Machine Learning and Data Mining
Lecture Notes

CSC 411/D11

Computer Science Department
University of Toronto

Version: February 6, 2012

Contents

Conventions and Notation

Scalars are written with lower-case italics, e.g., x. Column-vectors are written in bold, lower-case: $\mathbf{x}$, and matrices are written in bold uppercase: $\mathbf{B}$.

The set of real numbers is represented by $\mathbb{R}$; N-dimensional Euclidean space is written $\mathbb{R}^N$.

> *Aside:*
> Text in "aside" boxes provide extra background or information that you are not required to know for this course.

Acknowledgements

Graham Taylor and James Martens assisted with preparation of these notes.

1 Introduction to Machine Learning

Machine learning is a set of tools that, broadly speaking, allow us to "teach" computers how to perform tasks by providing examples of how they should be done. For example, suppose we wish to write a program to distinguish between valid email messages and unwanted spam. We could try to write a set of simple rules, for example, flagging messages that contain certain features (such as the word "viagra" or obviously-fake headers). However, writing rules to accurately distinguish which text is valid can actually be quite difficult to do well, resulting either in many missed spam messages, or, worse, many lost emails. Worse, the spammers will actively adjust the way they send spam in order to trick these strategies (e.g., writing "vi@gr@"). Writing effective rules — and keeping them up-to-date — quickly becomes an insurmountable task. Fortunately, machine learning has provided a solution. Modern spam filters are "learned" from examples: we provide the learning algorithm with example emails which we have manually labeled as "ham" (valid email) or "spam" (unwanted email), and the algorithms learn to distinguish between them automatically.

Machine learning is a diverse and exciting field, and there are multiple ways of defining it:

1. **The Artifical Intelligence View.** Learning is central to human knowledge and intelligence, and, likewise, it is also essential for building intelligent machines. Years of effort in AI has shown that trying to build intelligent computers by programming all the rules cannot be done; automatic learning is crucial. For example, we humans are not born with the ability to understand language — we learn it — and it makes sense to try to have computers learn language instead of trying to program it all it.

2. **The Software Engineering View.** Machine learning allows us to program computers by example, which can be easier than writing code the traditional way.

3. **The Stats View.** Machine learning is the marriage of computer science and statistics: computational techniques are applied to statistical problems. Machine learning has been applied to a vast number of problems in many contexts, beyond the typical statistics problems. Machine learning is often designed with different considerations than statistics (e.g., speed is often more important than accuracy).

Often, machine learning methods are broken into two phases:

1. **Training:** A model is learned from a collection of **training data**.

2. **Application:** The model is used to make decisions about some new **test data**.

For example, in the spam filtering case, the training data constitutes email messages labeled as ham or spam, and each new email message that we receive (and which to classify) is test data. However, there are other ways in which machine learning is used as well.

1.1 Types of Machine Learning

Some of the main types of machine learning are:

1. **Supervised Learning,** in which the training data is labeled with the correct answers, e.g., "spam" or "ham." The two most common types of supervised learning are **classification** (where the outputs are discrete labels, as in spam filtering) and **regression** (where the outputs are real-valued).

2. **Unsupervised learning,** in which we are given a collection of unlabeled data, which we wish to analyze and discover patterns within. The two most important examples are **dimension reduction** and **clustering**.

3. **Reinforcement learning,** in which an agent (e.g., a robot or controller) seeks to learn the optimal actions to take based the outcomes of past actions.

There are many other types of machine learning as well, for example:

1. **Semi-supervised learning**, in which only a subset of the training data is labeled

2. **Time-series forecasting**, such as in financial markets

3. **Anomaly detection** such as used for fault-detection in factories and in surveillance

4. **Active learning**, in which obtaining data is expensive, and so an algorithm must determine which training data to acquire

and many others.

1.2 A simple problem

Figure 1 shows a 1D regression problem. The goal is to fit a 1D curve to a few points. Which curve is best to fit these points? There are infinitely many curves that fit the data, and, because the data might be noisy, we might not even want to fit the data precisely. Hence, machine learning requires that we make certain choices:

1. How do we parameterize the model we fit? For the example in Figure 1, how do we parameterize the curve; should we try to explain the data with a linear function, a quadratic, or a sinusoidal curve?

2. What criteria (e.g., objective function) do we use to judge the quality of the fit? For example, when fitting a curve to noisy data, it is common to measure the quality of the fit in terms of the squared error between the data we are given and the fitted curve. When minimizing the squared error, the resulting fit is usually called a least-squares estimate.

3. Some types of models and some model parameters can be very expensive to optimize well. How long are we willing to wait for a solution, or can we use approximations (or hand-tuning) instead?

4. Ideally we want to find a model that will provide useful predictions in future situations. That is, although we might learn a model from *training data*, we ultimately care about how well it works on future *test data*. When a model fits training data well, but performs poorly on test data, we say that the model has *overfit* the training data; i.e., the model has fit properties of the input that are not particularly relevant to the task at hand (e.g., Figures 1 (top row and bottom left)). Such properties are refered to as *noise*. When this happens we say that the model does not *generalize* well to the test data. Rather it produces predictions on the test data that are much less accurate than you might have hoped for given the fit to the training data.

Machine learning provides a wide selection of options by which to answer these questions, along with the vast experience of the community as to which methods tend to be successful on a particular class of data-set. Some more advanced methods provide ways of automating some of these choices, such as automatically selecting between alternative models, and there is some beautiful theory that assists in gaining a deeper understanding of learning. In practice, there is no single "silver bullet" for all learning. Using machine learning in practice requires that you make use of your own prior knowledge and experimentation to solve problems. But with the tools of machine learning, you can do amazing things!

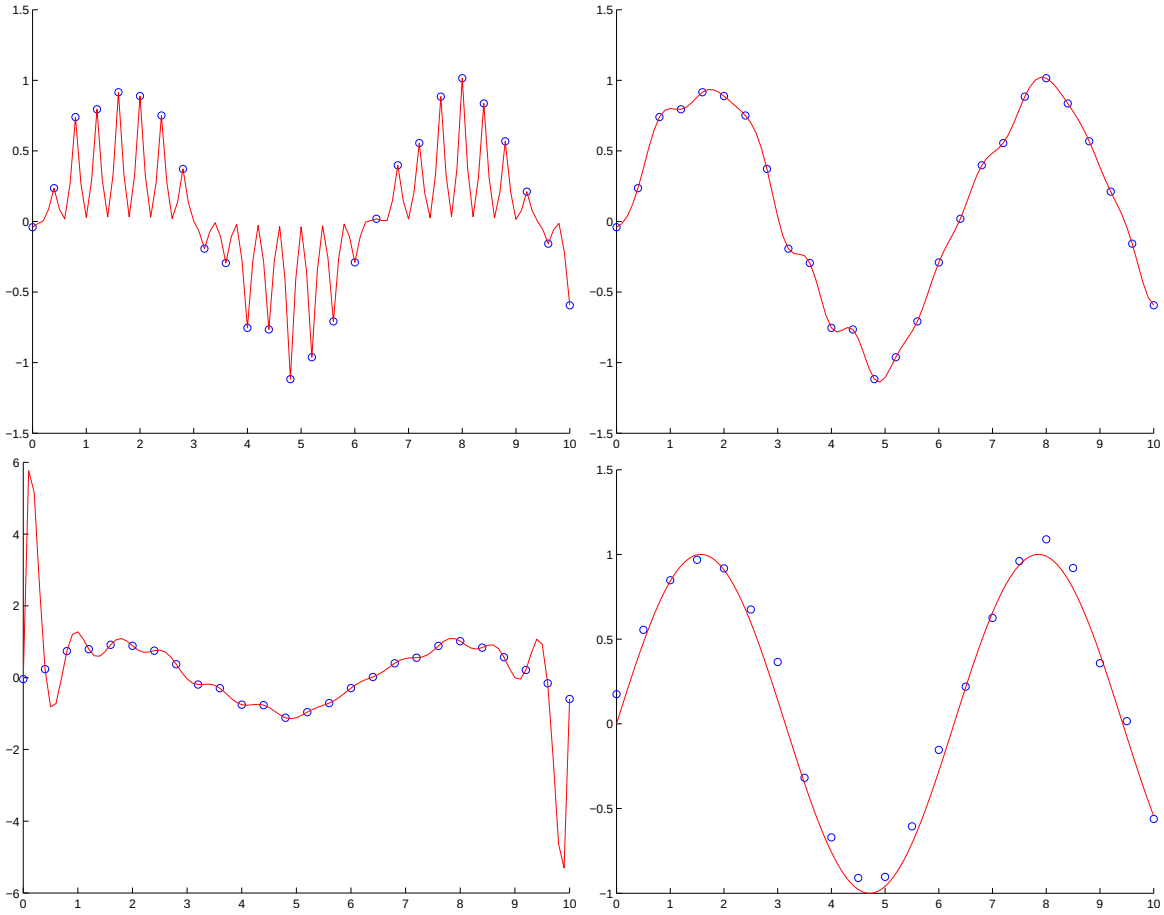

Figure 1: A simple regression problem. The blue circles are measurements (the training data), and the red curves are possible fits to the data. There is no one "right answer;" the solution we prefer depends on the problem. Ideally we want to find a model that provides good predictions for new inputs (i.e., locations on the x-axis for which we had no training data). We will often prefer simple, smooth models like that in the lower right.

2 Linear Regression

In regression, our goal is to learn a mapping from one real-valued space to another. Linear regression is the simplest form of regression: it is easy to understand, often quite effective, and very efficient to learn and use.

2.1 The 1D case

We will start by considering linear regression in just 1 dimension. Here, our goal is to learn a mapping $y = f(x)$, where x and y are both real-valued scalars (i.e., $x \in \mathbb{R}, y \in \mathbb{R}$). We will take f to be an linear function of the form:

$$y = wx + b \tag{1}$$

where w is a *weight* and b is a *bias*. These two scalars are the parameters of the model, which we would like to learn from training data. n particular, we wish to estimate w and b from the N training pairs $\{(x_i, y_i)\}_{i=1}^N$. Then, once we have values for w and b, we can compute the y for a new x.

Given 2 data points (i.e., N=2), we can exactly solve for the unknown slope w and offset b. (How would you formulate this solution?) Unfortunately, this approach is extremely sensitive to noise in the training data measurements, so you cannot usually trust the resulting model. Instead, we can find much better models when the two parameters are estimated from larger data sets. When $N > 2$ we will not be able to find unique parameter values for which $y_i = wx_i + b$ for all i, since we have many more constraints than parameters. The best we can hope for is to find the parameters that minimize the residual errors, i.e., $y_i - (wx_i + b)$.

The most commonly-used way to estimate the parameters is by *least-squares regression*. We define an energy function (a.k.a. objective function):

$$E(w, b) = \sum_{i=1}^{N} (y_i - (wx_i + b))^2 \tag{2}$$

To estimate w and b, we solve for the w and b that minimize this objective function. This can be done by setting the derivatives to zero and solving.

$$\frac{dE}{db} = -2 \sum_i (y_i - (wx_i + b)) = 0 \tag{3}$$

Solving for b gives us the estimate:

$$b^* = \frac{\sum_i y_i}{N} - w \frac{\sum_i x_i}{N} \tag{4}$$

$$= \bar{y} - w\bar{x} \tag{5}$$

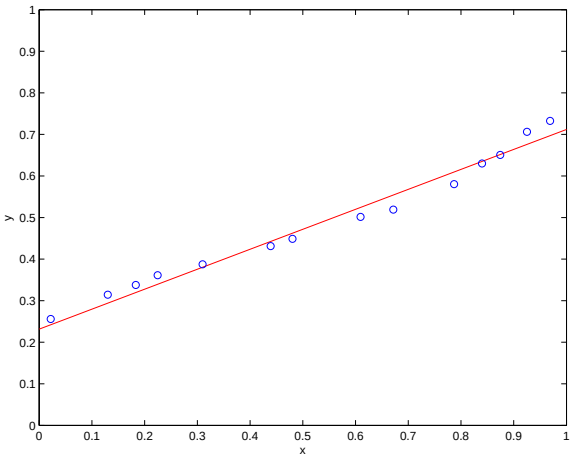

Figure 2: An example of linear regression: the red line is fit to the blue data points.

where we define $\bar{x}$ and $\bar{y}$ as the averages of the x's and y's, respectively. This equation for b^* still depends on w, but we can nevertheless substitute it back into the energy function:

$$E(w, b) = \sum_i ((y_i - \bar{y}) - w(x_i - \bar{x}))^2 \tag{6}$$

Then:

$$\frac{dE}{dw} = -2 \sum_i ((y_i - \bar{y}) - w(x_i - \bar{x}))(x_i - \bar{x}) \tag{7}$$

Solving $\frac{dE}{dw} = 0$ then gives:

$$w^* = \frac{\sum_i (y_i - \bar{y})(x_i - \bar{x})}{\sum_i (x_i - \bar{x})^2} \tag{8}$$

The values w^* and b^* are the least-squares estimates for the parameters of the linear regression.

2.2 Multidimensional inputs

Now, suppose we wish to learn a mapping from D-dimensional inputs to scalar outputs: $\mathbf{x} \in \mathbb{R}^D$, $y \in \mathbb{R}$. Now, we will learn a vector of weights $\mathbf{w}$, so that the mapping will be:[1]

$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b = \sum_{j=1}^{D} w_j x_j + b \quad . \tag{9}$$

[1] Above we used subscripts to index the training set, while here we are using the subscript to index the elements of the input and weight vectors. In what follows the context should make it clear what the index denotes.

For convenience, we can fold the bias b into the weights, if we augment the inputs with an additional 1. In other words, if we define

$$\tilde{\mathbf{w}} = \begin{bmatrix} w_1 \\ \vdots \\ w_D \\ b \end{bmatrix}, \quad \tilde{\mathbf{x}} = \begin{bmatrix} x_1 \\ \vdots \\ x_D \\ 1 \end{bmatrix} \tag{10}$$

then the mapping can be written:

$$f(\mathbf{x}) = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}} \,. \tag{11}$$

Given N training input-output pairs, the least-squares objective function is then:

$$E(\tilde{\mathbf{w}}) = \sum_{i=1}^{N} (y_i - \tilde{\mathbf{w}}^T \tilde{\mathbf{x}}_i)^2 \tag{12}$$

If we stack the outputs in a vector and the inputs in a matrix, then we can also write this as:

$$E(\tilde{\mathbf{w}}) = ||\mathbf{y} - \tilde{\mathbf{X}}\tilde{\mathbf{w}}||^2 \tag{13}$$

where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}, \quad \tilde{\mathbf{X}} = \begin{bmatrix} \mathbf{x}_1^T & 1 \\ \vdots & \\ \mathbf{x}_N^T & 1 \end{bmatrix} \tag{14}$$

and $||\cdot||$ is the usual Euclidean norm, i.e., $||\mathbf{v}||^2 = \sum_i v_i^2$. (You should verify for yourself that Equations 12 and 13 are equivalent).

Equation 13 is known as a linear least-squares problem, and can be solved by methods from linear algebra. We can rewrite the objective function as:

$$\begin{aligned} E(\mathbf{w}) &= (\mathbf{y} - \tilde{\mathbf{X}}\tilde{\mathbf{w}})^T (\mathbf{y} - \tilde{\mathbf{X}}\tilde{\mathbf{w}}) && (15) \\ &= \tilde{\mathbf{w}}^T \tilde{\mathbf{X}}^T \tilde{\mathbf{X}} \tilde{\mathbf{w}} - 2\mathbf{y}^T \tilde{\mathbf{X}} \tilde{\mathbf{w}} + \mathbf{y}^T \mathbf{y} && (16) \end{aligned}$$

We can optimize this by setting all values of $dE/dw_i = 0$ and solving the resulting system of equations (we will cover this in more detail later in Chapter 4). In the meantime, if this is unclear, start by reviewing your linear algebra and vector calculus). The solution is given by:

$$\mathbf{w}^* = (\tilde{\mathbf{X}}^T \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^T \mathbf{y} \tag{17}$$

(You may wish to verify for yourself that this reduces to the solution for the 1D case in Section 2.1; however, this takes quite a lot of linear algebra and a little cleverness). The matrix $\tilde{\mathbf{X}}^+ \equiv (\tilde{\mathbf{X}}^T \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^T$ is called the *pseudoinverse* of $\tilde{\mathbf{X}}$, and so the solution can also be written:

$$\tilde{\mathbf{w}}^* = \tilde{\mathbf{X}}^+ \mathbf{y} \tag{18}$$

In MATLAB, one can directly solve the system of equations using the slash operator:

$$\tilde{\mathbf{w}}^* = \tilde{\mathbf{X}} \backslash \mathbf{y} \tag{19}$$

There are some subtle differences between these two ways of solving the system of equations. We will not concern ourselves with these here except to say that I recommend using the slash operator rather than the pseudoinverse.

