finish the code precisely as specified by the predefined function interface. Save your function with the same name in the same directory. You may need to use the help system, or look into the `@MarkovChain/MarkovChain` class definition to see exactly how the class properties are stored internally.

   Since the initial state of a Markov chain, and its transitions conditioned on the current state, can be seen as discrete random variables, you can use the `DiscreteD` class and the `rand` method you implemented in the previous step to simplify your work here.

   Note that your function must be able to generate output sequences for either an *infinite-duration* or a *finite-duration* Markov chain. Of course, your function should only produce sequences of finite length, even if the HMM itself could in principle continue forever.

3. Open `@HMM/rand` and finish the code precisely as specified by the predefined function interface. Save your function with the same name in the same directory. Once again you may need to look into the `@HMM/HMM` class definition to see how the class is implemented internally.

   To complete this method you need to call the `rand` method you just implemented for the state-generator component (of class `MarkovChain`) of the HMM. You also need to call another `rand` method for the `OutputDistr` array of the HMM. Just make sure to call this method for the correct `OutputDistr` element, depending on the value of the

corresponding element in the state sequence. (However, the HMM should not generate any output for the final "state" that just indicates that the end of a finite-duration state sequence was reached.)

It is very important that your implementations always follow the interface specifications, both with regards functionality and input/output conventions.

When the `OutputDistr` is a `GaussD` array, the special `@GaussD/rand` method will be invoked automatically by MatLab. This method is very easy to implement for a scalar Gaussian distribution, using the standard MatLab function `randn`. However, it is a little more complicated for a Gaussian vector distribution. Therefore, the `@GaussD/rand` method has already been implemented for you.

A.1.2 Verify the MarkovChain and HMM Sources

To verify your code, use the following infinite-duration HMM $\lambda = \{q, A, B\}$ as a first test example:

$$q = \begin{pmatrix} 0.75 \\ 0.25 \end{pmatrix}; \quad A = \begin{pmatrix} 0.99 & 0.01 \\ 0.03 & 0.97 \end{pmatrix}; \quad B = \begin{pmatrix} b_1(x) \\ b_2(x) \end{pmatrix}$$

where $b_1(x)$ is a scalar Gaussian density function with mean $\mu_1 = 0$ and standard deviation $\sigma_1 = 1$, and $b_2(x)$ is a similar distribution with mean $\mu_2 = 3$ and standard deviation $\sigma_2 = 2$.

1. To verify your Markov chain code, calculate $P(S_t = j)$, $j \in \{1, 2\}$ for $t = 1, 2, 3 \ldots$ theoretically, by hand, to verify that $P(S_t = j)$ is actually constant for all t.

2. Use your Markov chain `rand` function to generate a sequence of $T = 10\,000$ state integer numbers from the test Markov chain. Calculate the relative frequency of occurrences of $S_t = 1$ and $S_t = 2$. The relative frequencies should of course be approximately equal to $P(S_t)$.

3. To verify your HMM `rand` method, first calculate $E[X_t]$ and $\mathrm{var}[X_t]$ theoretically. The conditional expectation formulas $\mu_X = E[X] = E_Z[E_X[X|Z]]$ and $\mathrm{var}[X] = E_Z[\mathrm{var}_X[X|Z]] + \mathrm{var}_Z[E_X[X|Z]]$ apply generally whenever some variable X depends on another variable Z and may be useful for the calculations. Then use your HMM `rand` function to generate a sequence of $T = 10\,000$ output scalar random numbers $\underline{x} = (x_1 \ldots x_t \ldots x_T)$ from the given HMM test example. Use the standard MatLab functions `mean` and `var` to calculate the mean and variance of your generated sequence. The result should agree approximately with your theoretical values.

4. To get an impression of how the HMM behaves, use `@HMM/rand` to generate a series of 500 contiguous samples X_t from the HMM, and plot them as a function of t. Do this many times until you have a good idea of what characterizes typical output of this HMM, and what structure there is to the randomness. Describe the behaviour in one or two sentences in your report. Also include one such plot in the report, labelled using `title`, `xlabel`, and `ylabel` to clearly show which variable is plotted along which axis. You should do this for every plot in the course project.

5. Create a new HMM, identical to the previous one except that it has $\mu_2 = \mu_1 = 0$. Generate and plot 500 contiguous values several times using `@HMM/rand` for this HMM. What is similar about how the two HMMs behave? What is different with this new HMM? Is it possible to estimate the state sequence $\underline{S}$ of the underlying Markov chain from the observed output variables $\underline{x}$ in this case?

6. Another aspect you must check is that your `rand`-function works for *finite-duration* HMMs. Define a new test HMM of your own and verify that your function returns reasonable results.

7. Finally, your `rand`-function should work also when the state-conditional output distributions generate random vectors. Define a new test HMM of your own where the outputs are Gaussian vector distributions, and verify that this also works with your code. (Note that a single instance of the `GaussD` class is capable of generating vector output; stacking several `GaussD`-objects is not correct.) At least one of the output distributions should have a non-diagonal covariance matrix such as

$$\Sigma = \begin{pmatrix} 2 & 1 \\ 1 & 4 \end{pmatrix}.$$

Your assignment report should include

- Copies of your `@DiscreteD/rand`, `@MarkovChain/rand`, and `@HMM/rand` functions, either attached as separate m-files, or in a zip archive. Merely pasting the code into a pdf or doc file is not sufficient. Also, please avoid the proprietary rar format.

- Your theoretically calculated $P(S_t = j)$ for the first infinite-duration HMM, and your corresponding measured relative frequencies.

- Your theoretically calculated $E[X_t]$ and $\text{var}[X_t]$, and your corresponding measured results.

- A plot of 500 contiguous values randomized from the first infinite-duration HMM, with a description of typical output behaviour.

- A discussion of the output behaviour of the second infinite-duration HMM, with answers to the associated questions.
- The definition of your *finite-duration* test HMM, together with the lengths of some test sequences you obtained, and relevant code. Discuss briefly why you think those lengths are reasonable.
- The definition of your vector-valued test HMM, and the code you used to verify that vector output distributions work with your implementation.

A.2 Feature Extraction

The goal of this project is to produce a complete MatLab system for pattern recognition in sequence data. In this particular assignment, you will be working on feature extraction for your recognizer. This is the initial step that takes an input signal and processes it into a form useful for applying standard pattern recognition techniques, such as the HMMs used here.

A.2.1 Feature Extraction in General

The feature extraction process is very important in any pattern recognition system. The input often includes too much data and we need to focus on the signal aspects we are interested in. This depends, of course, on the task.

The parameters defining the HMM must be adapted to match the selected classification features, which is done by the training procedure. Note that it is the HMM, and *not* the feature extractor, that supplies all of the learning and "intelligence" here—feature extraction is just a mechanical process to make it easier for the HMM to do its job by removing unnecessary, confusing, or otherwise unhelpful information. However, it is nevertheless important that the features are relevant to the problem and allow different classes to be distinguished.

Designing a suitable feature extractor is a significant part of the art of successful pattern recognition. Common sense, knowledge about the data, and a bit of thinking are all important components of the design process. A longer discussion of feature extraction is provided in Section 4.2.

In this course project, the specifics of the training data and the feature extraction process depend on the task. There are three different tasks to choose from:

Speech Recognition Design a system capable of recognizing spoken words from a small vocabulary. Instructions for this specific project task are provided in Section A.2.2.

Song Recognition Design a system capable of recognizing short hummed, played, or sung melodies. Instructions for this specific project task are provided in Section A.2.3.

Character Recognition Design a system capable of recognizing letters or characters drawn with the mouse (on-line character recognition). Instructions for this specific project task are provided in Section A.2.4.

You are required to select one of these three tasks that you would prefer to work on. Please make a choice as soon as possible and e-mail it to the course project assistant!

For the sake of variety, we will try to maintain an even distribution of groups over the three different tasks, and will not let everyone work on the

same topic. Tasks are therefore assigned on a first come, first serve basis. If your preferred problem is full, you will be assigned to another task where slots are available. Once the course assistants confirm which problem you will be working on, you may read the appropriate section below and solve the associated problems.

Pattern recognition and hidden Markov models can be used for many other sequence classification problems, in addition to the tasks proposed above. Other applications, to name a few, include recognizing bird species from their songs, music genre detection, distinguishing between speech and singing, distinguishing between different voices, DNA sequence classification, identifying the language of a text, and spam protection. More examples and inspiration might be found among the challenges at kaggle.com.

If you want to, you are very welcome to try your own application for the course project! In that case, please contact the course assistants, so we can help you develop your idea, make sure it fits with the course and with HMM classification, and that you won't have to do too much work.

A.2.2 Speech Recognition

The goal of this project task is to design a pattern recognition system capable of distinguishing between spoken words from a small vocabulary. This is a simple example of the more general area of speech recognition. Speech recognition technology is considered to have great potential, but has proven to be a very challenging practical problem. Speech recognizer performance remains significantly inferior to humans, especially in adverse conditions, and is an area of much active research.

For this particular assignment you will be working on understanding speech signals and the features used in speech recognition. The feature extraction technique commonly used in speech recognition today is known as *mel frequency cepstrum coefficients* (MFCC). It has been designed to mimic several aspects of human hearing and speech perception. The steps involved in this feature extraction technique are outlined below. However, since this is a complex procedure, a ready-made MFCC implementation called `GetSpeechFeatures` has been provided for you on the course project homepage.

The goal of this assignment is to give insight into the ideas behind feature extraction in modern speech recognition. Further information about these topics can be gained from the course EN2300 Speech Signal Processing given by the Electrical Engineering department at KTH.

Sound Signals

On CDs and in the computer, sound is usually represented by discrete samples of fluctuations in air pressure, forming "sound waves." These sampled

signals typically have a very high rate such as 1 411 200 bits per second for CD-quality stereo sound.

While this representation is good for high-fidelity sound playback, it is not very similar to how humans interpret sound, and contains lots of information that is of little use for speech recognition.

When we humans listen to music or speech we do not perceive each of these fluctuations, rather we hear different *frequencies* such as musical chords or the pitch of a voice.

To get a better impression of this, download the sound signal package `Sounds.zip` from the course project homepage. Each separate sound file in the package can be loaded into MatLab using the function `wavread`. You can use the function `sound` to listen to the sounds; use the sampling frequency returned by `wavread` to get the correct playback rate. Plot the female speech signal and the music signal. Label your plots to show which variable is plotted along which axis, and make sure the time axis has the correct scale and units. Then zoom in on a range of 20 ms or so in many different regions of the plots. You will find that the signal in most sections has a "wavy" appearance. It is the frequency of these oscillations that humans perceive.

The Fourier Transform

Information about the frequency content can be extracted through *Fourier analysis*, which you probably are familiar with from other courses.[1] In brief, it is a way to represent a signal not as a succession of distinct sample values at different times, but as a sum of component sine waves with different frequencies, amplitudes, and phases. For discretely sampled signals as discussed here the discrete Fourier transform (DFT) is used. This can be calculated in a computationally efficient manner using a technique known as the fast Fourier transform, FFT.

The downside of the Fourier transform is that, while it can extract the frequency contents of a sound, it discards all timing information in the process; it cannot tell us where in the sound these frequencies appear. Human hearing, meanwhile, is a compromise between time and frequency resolution.

To obtain a similar compromise in sound processing the entire signal is not Fourier analyzed as a whole. Instead, the DFT is applied separately to many short segments, or *frames*, of the signal from different times. When the spectral slices from each and every segments are lined up side by side to show the intensity of each frequency over time, the resulting "heat map" representation of the sound is known as a *spectrogram*, see Figure A.1.

Mathematically, this analysis is typically accomplished by multiplying the signal with a *window function*, which is zero everywhere except on a

[1] If you have not heard of Fourier analysis before, the topic is covered in many signal processing textbooks and on Wikipedia. A nice interactive demonstration of Fourier series approximations in Java is available at `http://www.jhu.edu/~signals/fourier2/`.

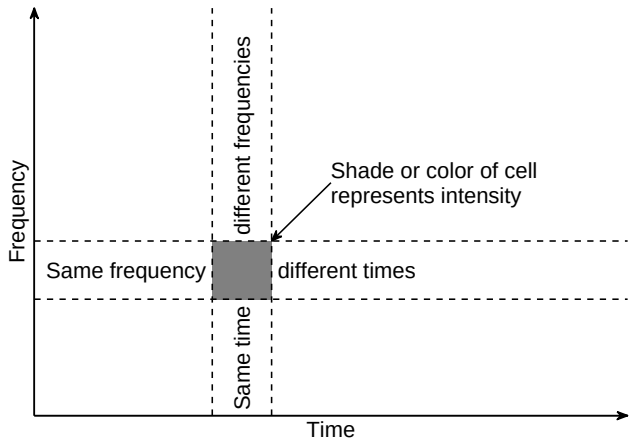


Figure A.1: Schematic illustration of the structure of a spectrogram.

short interval (often on the order of 10–20 ms), before applying the DFT. The window function is then translated and applied to the signal at several equidistant locations as shown in Figure A.2. As demonstrated, analysis windows often overlap. Note that the windows have the shape of a reasonably smooth single hump, since wavy or abruptly changing window functions will introduce artifacts (spurious frequencies) into the analysis. One common window function is the *Hann window* or *raised-cosine window*, which is given by

$$W(n) = \begin{cases} \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N_w}\right)\right) & \text{if } 0 \leq n < N_w, \\ 0 & \text{otherwise,} \end{cases}$$

for discrete n and window length N_w.

The function `GetSpeechFeatures` can compute short-time (windowed) spectrograms. Use it to find spectrograms for the music sample and the female speech sample you downloaded, and then plot the results using the command `imagesc`. Use a window length around 30 ms. You can run `help GetSpeechFeatures` before you start so you know how the function works. Make sure to put the time variable along the horizontal axis and use the additional outputs from `GetSpeechFeatures` to get correct time and frequency scales (and units) for your plots. Again, label your plots and their axes.

You will get an easier-to-interpret picture if you take the logarithm of the spectrogram intensity values before plotting. This corresponds to the decibel scale and the logarithmic properties of human intensity perception. The `colorbar` function and the command `axis xy` could also come in handy.

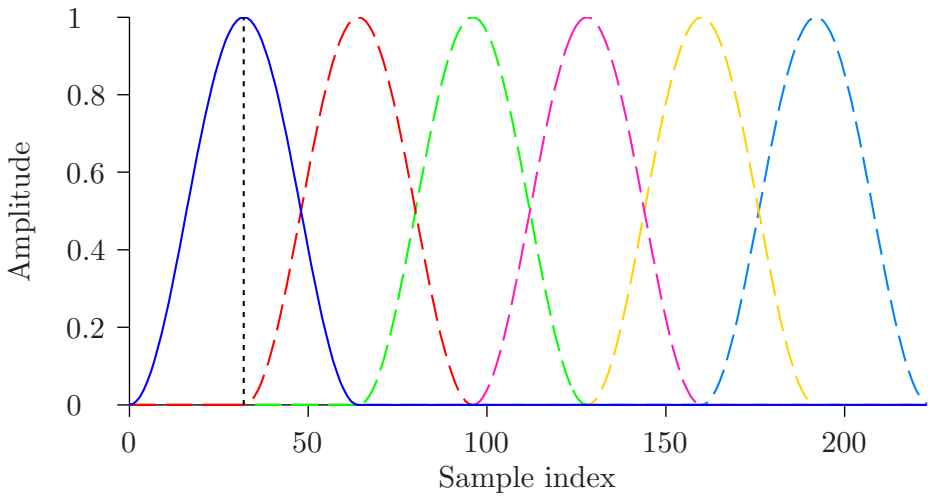


Figure A.2: A set of translated Hann windows of length 64 for short-time spectrogram analysis. A dotted line marks the centroid of the first window.