2.3 Multidimensional outputs

In the most general case, both the inputs and outputs may be multidimensional. For example, with D-dimensional inputs, and K-dimensional outputs $\mathbf{y} \in \mathbb{R}^K$, a linear mapping from input to output can be written as

$$\mathbf{y} = \tilde{\mathbf{W}}^T \tilde{\mathbf{x}} \tag{20}$$

where $\tilde{\mathbf{W}} \in \mathbb{R}^{(D+1)\times K}$. It is convenient to express $\tilde{\mathbf{W}}$ in terms of its column vectors, i.e.,

$$\tilde{\mathbf{W}} = [\tilde{\mathbf{w}}_1 \ \ldots \ \tilde{\mathbf{w}}_K] \equiv \begin{bmatrix} \mathbf{w}_1 & \ldots & \mathbf{w}_K \\ b_1 & \ldots & b_K \end{bmatrix} . \tag{21}$$

In this way we can then express the mapping from the input $\tilde{\mathbf{x}}$ to the j_{th} element of $\mathbf{y}$ as $y_j = \tilde{\mathbf{w}}_j^T \mathbf{x}$. Now, given N training samples, denoted $\{\tilde{\mathbf{x}}_i, \mathbf{y}_i\}_{i=1}^N$ a natural energy function to minimize in order to estimate $\tilde{\mathbf{W}}$ is just the squared residual error over all training samples and all output dimensions, i.e.,

$$E(\tilde{\mathbf{W}}) = \sum_{i=1}^{N} \sum_{j=1}^{K} (y_{i,j} - \tilde{\mathbf{w}}_j^T \tilde{\mathbf{x}}_i)^2 . \tag{22}$$

There are several ways to conveniently *vectorize* this energy function. One way is to express E solely as a sum over output dimensions. That is, let $\mathbf{y}'_j$ be the N-dimensional vector comprising the j^{th} component of each output training vector, i.e., $\mathbf{y}'_j = [y_{1,j}, y_{2,j}, ..., y_{N,j}]^T$. Then we can write

$$E(\tilde{\mathbf{W}}) = \sum_{j=1}^{K} ||\mathbf{y}'_j - \tilde{\mathbf{X}} \tilde{\mathbf{w}}_j||^2 \tag{23}$$

where $\tilde{\mathbf{X}}^T = [\tilde{\mathbf{x}}_1 \ \tilde{\mathbf{x}}_2 \ \ldots \ \tilde{\mathbf{x}}_N]$. With a little thought you can see that this really amounts to K distinct estimation problems, the solutions for which are given by $\tilde{\mathbf{w}}_j^* = \tilde{\mathbf{X}}^+ \mathbf{y}'_j$.

Another common convention is to stack up everything into a matrix equation, i.e.,

$$E(\tilde{\mathbf{W}}) = ||\mathbf{Y} - \tilde{\mathbf{X}} \tilde{\mathbf{W}}||_F^2 \tag{24}$$

where $\mathbf{Y} = [\mathbf{y}'_1 \ \ldots \mathbf{y}'_K]$, and $|| \cdot ||_F$ denotes the Frobenius norm: $||\mathbf{Y}||_F^2 = \sum_{i,j} \mathbf{Y}_{i,j}^2$. You should verify that Equations (23) and (24) are equivalent representations of the energy function in Equation (22). Finally, the solution is again provided by the pseudoinverse:

$$\tilde{\mathbf{W}}^* = \tilde{\mathbf{X}}^+ \mathbf{Y} \tag{25}$$

or, in MATLAB, $\tilde{\mathbf{W}}^* = \tilde{\mathbf{X}} \backslash \mathbf{Y}$.

3 Nonlinear Regression

Sometimes linear models are not sufficient to capture the real-world phenomena, and thus nonlinear models are necessary. In regression, all such models will have the same basic form, i.e.,

$$\mathbf{y} = f(\mathbf{x}) \tag{26}$$

In linear regression, we have $f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$; the parameters $\mathbf{W}$ and $\mathbf{b}$ must be fit to data.

What nonlinear function do we choose? In principle, $f(\mathbf{x})$ could be anything: it could involve linear functions, sines and cosines, summations, and so on. However, the form we choose will make a big difference on the effectiveness of the regression: a more general model will require more data to fit, and different models are more appropriate for different problems. Ideally, the form of the model would be matched exactly to the underlying phenomenon. If we're modeling a linear process, we'd use a linear regression; if we were modeling a physical process, we could, in principle, model $f(\mathbf{x})$ by the equations of physics.

In many situations, we do not know much about the underlying nature of the process being modeled, or else modeling it precisely is too difficult. In these cases, we typically turn to a few models in machine learning that are widely-used and quite effective for many problems. These methods include basis function regression (including Radial Basis Functions), Artificial Neural Networks, and k-Nearest Neighbors.

There is one other important choice to be made, namely, the choice of objective function for learning, or, equivalently, the underlying noise model. In this section we extend the LS estimators introduced in the previous chapter to include one or more terms to encourage smoothness in the estimated models. It is hoped that smoother models will tend to overfit the training data less and therefore generalize somewhat better.

3.1 Basis function regression

A common choice for the function $f(\mathbf{x})$ is a basis function representation[2]:

$$y = f(x) = \sum_k w_k b_k(x) \tag{27}$$

for the 1D case. The functions $b_k(x)$ are called basis functions. Often it will be convenient to express this model in vector form, for which we define $\mathbf{b}(x) = [b_1(x), \ldots, b_M(x)]^T$ and $\mathbf{w} = [w_1, \ldots, w_M]^T$ where M is the number of basis functions. We can then rewrite the model as

$$y = f(x) = \mathbf{b}(x)^T \mathbf{w} \tag{28}$$

Two common choices of basis functions are **polynomials** and **Radial Basis Functions (RBF)**. A simple, common basis for polynomials are the **monomials**, i.e.,

$$b_0(x) = 1, \quad b_1(x) = x, \quad b_2(x) = x^2, \quad b_3(x) = x^3, \quad \ldots \tag{29}$$

[2]In the machine learning and statistics literature, these representations are often referred to as linear regression, since they are linear functions of the "features" $b_k(x)$

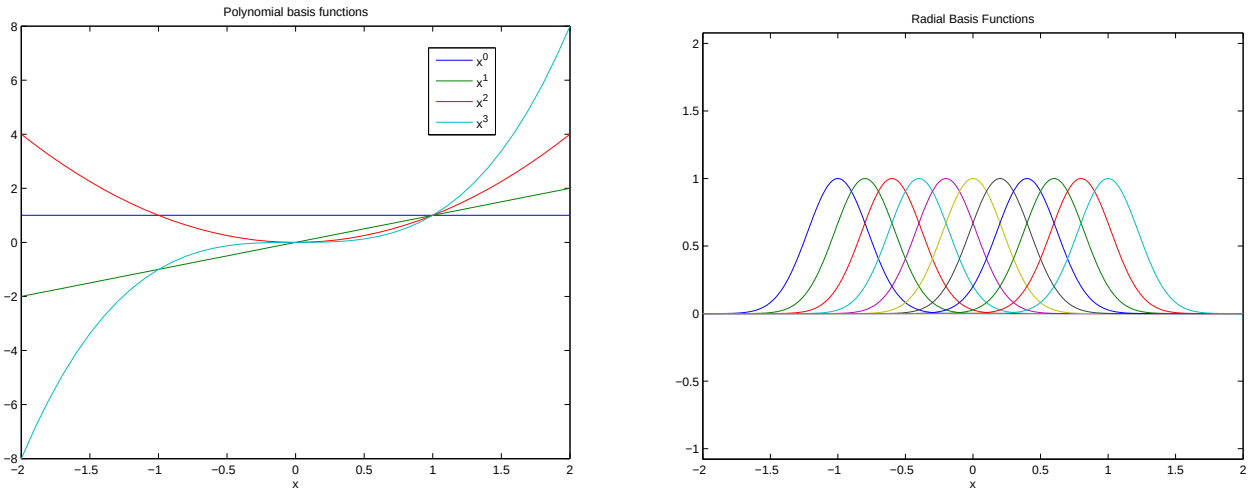

Figure 3: The first three basis functions of a polynomial basis, and Radial Basis Functions

With a monomial basis, the regression model has the form

$$f(x) = \sum w_k x^k \ , \tag{30}$$

Radial Basis Functions, and the resulting regression model are given by

$$b_k(x) = e^{-\frac{(x-c_k)^2}{2\sigma^2}} \ , \tag{31}$$

$$f(x) = \sum w_k e^{-\frac{(x-c_k)^2}{2\sigma^2}} \ , \tag{32}$$

where c_k is the **center** (i.e., the location) of the basis function and σ^2 determines the **width** of the basis function. Both of these are parameters of the model that must be determined somehow.

In practice there are many other possible choices for basis functions, including sinusoidal functions, and other types of polynomials. Also, basis functions from different families, such as monomials and RBFs, can be combined. We might, for example, form a basis using the first few polynomials and a collection of RBFs. In general we ideally want to choose a family of basis functions such that we get a good fit to the data with a small basis set so that the number of weights to be estimated is not too large.

To fit these models, we can again use least-squares regression, by minimizing the sum of squared residual error between model predictions and the training data outputs:

$$E(\mathbf{w}) = \sum_i (y_i - f(x_i))^2 = \sum_i \left(y_i - \sum_k w_k b_k(x) \right)^2 \tag{33}$$

To minimize this function with respect to $\mathbf{w}$, we note that this objective function has the same form as that for linear regression in the previous chapter, except that the inputs are now the $b_k(x)$ values.

In particular, E is still quadratic in the weights $\mathbf{w}$, and hence the weights $\mathbf{w}$ can be estimated the same way. That is, we can rewrite the objective function in matrix-vector form to produce

$$E(\mathbf{w}) = ||\mathbf{y} - \mathbf{B}\mathbf{w}||^2 \tag{34}$$

where $||\cdot||$ denotes the Euclidean norm, and the elements of the matrix $\mathbf{B}$ are given by $\mathbf{B}_{i,j} = b_j(x_i)$ (for row i and column j). In Matlab the least-squares estimate can be computed as $\mathbf{w}^* = \mathbf{B}\backslash\mathbf{y}$.

Picking the other parameters. The positions of the centers and the widths of the RBF basis functions cannot be solved directly for in closed form. So we need some other criteria to select them. If we optimize these parameters for the squared-error, then we will end up with one basis center at each data point, and with tiny width that exactly fit the data. This is a problem as such a model will not usually provide good predictions for inputs other than those in the training set.

The following heuristics instead are commonly used to determine these parameters without overfitting the training data. To pick the basis centers:

1. Place the centers uniformly spaced in the region containing the data. This is quite simple, but can lead to empty regions with basis functions, and will have an impractical number of data points in higher-dimensinal input spaces.

2. Place one center at each data point. This is used more often, since it limits the number of centers needed, although it can also be expensive if the number of data points is large.

3. Cluster the data, and use one center for each cluster. We will cover clustering methods later in the course.

To pick the width parameter:

1. Manually try different values of the width and pick the best by trial-and-error.

2. Use the average squared distances (or median distances) to neighboring centers, scaled by a constant, to be the width. This approach also allows you to use different widths for different basis functions, and it allows the basis functions to be spaced non-uniformly.

In later chapters we will discuss other methods for determining these and other parameters of models.

3.2 Overfitting and Regularization

Directly minimizing squared-error can lead to an effect called **overfitting,** wherein we fit the training data extremely well (i.e., with low error), yet we obtain a model that produces very poor predictions on future test data whenever the test inputs differ from the training inputs (Figure 4(b)). Overfitting can be understood in many ways, all of which are variations on the same underlying pathology:

1. The problem is insufficiently constrained: for example, if we have ten measurements and ten model parameters, then we can often obtain a perfect fit to the data.

2. Fitting noise: overfitting can occur when the model is so powerful that it can fit the data and also the random noise in the data.

3. Discarding uncertainty: the posterior probability distribution of the unknowns is insufficiently peaked to pick a single estimate. (We will explain what this means in more detail later.)

There are two important solutions to the overfitting problem: adding prior knowledge and handling uncertainty. The latter one we will discuss later in the course.

In many cases, there is some sort of prior knowledge we can leverage. A very common assumption is that the underlying function is likely to be **smooth**, for example, having small derivatives. Smoothness distinguishes the examples in Figure 4. There is also a practical reason to prefer smoothness, in that assuming smoothness reduces model complexity: it is easier to estimate smooth models from small datasets. In the extreme, if we make no prior assumptions about the nature of the fit then it is impossible to learn and generalize at all; smoothness assumptions are one way of constraining the space of models so that we have any hope of learning from small datasets.

One way to add smoothness is to parameterize the model in a smooth way (e.g., making the width parameter for RBFs larger; using only low-order polynomial basis functions), but this limits the expressiveness of the model. In particular, when we have lots and lots of data, we would like the data to be able to "overrule" the smoothness assumptions. With large widths, it is impossible to get highly-curved models no matter what the data says.

Instead, we can add **regularization**: an extra term to the learning objective function that prefers smooth models. For example, for RBF regression with scalar outputs, and with many other types of basis functions or multi-dimensional outputs, this can be done with an objective function of the form:

$$E(\mathbf{w}) = \underbrace{||\mathbf{y} - \mathbf{B}\mathbf{w}||^2}_{\text{data term}} + \underbrace{\lambda||\mathbf{w}||^2}_{\text{smoothness term}} \tag{35}$$

This objective function has two terms. The first term, called the data term, measures the model fit to the training data. The second term, often called the smoothness term, penalizes non-smoothness (rapid changes in $f(x)$). This particular smoothness term ($||\mathbf{w}||$) is called **weight decay**, because it tends to make the weights smaller.[3] Weight decay implicitly leads to smoothness with RBF basis functions because the basis functions themselves are smooth, so rapid changes in the slope of f (i.e., high curvature) can only be created in RBFs by adding and subtracting basis functions with large weights. (Ideally, we might directly penalize smoothness, e.g., using an objective term that directly penalizes the integral of the squared curvature of $f(\mathbf{x})$, but this is usually impractical.)

[3] Estimation with this objective function is sometimes called Ridge Regression in Statistics.

This **regularized least-squares** objective function is still quadratic with respect to $\mathbf{w}$ and can be optimized in closed-form. To see this, we can rewrite it as follows:

$$\begin{aligned} E(\mathbf{w}) &= (\mathbf{y} - \mathbf{B}\mathbf{w})^T(\mathbf{y} - \mathbf{B}\mathbf{w}) + \lambda \mathbf{w}^T\mathbf{w} && (36)\\ &= \mathbf{w}^T\mathbf{B}^T\mathbf{B}\mathbf{w} - 2\mathbf{w}^T\mathbf{B}^T\mathbf{y} + \lambda\mathbf{w}^T\mathbf{w} + \mathbf{y}^T\mathbf{y} && (37)\\ &= \mathbf{w}^T(\mathbf{B}^T\mathbf{B} + \lambda\mathbf{I})\mathbf{w} - 2\mathbf{w}^T\mathbf{B}^T\mathbf{y} + \mathbf{y}^T\mathbf{y} && (38) \end{aligned}$$

To minimize $E(\mathbf{w})$, as above, we solve the normal equations $\nabla E(\mathbf{w}) = \mathbf{0}$ (i.e., $\partial E/\partial w_i = 0$ for all i). This yields the following regularized LS estimate for $\mathbf{w}$:

$$\mathbf{w}^* = (\mathbf{B}^T\mathbf{B} + \lambda\mathbf{I})^{-1}\mathbf{B}^T\mathbf{y} \quad (39)$$

3.3 Artificial Neural Networks

Another choice of basis function is the sigmoid function. "Sigmoid" literally means "s-shaped." The most common choice of sigmoid is:

$$g(a) = \frac{1}{1 + e^{-a}} \quad (40)$$

Sigmoids can be combined to create a model called an **Artificial Neural Network (ANN)**. For regression with multi-dimensional inputs $\mathbf{x} \in \mathbb{R}_2^K$, and multidimensional outputs $\mathbf{y} \in \mathbb{R}^{K_1}$:

$$\mathbf{y} = f(\mathbf{x}) = \sum_j \mathbf{w}_j^{(1)} g\left(\sum_k w_{k,j}^{(2)} x_k + b_j^{(2)}\right) + \mathbf{b}^{(1)} \quad (41)$$

This equation describes a process whereby a linear regressor with weights $\mathbf{w}(2)$ is applied to $\mathbf{x}$. The output of this regressor is then put through the nonlinear Sigmoid function, the outputs of which act as features to another linear regressor. Thus, note that the *inner weights* $w^{(2)}$ are distinct parameters from the *outer weights* $\mathbf{w}_j^{(1)}$. As usual, it is easiest to interpret this model in the 1D case, i.e.,

$$y = f(x) = \sum_j w_j^{(1)} g\left(w_j^{(2)} x + b_j^{(2)}\right) + b^{(1)} \quad (42)$$

Figure 5(left) shows plots of $g(wx)$ for different values of w, and Figure 5(right) shows $g(x+b)$ for different values of b. As can be seen from the figures, the sigmoid function acts more or less like a step function for large values of w, and more like a linear ramp for small values of w. The bias b shifts the function left or right. Hence, the neural network is a linear combination of shifted (smoothed) step functions, linear ramps, and the bias term.

To learn an artificial neural network, we can again write a regularized squared-error objective function:

$$E(w, b) = ||\mathbf{y} - f(\mathbf{x})||^2 + \lambda||\mathbf{w}||^2 \quad (43)$$

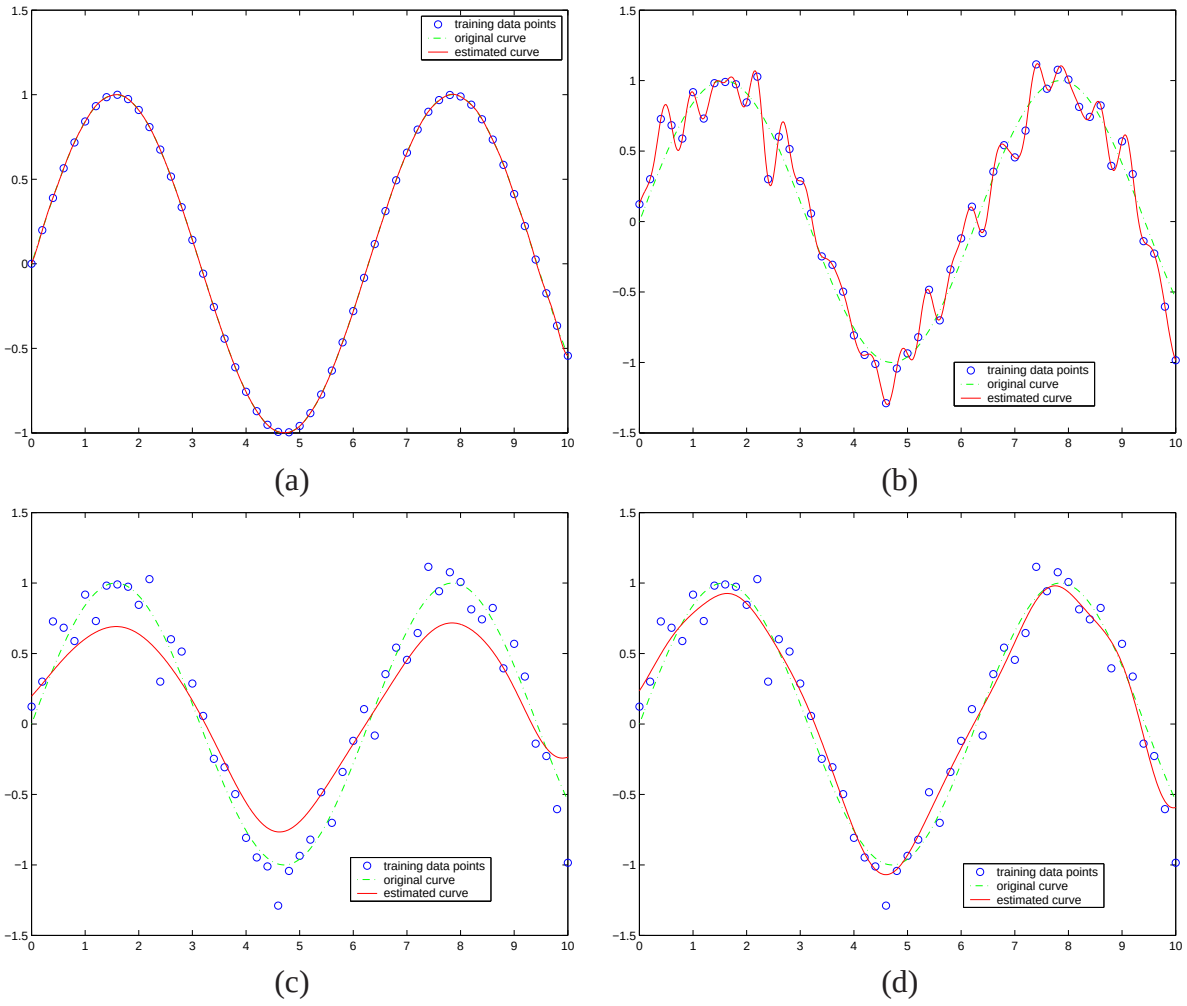

Figure 4: Least-squares curve fitting of an RBF. (a) Point data (blue circles) was taken from a sine curve, and a curve was fit to the points by a least-squares fit. The horizontal axis is x, the vertical axis is y, and the red curve is the estimated $f(\mathbf{x})$. In this case, the fit is essentially perfect. The curve representation is a sum of Gaussian basis functions. (b) **Overfitting**. Random noise was added to the data points, and the curve was fit again. The curve exactly fits the data points, which does not reproduce the original curve (a green, dashed line) very well. (c) **Underfitting**. Adding a smoothness term makes the resulting curve too smooth. (In this case, weight decay was used, along with reducing the number of basis functions). (d) Reducing the strength of the smoothness term yields a better fit.

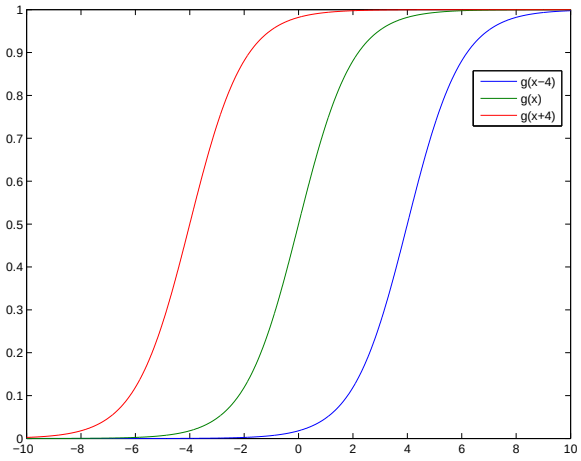

Figure 5: *Left:* Sigmoids $g(wx) = 1/(1+e^{-wx})$ for various values of w, ranging from linear ramps to smooth steps to nearly hard steps. *Right:* Sigmoids $g(x+b) = 1/(1+e^{-x-b})$ with different shifts b.

where $\mathbf{w}$ comprises the weights at both levels for all j. Note that we regularize by applying weight decay to the weights (both inner and outer), but not the biases, since only the weights affect the smoothness of the resulting function (why?).

Unfortuntely, this objective function cannot be optimized in closed-form, and numerical optimization procedures must be used. We will study one such method, gradient descent, in the next chapter.

3.4 K-Nearest Neighbors

At heart, many learning procedures — especially when our prior knowledge is weak — amount to smoothing the training data. RBF fitting is an example of this. However, many of these fitting procedures require making a number of decisions, such as the locations of the basis functions, and can be sensitive to these choices. This raises the question: why not cut out the middleman, and smooth the data directly? This is the idea behind K-**Nearest Neighbors** regression.

The idea is simple. We first select a parameter K, which is the only parameter to the algorithm. Then, for a new input $\mathbf{x}$, we find the K nearest neighbors to $\mathbf{x}$ in the training set, based on their Euclidean distance $||\mathbf{x} - \mathbf{x}_i||^2$. Then, our new output $\mathbf{y}$ is simply an average of the training outputs

for those nearest neigbors. This can be expressed as:

$$\mathbf{y} = \frac{1}{K} \sum_{i \in N_K(\mathbf{x})} \mathbf{y}_i \tag{44}$$

where the set $N_K(\mathbf{x})$ contains the indicies of the K training points closest to $\mathbf{x}$. Alternatively, we might take a weighted average of the K-nearest neighbors to give more influence to training points close to $\mathbf{x}$ than to those further away:

$$\mathbf{y} = \frac{\sum_{i \in N_K(\mathbf{x})} w(\mathbf{x}_i) \mathbf{y}_i}{\sum_{i \in N_K(\mathbf{x})} w(\mathbf{x}_i)} \quad , \quad w(\mathbf{x}_i) = e^{-||\mathbf{x}_i - \mathbf{x}||^2 / 2\sigma^2} \tag{45}$$

where σ^2 is an additional parameter to the algorithm. The parameters K and σ control the degree of smoothing performed by the algorithm. In the extreme case of $K = 1$, the algorithm produces a piecewise-constant function.

K-nearest neighbors is simple and easy to implement; it doesn't require us to muck about at all with different choices of basis functions or regularizations. However, it doesn't compress the data at all: we have to keep around the entire training set in order to use it, which could be very expensive, and we must search the whole data set to make predictions. (The cost of searching can be mitigated with spatial data-structures designed for searching, such as k-d-trees and locality-sensitive hashing. We will not cover these methods here).

4 Quadratics

The objective functions used in linear least-squares and regularized least-squares are multidimensional quadratics. We now analyze multidimensional quadratics further. We will see many more uses of quadratics further in the course, particularly when dealing with Gaussian distributions.

The general form of a one-dimensional quadratic is given by:

$$f(x) = w_2 x^2 + w_1 x + w_0 \tag{46}$$

This can also be written in a slightly different way (called standard form):

$$f(x) = a(x-b)^2 + c \tag{47}$$

where $a = w_2, b = -w_1/(2w_2), c = w_0 - w_1^2/4w_2$. These two forms are equivalent, and it is easy to go back and forth between them (e.g., given a, b, c, what are w_0, w_1, w_2?). In the latter form, it is easy to visualize the shape of the curve: it is a bowl, with minimum (or maximum) at b, and the "width" of the bowl is determined by the magnitude of a, the sign of a tells us which direction the bowl points (a positive means a convex bowl, a negative means a concave bowl), and c tells us how high or low the bowl goes (at $x = b$). We will now generalize these intuitions for higher-dimensional quadratics.

The general form for a 2D quadratic function is:

$$f(x_1, x_2) = w_{1,1} x_1^2 + w_{1,2} x_1 x_2 + w_{2,2} x_2^2 + w_1 x_1 + w_2 x_2 + w_0 \tag{48}$$

and, for an N-D quadratic, it is:

$$f(x_1, ...x_N) = \sum_{1 \le i \le N, 1 \le j \le N} w_{i,j} x_i x_j + \sum_{1 \le i \le N} w_i x_i + w_0 \tag{49}$$

Note that there are three sets of terms: the quadratic terms ($\sum w_{i,j} x_i x_j$), the linear terms ($\sum w_i x_i$) and the constant term (w_0).

Dealing with these summations is rather cumbersome. We can simplify things by using matrix-vector notation. Let $\mathbf{x}$ be an N-dimensional column vector, written $\mathbf{x} = [x_1, ...x_N]^T$. Then we can write a quadratic as:

$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c \tag{50}$$

where

$$\mathbf{A} = \begin{bmatrix} w_{1,1} & \dots & w_{1,N} \\ \vdots & w_{i,j} & \vdots \\ w_{N,1} & \dots & w_{N,N} \end{bmatrix} \tag{51}$$

$$\mathbf{b} = [w_1, ..., w_N]^T \tag{52}$$

$$c = w_0 \tag{53}$$

You should verify for yourself that these different forms are equivalent: by multiplying out all the elements of $f(\mathbf{x})$, either in the 2D case or, using summations, the general $N - D$ case.

For many manipulations we will want to do later, it is helpful for $\mathbf{A}$ to be symmetric, i.e., to have $w_{i,j} = w_{j,i}$. In fact, it should be clear that these off-diagonal entries are redundant. So, if we are a given a quadratic for which $\mathbf{A}$ is asymmetric, we can symmetrize it as:

$$f(\mathbf{x}) = \mathbf{x}^T(\frac{1}{2}(\mathbf{A} + \mathbf{A}^T))\mathbf{x} + \mathbf{b}^T\mathbf{x} + c = \mathbf{x}^T\tilde{\mathbf{A}}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c \tag{54}$$

and use $\tilde{\mathbf{A}} = \frac{1}{2}(\mathbf{A} + \mathbf{A}^T)$ instead. You should confirm for yourself that this is equivalent to the original quadratic.

As before, we can convert the quadratic to a form that leads to clearer interpretation:

$$f(\mathbf{x}) = (\mathbf{x} - \boldsymbol{\mu})^T\mathbf{A}(\mathbf{x} - \boldsymbol{\mu}) + d \tag{55}$$

where $\boldsymbol{\mu} = -\frac{1}{2}\mathbf{A}^{-1}\mathbf{b}, d = c - \boldsymbol{\mu}^T\mathbf{A}\boldsymbol{\mu}$, assuming that $\mathbf{A}^{-1}$ exists. Note the similarity here to the 1-D case. As before, this function is a bowl-shape in N dimensions, with curvature specified by the matrix $\mathbf{A}$, and with a single stationary point $\boldsymbol{\mu}$.[4] However, fully understanding the shape of $f(\mathbf{x})$ is a bit more subtle and interesting.

4.1 Optimizing a quadratic

Suppose we wish to find the stationary points (minima or maxima) of a quadratic

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c. \tag{56}$$

The stationary points occur where all partial derivatives are zero, i.e., $\partial f/\partial x_i = 0$ for all i. The gradient of a function is the vector comprising the partial derivatives of the function, i.e.,

$$\nabla f \equiv [\partial f/\partial x_1, \partial f/\partial x_2, \ldots, \partial f/\partial N]^T \ . \tag{57}$$

At stationary points it must therefore be true that $\nabla f = [0, \ldots, 0]^T$. Let us assume that $\mathbf{A}$ is symmetric (if it is not, then we can symmetrize it as above). Equation 56 is a very common form of cost function (e.g. the log probability of a Gaussian as we will later see), and so the form of its gradient is important to examine.

Due to the linearity of the differentiation operator, we can look at each of the three terms of Eq.56 separately. The last (constant) term does not depend on $\mathbf{x}$ and so we can ignore it because its derivative is zero. Let us examine the first term. If we write out the individual terms within the

[4] A stationary point means a setting of $\mathbf{x}$ where the gradient is zero.

vectors/matrices, we get:

$$(x_1 \dots x_N) \begin{pmatrix} a_{11} & \dots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} \tag{58}$$

$$= (x_1 a_{11} + x_2 a_{21} + \dots + x_N a_{N1} x_1 a_{12} + x_2 a_{22} + \dots \tag{59}$$

$$\dots + x_1 a_{1N} + x_2 a_{2N} + \dots + x_N a_{NN}) \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} \tag{60}$$

$$= x_1^2 a_{11} + x_1 x_2 a_{21} + \dots + x_1 x_N a_{N1} + x_1 x_2 a_{12} + x_2^2 a_{22} + \dots + x_N x_2 a_{N2} + \dots \tag{61}$$

$$\dots x_1 x_N a_{1N} + x_2 x_N a_{2N} + \dots + x_N^2 a_{NN} \tag{62}$$

$$= \sum_{ij} a_{ij} x_i x_j \tag{63}$$

The i^{th} element of the gradient corresponds to $\partial f / \partial x_i$. So in the expression above, for the terms in the gradient corresponding to each x_i, we only need to consider the terms involving x_i (others will have derivative zero), namely

$$x_i^2 a_{ii} + \sum_{j \neq i} x_i x_j (a_{ij} + a_{ji}) \tag{64}$$

The gradient then has a very simple form:

$$\frac{\partial \left(\mathbf{x}^T \mathbf{A} \mathbf{x} \right)}{\partial x_i} = 2 x_i a_{ii} + \sum_{j \neq i} x_j (a_{ij} + a_{ji}). \tag{65}$$

We can write a single expression for all of the x_i using matrix/vector form:

$$\frac{\partial \mathbf{x}^T \mathbf{A} \mathbf{x}}{\partial \mathbf{x}} = (\mathbf{A} + \mathbf{A}^T) \mathbf{x}. \tag{66}$$

You should multiply this out for yourself to see that this corresponds to the individual terms above. If we assume that $\mathbf{A}$ is symmetric, then we have

$$\frac{\partial \mathbf{x}^T \mathbf{A} \mathbf{x}}{\partial \mathbf{x}} = 2 \mathbf{A} \mathbf{x}. \tag{67}$$

This is also a very helpful rule that you should remember. The next term in the cost function, $\mathbf{b}^T \mathbf{x}$, has an even simpler gradient. Note that this is simply a dot product, and the result is a scalar:

$$\mathbf{b}^T \mathbf{x} = b_1 x_1 + b_2 x_2 + \dots + b_N x_N. \tag{68}$$

Only one term corresponds to each x_i and so $\partial f / \partial x_i = b_i$. We can again express this in matrix/vector form:

$$\frac{\partial \left(\mathbf{b}^T \mathbf{x}\right)}{\partial \mathbf{x}} = \mathbf{b}. \tag{69}$$

This is another helpful rule that you will encounter again. If we use both of the expressions we have just derived, and set the gradient of the cost function to zero, we get:

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = 2\mathbf{A}\mathbf{x} + \mathbf{b} = [0, \ldots, 0]^T \tag{70}$$

The optimum is given by the solution to this system of equations (called *normal equations*):

$$\mathbf{x} = -\frac{1}{2}\mathbf{A}^{-1}\mathbf{b} \tag{71}$$

In the case of scalar x, this reduces to $x = -b/2a$. For linear regression with multi-dimensional inputs above (see Equation 18): $\mathbf{A} = \mathbf{X}\mathbf{X}^T$ and $\mathbf{b} = -2\mathbf{X}\mathbf{y}^T$. As an exercise, convince yourself that this is true.

5 Basic Probability Theory

Probability theory addresses the following fundamental question: *how do we reason?* Reasoning is central to many areas of human endeavor, including philosophy (what is the best way to make decisions?), cognitive science (how does the mind work?), artificial intelligence (how do we build reasoning machines?), and science (how do we test and develop theories based on experimental data?). In nearly all real-world situations, our data and knowledge about the world is incomplete, indirect, and noisy; hence, uncertainty must be a fundamental part of our decision-making process. Bayesian reasoning provides a formal and consistent way to reasoning in the presence of uncertainty; probabilistic inference is an embodiment of common sense reasoning.

The approach we focus on here is **Bayesian**. Bayesian probability theory is distinguished by defining probabilities as **degrees-of-belief**. This is in contrast to **Frequentist statistics**, where the probability of an event is defined as its frequency in the limit of an infinite number of repeated trials.

5.1 Classical logic

Perhaps the most famous attempt to describe a formal system of reasoning is classical logic, originally developed by Aristotle. In classical logic, we have some statements that may be true or false, and we have a set of rules which allow us to determine the truth or falsity of new statements. For example, suppose we introduce two statements, named $\mathbf{A}$ and $\mathbf{B}$:

$\mathbf{A} \equiv$ "My car was stolen"
$\mathbf{B} \equiv$ "My car is not in the parking spot where I remember leaving it"

Moreover, let us assert the rule "$\mathbf{A}$ implies $\mathbf{B}$", which we will write as $\mathbf{A} \rightarrow \mathbf{B}$. Then, if $\mathbf{A}$ is known to be true, we may deduce logically that $\mathbf{B}$ must also be true (if my car is stolen then it won't be in the parking spot where I left it). Alternatively, if I find my car where I left it ("$\mathbf{B}$ is false," written $\bar{\mathbf{B}}$), then I may infer that it was not stolen ($\bar{\mathbf{A}}$) by the contrapositive $\bar{\mathbf{B}} \rightarrow \bar{\mathbf{A}}$.

Classical logic provides a model of how humans might reason, and a model of how we might build an "intelligent" computer. Unfortunately, classical logic has a significant shortcoming: it assumes that all knowledge is absolute. Logic requires that we know some facts about the world with absolute certainty, and then, we may deduce only those facts which must follow with absolute certainty.

In the real world, there are almost no facts that we know with absolute certainty — most of what we know about the world we acquire indirectly, through our five senses, or from dialogue with other people. One can therefore conclude that most of what we know about the world is **uncertain**. (Finding something that we know with certainty has occupied generations of philosophers.)

For example, suppose I discover that my car is not where I remember leaving it ($\mathbf{B}$). Does this mean that it was stolen? No, there are many other explanations — maybe I have forgotten where I left it or maybe it was towed. However, the knowledge of $\mathbf{B}$ makes $\mathbf{A}$ more *plausible* — even though I do not know it to be stolen, it becomes more likely a scenario than before. The

actual degree of plausibility depends on other contextual information — did I park it in a safe neighborhood?, did I park it in a handicapped zone?, etc.