The music signal contains several harmonics. A harmonic is a component frequency of the signal that is an integer multiple of the fundamental frequency (the pitch of the sound). If the fundamental frequency is f, the harmonics have frequencies f, $2f$, $3f$, etc. While the fundamental frequency determines the current pitch of the sound, the relative strengths of the harmonics determine most other sound qualities, for instance which instrument that is playing and generally what the signal "sounds like."

If you zoom in on the frequencies below 5 000 Hz (which is the most interesting part of the figure here) the music sample should show up as a collection of horizontal line segments, moving up and down in unison over time. The harmonics in the signal are represented by bands of high intensity—low frequency harmonics are typically the most intense. The harmonics are separated by bands of lower intensity, forming striped patterns. Look in your plot to identify and point out the existence of harmonics in the music signal.

Speech consists of both voiced and unvoiced sounds. Voiced sounds such as "aaaaa" are similar to musical instruments, in that they have a certain fundamental frequency and harmonic structure, here determined by the vibration of the vocal folds. In contrast, unvoiced sounds, e.g., "fffff," are mostly noise, moderated by the human vocal tract. You can feel the difference if you touch the laryngeal prominence on your neck while speaking.

Both voiced and unvoiced sounds should be visible in the spectrogram of the speech signal. Notice that the noisy, unvoiced sounds have no single determining frequency or pattern to them, but instead contain a significant amount of energy spread over almost all frequencies.

Voiced and unvoiced sounds should also be visible in the time-domain representation of the signal. To show this, take the female speech signal

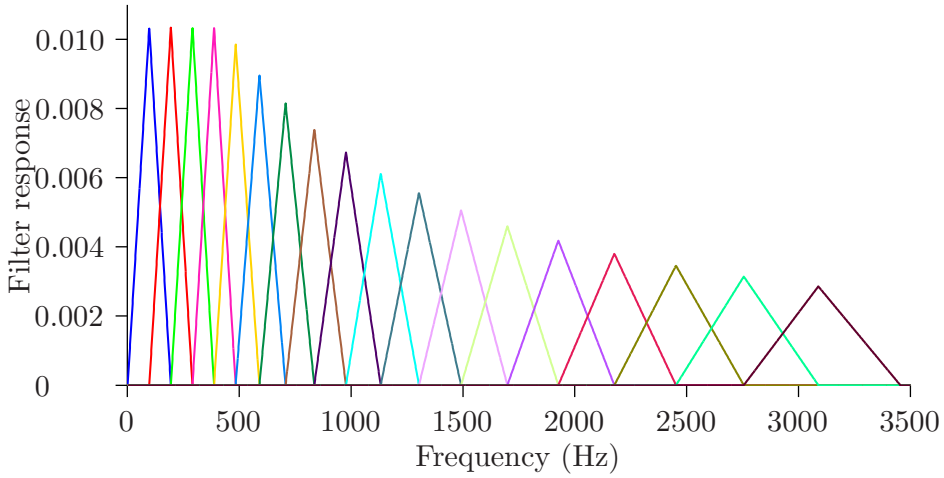


Figure A.3: Frequency responses of a mel-cepstrum-style filter bank.

from `Sounds.zip`, plot it, and zoom in on a range of about 20 ms in many different regions. Identify one region that corresponds to an unvoiced sound, and another that represents a voiced sound.

MFCCs

While the logarithmic short-time spectrogram representation gives a much better impression of how a sound is perceived than just a plot of the signal over time, it is still not particularly well suited for speech recognition—for one thing, the signal still has a much too high rate to be useful. Therefore further processing is applied to calculate the MFCCs. The idea of this processing is to transform each frame spectrum $I(f)$, which measures intensity as a function of frequency, to a frequency and intensity scale that better match human frequency and intensity perception (specifically, our discrimination properties), and then only keep the most general features of the spectrum in this space.

To begin with, human hearing has much coarser frequency resolution in high than in low registers. To imitate this, the frame DFT is smoothed roughly as if the signal had been analyzed through a filter bank with progressively broader sub-bands as shown in Figure A.3.[2] To approximately mimic the human auditory system, and to reduce the amount of data, the filter bank typically contains only around 30 bands in total. The specific locations of the bands are taken from psychoacoustic models of hearing such

[2]There are many MFCC variants, but most implementations actually apply the "mel" smoothing to the absolute *magnitude* of the DFT coefficients (ETSI-ES-201108, 2003), instead of the *power* as originally proposed (Davis and Mermelstein, 1980), although the summation of magnitude values violates Parseval's relation for the filter bank analogy.

as the *bark* or the *mel* scale (the M in MFCCs). Typically a setup with triangular bands like in the figure is used.

The output power values from each filter band are then logarithmically transformed. Finally, MFCCs are obtained by Fourier transforming[3] the log-intensity values within each windowed frame. The result of this process is known as the *cepstrum* for each frame. Typically, the first 13 or so of these coefficients are used as feature vectors in speech recognition systems. The zeroth cepstrum coefficient is related to the overall signal level, while the remainder relate to the general spectral envelope (spectral shape) of the sound. Sometimes the first and second time derivatives of the cepstral coefficients are also included in the feature vector.

Similar to the previously described spectrogram, a *cepstrogram* can be obtained by plotting the frame cepstra over time. Because low-order cepstral coefficients typically have significantly greater average magnitude than higher-order coefficients, it is advisable to always normalize each cepstral coefficient series extracted from each utterance to have zero mean and unit variance, both when plotting and as part of speech feature extraction. This connects with cepstral mean normalization for speaker adaptation discussed in example 4.4.

While the additional Fourier transform in the MFCCs may seem surprising, it brings a number of mathematically favourable properties for speech recognition. Importantly, low-order cepstral coefficients are mostly independent of the fundamental frequency, i.e., the pitch of the speech, while retaining other relevant speech information such as spectral envelope characteristics.[4] This makes it easier for pattern recognition algorithms to perform well regardless of individual pitch variations, for instance when both male and female speakers are considered. In fact, pitch is not important for word recognition in English at all, although it matters in so called *tonal languages*, such as Mandarin Chinese or (to a limited extent) Swedish.

Furthermore, unlike neighbouring frequencies in the short-time spectrum, which can be highly correlated, MFCCs generally have small correlations between coefficients. This is useful partially because some of these correlations are an artifact of the windowed Fourier analysis, and partially because the MFCC distributions can then better be approximated with techniques that do not take correlations into account. Such methods have significantly fewer parameters to estimate, which increases the precision of each individual estimate and decreases the risk of overfitting. GMMs are one example—these are typically constructed with diagonal covariance matrices for the MFCC component distributions.

To get a feel for how MFCCs work you should now plot and compare

[3] Technically, the *type-II discrete cosine transform* is often used.

[4] Since the harmonic structure in voiced segments causes rapid variations in intensity $I(f)$ as a function of frequency f—the stripes you saw earlier—pitch information sits in high-order cepstral coefficients, which we discard.

spectra and cepstra of the female speech and the music signal. You can use the `subplot` function to show both spectrograms and cepstrograms in parallel. Which representation do you think is the easiest for you, as a human, to interpret, and why?

It may be instructive to plot and compare spectrograms and cepstrograms of both a male and a female speaker uttering the same phrase, using the speech samples provided in `Sounds.zip`. First plot the two spectrograms. Can you see that they represent the same phrase? Could a computer discover this? Why/why not? Then plot the two cepstrograms. Can you see that they represent the same phrase now? What about a computer?

Also take one of the speech signals and use the MatLab command `corr` to calculate correlation matrices for the spectral and cepstral coefficient series. Specifically, the matrices should contain correlations coefficients between the different spectral or cepstral coefficient time series (*not* correlations between different time frames in the signal). For the spectrogram, use the log spectral intensities in the calculation. Plot the absolute values of the correlation matrices with `imagesc`. Which matrix, spectral or cepstral, looks the most diagonal to you? Use `colormap gray` to get an easier-to-interpret picture of these matrices.

Dynamic Features

One problem with using raw MFCCs as speech recognizer features is that pitch is not the only thing that makes our voices sound different, and there is a large variability in what MFCCs from the same speech sound can look like. It has been discovered that relative differences in acoustic properties between sounds often are more informative than the absolute MFCC values themselves. In particular, one can estimate the time derivative (velocity) and second derivative (acceleration) of each MFCC at each point in time, and use the resulting series as additional features. These quantities can be estimated by finite differences between frames, so called deltas, and differences of these differences, called delta-deltas, all of which can be computed with the helpful MatLab command `diff`. The resulting derivative-type features are known as *dynamic features*.

To improve accuracy, virtually all successful speech recognizers use feature vectors that include both static and dynamic features (deltas and delta-deltas) for every cepstral coefficient. It is recommended, but not required, that you also use such features in your recognizer. If you wish to do so, provide example MatLab code showing how you process the cepstral coefficients from `GetSpeechFeatures` into a vector of suitable normalized static and dynamic features. Note that these feature vectors can be quite high-dimensional, e.g., $13 \times 3 = 39$ elements per frame if 13 MFCCs are considered.

Your assignment report should include

- A copy of your MatLab code that plots the female speech and the music signal over time and also zooms in on representative signal patches illustrating oscillatory behaviour, as well as voiced and unvoiced speech segments. All code should be attached either in one or more separate m-files, or as a zip archive.
- A copy of your MatLab code that plots spectrograms for the same two signals. Put markers in the graph using `annotation` to demonstrate that you have identified the occurrence of harmonics in the music sample, as well as voiced and unvoiced segments in the speech.
- A copy of your MatLab code that compares the spectrogram and (normalized) cepstrogram representations of the two signals above, and the same for a female and a male speaker uttering the same phrase.
- A copy of your MatLab code that plots and compares correlation matrices for the spectral and cepstral coefficient series.
- Answers to the questions in the text associated with the plots.
- A working MatLab function which computes feature vector series that combine normalized static and dynamic features, if you wish to use this technique in your recognizer.
- Some thoughts on the possibility of confusing the MFCC representation in a speech recognizer. Can you think of a case where two utterances have noticeable differences to a human listener, and may come with different interpretations or connotations, but still have very similar MFCCs? (*Hint*: Think about what information the MFCCs remove.) What about the opposite situation—are there two signals that sound very similar to humans, but have substantially different MFCCs?

Always be sure to clearly state what each figure in your report depicts and which variable that is plotted on which axis.

A.2.3 Song Recognition

The goal of this project task is to design a pattern recognition system capable of distinguishing between a few brief snippets of whistled or hummed melodies. This is known as *query by humming*. While it may sound like an esoteric problem, there are numerous online and mobile song recognition services which support query by humming, including SoundHound and Midomi.com. Query by humming is also a useful feature for karaoke machines, and is even considered in the MPEG-7 standard.

In this particular assignment you will design a MatLab feature extractor suitable for this melody recognition task. For your assistance, you will be given a partial implementation of a feature extractor that computes some useful starting quantities for further processing. To complete the implementation it is useful to know a little about music signals, Fourier analysis, music theory, and human pitch and loudness perception, as described in the following sections. This information will help you design a good feature extractor.

Another classification project that would be possible using very similar features would be to have people sing a certain test melody, and then recognize which persons that are professional singers. Notify the course project assistant if you would rather work on that problem instead.

Music Signal Representation and Analysis

On CDs and in the computer, sound is usually represented by discrete samples of fluctuations in air pressure, forming "sound waves." These sampled signals typically have a very high rate such as 1 411 200 bits per second for CD-quality stereo sound.

While this representation is good for high-fidelity sound playback, it is not very similar to how humans interpret sound. (It is also likely to overload a simple melody recognizer with information.) When we listen to music or speech we do not notice each of these fluctuations, rather we perceive the rates of the fluctuations as different *frequencies*, which may form musical chords or relate to the pitch of a singing or talking voice.

To get a better impression of this, download the sound signal package `Songs.zip` from the course project homepage. Each separate sound file in the package can be loaded into MatLab using the function `wavread`. You can use the function `sound` to listen to the songs; use the sampling frequency returned by `wavread` to get the correct playback rate.

Pick a sound signal in the file and plot it. Label your plot to show which variable is shown along which axis, and make sure the time axis has the right scale and units. Then zoom in on a range of 200 samples or so in many different regions of your plot. You will find that the signal in most sections has a "wavy" appearance. It is the frequency of these oscillations that humans perceive.

The frequencies that make up a sound can be extracted from the signal using the important mathematical tool known as *Fourier series analysis*, which you probably are familiar with from other courses.[5] However, music is a dynamic process, where the sound and its component frequencies change over time. While standard Fourier analysis identifies the frequencies

[5] If you have not heard of Fourier analysis before, the topic is covered in many signal processing textbooks and on Wikipedia. A nice interactive demonstration of Fourier series approximations in Java is available at `http://www.jhu.edu/~signals/fourier2/`.

present in a given signal section, along with their average amplitudes, it does not say at what times in the signal these frequencies appear or disappear. To accommodate frequency content changing over time, sound signals are typically divided into many short pieces, called *windows* or *frames*, each of which is Fourier analyzed separately. This gives an impression of what the signal sounds like at each particular moment. Typically, the windows overlap a bit (often 50%) and are on the order of 20 ms long. Figure A.2 in Section A.2.2 provides an illustration of how overlapping windows may be arranged in practice.

Each frame of the sound contains a wealth of information about the signal at that particular point. Much of this information, extracted by the Fourier analysis, is not even perceived by the listener. Furthermore, many other aspects of the signal may be audible, but not important for our purposes: in addition to the melodies and notes played, the sequence of analysis frames also tells us about the instruments that are playing and their distinct timbres, the vocals (both lyrics and what the different voices sound like), and various special effects and noises present. The latter also includes the rhythm section, with drums and percussion, which may be very loud.

The notes that are playing constitute a very small part of the overall information content in a music signal. Yet this small piece of information is the most important part in defining the melody, and by extension the song, that is being played.

Filtering out relevant information in music is often a challenging problem. To make things easier, we shall not consider full-blown music recognition here. Instead, we concentrate on recognizing short snippets of melodies, hummed, played, whistled, or sung in an otherwise quiet environment. This avoids complexities such as recognizing and separating different instruments or speech sounds, removing percussion and effects, and handling chords and polyphony (several notes played simultaneously).

In a melody recognition task, music is boiled down to its bare essentials: the succession of notes played, their durations, and the durations of pauses between them. Decent features for recognizing melody snippets can then be computed just by knowing the dominant pitch and the intensity of the signal in each frame. (The intensity, or sound volume, is relevant for pause detection.) The MatLab package `GetMusicFeatures`, which can be downloaded from the course web page, extracts exactly this information from a given signal. It returns a matrix

$$\texttt{frIsequence} = \begin{pmatrix} f_1 & f_2 & \cdots & f_T \\ r_1 & r_2 & \cdots & r_T \\ I_1 & I_2 & \cdots & I_T \end{pmatrix},$$

where T is the number of analysis frames, which depends on the signal duration. For each frame t the matrix contains an estimated pitch f_t in Hz, an

estimated correlation coefficient r_t between adjacent pitch periods, and the frame root-mean-square intensity I_t. You can run `help GetMusicFeatures` to get to know more about how the function works.

However, to perform efficient melody recognition, it is necessary to apply some additional tweaks and processing to the data from `GetMusicFeatures`. This post-processing, forming the final steps in designing a good feature extractor for melody recognition, will be up to you to propose and implement. To do so, you need to know a little about music theory and human hearing.

Music Theory and Human Hearing

The musical qualities of a sequence of notes is not determined by the absolute frequencies involved, but the relative difference between them. More specifically, it is the quotient between different frequencies that decides what is harmonic and pleasant, and what is dissonant or out of tune.

The most fundamental relationship between two pitches is that one is double the frequency of the other. This interval is known as an *octave*, and forms the basis of all forms of tonal music. There are very fundamental reasons why this interval is so important, relating to harmonic analysis and the physics of vibrating strings and objects—if notes offset by an octave did not harmonize well together, many simple instruments (including the human voice) would, for instance, appear out of tune with themselves!

An octave is a big leap on any frequency scale. In Western music, the octave is further subdivided into twelve smaller steps, known as *semitones*. In the commonly used tempered tuning, the quotient between the pitches of two adjacent semitones is the same everywhere. This places all semitones at equally spaced distances on a logarithmic scale.

Combinations of semitones form the basis of all Western music. A melody is formed by concatenating a number of stretches of various semitones with different durations, potentially with silent segments in between, to form a sequence of notes. Any interval between notes that is not close to an integer number of semitones may be perceived as being out of tune. The musical notation used in sheet music is a way to write down these note sequences and represent them visually. (As each note in a melody often is shorter than a second, and musical pieces typically last several minutes, entire songs constitute very long, specific sequences of notes picked from the scale; sheet music often requires several sheets. For simplicity, the melody recognition task concentrates on short song snippets only.)

Another reason why musical scales and pitch perception is logarithmic is the wide frequency range of human hearing, which approximately fits in the interval from 20 to 20 000 Hz. This covers three orders of magnitude, or approximately ten octaves. In a linear representation, the lowest octave (20 to 40 Hz) covers a mere 20 Hz, a very small part of the entire 20 kHz range. By operating on a roughly logarithmic scale, humans are able to distinguish

between notes in all octaves with about equal accuracy.

In order to perform melody recognition that is robust to variations, it is important that your features account for the logarithmic nature of musical pitch and human frequency perception, typically by basing your features on the logarithm of the pitch track. Furthermore, the presence of distinct semitones allows devising a discrete representation of the sound, if you like.

Another vital aspect of music perception is that most people, possibly excluding those with perfect pitch, still perceive the same melody if all notes in it are *transposed* (moved) the same number of semitones up or down on the scale. This need not be an integer number of semitones, either, just as long as the frequency ratios always remain unchanged. It is therefore crucial to devise a feature extractor where the output remains largely unchanged if the entire input is transposed up or down by an arbitrary amount. Another way of saying this is that the offset of the pitch track should not matter. In practice, singers and other musicians may often use tuning forks and similar tools to ensure they all use the same offset when they play together.

A final note concerns signal intensity and loudness perception. The intensity component returned by `GetMusicFeatures` is proportional to the sound power in each window. Just like pitch perception, humans can perceive a wide range of sound energies: the scale from the faintest audible sound power to the pain threshold covers about 15 orders of magnitude. Loudness perception, like our pitch perception, therefore follows an approximately logarithmic scale. This improves discriminative accuracy between sounds on the fainter end of the intensity spectrum.

Because of the great differences in intensity that may be expected in your recordings, it is probably a good idea to base any intensity features on the logarithm of the signal intensity as well. Such features may be useful for handling pauses in the input melodies. This quantity may then be processed further, discretized, etc., depending on what kind of features you create. (Note that, in recordings, the output level and other characteristics of the recorded signal also depend on the microphone and its properties.)

Sometimes, there may not be a clear-cut threshold intensity separating notes and non-notes in the input. Moreover, the sound intensity can be high even if there is no note playing, for instance if the signal has a lot of noise. For more robust note detection, one may look at the estimated correlation coefficient between pitch periods, also computed by `GetMusicFeatures`. This correlation will be high (near one) in signal segments with a clear pitch, but lower in non-harmonic (noisy or silent) regions of the sound. A third strategy for coping with pauses can be to look at the pitch estimate itself, and see how it behaves in silent or unvoiced signal segments. Certain pitch extraction techniques may consistently indicate very high or very low pitch frequencies in such regions. Other pitch extractors just return noisy, seemingly random values when no tones are present.

Feature Extractor Design

You now know all the theory necessary to design and implement a set of features for melody recognition, based on signal pitch, correlation, and intensity series from `GetMusicFeatures`. Your features may be either continuous and vector-valued or discrete, but need to have the following properties:

1. They should allow distinguishing between different melodies, i.e., sequences of notes where the ratios between note frequencies may differ.
2. They should also allow distinguishing between note sequences with the same pitch track, but where note or pause durations differ.
3. They should be insensitive to transposition (all notes in a melody pitched up or down by the same number of semitones).
4. In quiet segments, the pitch track is unreliable and may be influenced by background noises in your recordings. This should not affect the features too much, or how they perceive the relative pitches of two notes separated by a pause.

It is also desirable, but not necessary, if your features satisfy:

5. They should not be particularly sensitive if the same melody is played at a different volume.
6. They should not be overly sensitive to brief episodes where the estimated pitch jumps an octave (the frequency suddenly doubles or halves). This is a common error in some pitch estimators, though it does not seem to be particularly prevalent in this melody recognition task.

To help you develop your features you should use the example recordings in `Songs.zip`: two of these are from the same melody, while one is from a different melody. Make a plot of the three pitch profiles of these recordings together, and another plot with the three corresponding intensity series. Place time along the horizontal axis in your plots. Also try changing the axes in the plots to use logarithmic scales (e.g., using `set(gca,'YScale','log')`), and look at the plots again. To see the melodies more clearly in the pitch profile plots here, you may want to look at the frequency range 100–300 Hz especially.

The graphs should give you an understanding of the data series you will be working with. Make sure you understand the relationship between the plots and what the example melodies sound like.

To assist you in your understanding, the code you downloaded also includes a command, `MusicFromFeatures`, capable of creating sound signals that closely match a given `frIsequence`. You can use this command to convert the output from `GetMusicFeatures` back into sounds. Because feature

extraction is lossy, the restored audio often sounds very different from the original. You should be able to hear that information about the melody is retained in the `frIsequence`, while most other information is gone.[6]

When you design your feature extractor, think about all the requirements above and what you can do to address them. Then code a MatLab function that implements your ideas for feature extraction and fits with the `PattRecClasses` framework. Use your knowledge, and your mathematical and engineering creativity to come up with a good method! Keep in mind that there are many ways to solve this problem. You can test your work and ideas on the example files in `Songs.zip`.

Your features need to match the output distributions you plan to use. Specify if you plan to use either `DiscreteD`, `GaussD`, or `GaussMixD` output distributions for your HMMs. Then make sure that the feature values you generate fit with your choice of output distribution, for instance that they have the same range.

When working on your features, it makes sense to think about the kind of processes described and generated by HMMs. To be described well by HMMs, your feature sequences should look like something an HMM can produce. As seen in the previous assignment, HMM output tends to consist of stationary segments with similar behaviour. Transitions between segments are instantaneous. If your feature sequences similarly have stationary or slowly changing regions with relatively well-defined boundaries, they will probably be modelled well by HMMs.

Conversely, if your features do not look like something that the `HMM` class from `PattRecClasses` can easily produce, chances are your classifier will work poorly. In particular, here are some things you should try to avoid:

- Do not mix and match discrete and continuous values in your feature sequences. Decide on either continuous-valued or discrete features, and stick with your choice.

- If your features are continuous, do not let the exact same numerical value appear more than once. (Do not have segments where the output value shows no variation at all, for example.) If the same numerical value appears more than once in the training data, this can lead to variance estimates being exactly zero, causing divisions by zero and errors in your classifier later on.

- If your features are continuous-valued, do not put in isolated points in the feature sequence with radically different behaviour from the

[6]Sampling from a trained model to generate new signals similar to the training data is known as *synthesis*. Sampling from an HMM trained on output from `GetMusicFeatures`, for example, yields new `frIsequence`s which can be played back using `MusicFromFeatures`. This shows what the model has learned. You are welcome to try this later in the project if you like. The results may surprise you.

rest. Even if it might be theoretically possible to create an HMM that describes such data, it will be difficult to learn such a model with the tools given in this course. If there are highly different, isolated points in the data sequence, these will look a lot like outliers, and the information in them is likely to be lost on the HMM.

- If your features are discrete-valued, only use scalar, positive integer values with a finite, fixed upper bound, as these are the only numbers the `DiscreteD` class can handle. You can always convert negative, non-integer, and vector-valued discrete variables to positive integers by enumerating all possible output values or output vectors: the set $\{0, 0.5, 1.5\}$ can be mapped to $\{1, 2, 3\}$, for instance.

- Don't necessarily try to remove "noise" from the output. As seen in A.1.2 point 5, noise can carry information about the state or class, and the HMM is designed to be able to model uncertainty and variability. Unless you are confident some piece of information is unhelpful, leave it in.

- Don't overthink things. Most of the complicated and intricate feature extractors that have been submitted have worked worse than the clean and simple schemes. In general, it is the HMM, rather than the feature extractor, that should supply any intelligence here.

Checking Your Feature Extractor

Once you have designed and started coding a feature extractor proposal, it is a good idea to graphically inspect the feature series produced by your extractor for the given example files. Figure out an informative way to plot or graphically illustrate your features, so that different feature series can be compared. If the regular `plot` command isn't good enough for you (say if you are using high-dimensional feature vectors), you can look into the command `imagesc` to see if it better fits your needs.

Plotting the features series from the example files should let you confirm that the series are reasonably similar between two examples of the same melody, but differ more between examples from different melodies. Keep in mind, though, that there is a difference between curves that are visually similar to the human eye, and mathematical similarities that a pattern recognition system can exploit.

To verify that your features are transposition independent, multiply the pitch track returned by `GetMusicFeatures` by 1.5, and use this in your feature extractor. The output should be virtually the same as with the original pitch. If it isn't, you likely have to rethink parts of your feature extractor. Make a plot to show that your features are equivalent before and after pitch track multiplication. Since two of the examples in `Songs.zip` are

from the same melody, these should have similar feature sequences as well. Verify this in another plot. If your feature function passes these tests, it is likely you have something that will give decent results in practice.

While feature extractor design is one of the most fun and creative aspects of this project, we also recognize it could be the most challenging assignment. As always, the teaching assistants are available to answer your questions. But before you ask, make sure to read about, think about, and work with the problem in MatLab as much as possible! The more effort you have made to understand, the better your questions will be, and the more useful the answers.

Your assignment report should include

- Plots of the pitch and intensity profiles of the three recordings from `Songs.zip`, two from the same melody, and one from another song. MatLab code files for these plots should also be included. Make sure that the scales and units are correct, and show which curve corresponds to which recording.

- A clear specification of the design of your feature extractor, how it integrates with the `PattRecClasses` output distributions, and how your scheme addresses each of the requirements listed previously. It must be clear that you understand your extractor and why it works.

- Working MatLab code for your feature extractor, either attached as one or more separate m-files, or in a zip archive.

- Plots illustrating how your extracted features behave over time for the three example files. It should be clear that two of the series are quite similar, whereas the third is significantly different. Again, code for generating these plots should be included.

- A plot that compares your feature output between a melody with a transposed pitch track and the original recording, showing equivalence, along with code for generating the plot.

- A discussion of aspects not captured by your feature extraction system, and possible cases where your feature extraction scheme may work poorly. Is there any way a human can confuse the system, and perform a melody or create a sound signal that we think sounds similar to another performance, but where the features produced by your function are quite different between the two performances? What about the opposite situation—can we create two recordings that sound like two different songs altogether, but which actually generate very similar feature sequences?

Always be sure to clearly state what each figure in your report depicts and which variable that is plotted on which axis.

A.2.4 Character Recognition

The goal of this project task is to design a pattern recognition system capable of distinguishing between a few different characters, drawn on the computer screen with the help of a mouse. This kind of task, where you have access to the pen movements used when writing a character, is known as *on-line character recognition*, in contrast to *off-line character recognition*, where only an image of the final character is available. This recognition problem is of obvious practical interest, for instance to simplify text input on portable devices such as the recently-popular tablet computers. It is also related to the topic of *mouse gestures*, where mouse movement patterns are used to issue commands to a computer.

In this particular assignment you will design and implement a feature extractor MatLab function that operates on data from mouse movements.

Feature Extractor Design

For the graphical symbol recognition task you will use an interface function `DrawCharacter` that can be downloaded from the course webpage. When the `DrawCharacter` function is called a blank window appears. The user can draw a character in this window, using the mouse as a pen, by holding down the left mouse button. Releasing the mouse button lifts the pen from the paper, and the mouse can be moved without leaving traces.

When the user is finished and closes the window, the function returns a matrix containing a sequence of Cartesian coordinates (and one additional variable), sampled at points along the motion of the mouse in the window, e.g.,

$$\texttt{xybpoints} = \begin{pmatrix} x_1 & x_2 & \cdots & x_L \\ y_1 & y_2 & \cdots & y_L \\ b_1 & b_2 & \cdots & b_L \end{pmatrix}.$$

The length L of the sequence depends on how complicated your symbol is, your drawing speed, and possibly the speed of your MatLab system.

The extra variable b in the series is a binary bit which is 1 at points where the left mouse button was pressed and 0 elsewhere. It thus indicates whether the user was drawing or not at any given point in time.

Your assignment is to use the coordinate and mouse-button data series to create features for recognizing the character drawn by the user. Of course, the features should facilitate good and robust classification performance as much as possible. There are therefore a number of requirements that your feature extractor must satisfy:

1. The feature output should be similar regardless of exactly *where* on the screen the user draws the symbols.

2. The features should disregard the absolute *size* of your symbols.

3. The path of the mouse when not drawing should not matter, since it leaves no visible results. However, you should still take into account the relative position between the end of one pen stroke and the start of the next. If not, the characters "T" and "+" would not be possible to distinguish, since the only difference between these characters is the offset between the vertical and the horizontal lines.

4. Data from before the user starts drawing and after the last stroke of the character has been drawn should be ignored.

5. The feature extractor should not fail if two adjacent frames are identical.

You may use either discrete or continuous features for your feature extractor. There is no single right answer to this design problem, so use your common sense, along with your mathematical and engineering creativity to come up with a good method!

Your feature extractor should be implemented as a MatLab function that operates on data series from `DrawCharacter`, and returns another series of features. As stated earlier, you should not build any advanced intelligence into your feature extractor; that is for the HMM to supply. The main information that your feature sequence should convey to the classifier is your pen (mouse) movements while you were drawing on the screen, and the relative position between different strokes. Notice that two identical-looking lines or curves may appear different to the system if drawn forwards or backwards.

Your features need to match the output distributions you plan to use. Specify if you plan to use either `DiscreteD`, `GaussD`, or `GaussMixD` output distributions for your HMMs. Then make sure that the feature values you generate fit with your choice of output distribution, for instance that they have the same range.

When working on your features, it makes sense to think about the kind of processes described and generated by HMMs. To be described well by HMMs, your feature sequences should look like something an HMM can produce. As seen in the previous assignment, HMM output tends to consist of stationary segments with similar behaviour. Transitions between segments are instantaneous. If your feature sequences similarly have stationary or slowly changing regions with relatively well-defined boundaries, they will probably be modelled well by HMMs.