Predicting the weather is another task that requires reasoning with uncertain information. While we can make some predictions with great confidence (e.g. we can reliably predict that it will not snow in June, north of the equator), we are often faced with much more difficult questions (will it rain today?) which we must infer from unreliable sources of information (e.g., the weather report, clouds in the sky, yesterday's weather, etc.). In the end, we usually cannot determine for certain whether it will rain, but we do get a degree of certainty upon which to base decisions and decide whether or not to carry an umbrella.

Another important example of uncertain reasoning occurs whenever you meet someone new — at this time, you immediately make hundreds of inferences (mostly unconscious) about who this person is and what their emotions and goals are. You make these decisions based on the person's appearance, the way they are dressed, their facial expressions, their actions, the context in which you meet, and what you have learned from previous experience with other people. Of course, you have no conclusive basis for forming opinions (e.g., the panhandler you meet on the street may be a method actor preparing for a role). However, we need to be able to make judgements about other people based on incomplete information; otherwise, normal interpersonal interaction would be impossible (e.g., how do you really *know* that everyone isn't out to get you?).

What we need is a way of discussing not just true or false statements, but statements that have varying levels of certainty. In addition, we would like to be able to use our beliefs to reason about the world and interpret it. As we gain new information, our beliefs should change to reflect our greater knowledge. For example, for any two propositions $\mathbf{A}$ and $\mathbf{B}$ (that may be true or false), if $\mathbf{A} \rightarrow \mathbf{B}$, then strong belief in $\mathbf{A}$ should increase our belief in $\mathbf{B}$. Moreover, strong belief in $\mathbf{B}$ may sometimes increase our belief in $\mathbf{A}$ as well.

5.2 Basic definitions and rules

The rules of probability theory provide a system for reasoning with uncertainty.There are a number of justifications for the use of probability theory to represent logic (such as Cox's Axioms) that show, for certain particular definitions of common-sense reasoning, that probability theory is the only system that is consistent with common-sense reasoning. We will not cover these here (see, for example, Wikipedia for discussion of the Cox Axioms).

The basic rules of probability theory are as follows.

- The probability of a statement $\mathbf{A}$ — denoted $P(\mathbf{A})$ — is a real number between 0 and 1, inclusive. $P(\mathbf{A}) = 1$ indicates absolute certainty that $\mathbf{A}$ is true, $P(\mathbf{A}) = 0$ indicates absolute certainty that $\mathbf{A}$ is false, and values between 0 and 1 correspond to varying degrees of certainty.

- The **joint probability** of two statements $\mathbf{A}$ and $\mathbf{B}$ — denoted $P(\mathbf{A}, \mathbf{B})$ — is the probability that both statements are true. (i.e., the probability that the statement "$\mathbf{A} \wedge \mathbf{B}$" is true). (Clearly, $P(\mathbf{A}, \mathbf{B}) = P(\mathbf{B}, \mathbf{A})$.)

- The **conditional probability** of $\mathbf{A}$ given $\mathbf{B}$ — denoted $P(\mathbf{A}|\mathbf{B})$ — is the probability that we would assign to $\mathbf{A}$ being true, **if** we knew $\mathbf{B}$ to be true. The conditional probability is defined as $P(\mathbf{A}|\mathbf{B}) = P(\mathbf{A}, \mathbf{B})/P(\mathbf{B})$.

- **The Product Rule:**
$$P(\mathbf{A}, \mathbf{B}) = P(\mathbf{A}|\mathbf{B})P(\mathbf{B}) \tag{72}$$
In other words, the probability that $\mathbf{A}$ and $\mathbf{B}$ are both true is given by the probability that $\mathbf{B}$ is true, multiplied by the probability we would assign to $\mathbf{A}$ if we knew $\mathbf{B}$ to be true. Similarly, $P(\mathbf{A}, \mathbf{B}) = P(\mathbf{B}|\mathbf{A})P(\mathbf{A})$. This rule follows directly from the definition of conditional probability.

- **The Sum Rule:**
$$P(\mathbf{A}) + P(\bar{\mathbf{A}}) = 1 \tag{73}$$
In other words, the probability of a statement being true and the probability that it is false must sum to 1. In other words, our certainty that $\mathbf{A}$ is true is in inverse proportion to our certainty that it is not true. A consequence: given a set of mutually-exclusive statements $\mathbf{A}_i$, exactly one of which must be true, we have
$$\sum_i P(\mathbf{A}_i) = 1 \tag{74}$$

- All of the above rules can be made conditional on additional information. For example, given an additional statement $\mathbf{C}$, we can write the Sum Rule as:
$$\sum_i P(\mathbf{A}_i|\mathbf{C}) = 1 \tag{75}$$
and the Product Rule as
$$P(\mathbf{A}, \mathbf{B}|\mathbf{C}) = P(\mathbf{A}|\mathbf{B}, \mathbf{C})P(\mathbf{B}|\mathbf{C}) \tag{76}$$

From these rules, we further derive many more expressions to relate probabilities. For example, one important operation is called **marginalization:**
$$P(\mathbf{B}) = \sum_i P(\mathbf{A}_i, \mathbf{B}) \tag{77}$$
if $\mathbf{A}_i$ are mutually-exclusive statements, of which exactly one must be true. In the simplest case — where the statement $\mathbf{A}$ may be true or false — we can derive:
$$P(\mathbf{B}) = P(\mathbf{A}, \mathbf{B}) + P(\bar{\mathbf{A}}, \mathbf{B}) \tag{78}$$

The derivation of this formula is straightforward, using the basic rules of probability theory:

$$P(\mathbf{A}) + P(\bar{\mathbf{A}}) = 1, \quad \text{Sum rule} \tag{79}$$
$$P(\mathbf{A}|\mathbf{B}) + P(\bar{\mathbf{A}}|\mathbf{B}) = 1, \quad \text{Conditioning} \tag{80}$$
$$P(\mathbf{A}|\mathbf{B})P(\mathbf{B}) + P(\bar{\mathbf{A}}|\mathbf{B})P(\mathbf{B}) = P(\mathbf{B}), \quad \text{Algebra} \tag{81}$$
$$P(\mathbf{A}, \mathbf{B}) + P(\bar{\mathbf{A}}, \mathbf{B}) = P(\mathbf{B}), \quad \text{Product rule} \tag{82}$$

Marginalization gives us a useful way to compute the probability of a statement $\mathbf{B}$ that is intertwined with many other uncertain statements.

Another useful concept is the notion of **independence**. Two statements are independent if and only if $P(\mathbf{A}, \mathbf{B}) = P(\mathbf{A})P(\mathbf{B})$. If $\mathbf{A}$ and $\mathbf{B}$ are independent, then it follows that $P(\mathbf{A}|\mathbf{B}) = P(\mathbf{A})$ (by combining the Product Rule with the defintion of independence). Intuitively, this means that, whether or not $\mathbf{B}$ is true tells you nothing about whether $\mathbf{A}$ is true.

In the rest of these notes, I will always use probabilities as statements about variables. For example, suppose we have a variable x that indicates whether there are one, two, or three people in a room (i.e., the only possibilities are $x = 1$, $x = 2$, $x = 3$). Then, by the sum rule, we can derive $P(x = 1) + P(x = 2) + P(x = 3) = 1$. Probabilities can also describe the range of a real variable. For example, $P(y < 5)$ is the probability that the variable y is less than 5. (We'll discuss continuous random variables and probability densities in more detail in the next chapter.)

To summarize:

The basic rules of probability theory:
- $P(\mathbf{A}) \in [0...1]$
- **Product rule:** $P(\mathbf{A}, \mathbf{B}) = P(\mathbf{A}|\mathbf{B})P(\mathbf{B})$
- **Sum rule:** $P(\mathbf{A}) + P(\bar{\mathbf{A}}) = 1$
- Two statements $\mathbf{A}$ and $\mathbf{B}$ are **independent** iff: $P(\mathbf{A}, \mathbf{B}) = P(\mathbf{A})P(\mathbf{B})$
- **Marginalizing:** $P(\mathbf{B}) = \sum_i P(\mathbf{A}_i, \mathbf{B})$
- Any basic rule can be made conditional on additional information.

For example, it follows from the product rule that $P(\mathbf{A}, \mathbf{B}|\mathbf{C}) = P(\mathbf{A}|\mathbf{B}, \mathbf{C})P(\mathbf{B}|\mathbf{C})$

Once we have these rules — and a suitable model — we can derive *any* probability that we want. With some experience, you should be able to derive any desired probability (e.g., $P(\mathbf{A}|\mathbf{C})$) given a basic model.

5.3 Discrete random variables

It is convenient to describe systems in terms of variables. For example, to describe the weather, we might define a discrete variable $\mathbf{w}$ that can take on two values sunny or rainy, and then try to determine $P(\mathbf{w} = \mathsf{sunny})$, i.e., the probability that it will be sunny today. **Discrete distributions** describe these types of probabilities.

As a concrete example, let's flip a coin. Let $\mathbf{c}$ be a variable that indicates the result of the flip: $\mathbf{c} = \mathsf{heads}$ if the coin lands on its head, and $\mathbf{c} = \mathsf{tails}$ otherwise. In this chapter and the rest of

these notes, I will use probabilities specifically to refer to values of variables, e.g., $P(\mathbf{c} = \mathsf{heads})$ is the probability that the coin lands heads.

What is the probability that the coin lands heads? This probability should be some real number θ, $0 \leq \theta \leq 1$. For most coins, we would say $\theta = .5$. What does this number mean? The number θ is a representation of our belief about the possible values of $\mathbf{c}$. Some examples:

$\theta = 0$ we are absolutely certain the coin will land tails
$\theta = 1/3$ we believe that tails is twice as likely as heads
$\theta = 1/2$ we believe heads and tails are equally likely
$\theta = 1$ we are absolutely certain the coin will land heads

Formally, we denote the probability of the coin coming up heads as $P(\mathbf{c} = \mathsf{heads})$, so $P(\mathbf{c} = \mathsf{heads}) = \theta$. In general, we denote the probability of a specific event event as $P(\mathsf{event})$. By the Sum Rule, we know $P(\mathbf{c} = \mathsf{heads}) + P(\mathbf{c} = \mathsf{tails}) = 1$, and thus $P(\mathbf{c} = \mathsf{tails}) = 1 - \theta$.

Once we flip the coin and observe the result, then we can be pretty sure that we know the value of $\mathbf{c}$; there is no practical need to model the uncertainty in this measurement. However, suppose we do not observe the coin flip, but instead hear about it from a friend, who may be forgetful or untrustworthy. Let $\mathbf{f}$ be a variable indicating how the friend claims the coin landed, i.e. $\mathbf{f} = \mathsf{heads}$ means the friend says that the coin came up heads. Suppose the friend says the coin landed heads — do we believe him, and, if so, with how much certainty? As we shall see, probabilistic reasoning obtains quantitative values that, qualitatively, matches our common sense very effectively.

Suppose we know something about our friend's behaviour. We can represent our beliefs with the following probabilities, for example, $P(\mathbf{f} = \mathsf{heads}|\mathbf{c} = \mathsf{heads})$ represents our belief that the friend says "heads" when the the coin landed heads. Because the friend can only say one thing, we can apply the Sum Rule to get:

$$P(\mathbf{f} = \mathsf{heads}|\mathbf{c} = \mathsf{heads}) + P(\mathbf{f} = \mathsf{tails}|\mathbf{c} = \mathsf{heads}) = 1 \quad (83)$$

$$P(\mathbf{f} = \mathsf{heads}|\mathbf{c} = \mathsf{tails}) + P(\mathbf{f} = \mathsf{tails}|\mathbf{c} = \mathsf{tails}) = 1 \quad (84)$$

If our friend always tells the truth, then we know $P(\mathbf{f} = \mathsf{heads}|\mathbf{c} = \mathsf{heads}) = 1$ and $P(\mathbf{f} = \mathsf{tails}|\mathbf{c} = \mathsf{heads}) = 0$. If our friend *usually* lies, then, for example, we might have $P(\mathbf{f} = \mathsf{heads}|\mathbf{c} = \mathsf{heads}) = .3$.

5.4 Binomial and Multinomial distributions

A binomial distribution is the distribution over the number of positive outcomes for a yes/no (binary) experiment, where on each trial the probability of a positive outcome is $p \in [0, 1]$. For example, for n tosses of a coin for which the probability of heads on a single trial is p, the distribution over the number of heads we might observe is a binomial distribution. The binomial distribution over the number of positive outcomes, denoted K, given n trials, each having a positive outcome with probability p is given by

$$P(K = k) = \binom{n}{k} p^k (1 - p)^{n-k} \quad (85)$$

for $k = 0, 1, \ldots, n$, where

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \,. \tag{86}$$

A multinomial distribution is a natural extension of the binomial distribution to an experiment with k mutually exclusive outcomes, having probabilities p_j, for $j = 1, \ldots, k$. Of course, to be valid probabilities $\sum p_j = 1$. For example, rolling a die can yield one of six values, each with probability 1/6 (assuming the die is fair). Given n trials, the multinomial distribution specifies the distribution over the number of each of the possible outcomes. Given n trials, k possible outcomes with probabilities p_j, the distribution over the event that outcome j occurs x_j times (and of course $\sum x_j = n$), is the multinomial distribution given by

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_k = x_k) = \frac{n!}{x_1!\, x_2!\, \ldots\, x_k!}\, p_1^{x_1}\, p_2^{x_2}\, \ldots\, p_k^{x_k} \tag{87}$$

5.5 Mathematical expectation

Suppose each outcome r_i has an associated real value $x_i \in \mathbb{R}$. Then the expected value of x is:

$$E[x] = \sum_i P(r_i) x_i \,. \tag{88}$$

The expected value of $f(x)$ is given by

$$E[f(x)] = \sum_i P(r_i) f(x_i) \,. \tag{89}$$

6 Probability Density Functions (PDFs)

In many cases, we wish to handle data that can be represented as a real-valued random variable, or a real-valued vector $\mathbf{x} = [x_1, x_2, ..., x_n]^T$. Most of the intuitions from discrete variables transfer directly to the continuous case, although there are some subtleties.

We describe the probabilities of a real-valued scalar variable x with a Probability Density Function (PDF), written $p(x)$. Any real-valued function $p(x)$ that satisfies:

$$p(x) \geq 0 \qquad \text{for all } x \tag{90}$$

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{91}$$

is a valid PDF. I will use the convention of upper-case P for discrete probabilities, and lower-case p for PDFs.

With the PDF we can specify the probability that the random variable x falls within a given range:

$$P(x_0 \leq x \leq x_1) = \int_{x_0}^{x_1} p(x)dx \tag{92}$$

This can be visualized by plotting the curve $p(x)$. Then, to determine the probability that x falls within a range, we compute the area under the curve for that range.

The PDF can be thought of as the infinite limit of a discrete distribution, i.e., a discrete distribution with an infinite number of possible outcomes. Specifically, suppose we create a discrete distribution with N possible outcomes, each corresponding to a range on the real number line. Then, suppose we increase N towards infinity, so that each outcome shrinks to a single real number; a PDF is defined as the limiting case of this discrete distribution.

There is an important subtlety here: a probability density is *not* a probability per se. For one thing, there is no requirement that $p(x) \leq 1$. Moreover, the probability that x attains any one specific value out of the infinite set of possible values is always zero, e.g. $P(x = 5) = \int_5^5 p(x)dx = 0$ for any PDF $p(x)$. People (myself included) are sometimes sloppy in referring to $p(x)$ as a probability, but it is not a probability — rather, it is a function that can be used in computing probabilities.

Joint distributions are defined in a natural way. For two variables x and y, the joint PDF $p(x, y)$ defines the probability that (x, y) lies in a given domain $\mathcal{D}$:

$$P((x, y) \in \mathcal{D}) = \int_{(x,y)\in\mathcal{D}} p(x, y)dxdy \tag{93}$$

For example, the probability that a 2D coordinate (x, y) lies in the domain $(0 \leq x \leq 1, 0 \leq y \leq 1)$ is $\int_{0\leq x\leq 1} \int_{0\leq y\leq 1} p(x, y)dxdy$. The PDF over a vector may also be written as a joint PDF of its variables. For example, for a 2D-vector $\mathbf{a} = [x, y]^T$, the PDF $p(\mathbf{a})$ is equivalent to the PDF $p(x, y)$.

Conditional distributions are defined as well: $p(x|\mathbf{A})$ is the PDF over x, if the statement $\mathbf{A}$ is true. This statement may be an expression on a continuous value, e.g. "$y = 5$." As a short-hand,

we can write $p(x|y)$, which provides a PDF for x for every value of y. (It must be the case that $\int p(x|y)dx = 1$, since $p(x|y)$ is a PDF over values of x.)

In general, for all of the rules for manipulating discrete distributions there are analogous rules for continuous distributions:

Probability rules for PDFs:
- $p(x) \geq 0$, for all x
- $\int_{-\infty}^{\infty} p(x)dx = 1$
- $P(x_0 \leq x \leq x_1) = \int_{x_0}^{x_1} p(x)dx$
- **Sum rule:** $\int_{-\infty}^{\infty} p(x)dx = 1$
- **Product rule:** $p(x, y) = p(x|y)p(y) = p(y|x)p(x)$.
- **Marginalization:** $p(y) = \int_{-\infty}^{\infty} p(x, y)dx$
- We can also add conditional information, e.g. $p(y|z) = \int_{-\infty}^{\infty} p(x, y|z)dx$
- **Independence:** Variables x and y are independent if: $p(x, y) = p(x)p(y)$.

6.1 Mathematical expectation, mean, and variance

Some very brief definitions of ways to describe a PDF:

Given a function $f(\mathbf{x})$ of an unknown variable $\mathbf{x}$, the **expected value** of the function with repect to a PDF $p(\mathbf{x})$ is defined as:

$$E_{p(\mathbf{x})}[f(\mathbf{x})] \equiv \int f(\mathbf{x})p(\mathbf{x})d\mathbf{x} \tag{94}$$

Intuitively, this is the value that we roughly "expect" $\mathbf{x}$ to have.

The mean $\boldsymbol{\mu}$ of a distribution $p(\mathbf{x})$ is the expected value of $\mathbf{x}$:

$$\boldsymbol{\mu} = E_{p(\mathbf{x})}[\mathbf{x}] = \int \mathbf{x}p(\mathbf{x})d\mathbf{x} \tag{95}$$

The variance of a scalar variable x is the expected squared deviation from the mean:

$$E_{p(x)}[(x - \mu)^2] = \int (x - \mu)^2 p(x)dx \tag{96}$$

The variance of a distribution tells us how uncertain, or "spread-out" the distribution is. For a very narrow distribution $E_{p(x)}[(x - \mu)^2]$ will be small.

The **covariance** of a vector $\mathbf{x}$ is a matrix:

$$\boldsymbol{\Sigma} = \text{cov}(\mathbf{x}) = E_{p(\mathbf{x})}[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T] = \int (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T p(x)d\mathbf{x} \tag{97}$$

By inspection, we can see that the diagonal entries of the covariance matrix are the variances of the individual entries of the vector:

$$\boldsymbol{\Sigma}_{ii} = \text{var}(x_{ii}) = E_{p(\mathbf{x})}[(x_i - \mu_i)^2] \tag{98}$$

The off-diagonal terms are covariances:

$$\mathbf{\Sigma}_{ij} = \text{cov}(x_i, x_j) = E_{p(x)}[(x_i - \mu_i)(x_j - \mu_j)] \tag{99}$$

between variables x_i and x_j. If the covariance is a large positive number, then we expect x_i to be larger than μ_i when x_j is larger than μ_j. If the covariance is zero and we know no other information, then knowing $x_i > \mu_i$ does not tell us whether or not it is likely that $x_j > \mu_j$.

One goal of statistics is to infer properties of distributions. In the simplest case, the **sample mean** of a collection of N data points $\mathbf{x}_{1:N}$ is just their average: $\bar{\mathbf{x}} = \frac{1}{N}\sum_i \mathbf{x}_i$. The **sample covariance** of a set of data points is: $\frac{1}{N}\sum_i (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T$. The covariance of the data points tells us how "spread-out" the data points are.

6.2 Uniform distributions

The simplest PDF is the **uniform distribution**. Intuitively, this distribution states that all values within a given range $[x_0, x_1]$ are equally likely. Formally, the uniform distribution on the interval $[x_0, x_1]$ is:

$$p(x) = \begin{cases} \frac{1}{x_1 - x_0} & \text{if } x_0 \leq x \leq x_1 \\ 0 & \text{otherwise} \end{cases} \tag{100}$$

It is easy to see that this is a valid PDF (because $p(x) > 0$ and $\int p(x)dx = 1$).

We can also write this distribution with this alternative notation:

$$x | x_0, x_1 \;\sim\; \mathcal{U}(x_0, x_1) \tag{101}$$

Equations 100 and 101 are equivalent. The latter simply says: x is distributed uniformly in the range x_0 and x_1, and it is impossible that x lies outside of that range.

The mean of a uniform distribution $\mathcal{U}(x_0, x_1)$ is $(x_1 + x_0)/2$. The variance is $(x_1 - x_0)^2/12$.

6.3 Gaussian distributions

Arguably the single most important PDF is the **Normal** (a.k.a., **Gaussian**) probability distribution function (PDF). Among the reasons for its popularity are that it is theoretically elegant, and arises naturally in a number of situations. It is the distribution that maximizes entropy, and it is also tied to the Central Limit Theorem: the distribution of a random variable which is the sum of a number of random variables approaches the Gaussian distribution as that number tends to infinity (Figure 6).

Perhaps most importantly, it is the analytical properties of the Gaussian that make it so ubiquitous. Gaussians are easy to manipulate, and their form so well understood, that we often assume quantities are Gaussian distributed, even though they are not, in order to turn an intractable model, or problem, into something that is easier to work with.

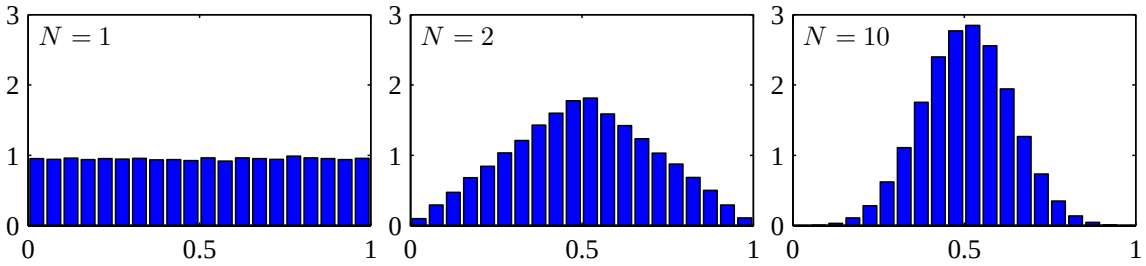

Figure 6: Histogram plots of the mean of N uniformly distributed numbers for various values of N. The effect of the Central Limit Theorem is seen: as N increases, the distribution becomes more Gaussian. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

The simplest case is a Gaussian PDF over a scalar value x, in which case the PDF is:

$$p(x|\mu,\sigma^2) \;=\; \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) \tag{102}$$

(The notation $\exp(a)$ is the same as e^a). The Gaussian has two parameters, the mean μ, and the variance σ^2. The mean specifies the center of the distribution, and the variance tells us how "spread-out" the PDF is.

The PDF for D-dimensional vector $\mathbf{x}$, the elements of which are jointly distributed with a the Gaussian denity function, is given by

$$p(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma}) \;=\; \frac{1}{\sqrt{(2\pi)^D|\boldsymbol{\Sigma}|}}\exp\left(-(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})/2\right) \tag{103}$$

where $\boldsymbol{\mu}$ is the mean vector, and $\boldsymbol{\Sigma}$ is the $D\times D$ covariance matrix, and $|A|$ denotes the determinant of matrix A. An important special case is when the Gaussian is isotropic (rotationally invariant). In this case the covariance matrix can be written as $\boldsymbol{\Sigma}=\sigma^2\mathbf{I}$ where $\mathbf{I}$ is the identity matrix. This is called a spherical or isotropic covariance matrix. In this case, the PDF reduces to:

$$p(\mathbf{x}|\boldsymbol{\mu},\sigma^2) \;=\; \frac{1}{\sqrt{(2\pi)^D\sigma^{2D}}}\exp\left(-\frac{1}{2\sigma^2}||\mathbf{x}-\boldsymbol{\mu}||^2\right)\;. \tag{104}$$

The Gaussian distribution is used frequently enough that it is useful to denote its PDF in a simple way. We will define a function G to be the Gaussian density function, i.e.,

$$G(\mathbf{x};\boldsymbol{\mu},\boldsymbol{\Sigma}) \equiv \frac{1}{\sqrt{(2\pi)^D|\boldsymbol{\Sigma}|}}\exp\left(-(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})/2\right) \tag{105}$$

When formulating problems and manipulating PDFs this functional notation will be useful. When we want to specify that a random vector has a Gaussian PDF, it is common to use the notation:

$$\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma} \sim \mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma}) \tag{106}$$

Equations 103 and 106 essentially say the same thing. Equation 106 says that $\mathbf{x}$ is Gaussian, and Equation 103 specifies (evaluates) the density for an input $\mathbf{x}$.

The covariance matrix $\boldsymbol{\Sigma}$ of a Gaussian must be symmetric and positive definite — this is equivalent to requiring that $|\boldsymbol{\Sigma}| > 0$. Otherwise, the formula does not correspond to a valid PDF, since Equation 103 is no longer real-valued if $|\boldsymbol{\Sigma}| \leq 0$.

6.3.1 Diagonalization

A useful way to understand a Gaussian is to diagonalize the exponent. The exponent of the Gaussian is quadratic, and so its shape is essentially elliptical. Through diagonalization we find the major axes of the ellipse, and the variance of the distribution along those axes. Seeing the Gaussian this way often makes it easier to interpret the distribution.

As a reminder, the eigendecomposition of a real-valued symmetric matrix $\boldsymbol{\Sigma}$ yields a set of orthonormal vectors $\mathbf{v}_i$ and scalars λ_i such that

$$\boldsymbol{\Sigma}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{107}$$

Equivalently, if we combine the eigenvalues and eigenvectors into matrices $\mathbf{U} = [\mathbf{u}_1, ..., \mathbf{u}_N]$ and $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1, ...\lambda_N)$, then we have

$$\boldsymbol{\Sigma}\mathbf{U} = \mathbf{U}\boldsymbol{\Lambda} \tag{108}$$

Since $\mathbf{U}$ is orthonormal:

$$\boldsymbol{\Sigma} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T \tag{109}$$

The inverse of $\boldsymbol{\Sigma}$ is straightforward, since $\mathbf{U}$ is orthonormal, and hence $\mathbf{U}^{-1} = \mathbf{U}^T$:

$$\boldsymbol{\Sigma}^{-1} = \left(\mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T\right)^{-1} = \mathbf{U}\boldsymbol{\Lambda}^{-1}\mathbf{U}^T \tag{110}$$

(If any of these steps are not familiar to you, you should refresh your memory of them.)

Now, consider the negative log of the Gaussian (i.e., the exponent); i.e., let

$$f(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \; . \tag{111}$$

Substituting in the diagonalization gives:

$$f(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \mathbf{U}\boldsymbol{\Lambda}^{-1}\mathbf{U}^T (\mathbf{x} - \boldsymbol{\mu}) \tag{112}$$

$$= \frac{1}{2}\mathbf{z}^T\mathbf{z} \tag{113}$$

where

$$\mathbf{z} = \mathrm{diag}(\lambda_1^{-\frac{1}{2}}, ..., \lambda_N^{-\frac{1}{2}})\mathbf{U}^T(\mathbf{x} - \boldsymbol{\mu}) \tag{114}$$

This new function $f(\mathbf{z}) = \mathbf{z}^T\mathbf{z}/2 = \sum_i z_i^2/2$ is a quadratic, with new variables z_i. Given variables $\mathbf{x}$, we can convert them to the $\mathbf{z}$ representation by applying Eq. 114, and, if all eigenvalues are

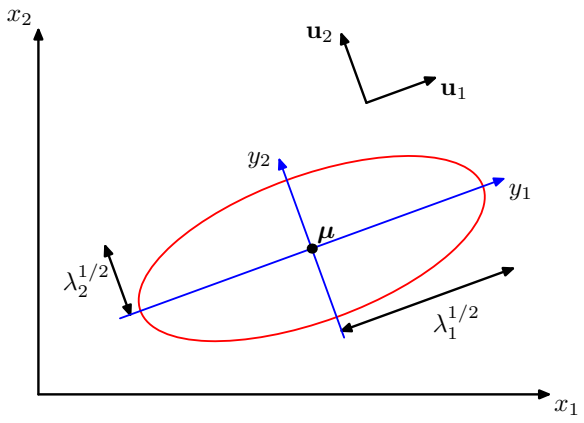

Figure 7: The red curve shows the elliptical surface of constant probability density for a Gaussian in a two-dimensional space on which the density is $\exp(-1/2)$ of its value at $\mathbf{x} = \boldsymbol{\mu}$. The major axes of the ellipse are defined by the eigenvectors $\mathbf{u}_i$ of the covariance matrix, with corresponding eigenvalues λ_i. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)(Note y_1 and y_2 in the figure should read z_1 and z_2.)

nonzero, we can convert back by inverting Eq. 114. Hence, we can write our Gaussian in this new coordinate system as[5]:

$$\frac{1}{\sqrt{(2\pi)^N}} \exp\left(-\frac{1}{2}||\mathbf{z}||^2\right) = \prod_i \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2} z_i^2\right) \tag{115}$$

It is easy to see that for the quadratic form of $f(\mathbf{z})$, its level sets (i.e., the surfaces $f(\mathbf{z}) = c$ for constant c) are hyperspheres. Equivalently, it is clear from 115 that $\mathbf{z}$ is a Gaussian random vector with an isotropic covariance, so the different elements of $\mathbf{z}$ are uncorrelated. In other words, the value of this transformation is that we have decomposed the original N-D quadratic with many interactions between the variables into a much simpler Gaussian, composed of d independent variables. This convenient geometrical form can be seen in Figure 7. For example, if we consider an individual z_i variable in isolation (i.e., consider a slice of the function $f(\mathbf{z})$), that slice will look like a 1D bowl.

We can also understand the local curvature of f with a slightly different diagonalization. Specifically, let $\mathbf{v} = \mathbf{U}^T(\mathbf{x} - \boldsymbol{\mu})$. Then,

$$f(\mathbf{u}) = \frac{1}{2}\mathbf{v}^T \boldsymbol{\Lambda}^{-1} \mathbf{v} = \frac{1}{2} \sum_i \frac{v_i^2}{\lambda_i} \tag{116}$$

If we plot a cross-section of this function, then we have a 1D bowl shape with variance given by λ_i. In other words, the eigenvalues tell us variance of the Gaussian in different dimensions.

[5]The normalizing $|\boldsymbol{\Sigma}|$ disappears due to the nature of change-of-variables in PDFs, which we won't discuss here.

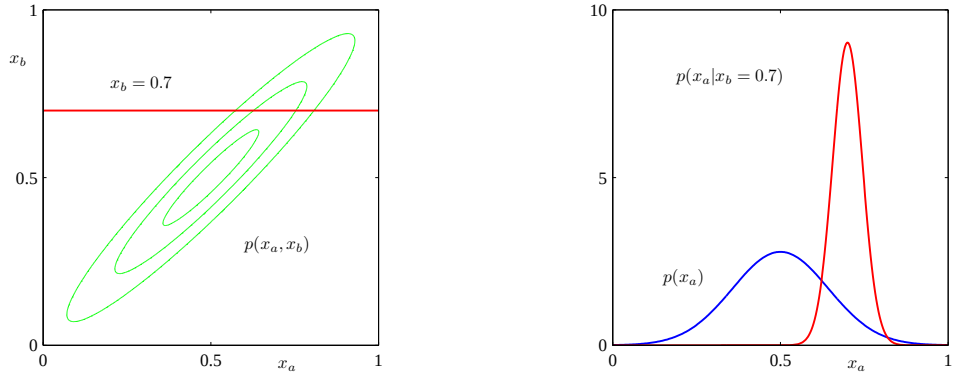

Figure 8: Left: The contours of a Gaussian distribution $p(x_a, x_b)$ over two variables. Right: The marginal distribution $p(x_a)$ (blue curve) and the conditional distribution $p(x_a|x_b)$ for $x_b = 0.7$ (red curve). (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

6.3.2 Conditional Gaussian distribution

In the case of the multivariate Gaussian where the random variables have been partitioned into two sets $\mathbf{x}_a$ and $\mathbf{x}_b$, the conditional distribution of one set conditioned on the other is Gaussian. The marginal distribution of either set is also Gaussian. When manipulating these expressions, it is easier to express the covariance matrix in inverse form, as a "precision" matrix, $\Lambda \equiv \Sigma^{-1}$. Given that $\mathbf{x}$ is a Gaussian random vector, with mean $\boldsymbol{\mu}$ and covariance Σ, we can express $\mathbf{x}$, $\boldsymbol{\mu}$, Σ and Λ all in block matrix form:

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{pmatrix}, \qquad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu_a} \\ \boldsymbol{\mu_b} \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{pmatrix}, \qquad \Lambda = \begin{pmatrix} \Lambda_{aa} & \Lambda_{ab} \\ \Lambda_{ba} & \Lambda_{bb} \end{pmatrix}, \quad (117)$$

Then one can show straightforwardly that the marginal PDFs for the components $\mathbf{x}_a$ and $\mathbf{x}_b$ are also Gaussian, i.e.,

$$\mathbf{x}_a \sim \mathcal{N}(\boldsymbol{\mu_a}, \Sigma_{aa}) \,, \qquad \mathbf{x}_b \sim \mathcal{N}(\boldsymbol{\mu_b}, \Sigma_{bb}). \quad (118)$$

With a little more work one can also show that the conditional distributions are Gaussian. For example, the conditional distribution of $\mathbf{x}_a$ given $\mathbf{x}_b$ satisfies

$$\mathbf{x}_a|\mathbf{x}_b \sim \mathcal{N}(\boldsymbol{\mu}_{a|b}, \Lambda_{aa}^{-1}) \quad (119)$$

where $\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu_a} - \Lambda_{aa}^{-1}\Lambda_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b)$. Note that Λ_{aa}^{-1} is not simply Σ_{aa}. Figure 8 shows the marginal and conditional distributions applied to a two-dimensional Gaussian.

Finally, another important property of Gaussian functions is that the product of two Gaussian functions is another Gaussian function (although no longer normalized to be a proper density function):

$$G(\mathbf{x};\, \boldsymbol{\mu}_1, \Sigma_2)\, G(\mathbf{x};\, \boldsymbol{\mu}_2, \Sigma_2) \;\propto\; G(\mathbf{x};\, \boldsymbol{\mu}, \Sigma), \quad (120)$$

where

$$\boldsymbol{\mu} = \Sigma \left(\Sigma_1^{-1} \boldsymbol{\mu_1} + \Sigma_2^{-1} \boldsymbol{\mu_2} \right), \tag{121}$$

$$\Sigma = (\Sigma_1^{-1} + \Sigma_2^{-1})^{-1}. \tag{122}$$

Note that the linear transformation of a Gaussian random variable is also Gaussian. For example, if we apply a transformation such that $\mathbf{y} = A\mathbf{x}$ where $\mathbf{x} \sim \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \Sigma)$, we have $\mathbf{y} \sim \mathcal{N}(\mathbf{y}|A\boldsymbol{\mu}, A\Sigma A^T)$.

7 Estimation

We now consider the problem of determining unknown parameters of the world based on measurements. The general problem is one of **inference**, which describes the probabilities of these unknown parameters. Given a model, these probabilities can be derived using Bayes' Rule. The simplest use of these probabilities is to perform **estimation**, in which we attempt to come up with single "best" estimates of the unknown parameters.