Conversely, if your features do not look like something that the `HMM` class from `PattRecClasses` can easily produce, chances are your classifier will work poorly. In particular, here are some things you should try to avoid:

- Do not mix and match discrete and continuous values in your feature sequences. Decide on either continuous-valued or discrete features, and stick with your choice.
- If your features are continuous, do not let the exact same numerical value appear more than once. (Do not have segments where the output value shows no variation at all, for example.) If the same numerical value appears more than once in the training data, this can lead to variance estimates being exactly zero, causing divisions by zero and errors in your classifier later on.
- If your features are continuous-valued, do not put in isolated points in the feature sequence with radically different behaviour from the rest. Even if it might be theoretically possible to create an HMM that describes such data, it will be difficult to learn such a model with the tools given in this course. If there are highly different, isolated points in the data sequence, these will look a lot like outliers, and the information in them is likely to be lost on the HMM.
- If your features are discrete-valued, only use scalar, positive integer values with a finite, fixed upper bound, as these are the only numbers the `DiscreteD` class can handle. You can always convert negative, non-integer, and vector-valued discrete variables to positive integers by enumerating all possible output values or output vectors: the set $\{0, 0.5, 1.5\}$ can be mapped to $\{1, 2, 3\}$, for instance.
- Don't necessarily try to remove "noise" from the output. As seen in A.1.2 point 5, noise can carry information about the state or class, and the HMM is designed to be able to model uncertainty and variability. Unless you are confident some piece of information is unhelpful, leave it in.
- Don't overthink things. Most of the complicated and intricate feature extractors that have been submitted have worked worse than the clean and simple schemes.

Verify Your Feature Extractor

When you have figured out and started coding a feature extractor proposal, you should do a number of tests to see that it works as expected and satisfies the requirements listed above. For this you may decide on a reasonably simple character such as "P" and try drawing it three times: once in the top

left quadrant of the drawing area, once in the bottom right quadrant, and finally twice as big as the other two examples, filling the entire window. For each example, save the data sequence returned by `DrawCharacter` for reference. You can later feed these saved data sequences into `DrawCharacter`'s companion function `DisplayCharacter` to plot the characters you drew.

Figure out an informative way to plot or otherwise represent your feature sequences graphically, so that one can compare different sequences and see how similar they are. If the regular `plot` command isn't good enough for you (say if you are using high-dimensional vectors), you can look into the command `imagesc` to see if it better fits your needs. Use your chosen graphical representation to compare the three different examples of the same character from the previous paragraph, to verify that they are all quite similar. They should not merely be similar to the human eye, which the original examples already are, but the plots should make clear that there are mathematical similarities between the feature sequences, which a pattern recognition system may pick up on.

Also compare the feature series against the features that are produced when you draw a different character, to ensure that there are significant differences between this feature series and the three considered above.

Finally, verify that mouse movements when the pen is lifted between different strokes do not matter, but only the relative offsets between strokes. To do this, write a multi-stroke character, but move the mouse around in crazy ways between strokes. Plot the resulting feature series, and verify that it does not differ much from the series that results when you draw the same character in a normal way. Also compare the features representing the characters "T" and "+". The different offsets of the horizontal lines in the two characters should be reflected by differences in the corresponding feature series. Point out these differences in your figure (or figures), preferably using the `annotation` command.

If the feature extractor you coded passes all these tests, it is likely that your final character recognition system will be capable of good performance.

While feature extractor design is one of the most fun and creative aspects of this project, we also recognize it could be the most challenging assignment. As always, the teaching assistants are available to answer your questions. But before you ask, make sure to read about, think about, and work with the problem in MatLab as much as possible! The more effort you have made to understand, the better your questions will be, and the more useful the answers.

Your assignment report should include

- A clear specification of the design of your feature extractor, how it integrates with the `PattRecClasses` output distributions, and how it addresses each of the requirements listed previously. It must be clear

that you understand your extractor procedure and why it works.

- Working MatLab code for your feature extractor, either attached as one or more separate m-files, or in a zip archive. The input to the feature extractor should be a data series from `DrawCharacter`.
- Plots of the big and small examples you drew of the same character, and informative illustrations of the corresponding feature sequences. This should confirm that the sequences are mostly similar, even though the three examples are visually different. MatLab code files for generating your plots should be included.
- Plots displaying a different character and its feature sequence. This should show substantial differences from the previous examples. Code should again be included.
- One or more figures comparing feature series resulting when writing the same character with wildly different pen movements between strokes, versus normal pen movements. This should demonstrate feature insensitivity to path while the pen is lifted. Also include code for generating the figures, and a MatLab mat-file with the original series from `DrawCharacter` for the two examples, so we can verify that the movements are different.
- One or more figures comparing the feature series of "T" and "+", along with code that generates the figures. There should be clear differences in the features, and an `annotation` that points to these differences.
- A discussion of aspects not captured by your feature extraction system. Is there any way a human can confuse the system, and write the same character twice in different ways, so that the end result looks the same to us, but the feature series are radically different? What about the opposite situation—can we draw two characters that have different meanings to us humans, but which generate very similar feature sequences?

Always be sure to clearly state what each figure in your report depicts and which variable that is plotted on which axis.

A.3 Algorithm Implementation

In this assignment you will implement one of two very important HMM algorithms: the Forward Algorithm and the Backward Algorithm. You will be assigned to one of these two algorithms by the teaching assistants. Read the appropriate section below and implement the associated functions.

A.3.1 The Forward Algorithm

In this section you will implement and verify a MatLab function to perform the *Forward Algorithm*. This algorithm calculates conditional state probabilities, given an observed feature sequence $(\boldsymbol{x}_1 \dots \boldsymbol{x}_t \dots)$ and an HMM λ, as

$$\hat{\alpha}_{j,t} = P(S_t = j | \boldsymbol{x}_1 \dots \boldsymbol{x}_t, \lambda).$$

You can also use the *forward scale factors* $(c_1 \dots c_t \dots)$, calculated by the algorithm, to determine the total probability $P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}} | \lambda)$ of the observed sequence given the HMM. The Forward Algorithm is also an essential step in HMM training, and you will need it for your final recognizer.

The Forward Algorithm consists of three steps, defined by equations in Sec. 5.4:

Initialization: Eqs. (5.42)–(5.44)

Forward Step: Eqs. (5.50)–(5.52)

Termination: Eq. (5.53), only needed for finite-duration HMMs.

Implement the Forward Algorithm

In the `PattRecClasses` framework the Forward Algorithm is a method of the `MarkovChain` class, because the algorithm does not need to know anything about the type of output probability distribution used by the HMM. The Forward Algorithm needs as input only a matrix `pX` with values *proportional to* the state-conditional probability mass or density values for each state and each frame in the observed feature sequence.

The proportionality scale factors, one for each frame, must be defined and stored by the calling function, if needed. This is not the responsibility of the Forward Algorithm. The purpose of the scaling is to avoid very small probability values, which can cause numerical problems in the computations.[7]

Your task is to complete the `@MarkovChain/forward` method as specified in the function interface and comments.

[7] Another approach to numerically stable HMMs is described in the technical report `http://bozeman.genome.washington.edu/compbio/mbt599_2006/hmm_scaling_revised.pdf` by Tobias P. Mann.

Note especially that your function must work for both *infinite-duration* and *finite-duration* Markov chains, and that the output results are slightly different in these two cases. Save your finished version of the function under the same file name in the same directory.

Verify the Implementation

The only way to test your implementation is to do a calculation by hand, and compare the result with the function output. However, this tedious calculation has already been done by previous students.

Create a finite-duration test HMM with a Markov chain given by

$$q = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0 & 0.9 & 0.1 \end{pmatrix}$$

and where the state-conditional output distribution is a scalar Gaussian with mean $\mu_1 = 0$ and standard deviation $\sigma_1 = 1$ for state 1, and another Gaussian with $\mu_2 = 3$ and $\sigma_2 = 2$ for state 2. Then use your own `forward` implementation to calculate $\hat{\alpha}_{j,t}$, $t = 1 \ldots 3$ and c_t, $t = 1 \ldots 4$, with an observed finite-duration feature sequence $\underline{x} = (-0.2, 2.6, 1.3)$. For this simple test example, the result of your `forward` function should be

```
alfaHat =
    1.0000    0.3847    0.4189
         0    0.6153    0.5811

c =
    1.0000    0.1625    0.8266    0.0581
```

assuming you applied `forward` directly to the scaled probabilities produced by `prob`. If you used the accompanying scale factors to undo the scaling before calling `forward`, the first three elements in `c` may differ. This is not recommended.

Since your implementation also must work for infinite-duration HMMs, it is a good idea to figure out a way to test this case as well.

Probability of a Feature Sequence

Later in the project you will need a function to calculate the log-probability of an observed sequence, $\log P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}|\lambda)$, given an HMM instance λ. The log-probability is used since the regular observation probability $P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}|\lambda)$ very often is too small to be represented in a computer. For this reason $P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}|\lambda)$ should never be used directly in any computations.

Complete the small MatLab function `@HMM/logprob` to perform the log-probability calculation by calling your `forward` method with the scaled conditional probability values supplied by the `OutputDistr` `prob`-method (try

`help GaussD/prob` for more information). However, do not forget to include the scale factors in the subsequent calculation. For the HMM and observation sequence above the result should be that $\log P(\underline{X} = \underline{x}|\lambda) \approx -9.1877$, using the natural logarithm, if your implementation works for this example. Your function should also work when the input is an array of HMM objects, and compute the probability of the feature sequence under each model.

Note that if your `c`-vector looks like

```
c =
    0.3910    0.0318    0.1417    0.0581
```

it is likely that your implementation of the Forward Algorithm is correct, but that you are applying it to the true, unscaled probabilities by accounting for the scaling factors already before running `forward`. This can lead to problems later on. To get numerically robust calculations of very small log-probabilities, which is vital for later parts of the project, you will have to apply `forward` to the *scaled* probability values `pX`, as given by `prob`, first—only thereafter should the scaling factors be taken into account.

Your assignment report should include

- A copy of your finished `@MarkovChain/forward` function.
- A copy of your finished `@HMM/logprob` function.

A.3.2 The Backward Algorithm

In this project step you will implement and verify a MatLab function to perform the *Backward Algorithm*. The Backward Algorithm will be needed later for HMM training.

The Backward Algorithm is used to calculate a matrix $\boldsymbol{\beta}$ of conditional probabilities of the final part of an observed sequence $(\boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T)$, given an HMM λ and the state $S_t = i$ at time t. The result is known as the *backward variables*, defined through

$$\beta_{i,t} = P(\boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T | S_t = i, \lambda)$$

for an infinite-duration HMM. $\beta_{i,t}$ has a slightly different interpretation for finite-duration HMMs, as explained in Sec. 5.5.

In practice it is numerically preferable to calculate the *scaled backward variables* $\hat{\beta}_{i,t}$ instead, which are proportional to the regular backward variables. The Backward Algorithm calculates these variables in two steps, defined by equations in Sec. 5.5:

Initialization: Eqs. (5.64) and (5.65) define the slightly different initializations needed for infinite-duration and finite-duration HMMs.

Backward Step: Eq. (5.70) applies to any type of HMM.

Implement the Backward Algorithm

In the `PattRecClasses` framework the Backward Algorithm is a method of the `MarkovChain` class, since the algorithm does not need to know anything about the type of output probability distribution used by the HMM. The Backward algorithm takes as input a matrix with values proportional to the state-conditional probability mass or density values for each state and each element in the observed feature sequence, along with a corresponding sequence of scale factors $(c_1 \dots c_T)$ computed by the Forward Algorithm in the previous section.

Your task is to complete the `@MarkovChain/backward` method as specified in the function interface and comments. Because of the object-oriented nature of the HMM implementation, you need not have a working Forward Algorithm in order to write the Backward Algorithm code, as seen below.

Note especially that your function must accept either an *infinite-duration* or a *finite-duration* HMM. The output has exactly the same format in both cases, although the theoretical interpretation of the output is slightly different. Save your finished version of the function under the same file name in the same directory.

Verify the Implementation

One of the most rigorous ways to test your implementation is to do a calculation by hand, and compare the result with the function output. Fortunately for you, this tedious calculation has already been carried out by previous students, and you will just use their results here.

Create a finite-duration test HMM with a Markov chain given by

$$q = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0 & 0.9 & 0.1 \end{pmatrix},$$

where the state-conditional output distribution is a scalar Gaussian with mean $\mu_1 = 0$ and standard deviation $\sigma_1 = 1$ for state 1, and another Gaussian with $\mu_2 = 3$ and $\sigma_2 = 2$ for state 2. Previous calculations show that, for this HMM and the observation sequence $\underline{x} = (-0.2, 2.6, 1.3)$, the Forward Algorithm gives the scale factors $\underline{c} = (1, 0.1625, 0.8266, 0.0581)$. Feeding these results into the Backward algorithm, further computations show that the final scaled backward variables $\hat{\beta}_{j,t}$, $t = 1 \dots 3$ for this simple test example should be about

```
betaHat =
    1.0003    1.0393         0
    8.4182    9.3536    2.0822
```

These numerical results require that all calculations use the scaled state-conditional probability values `pX` as supplied by the `OutputDistr prob`-method (try `help GaussD/prob` for more information). If you used the

accompanying scale factors from `prob` to undo the scaling before calling `backward`, you may get different results. This is not recommended, as the unscaled values may be very small, leading to numerical problems.

Since your implementation also must work for infinite-duration HMMs, it is recommended that you figure out a way to test this case as well.

Your assignment report should include

- A copy of your finished `@MarkovChain/backward` function.

A.4 Code Verification and Signal Database

This assignment consists of two very different parts. In one part you will verify, and possibly correct, code for your MatLab HMM implementation. This is described in Section A.4.1. In the other part you will assemble a database of example signals for training and testing your final pattern recognition system. The details of this database are task dependent: for the word recognition task see Section A.4.2, for melody recognition consult Section A.4.3, and for character recognition turn to Section A.4.4.

In case you finish the two parts of this assignment early, it might be a good idea to use any extra time to start working on your system and presentation for the final assignment. While the final assignment should not be conceptually or mathematically challenging, it is likely to require a lot of "duct-tape" code to perform all necessary training and testing steps, and to provide an interactive demo. It may therefore be a good idea to start the next assignment ahead of time, if possible.

A.4.1 Code Verification

It is easy to make mistakes in calculations and computer code, and it is therefore crucial to always check your work. For the same reason, it is frequently important to check code from other people as well. Studying someone else's code to understand how it operates, and then correcting possible errors, is a very common task for engineers and scientists today.[8]

In practice it is very important to find and eliminate all problems in whatever task you are working on, since engineers often work on projects where errors can have substantial economic consequences, and sometimes even cause bodily harm or death. Producing neat and correct code, and eliminating bugs wherever you may find them, is also a way to take pride in your work.

For the code verification part of the current assignment you will work with either the Forward Algorithm or the Backward Algorithm—whichever you did *not* work with before. Instead of writing your own functions from scratch, however, you are here given code written by someone else. You will have to verify that this code is correct, fix any problems you may find, and hand in your corrected implementation. Your final code will be judged to the same standards as implementations submitted for this task on the previous assignment; the state of the code when you received it does not matter for the grading.

To get started, you should read the appropriate sections in the previous assignment, so you know what the purpose of the code is and how it should

[8] A highly recommended article about the world-leading coding and verification practices at the NASA space shuttle group can be found at `http://www.fastcompany.com/node/28121/print`.

behave. You may then inspect the code you received to make sure it does the right thing, and write tests to verify it works as expected in practice. It is important that you find and point out *all* the errors, if there are any.

Your assignment report should include

- Correct code for the Forward or Backward Algorithm task, based on the code you have been given by the course assistants.
- Everything else that is required for the corresponding original implementation task.
- A brief description of your verification procedure and code for any tests you carried out.
- A list of problems in the code you were given, how you identified them, and how you corrected them. Were they syntax errors, run-time errors, or logical flaws? What kind of consequences could be expected if any of the errors had not been fixed? How problematic would you consider these consequences to be?

A.4.2 Speech Database

To build a speech recognizer it is necessary to have some speech data to train it on. To make things more interesting you are here required to assemble this data yourself. Some of the data will also be used to estimate the performance of your recognition system.

For simplicity, you will only build a recognizer that can distinguish between a handful of different words or phrases spoken in isolation. The training data is then a database of examples of the words or phrases to be recognized. You are free to define the vocabulary yourself. Perhaps you can come up with a few words from a fun and simple application of speech recognition? Be creative!

Most fun is probably to design a word recognizer that understands your own voice. To do this reliably you will need to record some of your own speech. All you need is a PC with a microphone. Sound recording and editing software usually comes with your operating system, but alternatives are given on the project homepage. Contact the project assistant if you have no possibility of recording your own training data.

When recording, try to minimize errors and disturbances in your sound files and record at least fifteen examples of each word or phrase! For simplicity it is probably best if you use on the order of ten words and save your recordings in 22 050 Hz 16 bit mono wav files, one for each word or phrase you say. Listen to each file you generate so that there is no distortion or other problems. To get a less speaker-dependent, and thus more generally

applicable recognizer, it is a good idea to use training data from several different speakers. See if you can get together with a few fellow students and do your recordings together!

Your assignment report should include

- A brief specification of the database you recorded. Which words or phrases did you use and how did you set up the recording process? How many repetitions did you record for each word? What kind of variation is there in the database between different examples of the same word?

A.4.3 Song Database

To build a melody recognizer it is first necessary to have some melody data to train it on. To make things more interesting you are here required to record this data yourself. Some of the data will also be used to estimate the performance of your recognition system.

For simplicity, you will only build a recognizer that can distinguish between a handful of different melody snippets in a quiet background. The training data is then a database with a number of example recordings for each melody to be recognized. You are free to define the song library yourself. Perhaps you can come up with a few fun and simple songs or music snippets to distinguish? Be creative! There is also the option to do "trained singer recognition," as mentioned in assignment A.2.3.

It is probably easiest if you settle for either humming or whistling melodies in your recordings. Playing the melodies on an instrument might also work, but using chords or similar might be a bad idea since the `GetMusicFeatures` function cannot handle polyphonic sounds. Singing is another possibility, but you might get lower accuracy because of the variety of speech sounds that occur in songs, some of which (e.g., "s") may be quite energetic but do not have a well-defined pitch. This is likely to confuse your simple, pitch-based feature extractor.

Most fun is probably to design a recognizer that responds well to your own voice or instrument and performance style. To do this reliably you will need to record some of your own humming, whistling, singing, or playing. All you need is a PC with a microphone. Sound recording and editing software usually comes with your operating system, but alternatives are given on the project homepage. Contact the project assistant if you have no possibility of recording your own training data.

When recording, try to minimize noise and disturbances in your sound files and record at least fifteen examples of each melody snippet! For simplicity it might be best if you use on the order of ten different melodies in your database, or maybe a little less. Save your recordings in 22 050 Hz

16 bit mono wav files, one for each example. Also listen to each file you generate so that there is no distortion or other problems!

It is probably a good idea to keep your snippets short; select a single section of each melody (often from the beginning) that takes about ten seconds or less to perform, and record at least fifteen takes of it. If you vary your style a little in each take you are likely to get a recognizer that is more robust to different performance styles. Similarly, you might get a less style dependent, and thus more generally applicable recognizer, by using training data from several different persons. See if you can get together with a few fellow students and do your recordings together!

Your assignment report should include

- A brief specification of your recording database. Which songs did you choose and why? How are the melodies performed (whistled, hummed, etc.)? Where and how did you record the examples? How many examples of each melody were recorded? What kind of variation is present in different examples of the same melody?

- An example file for each melody in your database. You may compress the audio files you send into mp3 or ogg format, or simply reduce sampling frequency and bit-depth to save space. (Note that zip file compression is notoriously poor for audio files.)

A.4.4 Character Database

To build an on-line character recognition system, it is necessary to have some pen-trace data to train it on. You will here create this database yourself. This data can also be used to estimate the performance of your recognizer.

First, try to think of a fun and interesting application of on-line character recognition. Be creative! Then pick a library of about ten symbols, characters, or glyphs relevant to your chosen application that your system will learn to distinguish between.

For each symbol in your library, you should write it at least fifteen times to build a collection of examples. Store each raw data series generated by `DrawCharacter`, so you can plot any example character later on. You can use a cell array in MatLab (delimited by the characters `{` and `}`) to store matrices of different sizes in the same multidimensional array, and save your data to disk with the command `save`. Check every example so that there are no mistakes.

If you vary your style a little between each example you draw, you are likely to get a recognizer that is more robust to stylistic variation. Similarly, you may get a less handwriting-dependent and more generally applicable recognizer by using training data from several different persons. See if you

can get together with a few fellow students and assemble your databases together!

Even if there is stylistic variation, please make sure, for each character, that you always draw everything in the same basic order and direction. If not, you will generate a database that mixes very different-looking feature sequences. This variation is something simple left-right HMMs cannot describe—it will just confuse these models and make it very difficult for them to learn anything. Since the plan is for you to use standard left-right HMMs for this project, we suggest you avoid these difficulties. If you want a system that can recognize characters drawn backwards or with different stroke orders, you need to design advanced HMMs, with complicated transition matrices that essentially represent mixtures of left-right models.

Your assignment report should include

- A brief specification of your character database. Which characters did you choose and what is the application? How many examples are there of each character? What kind of variation is present between examples?
- Illustrative figures showing an example or two of each character you want to distinguish.

A.5 System Demonstration

Now that you have verified your toolbox, the final project assignment is to use it to implement a classification system for spoken words, hummed, played, or whistled melodies, written characters, or whatever other data you are using. Use the function you worked on in assignment A.2 to extract features. Also create a set of HMMs, one for each class, with a matching output distribution from either the `DiscreteD`, `GaussD`, or `GaussMixD` class, depending on the nature and characteristics of your feature data.

1. Figure out a way to partition your dataset into two parts: a training set and a test set. The training set should be the largest of the two and is used for learning the parameters of the HMMs. The examples in the test set are then used to assess the performance of the resulting recognizer for your particular task. Both sets should contain several examples of every class your system is designed to recognize.

2. Initialize and train your classification system using your training set only, creating one left-right HMM for each class.

3. Apply the resulting classifier to your test data and note the number of misclassifications. This is a form of cross-validation; see Section 4.5.1. To get a more accurate estimate of classifier performance, you can repeat the cross-validation process many times, using different partitionings, and compute the average error rate over all trials.

4. Take a look at the misclassified examples. Sometimes one can find a simple reason why these have been particularly difficult to classify. Other times, this is less clear.

5. Build a simple live demo in MatLab which accepts data from the user, either using `audiorecorder` or by calling `DrawCharacter`, and then shows the classification of the input and plots the log-likelihoods of the various classes for comparison. If the demo involves audio, it should play back each recorded sound, so that one can be sure there is no distortion or other sound issues.

6. Play around with your demo and try to estimate how it performs in practice for your own voice, playing, or handwriting.

7. Prepare to describe and demonstrate your complete classification system at one of the scheduled presentation seminars. Do not forget to register for the seminar in advance!

8. At the presentation you should introduce your problem, describe your system design, report the test set performance, and do a live demonstration of your system. You are encouraged to use your own laptop

for the presentation, if possible. A microphone, speakers, and a video projector will be available. Always bring all the files your require on a USB memory stick or CD, as a backup in case your particular laptop does not work with the projector. If necessary, also bring an appropriate adapter to connect to the VGA cable on the projector.

Practical Hints

If your project involves an audio database, it might be easier to work with the data in MatLab if you use a consistent pattern for storing and naming your recordings. For instance, you could create one sub-directory for each class, and number the example recordings in each folder as "`1.wav`", "`2.wav`", etc. The MatLab command `dir` could potentially also be helpful.

To create your models you can often use the function `MakeLeftRightHMM`, but first you must decide on a suitable *number of states* in your models. This need not be the same for all classes, but it could be. A useful rule of thumb is that each model should have at least one state for each distinct regime you expect in your feature series, including silences for audio. While HMMs are good at describing sharp switches from one segment to another, they assume that the mean output value is constant within each segment (HMM signals are piecewise i.i.d.). Gradual transitions between different behaviours, such as diphthongs, pitch slides, or many pen paths, will likely require more than one state to be modelled well. It is probably a good idea to try several different numbers of states, and try to see what works best in practice.

You also need to think about the HMM output distributions, which should match your features and their behaviour. For continuous feature values, you need to consider if you expect the state-conditional feature distributions to be approximately normal or not. Continuous output distributions that are multimodal (have more than one peak) or skewed (asymmetric) are likely better described by multi-component GMMs. For multidimensional feature vectors, you need to think about whether or not you should model possible correlations between the vector components. In the word recognition task, be sure to use normalized MFCCs, and consider including dynamic features. You may have to experiment with the analysis window length and the number of coefficients to find something that works well.

For the testing, it may be convenient to store all your trained HMM instances in a single array, e.g., as

```
hmms(1)=h1; %HMM for the first class
hmms(2)=h2; %HMM for the next class
%And so on
```

As your `@HMM/logprob` method was designed to work also with an HMM array, you can then obtain probabilities of features `xtest` from a given example with all your HMMs in one single call, simply as

```
lP=logprob(hmms,xtest)
```

When you are done you can save your trained system to disk using the MatLab command `save`.

Solutions to Common Problems

Do not expect everything to work well immediately. It is not uncommon to see some log-probabilities being infinite, or even `NaN` ("not a number"), at first. These problems are often due to problematic features or models, or a combination of features and models that do not work well together, as the EM-algorithm can be quite sensitive to such issues.

If you see unreasonable or undefined probabilities or parameter values, you must work out what the causes are and address them before your demonstration. After all, problem solving is an important part of pattern recognizer development! Some of the most common issues, and possible resolutions, are described below.

If `NaN`s appear at any point during training, this often signifies a big problem somewhere, and may lead to models where some or all parameters are `NaN`. Such models cannot be used at all, and should absolutely be avoided.

`NaN`s during training are particularly common with Gaussian or GMM output distributions. The mathematical basis of the issue is often that the parameters of a state-conditional Gaussian distribution or GMM component may be inferred from a subset of the training data which has virtually no variation, for instance a single sample. If the material used to estimate some particular Gaussian parameters shows no variation, the variance is estimated to be zero. This leads to a division by zero in the Gaussian density function, and causes training to break down.

Sometimes, these issues are related to an overabundance of GMM components, or due to using too many HMM states. If you are using GMMs and experience this problem, you might be able to recover by decreasing the number of GMM components and trying again. The most common cause, however, is inappropriate features with one or more elements that (sometimes or always) take on discrete values with no variation. Such features are just not suitable to model with Gaussian distributions. In theory, it is possible to define advanced models that can describe such features, but this is not something the current `PattRecClasses` code can handle.

If your features are generally scalar and discrete in nature, it is probably advisable to switch to use `DiscreteD` output distributions instead; these have no variances to estimate, and are conceptually a much more appropriate choice for integer features. Otherwise, your best bet is to redesign your features to ensure that there always is some variation present. A simple but somewhat inelegant workaround is to add a bit of random noise to the features: small enough so that the features remain substantially the same in the big picture, but big enough to ensure there always is some variation.

Another issue altogether is that models may train just fine, but produce infinite log-probabilities during testing, and sometimes even `NaN`s. This is typically because data sequences that are deemed impossible under a given model will have probability zero, the logarithm of which is minus infinity.

Numerical problems with very small probabilities in the Forward Algorithm can also lead to a division by zero, and produce `NaN`s. If `NaN`s only occur for out-of-class examples, and you are using *continuous*, not discrete, output distributions, you can probably ignore these values.

Infinitely negative log-probabilities are particularly common with discrete distributions. As discussed in Section 4.8.4, the reason is that, to maximize likelihood, probability mass should only be allocated to those outcomes actually observed during training. Any discrete symbol or event not observed in the training data, or not observed for a particular hidden state, will typically be assigned zero probability mass. If previously unobserved events or combinations of events happen to occur in the validation data, for instance due to errors or noise, the entire data sequences where they occur may be deemed impossible by the model. This is surprisingly common in practice. It may even happen that *all* models assign zero probability to the validation data, which would be highly undesirable.

We see that the maximum likelihood parameter symbol probability estimates produced by EM training are too extreme and overconfident, in a sense. This is very similar to the situation described in Example 8.1. It can also be resolved similarly by using a Bayesian approach, where a prior is introduced for the discrete distribution parameters (symbol probabilities), and the parameter estimates become MAP rather than maximum likelihood.

If used correctly, the prior ensures that all outcomes will be assigned nonzero probability, but that uncommon events still are associated with suitably low probabilities. The influence of the prior depends on how much data that is available: the more applicable training data that is available, the smaller the effect of the prior becomes.

Because the prior effectively acts as fractional observations that are added to all empirical frequency counts, the approach is sometimes known as *pseudocounts*. Since this procedure evens out the differences between big and small probabilities a bit, it is an example of so-called *smoothing*. Various kinds of smoothing are common post-processing steps in many practical applications of pattern recognition.

The `DiscreteD` class has facilities for using pseudocounts in HMM training. It is recommended that you use these in your project if you are using discrete output distributions.

At the project presentation you should

- Have all project and presentation files with you, easily accessible on a USB memory stick, CD, or similar.

- Introduce yourself to the class (maximum one minute per person). What is your Masters and specialization? If you are here on exchange, what is your home university, where is it located, and what do you study there?
- Talk about your application and your data. Which pattern recognition application inspired your choice of examples? Play or display an example or two from your database.
- Briefly describe your feature extraction scheme. Are the features discrete or continuous? Scalars or vectors? Name a number of ways the data can vary, along with the innovations or techniques used by the feature extractor to be more robust against these kinds of variation.
- Talk about your HMM design. How many states did you use, and why? What about the output distributions?
- Outline how you trained and tested your system. In particular, explain how you partitioned the data into training and test sets and describe the reasoning behind your choice.
- Plot or illustrate some example training sequences from one class, and compare these against random output sequences generated by the corresponding trained HMM using `@HMM/rand`. In what ways are they similar, and how do they differ? Discuss what aspects of the data the HMM has learned to describe.
- Report the average classification error of your recognizer over the test set. Also report the error rate for the most commonly misclassified class.
- Play back or show some misclassified instances to illustrate the errors. You may present a *confusion matrix* C, a table with elements c_{ij} showing how often examples from class i were classified as class j.
- Do a live demonstration of how the classifier works with your own voice, playing, or handwriting.
- Present your conclusions: How did the choices you made in the design process affect your classifier? What are the strengths and weaknesses of your system? What have you learned?

Presentations should preferably be at most ten minutes. No written report is necessary for this assignment. However, please be sure send us your presentation slides in an e-mail. Also be prepared to provide the code and data for your classification system, as we may ask for this to check your work in case anything is unclear.