7.1 Learning a binomial distribution

For a simple example, we return to coin-flipping. We flip a coin N times, with the result of the i-th flip denoted by a variable $\mathbf{c}_i$: "$\mathbf{c}_i = \mathsf{heads}$" means that the i-th flip came up heads. The probability that the coin lands heads on any given trial is given by a parameter θ. We have no prior knowledge as to the value of θ, and so our prior distribution on θ is uniform.[6] In other words, we describe θ as coming from a uniform distribution from 0 to 1, so $p(\theta) = 1$; we believe that all values of θ are equally likely if we have not seen any data. We further assume that the individual coin flips are independent, i.e., $P(\mathbf{c}_{1:N}|\theta) = \prod_i p(\mathbf{c}_i|\theta)$. (The notation "$\mathbf{c}_{1:N}$" indicates the set of observations $\{\mathbf{c}_1, ..., \mathbf{c}_N\}$.) We can summarize this model as follows:

Model: Coin-Flipping		
θ	$\sim$	$\mathcal{U}(0,1)$
$P(\mathbf{c} = \mathsf{heads})$	$=$	θ
$P(\mathbf{c}_{1:N}\|\theta)$	$=$	$\prod_i p(\mathbf{c}_i\|\theta)$

(123)

Suppose we wish to learn about a coin by flipping it 1000 times and observing the results $\mathbf{c}_{1:1000}$, where the coin landed heads 750 times? What is our belief about θ, given this data? We now need to solve for $p(\theta|\mathbf{c}_{1:1000})$, i.e., our belief about θ *after* seeing the 1000 coin flips. To do this, we apply the basic rules of probability theory, beginning with the Product Rule:

$$P(\mathbf{c}_{1:1000}, \theta) \;=\; P(\mathbf{c}_{1:1000}|\theta)\, p(\theta) \;=\; p(\theta|\mathbf{c}_{1:1000})\, P(\mathbf{c}_{1:1000}) \tag{124}$$

Solving for the desired quantity gives:

$$p(\theta|\mathbf{c}_{1:1000}) \;\;=\;\; \frac{P(\mathbf{c}_{1:1000}|\theta)p(\theta)}{P(\mathbf{c}_{1:1000})} \tag{125}$$

The numerator may be written using

$$P(\mathbf{c}_{1:1000}|\theta)\, p(\theta) \;=\; \prod_i P(\mathbf{c}_i|\theta) \;=\; \theta^{750}(1-\theta)^{1000-750} \tag{126}$$

[6] We would usually expect a coin to be fair, i.e., the prior distribution for θ is peaked near 0.5.

Figure 9: Posterior probability of θ from two different experiments: one with a single coin flip (landing heads), and 1000 coin flips (750 of which land heads). Note that the latter distribution is much more peaked.

The denominator may be solved for by the marginalization rule:

$$P(\mathbf{c}_{1:1000}) \;=\; \int_0^1 P(\mathbf{c}_{1:1000}, \theta) d\theta \;=\; \int_0^1 \theta^{750}(1-\theta)^{1000-750} d\theta \;=\; Z \tag{127}$$

where Z is a constant (evaluating it requires more advanced math, but it is not necessary for our purposes). Hence, the final probability distribution is:

$$p(\theta|\mathbf{c}_{1:1000}) \;=\; \theta^{750}(1-\theta)^{1000-750}/Z \tag{128}$$

which is plotted in Figure 9. This form gives a probability distribution over θ that expresses our belief about θ *after* we've flipped the coin 1000 times.

Suppose we just take the peak of this distribution; from the graph, it can be seen that the peak is at $\theta = .75$. This makes sense: if a coin lands heads 75% of the time, then we would probably estimate that it will land heads 75% of the time of the future. More generally, suppose the coin lands heads H times out of N flips; we can compute the peak of the distribution as follows:

$$\arg\max_\theta p(\theta|\mathbf{c}_{1:N}) \;=\; H/N \tag{129}$$

(Deriving this is a good exercise to do on your own; hint: minimize the negative log of $p(\theta|\mathbf{c}_{1:N})$).

7.2 Bayes' Rule

In general, given that we have a model of the world described by some unknown variables, and we observe some data; our goal is to determine the model from the data. (In coin-flip example, the model consisted of the likelihood of the coin landing heads, and the prior over θ, while the data consisted of the results of N coin flips.) We describe the probability model as $p(\mathsf{data}|\mathsf{model})$ — if we knew model, then this model will tell us what data we expect. Furthermore, we must have some prior beliefs as to what model is ($p(\mathsf{model})$), even if these beliefs are completely non-committal (e.g., a uniform distribution). Given the data, what do we know about model?

Applying the product rule as before gives:

$$p(\mathsf{data}, \mathsf{model}) = p(\mathsf{data}|\mathsf{model})p(\mathsf{model}) = p(\mathsf{model}|\mathsf{data})p(\mathsf{data}) \tag{130}$$

Solving for the desired distribution, gives a seemingly simple but powerful result, known widely as **Bayes' Rule:**

Bayes' Rule:
$$p(\mathsf{model}|\mathsf{data}) = \frac{p(\mathsf{data}|\mathsf{model})p(\mathsf{model})}{p(\mathsf{data})}$$

The different terms in Bayes' Rule are used so often that they all have names:

$$\underbrace{p(\mathsf{model}|\mathsf{data})}_{\text{posterior}} = \frac{\overbrace{P(\mathsf{data}|\mathsf{model})}^{\text{likelihood}}\overbrace{p(\mathsf{model})}^{\text{prior}}}{\underbrace{p(\mathsf{data})}_{\text{evidence}}} \tag{131}$$

- The **likelihood** distribution describes the likelihood of data given model — it reflects our assumptions about how the data **c** was generated.
- The **prior distribution** describes our assumptions about model before observing the data data.
- The **posterior distribution** describes our knowledge of model, incorporating both the data and the prior.
- The **evidence** is useful in model selection, and will be discussed later. Here, its only role is to normalize the posterior PDF.

7.3 Parameter estimation

Quite often, we are interested in finding a single estimate of the value of an unknown parameter, even if this means discarding all uncertainty. This is called **estimation:** determining the values

of some unknown variables from observed data. In this chapter, we outline the problem, and describe some of the main ways to do this, including Maximum A Posteriori (MAP), and Maximum Likelihood (ML). Estimation is the most common form of learning — given some data from the world, we wish to "learn" how the world behaves, which we will describe in terms of a set of unknown variables.

Strictly speaking, parameter estimation is not justified by Bayesian probability theory, and can lead to a number of problems, such as overfitting and nonsensical results in extreme cases. Nonetheless, it is widely used in many problems.

7.3.1 MAP, ML, and Bayes' Estimates

We can now define the MAP learning rule: choose the parameter value θ that maximizes the posterior, i.e.,

$$\hat{\theta} = \arg\max_{\theta} p(\theta|\mathcal{D}) \tag{132}$$

$$= \arg\max_{\theta} P(\mathcal{D}|\theta)p(\theta) \tag{133}$$

Note that we don't need to be able to evaluate the evidence term $p(\mathcal{D})$ for MAP learning, since there are no θ terms in it.

Very often, we will assume that we have no prior assumptions about the value of θ, which we express as a **uniform prior**: $p(\theta)$ is a uniform distribution over some suitably large range. In this case, the $p(\theta)$ term can also be ignored from MAP learning, and we are left with only maximizing the likelihood. Hence, the **Maximum Likelihood** (ML) learning principle (i.e., estimator) is

$$\hat{\theta}_{ML} = \arg\max_{\theta} P(\mathcal{D}|\theta) \tag{134}$$

It often turns out that it is more convenient to minimize the negative-log of the objective function. Because "$-\ln$" is a monotonic decreasing function, we can pose MAP estimation as:

$$\hat{\theta}_{MAP} = \arg\max_{\theta} P(\mathcal{D}|\theta)p(\theta) \tag{135}$$

$$= \arg\min_{\theta} -\ln\left(P(\mathcal{D}|\theta)p(\theta)\right) \tag{136}$$

$$= \arg\min_{\theta} -\ln P(\mathcal{D}|\theta) - \ln p(\theta) \tag{137}$$

We can see that the objective conveniently breaks into a part corresponding to the likelihood and a part corresponding to the prior.

One problem with this approach is that all model uncertainty is ignored. We are choosing to put all our faith in the most probable model. This sometimes has surprising and undesirable consequences. For example, in the coin tossing example above, if one were to flip a coin just once and see a head, then the estimator in Eqn. (129) would tell us that the probability of the outcome being heads is 1. Sometimes a more suitable estimator is the expected value of the posterior distribution, rather than its maximum. This is called the **Bayes' estimate**.

In the coin tossing case above, you can show that the expected value of θ, under the posterior provides an estimate of the probability that is biased toward 1/2. That is:

$$\int_0^1 p(\theta|\mathbf{c}_{1:N})\,\theta\,d\theta \;=\; \frac{H+1}{N+2} \tag{138}$$

You can see that this value is always somewhat biased toward 1/2, but converges to the MAP estimate as N increases. Interestingly, even when there are is no data whatsoever, in which case the MAP estimate is undefined, the Bayes' estimate is simply 1/2.

7.4 Learning Gaussians

We now consider the problem of learning a Gaussian distribution from N training samples $\mathbf{x}_{1:N}$. Maximum likelihood learning of the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ entails maximizing the likelihood:

$$p(\mathbf{x}_{1:N}|\boldsymbol{\mu},\boldsymbol{\Sigma}) \tag{139}$$

We assume here that the data points come from a Gaussian. We further assume that they are drawn independently. We can therefore write the joint likelihood over the entire set of data as the produce of the likelihoods for each individual datum, i.e.,

$$\begin{aligned} p(\mathbf{x}_{1:N}|\boldsymbol{\mu},\boldsymbol{\Sigma}) &= \prod_{i=1}^{N} p(\mathbf{x}_{1:N}|\boldsymbol{\mu},\boldsymbol{\Sigma}) && (140)\\ &= \prod_{i=1}^{N} \frac{1}{\sqrt{(2\pi)^M|\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x}_i-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i-\boldsymbol{\mu})\right), && (141)\end{aligned}$$

where M is the dimensionality of the data $\mathbf{x}_i$. It is somewhat more convenient to minimize the negative log-likelihood:

$$\begin{aligned} L(\boldsymbol{\mu},\boldsymbol{\Sigma}) &\equiv -\ln p(\mathbf{x}_{1:N}|\boldsymbol{\mu},\boldsymbol{\Sigma}) && (142)\\ &= -\sum_i \ln p(\mathbf{x}_i|\boldsymbol{\mu},\boldsymbol{\Sigma}) && (143)\\ &= \sum_i \frac{(\mathbf{x}_i-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i-\boldsymbol{\mu})}{2} + \frac{N}{2}\ln|\boldsymbol{\Sigma}| + \frac{NM}{2}\ln(2\pi) && (144)\end{aligned}$$

Solving for $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ by setting $\partial L/\partial\boldsymbol{\mu} = 0$ and $\partial L/\partial\boldsymbol{\Sigma} = 0$ (subject to the constraint that $\boldsymbol{\Sigma}$ is symmetric) gives the maximum likelihood estimates[7]:

$$\begin{aligned} \boldsymbol{\mu}^* &= \frac{1}{N}\sum_i \mathbf{x}_i && (145)\\ \boldsymbol{\Sigma}^* &= \frac{1}{N}\sum_i (\mathbf{x}_i-\boldsymbol{\mu}^*)(\mathbf{x}_i-\boldsymbol{\mu}^*)^T && (146)\end{aligned}$$

[7]Warning: the calculation for the optimal covariance matrix involves Lagrange multipliers and is not easy.

The ML estimates make intuitive sense; we estimate the Gaussian's mean to be the sample mean of the data, and the Gaussian's covariance to be the sample covariance of the data. Maximum likelihood estimates usually make sense intuitively. This is very helpful when debugging your math — you can sometimes find bugs in derivations simply because the ML estimates do not look right.

7.5 MAP nonlinear regression

Let us revisit the nonlinear regression model from Section 3.1, but now admitting that there exists noise in measurements and modelling errors. We'll now write the model as

$$y = \mathbf{w}^T \mathbf{b}(\mathbf{x}) + n \tag{147}$$

where n is a Gaussian random variable, i.e.,

$$n \sim \mathcal{N}(0, \sigma^2) \; . \tag{148}$$

We add this random variable to the regression equation in (147) to represent the fact that most models and most measurements involve some degree of error. We'll refer to this error as *noise*.

It is straightforward to show from basic probability theory that Equation (147) implies that, given $\mathbf{x}$ and $\mathbf{w}$, y is also Gaussian (i.e., has a Gaussian density), i.e.,

$$p(y \,|\, \mathbf{x}, \mathbf{w}) \;=\; G(y;\, \mathbf{w}^T \mathbf{b}(\mathbf{x}),\, \sigma^2) \;\equiv\; \frac{1}{\sqrt{2\pi}\sigma}\, e^{-(y - \mathbf{w}^T \mathbf{b}(\mathbf{x}))^2 / 2\sigma^2} \tag{149}$$

(G is defined in the previous chapter.) It follows that, for a collection of N independent training points, $(y_{1:N}, \mathbf{x}_{1:N})$, the likelihood is given by

$$\begin{aligned} p(y_{1:N} \,|\, \mathbf{w}, \mathbf{x}_{1:N}) \;&=\; \prod_{i=1}^{N} G(y_i;\, \mathbf{w}^T \mathbf{b}(\mathbf{x}_i),\, \sigma^2) \\ &=\; \frac{1}{(2\pi\sigma^2)^{N/2}} \, \exp\left(- \sum_{i=1}^{N} \frac{(y_i - \mathbf{w}^T \mathbf{b}(\mathbf{x}_i))^2}{2\sigma^2} \right) \end{aligned} \tag{150}$$

Furthermore, let us assume the following (weight decay) prior distribution over the unknown weights $\mathbf{w}$:

$$\mathbf{w} \sim \mathcal{N}(0, \alpha \mathbf{I}) \; . \tag{151}$$

That is, for $\mathbf{w} \in \mathbb{R}^M$,

$$p(\mathbf{w}) \;=\; \prod_{k=1}^{M} \frac{1}{\sqrt{2\pi\alpha}}\, e^{-w_k^2 / 2\alpha} \;=\; \frac{1}{(2\pi\alpha)^{M/2}}\, e^{-\mathbf{w}^T \mathbf{w} / 2\alpha} \; . \tag{152}$$

Now, to estimate the model parameters (i.e., $\mathbf{w}$), let's consider the posterior distribution over $\mathbf{w}$ conditioned on our N training pairs, $(\mathbf{x}_i, y_i)$. Based on the formulation above, assuming independent training samples, it follows that

$$p(\mathbf{w}|y_{1:N}, \mathbf{x}_{1:N}) = \frac{p(y_{1:N}|\mathbf{w}, \mathbf{x}_{1:N})p(\mathbf{w}|\mathbf{x}_{1:N})}{p(y_{1:N}|\mathbf{x}_{1:N})} \quad (153)$$

$$= \frac{\left(\prod_i p(y_i|\mathbf{w}, \mathbf{x}_i)\right) p(\mathbf{w})}{p(y_{1:N}|\mathbf{x}_{1:N})} \,. \quad (154)$$

Note that $p(\mathbf{w}|\mathbf{x}_{1:N}) = p(\mathbf{w})$, since we can assume that $\mathbf{x}$ alone provides no information about $\mathbf{w}$.

In MAP estimation, we want to find the parameters $\mathbf{w}$ that maximize their posterior probability:

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} p(\mathbf{w}|y_{1:N}, \mathbf{x}_{1:N}) \quad (155)$$

$$= \arg\min_{\mathbf{w}} -\ln p(\mathbf{w}|y_{1:N}, \mathbf{x}_{1:N}) \quad (156)$$

The negative log-posterior is:

$$L(\mathbf{w}) = -\ln p(\mathbf{w}|y_{1:N}, \mathbf{x}_{1:N}) \quad (157)$$

$$= \left(\sum_i \frac{1}{2\sigma^2}(y_i - \mathbf{w}^T\mathbf{b}(\mathbf{x}_i))^2\right) + \frac{N}{2}\ln(2\pi\sigma^2) \quad (158)$$

$$+\frac{1}{2\alpha}||\mathbf{w}||^2 + \frac{M}{2}\ln(2\pi\alpha) + \ln p(y_{1:N}|\mathbf{x}_{1:N}) \quad (159)$$

Now, we can discard terms that do not depend on $\mathbf{w}$, since they are irrelevant for optimization:

$$L(\mathbf{w}) = \left(\sum_i \frac{1}{2\sigma^2}(y_i - \mathbf{w}^T\mathbf{b}(\mathbf{x}_i))^2\right) + \frac{1}{2\alpha}||\mathbf{w}||^2 + \text{constants} \quad (160)$$

Furthermore, we can multiply by a constant, without changing where the optima are, so let us multiply the whole expression by $2\sigma^2$. Then, if we define $\lambda = \sigma^2/\alpha$, we have the exact same objective function as used in nonlinear regression with regularization. Hence, nonlinear least-squares with regularization is a form of MAP estimation, and can be optimized the same way. When the measurements are very reliable, then σ is small and we give the regularizer less influence on the estimate. But when the data are relatively noisy, so σ is larger, then regularizer has more influence.

8 Classification

In classification, we are trying to learn a map from an input space to some finite output space. In the simplest case we simply detect whether or not the input has some property or not. For example, we might want to determine whether or not an email is spam, or whether an image contains a face. A task in the health care field is to determine, given a set of observed symptoms, whether or not a person has a disease. These detection tasks are *binary classification* problems.

In *multi-class classification* problems we are interested in determining to which of multiple categories the input belongs. For example, given a recorded voice signal we might wish to recognize the identity of a speaker (perhaps from a set of people whose voice properties are given in advance). Another well studied example is optical character recognition, the recognition of letters or numbers from images of handwritten or printed characters.