Bibliography

Bishop, C. M. (2006). *Pattern recognition and machine learning.* Springer, New York, NY, USA. p. 173, 191

Davis, S. B. and Mermelstein, P. (1980). Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(4):357–366. p. 220

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Scociety. Series B*, 39(1):1–38. p. 139, 141

Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87. p. 58

Duda, R. O., Hart, P. E., and Stork, D. G. (2000). *Pattern Classification. Second Edition.* John Wiley & Sons, New York. p. 27

Ephraim, Y. and Merhav, N. (2002). Hidden Markov processes. *IEEE Transactions on Information Theory*, 48(6):1518–1569. p. 91

Ephraim, Y. and Roberts, W. J. J. (2005). Revisiting autoregressive hidden Markov modeling of speech signals. *IEEE Signal processing letters*, 12(2):166–169. p. 91

ETSI-ES-201108 (2003). Speech processing, transmission and quality aspects (STQ); distributed speech recognition; front-end feature extraction algorithms; compression algorithms. Technical Report ES-201108 v.1.1.3, European Telecommunications Standards Institute. p. 220

Gersho, A. and Gray, R. (1992). *Vector quantization and signal compression.* Kluwer Academic Publ. p. 95

Jeffreys, H. (1946). An invariant form for the prior probability in estimation problems. *Proceedings of the Royal Society of London Series A – Mathematical and Physical Sciences*, 186(1007):453–461. p. 176

Linde, Y., Buzo, A., and Gray, R. (1980). An algorithm for vector quantizer design. *IEEE Transactions on Communications*, COM-28:84–95. p. 95

Ma, J., Xu, L., and Jordan, M. I. (2000). Asymptotic convergence rate of the EM algorithm for Gaussian mixtures. *Neural Computation*, 12:2881–2907. p. 139

MacKay, D. J. C. (2006). *Information theory, inference, and learning algorithms*. Cambridge University Press. p. 11, 191, 267

Rabiner, L. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77:257–286. p. 91, 133, 159

Rabiner, L. and Juang, B. (1986). An introduction to hidden Markov models. *IEEE ASSP Magazine*, 3(1):4–16. p. 91, 133, 159

Råde, L. and Westergren, B. (1995). *BETA Mathematics handbook for science and engineering*. Studentlitteratur, Lund, Sweden. p. 8, 9, 14, 43, 91, 143, 151, 157, 162, 170

Redner, R. A. and Walker, H. F. (1984). Mixture densities, maximum likelihood and the EM algorithm. *SIAM Review*, 26(2):195–239. p. 149

Wasserman, L. (2000). Asymptotic inference for mixture models by using data-dependent priors. *Journal of the Royal Statistical Society: Series B*, 62(1):159–180. p. 197, 205

Wasserman, L. (2012). Mixture models: the twilight zone of statistics. `http://normaldeviate.wordpress.com/2012/08/04/mixture-models-the-twilight-zone-of-statistics/`. p. 197, 205

Wu, C. F. J. (1983). On the convergence properties of the EM algorithm. *The Annals of Statistics*, 11(1):95–103. p. 139

Young, S., Evermann, G., Kershaw, D., Moore, G., Odell, J., Ollason, D., Povey, D., Valtchev, V., and Woodland, P. (2002). *The HTK book*. Cambridge University Engineering Dept. p. 133, 159

Answers to some Problems

Chapter 1

1.1.a

$$\boldsymbol{\mu}_X = \mathbf{0},\ (K \times 1); \quad C_X = PDP^T,\ (K \times K); \quad \text{with } D = \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix}$$

1.1.b

$$E[\boldsymbol{X}] = \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \quad C_X = \begin{pmatrix} 10 & 8 \\ 8 & 10 \end{pmatrix}$$

1.2.b

$$\lambda_1 = 18, \quad \lambda_2 = 2; \quad \boldsymbol{e}_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}; \quad \boldsymbol{e}_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

1.3.a $\mu_X = \begin{pmatrix} 1 \\ 2 \end{pmatrix}; \quad C_X = \begin{pmatrix} 5 & 4 \\ 4 & 4 \end{pmatrix}$

1.4.b For $K = 1$: two points on the real axis; For $K = 2$: an ellipse; For $K = 3$: an ellipsoid.

1.5 Unit vectors in main directions: $(0.6, 0.8)^T$ and $(-0.8, 0.6)^T$.
Maximum diameter is 4 times the minimum diameter.

1.7

$$w_1 = w_2 = w_3 = w_4 = 0.25$$
$$\mu_1 = \begin{pmatrix} -3 \\ 0 \end{pmatrix}; \quad \mu_2 = \begin{pmatrix} +3 \\ 0 \end{pmatrix}; \quad \mu_3 = \begin{pmatrix} 0 \\ -3 \end{pmatrix}; \quad \mu_4 = \begin{pmatrix} 0 \\ +3 \end{pmatrix};$$
$$C_1 = C_2 = \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix}; \quad C_3 = C_4 = \begin{pmatrix} \sigma_2^2 & 0 \\ 0 & \sigma_1^2 \end{pmatrix};$$

where $\sigma_1 = 1$ and $\sigma_2 = 2$.

1.8

$$f_Y(y) = \frac{1}{\sqrt{2\pi}\sigma} \frac{1}{2\sqrt{y}} e^{-\frac{y+\mu^2}{2\sigma^2}} \left(e^{\frac{\mu\sqrt{y}}{\sigma^2}} + e^{-\frac{\mu\sqrt{y}}{\sigma^2}} \right)$$

Chapter 2

2.2.a

$$f_Z(z) = \frac{1}{\sigma_Z\sqrt{2\pi}} e^{-\frac{(z-\mu_Z)^2}{2\sigma_Z^2}}, \quad \mu_Z = \mu_X + \mu_Y, \quad \sigma_Z^2 = \sigma_X^2 + \sigma_Y^2$$

2.2.b

$$f_{Z|X}(z|x_1) = \frac{1}{\sigma_Y\sqrt{2\pi}} e^{-\frac{(z-x_1-\mu_Y)^2}{2\sigma_Y^2}}$$

2.2.c

$$f_{X|Z}(x|z_1) = \frac{1}{\sigma_{X|Z}\sqrt{2\pi}} e^{-\frac{(x-\mu_{X|Z})^2}{2\sigma_{X|Z}^2}}$$
$$\mu_{X|Z} = \frac{\sigma_X^2(z_1-\mu_Y) + \sigma_Y^2\mu_X}{\sigma_X^2+\sigma_Y^2}$$
$$\sigma_{X|Z}^2 = \frac{\sigma_X^2\sigma_Y^2}{\sigma_X^2+\sigma_Y^2}$$

2.3.a

$$f_{X|S}(x|i) = \begin{cases} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}, & i=0 \\ \frac{1}{2\sqrt{2\pi}} e^{-\frac{(x-1)^2}{8}}, & i=1 \end{cases}$$

2.3.b

$$f_X(x) = \frac{1}{2}\frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} + \frac{1}{2}\frac{1}{2\sqrt{2\pi}} e^{-\frac{(x-1)^2}{8}}$$

2.3.c

$$P_{S|X}(0|x) = \frac{1}{1+\frac{1}{2}e^{\frac{3x^2+2x-1}{8}}}, \quad P_{S|X}(0|0.3) \approx 0.670$$

2.4.a

$$f_{Z|S}(z|i) = \frac{1}{i\sqrt{2\pi}} e^{-\frac{(z-i)^2}{2i^2}}$$

2.4.b

$$f_Z(z) = \frac{0.8}{\sqrt{2\pi}} e^{-\frac{(z-1)^2}{2}} + \frac{0.1}{\sqrt{2\pi}} e^{-\frac{(z-2)^2}{8}}$$

2.4.c

$$P_{S|Z}(1|z_1) = \frac{8}{8+e^{(3z_1^2-4z_1)/8}}$$
$$P_{S|Z}(2|z_1) = 1 - P_{S|Z}(1|z_1) = \frac{1}{1+8e^{(-3z_1^2+4z_1)/8}}$$

2.4.d The conditional probability for X is most easily expressed as a probability *mass* function, as only two discrete values of X are possible, either $X = z_1$ or $X = z_1/2$:

$$\begin{aligned} P_{X|Z}(z_1|z_1) &= \frac{8}{8 + e^{(3z_1^2 - 4z_1)/8}} \\ P_{X|Z}(z_1/2|z_1) &= \frac{1}{1 + 8e^{(-3z_1^2 + 4z_1)/8}} \\ P_{X|Z}(x|z_1) &= 0, \quad \text{otherwise} \end{aligned}$$

This can also be formally expressed as a probability *density* function, using a Dirac δ, as

$$f_{X|Z}(x|z_1) = \frac{8\delta(x - z_1)}{8 + e^{(3z_1^2 - 4z_1)/8}} + \frac{\delta(x - z_1/2)}{1 + 8e^{(-3z_1^2 + 4z_1)/8}}$$

2.5.a

$$P_{S_{13}|S_{12}}(k|2) = \begin{cases} 0.1 & k = 1 \\ 0.8 & k = 2 \\ 0.1 & k = 3 \end{cases}$$

2.5.b

$$P_{S_{13}|S_{11},S_{12}}(k|1,2) = P_{S_{13}|S_{12}}(k|2)$$

2.5.c

$$P_{S_{11}|S_{12}}(k|2) = \begin{cases} 1/9 & k = 1 \\ 8/9 & k = 2 \\ 0 & k = 3 \end{cases}$$

2.5.d

$$P_{S_{11}|S_{10},S_{12}}(k|3,2) = \begin{cases} 1 & k = 1 \\ 0 & k = 2 \\ 0 & k = 3 \end{cases}$$

Chapter 3

3.1 If $P\left[(12345) \mid human\right] = q_1$ and $P\left[(24153) \mid human\right] = q_2$, then the a posteriori probability is

$$P\left[(human;\, computer) \mid (12345;\, 24153\right] = \frac{q_1}{q_1 + q_2}$$

Thus, if $q_1 > q_2$, which is a reasonable assumption, then the optimal decision is $(human;\, computer)$.

3.2.a Assume that $\mu_1 < \mu_2$. Then the optimal ML decision rule is

$$d(x) = \begin{cases} 1, & x < (\mu_1 + \mu_2)/2 \\ 2, & \text{otherwise} \end{cases}$$

3.2.b

$$P\,[error] = 1 - \Phi(\mid \mu_2 - \mu_1 \mid /2\sigma) = 1 - \Phi(2) \approx 0.023$$

3.3.b If $r/c = 0$, then the classifier should always choose "cannot decide", regardless of the observation.

3.3.c If $r \geq c$, then the classifier never chooses "cannot decide", regardless of the observation.

3.3.f

$$P\,[reject] = \Phi(1 + x_0) - \Phi(1 - x_0), \quad x_0 = 0.5 \ln 3 \approx 0.549$$

With these feature distributions, the "reject" desicion is never chosen, if $r/c > 0.5$.

3.4.a

$$f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) = \frac{1}{(2\pi)^{\cdot/2}\sqrt{\det \boldsymbol{C}}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{a}_i)^T \boldsymbol{C}^{-1}(\boldsymbol{x}-\boldsymbol{a}_i)} \quad \boldsymbol{C} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix}$$

3.4.b The probability density is constant on ellipses in the x_1, x_2 plane, with main axes directed along the eigenvectors of $\boldsymbol{C}$. Normalized eigenvectors are

$$e_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

3.4.c Hint: Use transformation $g(\boldsymbol{x}) = g_0(\boldsymbol{x}) - g_1(\boldsymbol{x})$, where

$$g_i(\boldsymbol{x}) = \ln f_{\boldsymbol{X}|S}(\boldsymbol{x} \mid i) P_S(i)$$

3.5.a

$$d(\boldsymbol{x}) = \begin{cases} 1, & g(\boldsymbol{x}) > 0 \\ 2, & g(\boldsymbol{x}) < 0 \end{cases}, \quad \text{where } g(\boldsymbol{x}) = (\boldsymbol{a}_1 - \boldsymbol{a}_2)^T [\boldsymbol{x} - (\boldsymbol{a}_1 + \boldsymbol{a}_2)/2]$$

3.5.b Hint: Determine the conditional probability distributions for the scalar decision variable $Y = g(\boldsymbol{X})$.

3.6.a Use discriminant function $g(k) = k(\ln p - \ln(1-p)) + K(\ln(1-p) + \ln 2)$. This leads to the decision rule:

$$d(k) = \begin{cases} 1, & k > k_{th} \\ 0, & k < k_{th} \end{cases}, \quad \text{where } k_{th} = K \frac{\ln \frac{1}{2(1-p)}}{\ln \frac{p}{1-p}}$$

3.6.b Here $k_{th} = 2.75$, and

$$P\left[error\right] = 0.5P\left[k \geq 3 \mid p = 0.5\right] + 0.5P\left[k \leq 2 \mid p = 0.6\right] \approx 0.409$$

3.8.b

$$\boldsymbol{q} = C_W^{-1}(\boldsymbol{u}_0 - \boldsymbol{u}_1)$$

3.9.a

$$d(x_1, x_2) = \begin{cases} 1, & x_1 > x_2 \\ 2, & x_1 < x_2 \end{cases}$$

3.9.b

$$P\left[correct\right] = \Phi(d'/\sqrt{2})$$

3.10.a Hint: Show that optimal discriminant functions can have the form $g_i(\boldsymbol{x}) = d'x_i$.

3.10.c

$$P\left[correct\right] = \int_{-\infty}^{\infty} \Phi^{M-1}(x) \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(x-d')^2}$$

3.11 Using an observed sequence $\underline{x} = (x_0, \ldots, x_{L-1})$, and a discriminant function

$$g(\underline{x}) = L \ln \frac{\sigma_2}{\sigma_1} + \left(\frac{1}{2\sigma_2^2} - \frac{1}{2\sigma_1^2}\right) \sum_{n=0}^{L-1} x_n^2$$

the optimal decision is to guess state 1, iff $g(\underline{x}) > 0$.

3.13.a We have observed an outcome $\underline{x} = (x_1, \ldots, x_T)$ of the random sequence $\underline{X}$. As usual, we can define preliminary discriminant functions as

$$g_1'(\underline{x}) = \ln \prod_{t=1}^{T} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x_t-\mu_1)^2}{2\sigma^2}} = \sum_{t=1}^{T} -\frac{(x_t-\mu_1)^2}{2\sigma^2} + \text{const.}$$

and similarly for $g_0'()$. Here, $\mu_i = E\left[X_t \mid S = i\right]$ and $\sigma^2 = \text{var}\left[X_t\right]$ have exactly known values given in the problem text. Optimal decisions can be made using a single *linear* discrimininant function giving a scalar decision variable

$$y = g(\underline{x}) = \sigma^2 \left(g_1'(\underline{x}) - g_0'(\underline{x})\right) = \sum_{t=1}^{T} x_t$$

We now consider the observed decision variable as an outcome of a scalar random variable Y. As this variable is a sum of Gaussian random variables, it is Gaussian, with means and variance

$$E\left[Y\right] = \begin{cases} \mu_{y0} = -T, & S = 0 \\ \mu_{y1} = +T, & S = 1 \end{cases}$$
$$\text{var}\left[Y\right] = \sigma_y^2 = T\sigma^2$$

The probability of correct decisions is now

$$P_c = P\left[Y > 0 \mid S = 1\right] P\left[S = 1\right] + P\left[Y < 0 \mid S = 0\right] P\left[S = 0\right]$$

Because of the symmetry of the distributions, and as $P\left[S = 1\right] = P\left[S = 0\right] = 0.5$, the probability can be calculated, for example, as

$$P_c = P\left[Y < 0 \mid S = 0\right] = \int_{-\infty}^{0} \frac{1}{\sigma_y \sqrt{2\pi}} e^{-\frac{(y-\mu_{y0})^2}{2\sigma_y^2}} dy = \\ = \Phi\left(\frac{T}{\sqrt{T}\sigma}\right) = \Phi(1.5) \approx 0.9332$$

Here, $\Phi()$ is the cumulative distribution function of the standardized Gaussian distribution $N(0, 1)$. Thus, the error probability of this *optimal* classifier is $P\left[\text{error}\right] \approx 0.067$.