The input $\mathbf{x}$ might be a vector of real numbers, or a discrete feature vector. In the case of binary classification problems the output y might be an element of the set $\{-1, 1\}$, while for a multi-dimensional classification problem with N categories the output might be an integer in $\{1, \ldots, N\}$.

The general goal of classification is to learn a *decision boundary*, often specified as the level set of a function, e.g., $a(\mathbf{x}) = 0$. The purpose of the decision boundary is to identity the regions of the input space that correspond to each class. For binary classification the decision boundary is the surface in the feature space that separates the test inputs into two classes; points $\mathbf{x}$ for which $a(\mathbf{x}) < 0$ are deemed to be in one class, while points for which $a(\mathbf{x}) > 0$ are in the other. The points on the decision boundary, $a(\mathbf{x}) = 0$, are those inputs for which the two classes are equally probable.

In this chapter we introduce several basis methods for classification. We focus mainly on on binary classification problems for which the methods are conceptually straightforward, easy to implement, and often quite effective. In subsequent chapters we discuss some of the more sophisticated methods that might be needed for more challenging problems.

8.1 Class Conditionals

One approach is to describe a "generative" model for each class. Suppose we have two mutually-exclusive classes C_1 and C_2. The prior probability of a data vector coming from class C_1 is $P(C_1)$, and $P(C_2) = 1 - P(C_1)$. Each class has a distribution for its data: $p(\mathbf{x}|C_1)$, and $p(\mathbf{x}|C_2)$. In other words, to sample from this model, we would first randomly choose a class according to $P(C_1)$, and then sample a data vector $\mathbf{x}$ from that class.

Given labeled training data $\{(\mathbf{x}_i, y_i)\}$, we can estimate the distribution for each class by maximum likelihood, and estimate $P(C_1)$ by computing the ratio of the number of elements of class 1 to the total number of elements.

Once we have trained the parameters of our *generative* model, we perform classification by comparing the posterior class probabillities:

$$P(C_1|\mathbf{x}) > P(C_2|\mathbf{x}) \; ? \tag{161}$$

That is, if the posterior probability of C_1 is larger than the probability of C_2, then we might classify the input as belonging to class 1. Equivalently, we can compare their ratio to 1:

$$\frac{P(C_1|\mathbf{x})}{P(C_2|\mathbf{x})} > 1 \; ? \tag{162}$$

If this ratio is greater than 1 (i.e. $P(C_1|\mathbf{x}) > P(C_2|\mathbf{x})$) then we classify $\mathbf{x}$ as belonging to class 1, and class 2 otherwise.

The quantities $P(C_i|\mathbf{x})$ can by computed using Bayes' Rule as:

$$P(C_i|\mathbf{x}) = \frac{p(\mathbf{x}|C_i)\, P(C_i)}{p(\mathbf{x})} \tag{163}$$

so that the ratio is:

$$\frac{p(\mathbf{x}|C_1)\, P(C_1)}{p(\mathbf{x}|C_2)\, P(C_2)} \tag{164}$$

Note that the $p(\mathbf{x})$ terms cancel and so do not need to be computed. Also, note that these computations are typically done in the logarithmic domain as this is often faster and more numerically stable.

Gaussian Class Conditionals. As a concrete example, consider a generative model in which the inputs associated with the i^{th} class (for $i = 1, 2$) are modeled with a Gaussian distribution, i.e.,

$$p(\mathbf{x}|C_i) = G(\mathbf{x}; \mu_i, \mathbf{\Sigma}_i) \; . \tag{165}$$

Also, let's assume that the prior class probabilities are equal:

$$P(C_i) = \frac{1}{2} \; . \tag{166}$$

The values of μ_i and $\mathbf{\Sigma}_i$ can be estimated by maximum likelihood on the individual classes in the training data.

Given this models, you can show that the log of the posterior ratio (164) is given by

$$a(\mathbf{x}) = -\frac{1}{2}(\mathbf{x} - \mu_1)^T \mathbf{\Sigma}_1^{-1}(\mathbf{x} - \mu_1) - \frac{1}{2}\ln|\mathbf{\Sigma}_1| + \frac{1}{2}(\mathbf{x} - \mu_2)^T \mathbf{\Sigma}_2^{-1}(\mathbf{x} - \mu_2) + \frac{1}{2}\ln|\mathbf{\Sigma}_2| \tag{167}$$

The sign of this function determines the class of $\mathbf{x}$, since the ratio of posterior class probabilities is greater than 1 when this log is greater than zero. Since $a(\mathbf{x})$ is quadratic in $\mathbf{x}$, the decision boundary (i.e., the set of points satisfying $a(\mathbf{x}) = 0$) is a conic section (e.g., a parabola, an ellipse, a line, etc.). Furthermore, in the special case where $\mathbf{\Sigma}_1 = \mathbf{\Sigma}_2$, the decision boundary is linear (why?).

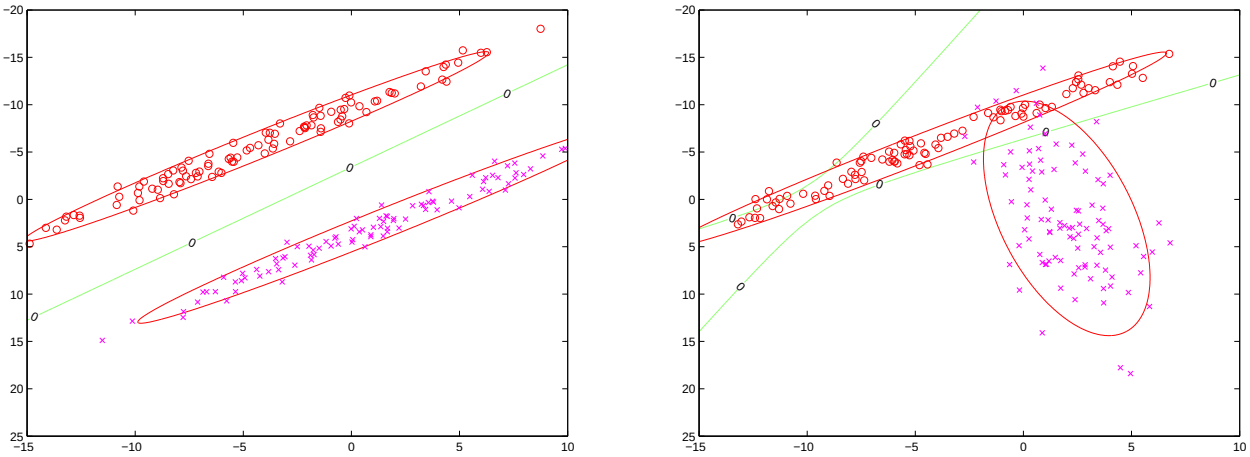

Figure 10: GCC classification boundaries for two cases. Note that the decision boundary is linear when both classes have the same covariance.

8.2 Logistic Regression

Noting that $p(\mathbf{x})$ can be written as (why?)

$$p(\mathbf{x}) \;=\; p(\mathbf{x}, C_1) + p(\mathbf{x}, C_2) \;=\; p(\mathbf{x}|C_1)P(C_1) + p(\mathbf{x}|C_2)P(C_2)\;, \tag{168}$$

we can express the posterior class probability as

$$P(C_1|\mathbf{x}) = \frac{p(\mathbf{x}|C_1)P(C_1)}{p(\mathbf{x}|C_1)P(C_1) + p(\mathbf{x}|C_2)P(C_2)}\;. \tag{169}$$

Dividing both the numerator and denominator by $p(\mathbf{x}|C_1)P(C_1)$ we obtain:

$$P(C_1|\mathbf{x}) \;=\; \frac{1}{1+e^{-a(\mathbf{x})}} \tag{170}$$

$$\;=\; g(a(\mathbf{x})) \tag{171}$$

where $a(\mathbf{x}) = \ln \frac{p(\mathbf{x}|C_1)P(C_1)}{p(\mathbf{x}|C_2)P(C_2)}$ and $g(a)$ is the sigmoid function. Note that $g(a)$ is monotonic, so that the probability of class C_1 grows as a grows and is precisely $\frac{1}{2}$ when $a = 0$. Since $P(C_1|\mathbf{x}) = \frac{1}{2}$ represents equal probability for both classes, this is the boundary along which we wish to make decisions about class membership.

For the case of Gaussian class conditionals where both Gaussians have the same covariance, a is a linear function of $\mathbf{x}$. In this case the classification probability can be written as

$$P(C_1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}} = g(\mathbf{w}^T\mathbf{x}+b)\;, \tag{172}$$

or, if we augment the data vector with a 1 and the weight vector with b,

$$P(C_1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}\;. \tag{173}$$

At this point, we can forget about the generative model (e.g., the Gaussian distributions) that we started with, and **use this as our entire model.** In other words, rather than learning a distribution over each class, we learn only **the conditional probability of y given $\mathbf{x}$**. As a result, we have fewer parameters to learn since the number of parameters in logistic regression is linear in the dimension of the input vector, while learning a Gaussian covariance requires a quadratic number of parameters. With fewer parameters we can learn models more effectively with less data. On the other hand, we cannot perform other tasks that we could with the generative model (e.g., sampling from the model; classify data with noisy or missing measurements).

We can learn logistic regression with maximum likelihood. In particular, given data $\{\mathbf{x}_i, y_i\}$, we minimize the negative log of:

$$\begin{aligned} p(\{\mathbf{x}_i, y_i\}|\mathbf{w}, b) &\propto p(\{y_i\} \,|\, \{\mathbf{x}_i\}, \mathbf{w}, b) \\ &= \prod_i p(y_i|\mathbf{x}_i, \mathbf{w}, b) \\ &= \prod_{i:y_i=C_1} P(C_1|\mathbf{x}_i) \prod_{i:y_i=C_2} (1 - P(C_1|\mathbf{x}_i)) \end{aligned} \tag{174}$$

In the first step above we have assumed that the input features are independent of the weights in the logistic regressor, i.e., $p(\{\mathbf{x}_i\}) = p(\{\mathbf{x}_i\}|\mathbf{w}, b)$. So this term can be ignored in the likelihood since it is constant with respect to the unknowns. In the second step we have assumed that the input-output pairs are independent, so the joint likelihood is the product of the likelihoods for each input-output pair.

The decision boundary for logistic regression is linear; in 2D, it is a line. To see this, recall that the decision boundary is the set of points $P(C_1|\mathbf{x}) = 1/2$. Solving for $\mathbf{x}$ gives the points $\mathbf{w}^T\mathbf{x} + b = 0$, which is a line in 2D, or a hyperplane in higher dimensions.

Although this objective function cannot be optimized in closed-form, it is convex, which means that it has a single minimum. Therefore, we can optimize it with gradient descent (or any other gradient-based search technique), which will be guaranteed to find the global minimum.

If the classes are linearly separable, this approach will lead to very large values of the weights $\mathbf{w}$, since as the magnitude of $\mathbf{w}$ tends to infinity, the function $g(a(x))$ behaves more and more like a step function and thus assigns higher likelihood to the data. This can be prevented by placing a weight-decay prior on $\mathbf{w}$: $p(\mathbf{w}) = G(\mathbf{w};\, 0, \sigma^2)$.

Multiclass classification. Logistic regression can also be applied to multiclass classification, i.e., where we wish to classify a data point as belonging to one of K classes. In this case, the probability of data vector $\mathbf{x}$ being in class i is:

$$P(C_i|\mathbf{x}) = \frac{e^{-\mathbf{w}_i^T\mathbf{x}}}{\sum_{k=1}^{K} e^{-\mathbf{w}_k^T\mathbf{x}}} \tag{175}$$

You should be able to see that this is equivalent to the method described above in the two-class case. Furthermore, it is straightforward to show that this is a sensible choice of probability: $0 \leq P(C_i|\mathbf{x})$, and $\sum_k P(C_k|\mathbf{x}) = 1$ (verify these for yourself).

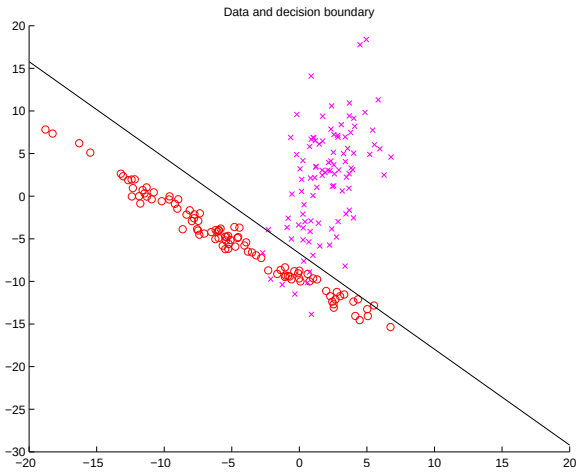

Figure 11: Classification boundary for logistic regression.

8.3 Artificial Neural Networks

Logistic regression works for linearly separable datasets, but may not be sufficient for more complex cases. We can generalize logistic regression by replacing the linear function $\mathbf{w}^T\mathbf{x} + b$ with any other function. If we replace it with a neural network, we get:

$$P(C_1|\mathbf{x}) = g\left(\sum_j w_j^{(1)} g\left(\sum_k w_{k,j}^{(2)} x_k + b_j^{(2)}\right) + b^{(1)}\right) \tag{176}$$

This representation is no longer connected to any particular choice of class-conditional model; it is purely a model of the class probability given the measurement.

8.4 K-Nearest Neighbors Classification

We can apply the KNN idea to classification as well. For class labels $\{-1, 1\}$, the classifier is:

$$y_{new} = \text{sign}\left(\sum_{i \in N_K(\mathbf{x})} y_i\right) \tag{177}$$

where

$$\text{sign}(z) = \begin{cases} -1 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{178}$$

Alternatively, we might take a weighted average of the K-nearest neighbors:

$$y = \text{sign}\left(\sum_{i \in N_K(\mathbf{x})} w(\mathbf{x}_i) y_i\right), \quad w(\mathbf{x}_i) = e^{-||\mathbf{x}_i - \mathbf{x}||^2/2\sigma^2} \tag{179}$$

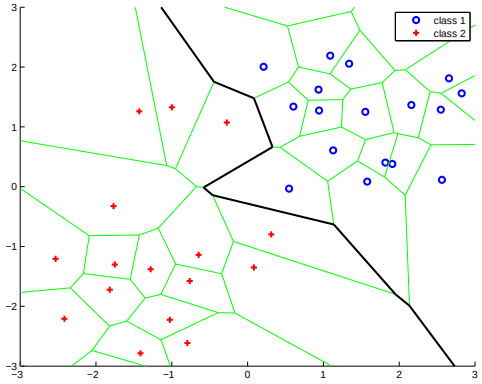

Figure 12: For two classes and planar inputs, the decision boundary for a 1NN classifiifier (the bold black curve) is a subset of the perpendicular bisecting line segments (green) between pairs of neighbouring points (obtained with a Voronoi tesselation).

where σ^2 is an additional parameter to the algorithm.

For KNN the decision boundary will be a collections of hyperplane patches that are perpendicular bisectors of pairs of points drawn from the two classes. As illustrated in Figure 12, this is a set of bisecting line segments for 2D inputs. Figure 12, shows a simple case but it is not hard to imagine that the decision surfaces can get very complex, e.g., if a point from class 1 lies somewhere in the middle of the points from class 2. By increasing the number of nearest neighbours (i.e., K) we are effectively smoothing the decision boundary, hopefully thereby improving generalization.

8.5 Generative vs. Discriminative models

The classifiers described here illustrate a distinction between two general types of models in machine learning:

1. **Generative models**, such as the GCC, describe the complete probability of the data $p(\mathbf{x}, y)$.

2. **Discriminative models**, such as LR, ANNs, and KNN, describe the conditional probability of the output given the input: $p(y|\mathbf{x})$

The same distinction occurs in regression and classification, e.g., KNN is a discriminative method that can be used for either classification or regression.

The distinction is clearest when comparing LR with GCC with equal covariances, since they are both linear classifiers, but the training algorithms are different. This is because they have different goals; LR is optimized for classification performance, where as the GCC is a "complete" model of the probability of the data that is then pressed into service for classification. As a consequence, GCC may perform poorly with non-Gaussian data. Conversely, LR is not premised on any particular form of distribution for the two class distributions. On the other hand, LR can **only** be

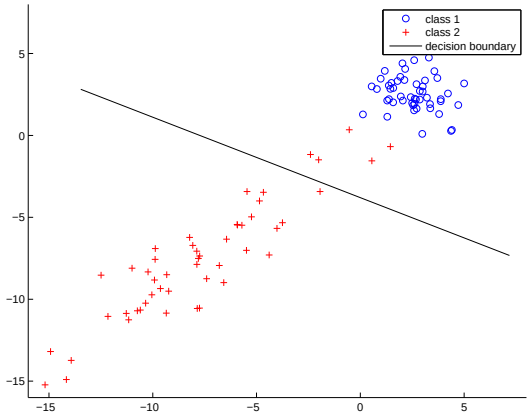

Figure 13: In this example there are two classes, one with a small isotropic covariance, and one with an anistropic covariance. One can clearly see that the data are linearly separable (i.e., a line exists that correctly separates the input training samples). Despite this, LS regression does not separate the training data well. Rather, the LS regression decision boundary produces 5 incorrectly classified training points.

used for classification, whereas the GCC can be used for other tasks, e.g., to sample new $\mathbf{x}$ data, to classify noisy inputs or inputs with outliers, and so on.

The distinctions between generative and discriminative models become more significant in more complex problems. Generative models allow us to put more prior knowledge into how we build the model, but classification may often involve difficult optimization of $p(y|\mathbf{x})$; discriminative methods are typically more efficient and generic, but are harder to specialize to particular problems.

8.6 Classification by LS Regression

One tempting way to perform classification is with least-squares rgression. That is, we could treat the class labels $y \in \{-1, 1\}$ as real numbers, and estimate the weights by minimizing

$$E(\mathbf{w}) = \sum_i (y_i - \mathbf{x}_i^T \mathbf{w})^2 \,, \tag{180}$$

for labeled training data $\{\mathbf{x}_i, y_i\}$. Given the optimal regression weights, one could then perform regression on subsequent test inputs and use the sign of the output to determine the output class.