3.13.b The amateur classifier uses a sub-optimal decision variable

$$Z = \sum_{t=1}^{T} Z_t$$

where

$$Z_t = \begin{cases} 0, & X_t \leq 0 \\ 1, & X_t > 0 \end{cases}$$

As Z is a sum of independent binary random variables, it has a Binomial distribution with probability-mass function

$$P\left[Z = n \mid S = 1\right] = \binom{T}{n} p_1^n (1 - p_1)^{T-n}$$

with the parameter $p_1 = P\left[X_t > 0 \mid S = 1\right]$, and a similar distribution for $S = 0$. For symmetry reasons, we have

$$p_1 = P\left[X_t > 0)\right] \mid S = 1] = P\left[X_t \leq 0 \mid S = 0\right] = \\ = \int_{-\infty}^{0} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu_0)^2}{2\sigma^2}} dx = \Phi\left(\frac{1}{2}\right) \approx 0.6915$$

For symmetry reasons again, we can calculate the error probability, using the cumulative probability for the binomial distribution with $T = 9$, as

$$P\left[\text{error}\right] = \sum_{n=0}^{4} P\left[Z = n \mid S = 1\right] = \sum_{n=0}^{4} \binom{9}{n} p_1^n (1 - p_1)^{9-n} \approx 0.11$$

This approximate value can be interpolated from the cumulative distribution function tabulated in, e.g., *Beta Handbook*, page 439. In this example the sub-optimal *amateur classifier* has nearly twice the error probability, compared to the optimal solution.

Chapter 4

4.1.a Since the probabilities of independent events multiply, the probability of the entire dataset is

$$P\left[\mathcal{D} \mid \mu\right] = \prod_{n=1}^{N} f_X(x_n \mid \mu) = \frac{1}{\mu^N} \exp(-\frac{1}{\mu} \sum_{n=1}^{N} x_n).$$

4.1.b Maximizing the log-likelihood gives

$$\widehat{\mu}_{\mathrm{ML}}(\mathcal{D}) = \frac{1}{N} \sum_{n=1}^{N} x_n$$

which is just the sample mean $\overline{x}$. For this reason, μ is sometimes called the expected lifetime.

4.1.c The MLE is the sample mean

$$\widehat{\mu}_{\mathrm{ML}} = \frac{66\,300}{3} = 22\,100,$$

yielding an expected lifetime of about two and a half years.

4.2.a Similar to the exponential distribution, the MLE is given by the sample mean

$$\widehat{\lambda}_{\mathrm{ML}} = \frac{1}{N} \sum_{n=1}^{N} x_n.$$

There is actually a deep connection between the Poisson distribution and the exponential distribution in the previous problem: if the time between component failures is exponentially distributed with parameter μ, the number of failures in a time interval T is Poisson distributed with parameter $\lambda = \frac{T}{\mu}$.

Note that the exponential distribution and the Poisson distribution both are very simple models with easy-to-compute formulas for the maximum-likelihood parameter estimate. We will see much more advanced and interesting parameter-estimation problems in later chapters.

4.2.b The MLE is simply

$$\widehat{\lambda}_{\mathrm{ML}} = \frac{16}{4} = 4.$$

4.2.c Because we assume that the rate parameter is given without error, the requested probability is simply

$$P\left[X < 2 \;\middle|\; \widehat{\lambda}_{\mathrm{ML}}\right] = \mathrm{Po}(0 \mid 4) + \mathrm{Po}(1 \mid 4) \approx 9.2\%.$$

This approach disregards the fact that the parameter estimate may be quite uncertain. In later chapters we will see more interesting examples of these so-called predictive distributions, which take uncertainty into account.

4.3.a This problem is a straightforward application of Bayes' law. The answer is 90% for the case of two tests being positive, 10% if only one test is positive, and $\frac{1}{730} \approx 0.14\%$ if both tests are clean.

4.3.b This situation is the same as a scenario in which the second test does not exist. Bayes' law shows that the answer is 50% for the case of two tests being positive, and $\frac{1}{82} \approx 1.2\%$ if both tests are clean.

4.3 The practical consequence would be that more innocent people than expected fail the tests, while at the same time more cheaters than we think slip through undetected. Our assumed models would tell us that we can be fairly certain that those who fail both tests are genuine cheaters, when in reality half of those people would be innocent. The main effect is thus a type of overconfidence: we think that the tests (features) give independent and complementary information, but it's really just the same piece of information repeated twice.

To avoid being fooled by the models we make, we have to check them against reality. In this particular example, the fact that a lot fewer athletes than expected would fail only one test can be interpreted as a sign that our model is wrong and needs to be revised. More generally, one of the strengths of proper cross-validation is that it (to an extent) protects against overconfidence in the performance of the models we build.

Chapter 5

5.1.a Left-right, because the transition probability matrix has a left-right structure.

5.1.b $P(\boldsymbol{z}|\lambda) = 0.0069$

5.1.c There are 11 possible state sequences.

5.1.d Most probable state sequence is $(1, 2, 3, 3, 3)$.

5.2.a $P(\boldsymbol{z}|\lambda_1) \approx 0.005704$, and $P(\boldsymbol{z}|\lambda_2) \approx 0.002824$, so the classifier must guess that the observation came from environment 1.

5.2.b

$$P(\lambda_1|\boldsymbol{z}) = \frac{P(\boldsymbol{z}|\lambda_1)}{P(\boldsymbol{z}|\lambda_1) + P(\boldsymbol{z}|\lambda_2)} \approx 0.67$$

5.3.a

$$P(S_{19} = j|x_1x_2 \ldots x_{18}x_{19}, \lambda) \approx \begin{cases} 0.3166, & j = 1 \\ 0.6834, & j = 2 \end{cases}$$

5.3.b

$$P(S_{19} = j | x_1 x_2 \dots x_{18} x_{19} x_{20}, \lambda) = \begin{cases} 0.1505, & j = 1 \\ 0.8495, & j = 2 \end{cases}$$

5.4.a Yes, it is stationary, with state probabilites equal to the indicated initial state probabilites.

5.4.b *Not ergodic*, because the Markov chain is *reducible*. It is not possible to go from states 1 or 2 to states 3 or 4, and vice versa.

5.4.c

$$\boldsymbol{p} = \begin{pmatrix} a \\ a \\ b \\ 2b \end{pmatrix}, \text{ with } \quad 2a + 3b = 1, \quad a \geq 0, \quad b \geq 0.$$

5.5.b Matlab calculates eigenvectors normalized with L_2-norm 1, column-wise, as

$$V = \begin{pmatrix} 0.8944 & -0.7071 & 0 \\ 0 & 0 & 1 \\ 0.4472 & 0.7071 & 0 \end{pmatrix}$$

with corresponding eigenvalues $(1, 0.7, 1)$.

5.5.c Any linear combination of the two eigenvectors is a stationary distribution, i.e., any vector of the following form:

$$\boldsymbol{p} = (1 - b) \begin{pmatrix} 2/3 \\ 0 \\ 1/3 \end{pmatrix} + b \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2(1 - b)/3 \\ b \\ (1 - b)/3 \end{pmatrix}$$

5.5.d The Markov chain is *reducible*, thus not ergodic.

5.6.a A geometric distribution with probability mass

$$f_{D_i}(d) = a_{ii}^{d-1}(1 - a_{ii})$$

5.6.b

$$E[D_i] = \frac{1}{1 - a_{ii}}$$

5.7.a

$$f_D(d) = P(D = d) = \mathbf{1}(\boldsymbol{I} - \boldsymbol{C})\boldsymbol{C}^{d-1}q$$

where $\mathbf{1}$ is a row vector where all N elements are equal to 1, and $\boldsymbol{I}$ is the identity matrix, and we have denoted the transposed transition matrix, without the last column, as the square matrix $\boldsymbol{C}$ with elements $c_{ji} = a_{ij}$ for $1 \leq i, j \leq N$.

5.7.b

$$E\left[D\right] = \mathbf{1}(\boldsymbol{I} - \boldsymbol{C})^{-1}q$$

5.8.a

$$\underline{\hat{w}} = \underset{w_1,\ldots,w_J}{\operatorname{argmax}} \sum_{m=1}^{M} e^{\sum_{j=1}^{J} L_{j,w_j,m}}$$

5.8.b

$$\hat{w}_j = \underset{n}{\operatorname{argmax}} \sum_{m=1}^{M} \frac{e^{L_{jnm}}}{\sum_l e^{L_{jlm}}} p_m$$

where

$$p_m = P(S = m|\underline{\boldsymbol{x}}_1, \ldots, \underline{\boldsymbol{x}}_J) = \frac{\prod_{j=1}^{J}\left(\sum_{w_j=1}^{N} e^{L_{j,w_j,m}}\right)}{\sum_k \prod_{j=1}^{J}\left(\sum_{w_j=1}^{N} e^{L_{j,w_j,k}}\right)}$$

Chapter 6

6.1

$$A = \begin{pmatrix} 0.8 & 0.2 & 0 & 0 & 0 & 0 \\ 0 & 0.9 & 0.1 & 0 & 0 & 0 \\ 0 & 0 & 0.9 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0.9 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0.8 & 0.2 \end{pmatrix}$$

Chapter 7

7.1.a

$$\frac{\partial \boldsymbol{x}^T \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} = (\boldsymbol{A}^T + \boldsymbol{A})\boldsymbol{x}$$

7.1.b

$$\frac{\partial \boldsymbol{x}^T \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{A}} = \boldsymbol{x}\boldsymbol{x}^T$$

7.1.c

$$\frac{\partial \det \boldsymbol{A}}{\partial \boldsymbol{A}} = \begin{pmatrix} C_{11} & C_{12} & \cdots & \cdots \\ C_{21} & \cdots & \cdots & \cdots \\ \vdots & \cdots & C_{ij} & \cdots \\ \vdots & \cdots & \cdots & \cdots \end{pmatrix}$$

where $C_{ij} = (-1)^{i+j} D_{ij}$ is a *cofactor* of $\boldsymbol{A}$, and D_{ij} is the determinant of the sub-matrix obtained by deleting row i and column j of $\boldsymbol{A}$.

7.1.d

$$\frac{\partial \ln \det \boldsymbol{A}}{\partial \boldsymbol{A}} = (\boldsymbol{A}^{-1})^T$$

7.1.e

$$\frac{\partial \ln \det \boldsymbol{A}}{\partial \boldsymbol{A^{-1}}} = -\boldsymbol{A}^T$$

7.2.a

$$\theta' = \frac{\theta^{K_o}(1-\theta)^{N_o-K_o}K_o + 0.5^{N_o-N_e}\theta^{K_e}(1-\theta)^{N_e-K_e}K_e}{\theta^{K_o}(1-\theta)^{N_o-K_o}N_o + 0.5^{N_o-N_e}\theta^{K_e}(1-\theta)^{N_e-K_e}N_e}$$

7.2.b

$$\theta' = \frac{0.5^5 \cdot 2 + 0.5^5 \cdot 4}{0.5^5 \cdot 5 + 0.5^5 \cdot 5} = 0.6$$

After a few iterations $\theta' \to 0.7658$.

7.3 The conditional probability that a disturbance occurred at time t, given the measurement result, is

$$\gamma_1(x_t) = P(S_t = 1 | X_t = x_t, \hat{\mu}) = \frac{d\frac{1}{(\alpha\sigma)\sqrt{2\pi}}e^{-\frac{(x_t-\hat{\mu})^2}{2(\alpha\sigma)^2}}}{(1-d)\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x_t-\hat{\mu})^2}{2\sigma^2}} + d\frac{1}{(\alpha\sigma)\sqrt{2\pi}}e^{-\frac{(x_t-\hat{\mu})^2}{2(\alpha\sigma)^2}}}$$

$$\gamma_0(x_t) = 1 - \gamma_1(x_t)$$

The EM function is

$$\begin{aligned} Q(\hat{\mu}', \hat{\mu}) &= E_{\underline{S}}[\ln P(\underline{x}, \underline{S}|\hat{\mu}')|\underline{x}, \hat{\mu}] = \\ &= \sum_{i_1=0}^{1} \dots \sum_{i_T=0}^{1} P(S_1 = i_1 \dots S_T = i_T | x_1 \dots x_T, \hat{\mu}) \cdot \\ &\quad \cdot \ln P(x_1 \dots x_T, \underline{S} = (i_1 \dots i_T)|\hat{\mu}') \end{aligned}$$

Because of the statistical independence between samples we have

$$\ln P(x_1 \dots x_T, \underline{S} = (i_1 \dots i_T)|\hat{\mu}') = \sum_{t=1}^{T} \ln P(X_t = x_t \cap S_t = i_t|\hat{\mu}')$$

Therefore,

$$
\begin{aligned}
Q(\hat{\mu}', \hat{\mu}) &= E_{\underline{S}}[\ln P(\underline{x}, \underline{S}|\hat{\mu}')|\underline{x}, \hat{\mu}] = \\
&= \sum_{t=1}^{T} \sum_{i_1=0}^{1} \dots \sum_{i_T=0}^{1} P(S_1 = i_1 \dots S_T = i_T|x_1 \dots x_T, \hat{\mu}) \cdot \\
&\quad \cdot \ln P(X_t = x_t \cap S_t = i_t|\hat{\mu}') = \\
&= \sum_{t=1}^{T} \sum_{i_t=0}^{1} P(S_t = i_t|x_t, \hat{\mu}) \ln P(X_t = x_t \cap S_t = i_t|\hat{\mu}') = \\
&= \sum_{t=1}^{T} \gamma_0(x_t) \ln(1-d) \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x_t - \hat{\mu}')^2}{2\sigma^2}} + \gamma_1(x_t) \ln d \frac{1}{(\alpha\sigma)\sqrt{2\pi}} e^{-\frac{(x_t - \hat{\mu}')^2}{2(\alpha\sigma)^2}} = \\
&= \sum_{t=1}^{T} \gamma_0(x_t) \ln \frac{(1-d)}{\sqrt{2\pi}\sigma} - \gamma_0(x_t) \frac{(x_t - \hat{\mu}')^2}{2\sigma^2} + \\
&\quad + \gamma_1(x_t) \ln \frac{d}{\sqrt{2\pi}\alpha\sigma} - \gamma_1(x_t) \frac{(x_t - \hat{\mu}')^2}{2(\alpha\sigma)^2}
\end{aligned}
$$

A necessary condition for maximum is given by

$$
\frac{\partial Q(\hat{\mu}', \hat{\mu})}{\partial \hat{\mu}'} = \sum_{t=1}^{T} \gamma_0(x_t) \frac{(x_t - \hat{\mu}')}{\sigma^2} + \gamma_1(x_t) \frac{(x_t - \hat{\mu}')}{\alpha^2\sigma^2} = 0
$$

This is satisfied for

$$
\begin{aligned}
\sum_{t=1}^{T} \frac{\gamma_0(x_t) x_t}{\sigma^2} + \frac{\gamma_1(x_t) x_t}{\alpha^2\sigma^2} &= \sum_{t=1}^{T} \frac{\gamma_0(x_t)\hat{\mu}'}{\sigma^2} + \frac{\gamma_1(x_t)\hat{\mu}'}{\alpha^2\sigma^2} \\
\hat{\mu}' &= \frac{\sum_{t=1}^{T} (\gamma_0(x_t) + \gamma_1(x_t)/\alpha^2) x_t}{\sum_{t=1}^{T} (\gamma_0(x_t) + \gamma_1(x_t)/\alpha^2)}
\end{aligned}
$$

7.4

$$
\begin{aligned}
c_1' &= \frac{\sum_j \gamma_j x_j}{D \sum_j \gamma_j} \\
c_2' &= \frac{\sum_j (1 - \gamma_j) x_j}{D \sum_j (1 - \gamma_j)}
\end{aligned}
$$

where

$$
\gamma_j = P\left[S_j = 1 \mid x_j, \lambda\right]
$$

The conditional probabilities can be expanded with Bayes' Rule as

$$
\gamma_j = P\left[S_j = 1 \mid x_j, \lambda\right] = \frac{P\left[x_j \mid S_j = 1, \lambda\right] P\left[S_j = 1\right]}{\sum_{i=1}^{2} P\left[x_j \mid S_j = i, \lambda\right] P\left[S_j = i\right]}
$$

7.5

$$
c' = \frac{1}{T} \sum_{t=1}^{T} x_t (\gamma_{1,t} - \gamma_{2,t})
$$

Chapter 8

8.1.a

$$\hat{w}_{ML} = \frac{z}{T} = \frac{1}{T}\sum_{t=1}^{T} x_t$$

8.1.b

$$f_W(w) \propto \frac{1}{w^{0.5}(1-w)^{0.5}}$$

8.2.b

$$P\left[H_0 \mid z_1 = 4, z_2 = 6\right] = 0.579$$
$$P\left[H_1 \mid z_1 = 4, z_2 = 6\right] = 0.421$$

with $T = 10$. One might have thought that the more flexible model H_1 should be able to fit the observations better than the more restricted model H_0. The result that the *simpler* model is more probable, although the data seem to indicate a difference between the two listeners, arises because the Bayesian treatment automatically includes a form of *Occam's razor* effect, and penalizes the more complex model. A deeper discussion of Occam's razor and Bayesian Learning is given by MacKay (2006).

8.3.a

$$\mu_T = \frac{\mu_0 c^2/\sigma_0^2 + \sum_{t=1}^{T} x_t x_{t-1}}{c^2/\sigma_0^2 + \sum_{t=1}^{T} x_{t-1}^2}$$
$$\sigma_T^2 = \frac{c^2}{c^2/\sigma_0^2 + \sum_{t=1}^{T} x_{t-1}^2}$$

To account for the fact that we had no prior knowledge about A we can just let $\sigma_0 \to \infty$ in these expressions.

8.3.b

$$\frac{1}{\sigma_T^2} = \frac{1}{\sigma_{T-1}^2} + \frac{x_{T-1}^2}{c^2}$$
$$\mu_T = \frac{\sigma_T^2}{\sigma_{T-1}^2}\mu_{T-1} + \frac{\sigma_T^2}{c^2} x_T x_{T-1}$$

8.3.c

$$E\left[X_{T+1} \mid \underline{x}\right] = \mu_T x_T$$
$$\text{var}\left[X_{T+1} \mid \underline{x}\right] = \sigma_T^2 x_T^2 + c^2$$

8.4.a The prior density function for $\boldsymbol{W}$ has the Dirichlet form

$$f_{\boldsymbol{W}}(w_1, \dots w_6) \propto \prod_{k=1}^{6} w_k^{a_k(0)-1}$$

specified by the prior hyper-parameter vector $\boldsymbol{a}(0)$ with no previous observations. If we assume that the prior density should be exactly uniform, we may assign non-informative values $a_k(0) = 1$ for all k.

The Likelihood function for the parameter vector and all observations is

$$f_{\boldsymbol{W},\underline{\boldsymbol{X}}}(\boldsymbol{w}, \underline{\boldsymbol{x}}) \propto \left(\prod_{k=1}^{6} w_k^{a_k(0)-1} \right) \prod_{t=1}^{T} \prod_{k=1}^{6} w_k^{x_{k,t}} = \prod_{k=1}^{6} w_k^{a_k(0)-1+\sum_{t=1}^{T} x_{k,t}}$$

We now regard this expression as a function only of the unknown parameters, and identify it as proportional to the posterior density for $\boldsymbol{W}$. We see that this posterior density function can again be expressed in the Dirichlet form as

$$f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\boldsymbol{w} \mid \underline{\boldsymbol{x}}) \propto \prod_{k=1}^{6} w_k^{a_k(T)-1}$$

with the new hyper-parameters

$$a_k(T) = a_k(0) + \sum_{t=1}^{T} x_{k,t} = a_k(0) + N_k$$

Including the normalization factor given in Sec. "Dirichlet Distribution", we can express the exact posterior density as

$$f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\boldsymbol{w} \mid \underline{\boldsymbol{x}}) = \frac{\Gamma(\sum_k a_k(T))}{\prod_k \Gamma(a_k(T))} \prod_{k=1}^{6} w_k^{a_k(T)-1}$$

If we choose the prior hyperparameters as $a_k(0) = 1$ to get a uniform prior density, the posterior density is

$$f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\boldsymbol{w} \mid \underline{\boldsymbol{x}}) = \frac{\Gamma(N+6)}{\prod_k \Gamma(N_k+1)} \prod_{k=1}^{6} w_k^{N_k}$$

where $N = \sum_{k=1}^{6} N_k$.

8.4.b The predictive probability is obtained by integrating over all possible posterior parameter values. Abbreviating the posterior hyperparameter value as $a_k(T) = a_k = N_k + 1$, we can express the probability as

$$P\left[X_{6,T+1} = 1 \mid \underline{\boldsymbol{x}}\right] = \int \cdots \int_0^1 P\left[X_{6,T+1} = 1 \mid \boldsymbol{w}\right] f_{\boldsymbol{W}|\underline{\boldsymbol{X}}}(\boldsymbol{w} \mid \underline{\boldsymbol{x}}) d\boldsymbol{w} =$$

$$= \int \cdots \int_0^1 w_6 \frac{1}{B(\boldsymbol{a})} \prod_{k=1}^{6} w_k^{a_k-1} d\boldsymbol{w} = \int \cdots \int_0^1 \frac{1}{B(\boldsymbol{a})} \prod_{k=1}^{6} w_k^{b_k-1} d\boldsymbol{w}$$

where we have defined $b_k = a_k$ for $k = 1 \ldots 5$ and $b_6 = a_6 + 1$. The multiple integral can be evaluated simply as the Dirichlet normalizing factor, for the hyperparameter set $\boldsymbol{b}$. Thus,

$$P\left[X_{6,T+1} = 1 \mid \underline{\boldsymbol{x}}\right] = \frac{B(\boldsymbol{b})}{B(\boldsymbol{a})} = \\ = \frac{\Gamma(N_1+1)\cdots\Gamma(N_5+1)\Gamma(N_6+2)}{\Gamma(N_1+1)\cdots\Gamma(N_5+1)\Gamma(N_6+1)} \frac{\Gamma(N+6)}{\Gamma(N+7)} = \\ = \frac{\Gamma(N_6+2)}{\Gamma(N_6+1)} \frac{\Gamma(N+6)}{\Gamma(N+7)}$$

with $N = \sum_{k=1}^{6} N_k$. With the given numerical values, using the Matlab `gammaln` function, we obtain

$$P\left[X_{6,T+1} = 1 \mid \underline{\boldsymbol{x}}\right] = \frac{\Gamma(17)}{\Gamma(16)} \frac{\Gamma(66)}{\Gamma(67)} \approx 0.2424$$

which is greater than 1/6, but still slightly less than $15/60 = 0.25$.

8.5.a The posterior density for U, given $X_1 = x_1$, is Gaussian with mean μ_1 and variance σ_1^2:

$$\mu_1 = \frac{\sigma_0^2 x_1 + \sigma^2 \mu_0}{\sigma^2 + \sigma_0^2}$$
$$\sigma_1^2 = \frac{\sigma^2 \sigma_0^2}{\sigma^2 + \sigma_0^2}$$

8.5.b The posterior density for U, given $\underline{x} = (x_1, \ldots, x_N)$, is Gaussian with mean μ_N and variance σ_N^2:

$$\mu_N = \frac{\sigma_0^2 \sum_n x_n + \sigma^2 \mu_0}{N\sigma_0^2 + \sigma^2}$$
$$\sigma_N^2 = \frac{\sigma^2 \sigma_0^2}{\sigma^2 + N\sigma_0^2}$$

8.6.b

$$a_1 = a_0 + x$$
$$b_1 = b_0 + 1$$

8.6.c The Jeffreys prior can be seen as the asymptotic limit of a gamma density with shape $a_0 = 0.5$ and inverse scale $b_0 \to 0$.

8.7 Necessary conditions used in the development of the expected values:

- The density function $f_{\boldsymbol{X}|\boldsymbol{W}}(\boldsymbol{x} \mid \boldsymbol{w})$ and its logarithm must be twice differentiable with regard to all elements of $\boldsymbol{w}$.
- The functions must be "decent" enough to allow differentiation inside the integral.
- The integration range Ω_X must not depend on the parameters $\boldsymbol{w}$.

Chapter 9

9.1.a

$$f_{\Theta|\underline{X}}(\theta \mid \underline{x}) \propto \frac{1}{c} \prod_{t=1}^{T} \left(0.5 \frac{1}{\sqrt{2\pi}} e^{-x_t^2/2} + 0.5 \frac{1}{\sqrt{2\pi}} e^{-(x_t-\theta)^2/2} \right)$$

but this cannot be normalized with regard to θ, as the first exponential term is non-zero and independent of θ.

9.1.c Update equation for the posterior mean of Θ:

$$\mu \leftarrow \frac{\sum_t \gamma_t(\mu) x_t}{1/\sigma_0^2 + \sum_t \gamma_t(\mu)}$$

with responsibility weight factors

$$\gamma_t(\mu) = \frac{e^{-(x_t-\mu)^2/2}}{e^{-x_t^2/2} + e^{-(x_t-\mu)^2/2}}$$

9.2.a Summary of VI algorithm update equations for hyper-parameters:
For posterior A with Gaussian density $q_A(a)$:

$$\eta_t = \frac{1-\gamma_t}{c^2} + \frac{\gamma_t}{c^2+d^2}$$

$$\frac{1}{\sigma_T^2} = \frac{1}{\sigma_0^2} + \sum_{t=1}^{T} \eta_t x_{t-1}^2$$

$$\mu_T = \frac{\sum_t \eta_t x_t x_{t-1}}{\frac{1}{\sigma_0^2} + \sum_t \eta_t x_{t-1}^2}$$

For posterior W with beta density $q_W(w]$:

$$\alpha_T = \alpha_0 + \sum_{t=1}^{T} \gamma_t$$

$$\beta_T = \beta_0 + \sum_{t=1}^{T} (1-\gamma_t)$$

For binary Z_t with probability mass $q_{Z_t}(z_t)$ and mean $E[Z_t] = \gamma_t$:

$$
\begin{aligned}
y_t^2 &= x_t^2 - 2\mu_T x_t x_{t-1} + (\mu_T^2 + \sigma_T^2) x_{t-1}^2 \\
r_{1,t} &= \psi(\alpha_T) - \ln\sqrt{c^2 + d^2} - \frac{y_t^2}{2(c^2 + d^2)} \\
r_{0,t} &= \psi(\beta_T) - \ln c - \frac{y_t^2}{2c^2} \\
\gamma_t &= \frac{e^{r_{1,t}}}{e^{r_{0,t}} + e^{r_{1,t}}}
\end{aligned}
$$

9.2.b

$$
\begin{aligned}
E[X_{T+1} \mid \underline{x}] &= \mu_T x_T \\
\text{var}[X_{T+1} \mid \underline{x}] &= \sigma_T^2 x_T^2 + c^2 + d^2 \alpha_T / \beta_T
\end{aligned}
$$

where $E[Z_{T+1}] = E_{q_W}[W] = \alpha_T/\beta_T$, as determined by the final posterior beta distribution for W.

Index

applications, 216
auto-regressive process, 186, 205

backward algorithm, 110, 239
backward variable, 112
Baum-Welch, 127, 153
Bayes rule, 21, 31, 45
Bayesian Learning
    approximate, 191
Bayesian learning, 79, 84, 167
    classification, 185
Bayesian model comparison, 185
Bernoulli distribution, 63
beta distribution, 170, 181
bias-variance trade-off, 81
big data, 81

categorical distribution, 69
character recognition, 232
conditional probability, 21
conditional risk, 32
conjugate distribution, 171, 173, 181
Cramér-Rao bound, 81, 177
cross-validation, 71, 77, 247
curse of dimensionality, 73

data synthesis, 62, 72, 229
decision, 38, 42
decision criterion, 37, 40
decision function, 4
decision region, 35
digamma function, 182
dimensionality reduction, 59
Dirichlet distribution, 182
discriminant function, 5, 28, 34, 37, 41
domain adaptation, 82

eigenvalue, 101
eigenvector, 101
EM, 139, 142, 200
ensemble methods, 72, 81
entropy
    relative, 194
ergodic, 97, 101
exit state, 131
Expectation Maximization, 139, 200
Expectation Propagation, 191

feature, 28
    discrete, 45, 63, 64
    dynamic, 222
    extraction, 59, 215
    Gaussian scalar, 37
    Gaussian vector, 40, 43, 64
    GMM, 64
Fisher information, 177, 178, 189
forward algorithm, 105, 237
    numerical problems, 237, 250
forward scale factor, 107
forward variable, 107
    scaled, 107
Fourier analysis, 59, 217, 224

gamma distribution, 179, 182
gamma function, 182
Gaussian, 5, 37, 64, 149
Gaussian Mixture, 11, 64, 94, 149, 159
    tied, 94, 166

training, 149
geometric methods, 62
GMM, 11, 78, 94, 149, 159
numerical problems, 249
tied, 94, 165

hapax, 84
HMM, 89, 127
aperiodic, 104
design, 246, 248
ergodic, 97, 101
finite, 97, 99, 107, 111, 112, 128, 130, 131
infinite, 97, 102, 107, 112, 128
irreducible, 102
left-right, 97, 99
periodic, 102
reducible, 102
standard problems, 209
stationary, 100, 101
training, 127, 153
hyper-parameters, 172

improper prior, 175
independent, 7, 40

Jensen's inequality, 195

Kullback-Leibler divergence, 194

Lagrange multiplier, 69
Laplace approximation, 205
learning
Bayesian, 167
likelihood, 66, 169
linear, 44

Mahalanobis distance, 43
MAP
decision, 33
estimate, 70, 250
Markov model, 89
maximum likelihood
decision, 34
parameter estimation, 66
training, 167
MCE, 75
mel frequency cepstrum coefficients, 216, 220
MFCC, 216, 220
minimum risk, 32
ML decision, 34
MLE, 66, 78
MMI, 76
Monte Carlo methods, 191
Moravec's paradox, 60

Occam's razor, 79
outlier, 84, 230
overfitting, 11, 77, 167
overtraining, 77, 136

parameter, 11, 62, 167
estimation, 65
PCA, 10, 59
Poisson distribution, 64, 188
posterior density, 170, 173
predictive density, 173
principal-component analysis, 10
prior density, 172, 173
improper, 175
Jeffreys, 178
objective, 174, 176
proper, 175
subjective, 173
uniform, 169, 171
probability
a posteriori, 22, 31
a priori, 22, 29
correct, 39, 42
density, 7, 8
error, 37, 40
initial, 93
numerical problems, 239, 241
observation, 93–95
transition, 93
proper density, 175

query by humming, 223

robust classification, 84

sampling, 191
scale parameter, 179, 183
shape parameter, 183
smoothing, 84, 250
speaker adaptation, 82
stationary, 101
statistical model
    discriminative, 62, 75
    generative, 62, 72, 172
Student's t-distribution, 84
subspace, 10
support vector machine, 76, 78
SVM, 76, 78

tied, 94, 165
training, 65
    Bayesian, 167
    discriminative, 75, 78
    GMM, 149, 159
    HMM, 127, 153
    maximum likelihood, 66, 167
    maximum mutual information, 76
    minimum classification error, 75
training data, 171
    notation, 65, 129, 153
transition, 93

variational inference, 191
    factorized approximation, 195
vector quantizer, 95
VI, 191
Viterbi, 115
Voronoi, 95
VQ, 95