In simple cases this can perform well, but in general it will perform poorly. This is because the objective function in linear regression measures the distance from the modeled class labels (which can be any real number) to the true class labels, which may not provide an accurate measure of how well the model has classified the data. For example, a linear regression model will tend to produce predicted labels that lie outside the range of the class labels for "extreme" members of a given class (e.g. 5 when the class label is 1), causing the error to be measured as high even when the classification (given, say, by the sign of the predicted label) is correct. In such a case the decision

boundary may be shifted towards such an extreme case, potentially reducing the number of correct classifications made by the model. Figure 13 demonstrates this with a simple example.

The problem arises from the fact that the constraint that $y \in (-1, 1)$ is not built-in to the model (the regression algorithm knows nothing about it), and so wastes considerable representational power trying to reproduce this effect. It is much better to build this constraint into the model.

8.7 Naïve Bayes

One problem with class conditional models, as described above, concerns the large number of parameters required to learn the likelihood model, i.e., the distribution over the inputs conditioned on the class. In Gaussian Class Conditional models, with d-dimensional input vectors, we need to estimate the class mean and class covariance matrix for each class. The mean will be a d-dimensional vector, but the number of unknowns in the covariance matrix grows quadratically with d. That is, the covariance is a $d \times d$ matrix (although because it is symmetric we do not need to estimate all d^2 elements).

Naïve Bayes aims to simplify the estimation problem by assuming that the different input features (e.g., the different elements of the input vector), are conditionally independent. That is, they are assumed to be independent when conditioned on the class. Mathematically, for inputs $\mathbf{x} \in \mathbb{R}^d$, we express this as

$$p(\mathbf{x}|C) = \prod_{i=1}^{d} p(x_i|C) \; . \tag{181}$$

With this assumption, rather than estimating one d-dimensional density, we instead estimate d 1-dimensional densities. This is important because each 1D Gaussian only has two parameters, its mean and variance, both of which are scalars. So the model has $2d$ unknowns. In the Gaussian case, the Naïve Bayes model effectively replaces the general $d \times d$ covariance matrix by a diagonal matrix. There are d entries along the diagonal of the covariance matrix; the i^{th} entry is the variance of $x_i|C$. This model is not as expressive but it is much easier to estimate.

8.7.1 Discrete Input Features

Up to now, we have looked at algorithms for real-valued inputs. We now consider the Naïve Bayes classification algorithm for discrete inputs. In discrete Naïve Bayes, the inputs are a discrete set of "features", and each input either has or doesn't have each feature. For example, in document classification (including spam filtering), a feature might be the presence or absence of a particular word, and the feature vector for a document would be a list of which words the document does or doesn't have.

Each data vector is described by a list of discrete features $F_{1:D} = [F_1, ..., F_D]$. Each feature F_i has a set of possible values that it can take; to keep things simple, we'll assume that each feature is binary: $F_i \in \{0, 1\}$. In the case of document classification, each feature might correspond to the presence of a particular word in the email (e.g., if $F_3 = 1$, then the email contains the word

"business"), or another attribute (e.g., $F_4 = 1$ might mean that the mail headers appear forged). Similarly, a classifier to distinguish news stories between sports and financial news might be based on particular words and phrases such as "team," "baseball," and "mutual funds."

To understand the complexity of discrete class conditional models in general (i.e., without using the Naïve Bayes model), consider the distribution over 3 inputs, for class $C = 1$, i.e., $P(F_{1:3} \,|\, C = 1)$. (There will be another model for $C = 0$, but for our little thought experiment here we'll just consider the model for $C = 1$.) Using basic rules of probability, we find that

$$\begin{aligned} P(F_{1:3} \,|\, C = 1) &= P(F_1 \,|\, C = 1, F_2, F_3)\, P(F_2, F_3 \,|\, C = 1) \\ &= P(F_1 \,|\, C = 1, F_2, F_3)\, P(F_2 \,|\, C = 1, F_3)\, P(F_3 \,|\, C = 1) \end{aligned} \tag{182}$$

Now, given $C = 1$ we know that F_3 is either 0 or 1 (ie. it is a coin toss), and to model it we simply want to know the probability $P(F_3 = 1 \,|\, C = 1)$. Of course the probability that $F_3 = 0$ is simply $1 - P(F_3 = 1 \,|\, C = 1)$. In other words, with one parameter we can model the third factor above, $P(F_3 \,|\, C = 1)$.

Now consider the second factor $P(F_2 \,|\, C = 1, F_3)$. In this case, because F_2 depends on F_3, and there are two possible states of F_3, there are two distributions we need to model, namely $P(F_2 \,|\, C = 1, F_3 = 0)$ and $P(F_2 \,|\, C = 1, F_3 = 1)$. Acordingly, we will need two parameters, one for $P(F_2 = 1 \,|\, C = 1, F_3 = 0)$ and one for $P(F_2 = 1 \,|\, C = 1, F_3 = 1)$. Using the same logic, to model $P(F_1 \,|\, C = 1, F_2, F_3)$ will require one model parameter for each possible setting of (F_2, F_3), and of course there are 2^2 such settings. For D-dimensional binary inputs, there are $O(2^{D-1})$ parameters that one needs to learn. The number of parameters required grows prohibitively large as D increases.

The Naïve Bayes model, by comparison, only have D parameters to be learned. The assumption of Naïve Bayes is that the feature vectors are all conditionally independent given the class. The independence assumption is often very naïve, but yet the algorithm often works well nonetheless. This means that the likelihood of a feature vector for a particular class j is given by

$$P(F_{1:D}|C = j) = \prod_i P(F_i|C = j) \tag{183}$$

where C denotes a class $C \in \{1, 2, ...K\}$. The probabilities $P(F_i|C)$ are parameters of the model:

$$P(F_i = 1|C = j) = a_{i,j} \tag{184}$$

We must also define class priors $P(C = j) = b_j$.

To classify a new feature vector using this model, we choose the class with maximum probability given the features. By Bayes' Rule this is:

$$P(C = j|F_{1:D}) = \frac{P(F_{1:D}|C = j)P(C = j)}{P(F_{1:D})} \tag{185}$$

$$= \frac{\left(\prod_i P(F_i|C = j)\right) P(C = j)}{\sum_{\ell=1}^K P(F_{1:D}, C = \ell)} \tag{186}$$

$$= \frac{\left(\prod_{i:F_i=1} a_{i,j} \prod_{i:F_i=0} (1 - a_{i,j})\right) b_j}{\sum_{\ell=1}^K \left(\prod_{i:F_i=1} a_{i,\ell} \prod_{i:F_i=0} (1 - a_{i,\ell})\right) b_\ell} \tag{187}$$

If we wish to find the class with maximum posterior probability, we need only compute the numerator. The denominator in (187) is of course the same for all classes j. To compute the denominator one simply divides the numerators for each class by their sum.

The above computation involves the product of many numbers, some of which might be quite small. This can lead to underflow. For example, if you take the product $a_1 a_2 ... a_N$, and all $a_i << 1$, then the computation may evaluate to zero in floating point, even though the final computation after normalization should not be zero. If this happens for all classes, then the denominator will be zero, and you get a divide-by-zero error, even though, mathematically, the denominator cannot be zero. To avoid these problems, it is safer to perform the computations in the log-domain:

$$\alpha_j = \left(\sum_{i:F_i=1} \ln a_{i,j} + \sum_{i:F_i=0} \ln(1 - a_{i,j}) \right) + \ln b_j \tag{188}$$

$$\gamma = \min_j \alpha_j \tag{189}$$

$$P(C = j|F_{1:D}) = \frac{\exp(\alpha_j - \gamma)}{\sum_\ell \exp(\alpha_\ell - \gamma)} \tag{190}$$

which, as you can see by inspection, is mathematically equivalent to the original form, but will not evaluate to zero for at least one class.

8.7.2 Learning

For a collection of N training vectors F_k, each with an associated class label C_k, we can learn the parameters by maximizing the data likelihood (i.e., the probability of the data given the model). This is equivalent to estimating multinomial distributions (in the case of binary features, binomial distributions), and reduces to simple counting of features.

Suppose there are N_k examples of each class, and N examples total. Then the prior estimate is simply:

$$b_k = \frac{N_k}{N} \tag{191}$$

Similarly, if class k has $N_{i,k}$ examples where $F_i = 1$, then

$$a_{i,k} = \frac{N_{i,k}}{N_k} \tag{192}$$

With large numbers of features and small datasets, it is likely that some features will never be seen for a class, giving a class probability of zero for that feature. We might wish to regularize, to prevent this extreme model from occurring. We can modify the learning rule as follows:

$$a_{i,k} = \frac{N_{i,k} + \alpha}{N_k + 2\alpha} \tag{193}$$

for some small value α. In the extreme case where there are no examples for which feature i is seen for class k, the probability $a_{i,k}$ will be set to $1/2$, corresponding to no knowledge. As the number of examples N_k becomes large, the role of α will become smaller and smaller.

In general, given in a multinomial distribution with a large number of classes and a small training set, we might end up with estimates of prior probability b_k being zero for some classes. This might be undesirable for various reasons, or be inconsistent with our prior beliefs. Again, to avoid this situation, we can regularize the maximum likelihood estimator with our prior believe that all classes should have a nonzero probability. In doing so we can estimate the class prior probabilities as

$$b_k = \frac{N_k + \beta}{N + K\beta} \tag{194}$$

for some small value of β. When there are no observations whatsoever, all classes are given probability $1/K$. When there are observations the estimated probabilities will lie between N_k/N and $1/K$ (converging to N_k/N as $N \to \infty$).

Derivation. Here we derive just the per-class probability assuming two classes, ignoring the feature vectors; this case reduces to estimating a binomial distribution. The full estimation can easily be derived in the same way.

Suppose we observe N examples of class 0, and M examples of class 1; what is b_0, the probability of observing class 0? Using maximum likelihood estimation, we maximize:

$$\prod_i P(C_i = k) = \left(\prod_{i:C_i=0} P(C_i = 0)\right)\left(\prod_{i:C_i=1} P(C_i = 1)\right) \tag{195}$$

$$= b_0^N b_1^M \tag{196}$$

Furthermore, in order for the class probabilities to be a valid distribution, it is required that $b_0 + b_1 = 1$, and that $b_k \geq 0$. In order to enforce the first constraint, we set $b_1 = 1 - b_0$:

$$\prod_i P(C_i = k) = b_0^N (1 - b_0)^M \tag{197}$$

The log of this is:

$$L(b_0) = N \ln b_0 + M \ln(1 - b_0) \tag{198}$$

To maximize, we compute the derivative and set it to zero:

$$\frac{dL}{db_0} = \frac{N}{b_0} - \frac{M}{1 - b_0} = 0 \tag{199}$$

Multiplying both sides by $b_0(1 - b_0)$ and solving gives:

$$b_0^* = \frac{N}{N + M} \tag{200}$$

which, fortunately, is guaranteed to satisfy the constraint $b_0 \geq 0$.

9 Gradient Descent

There are many situations in which we wish to minimize an objective function with respect to a parameter vector:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} E(\mathbf{w}) \tag{201}$$

but no closed-form solution for the minimum exists. In machine learning, this optimization is normally a data-fitting objective function, but similar problems arise throughout computer science, numerical analysis, physics, finance, and many other fields.

The solution we will use in this course is called **gradient descent**. It works for any differentiable energy function. However, it does not come with many guarantees: it is only guaranteed to find a local minima in the limit of infinite computation time.

Gradient descent is iterative. First, we obtain an initial estimate $\mathbf{w}_1$ of the unknown parameter vector. How we obtain this vector depends on the problem; one approach is to randomly-sample values for the parameters. Then, from this initial estimate, we note that the direction of steepest descent from this point is to follow the negative gradient $-\nabla E$ of the objective function evaluated at $\mathbf{w}_1$. The gradient is defined as a vector of derivatives with respect to each of the parameters:

$$\nabla E \equiv \begin{bmatrix} \frac{dE}{dw_1} \\ \vdots \\ \frac{dE}{dw_N} \end{bmatrix} \tag{202}$$

The key point is that, if we follow the negative gradient direction in a small enough distance, *the objective function is guaranteed to decrease*. (This can be shown by considering a Taylor-series approximation to the objective function).

It is easiest to visualize this process by considering $E(\mathbf{w})$ as a surface parameterized by $\mathbf{w}$; we are trying to finding the deepest pit in the surface. We do so by taking small downhill steps in the negative gradient direction.

The entire process, in its simplest form, can be summarized as follows:

pick initial value $\mathbf{w}_1$
$i \leftarrow 1$
loop
  $\mathbf{w}_{i+1} \leftarrow \mathbf{w}_i - \lambda \nabla E|_{\mathbf{w}_i}$
  $i \leftarrow i + 1$
end **loop**

Note that this process depends on three choices: the initialization, the termination conditions, and the step-size λ. For the termination condition, one can run until a preset number of steps has elapsed, or monitor convergence, i.e., terminate when

$$|E(\mathbf{w}_{i+1}) - E(\mathbf{w}_i)| < \epsilon \tag{203}$$

for some preselected constant ϵ, or terminate when either condition is met.

The simplest way to determine the step-size λ is to pick a single value in advance, and this approach is often taken in practice. However, it is somewhat unreliable: if we choose step-size too large, than the objective function might actually get worse on some steps; if the step-size is too small, then the algorithm will take a very long time to make any progress.

The solution is to use **line search**, namely, at each step, to search for a step-size that reduces the objective function as much as possible. For example, a simple gradient search with line search procedure is:

pick initial value $\mathbf{w}_1$
$i \leftarrow 1$
loop
  $\Delta \leftarrow \nabla E|_{\mathbf{w}_i}$
  $\lambda \leftarrow 1$
  while $E(\mathbf{w}_i - \lambda\Delta) \geq E(\mathbf{w}_i)$
    $\lambda \leftarrow \frac{\lambda}{2}$
  end while
  $\mathbf{w}_{i+1} \leftarrow \mathbf{w}_i - \lambda\Delta$
  $i \leftarrow i + 1$
end loop

A more sophisticated approach is to reuse step-sizes between iterations:

pick initial value $\mathbf{w}_1$
$i \leftarrow 1$
$\lambda \leftarrow 1$
loop
  $\Delta \leftarrow \nabla E|_{\mathbf{w}_i}$
  $\lambda \leftarrow 2\lambda$
  while $E(\mathbf{w}_i - \lambda\Delta) \geq E(\mathbf{w}_i)$
    $\lambda \leftarrow \frac{\lambda}{2}$
  end while
  $\mathbf{w}_{i+1} \leftarrow \mathbf{w}_i - \lambda\Delta$
  $i \leftarrow i + 1$
end loop

There are many, many more advanced methods for numerical optimization. For unconstrained optimization, I recommend the L-BFGS-B library, which is available for download on the web. It is written in Fortran, but there are wrappers for various languages out there. This method will be vastly superior to gradient descent for most problems.

9.1 Finite differences

The gradient of any function can be computed approximately by numerical computations. This is useful for debugging your gradient computations, and in situations where it's too difficult or tedious to implement the complete derivative. The numerical approximation follows directly from the definition of derivative:

$$\left.\frac{dE}{dw}\right|_w \approx \frac{E(w+h) - E(w)}{h} \tag{204}$$

for some suitably small stepsize h. Computing this value for each element of the parameter vector gives you an approximate estimate of the gradient ∇E.

It is strongly recommend that you use this method to debug your derivative computations; many errors can be detected this way! (This test is analogous to the use of "assertions").

Aside:
The term **backpropagation** is sometimes used to refer to an efficient algorithm for computing derivatives for Artificial Neural Networks. Confusingly, this term is also used to refer to gradient descent (without line search) for ANNs.

10 Cross Validation

Suppose we must choose between two possible ways to fit some data. How do we choose between them? Simply measuring how well they fit they data would mean that we always try to fit the data as closely as possible — the best method for fitting the data is simply to memorize it in big look-up table. However, fitting the data is no guarantee that we will be able to **generalize** to new measurements. As another example, consider the use of polynomial regression to model a function given a set of data points. Higher-order polynomials will always fit the data as well or better than a low-order polynomial; indeed, an $N-1$ degree polynomial will fit N data points exactly (to within numerical error). So just fitting the data as well as we can usually produces models with many parameters, and they are not going to generalize to new inputs in almost all cases of interest.

The general solution is to evaluate models by testing them on a new data set (the "test set"), distinct from the training set. This measures how **predictive** the model is: Is it useful in new situations? More generally, we often wish to obtain empirical estimates of performance. This can be useful for finding errors in implementation, comparing competing models and learning algorithms, and detecting over or under fitting in a learned model.

10.1 Cross-Validation

The idea of empirical performance evaluation can also be used to determine model parameters that might otherwise to hard to determine. Examples of such model parameters include the constant K in the K-Nearest Neighbors approach or the σ parameter in the Radial Basis Function approach.

Hold-out Validation. In the simplest method, we first partition our data randomly into a "training set" and a "validation set." Let K be the unknown model parameter. We pick a set of range of possible values for K (e.g., $K = 1, ..., 5$). For each possible value of K, we learn a model with that K on the training set, and compute that model's error on the validation set. For example, the error on validation set might be just the squared-error, $\sum_i ||y_i - f(x_i)||^2$. We then pick the K which has the smallest validation set error. The same idea can be applied if we have more model parameters (e.g., the σ in KNN), however, we must try many possible combinations of K and σ to find the best.

There is a significant problem with this approach: we use less training data when fitting the other model parameters, and so we will only get good results if our initial training set is rather large. If large amounts of data are expensive or impossible to obtain this can be a serious problem.

N-Fold Cross Validation. We can use the data much more efficiently by N-fold cross-validation. In this approach, we randomly partition the training data into N sets of equal size and run the learning algorithm N times. Each time, a different one of the N sets is deemed the test set, and the model is trained on the remaining $N-1$ sets. The value of K is scored by averaging the error across the N test errors. We can then pick the value of K that has the lowest score, and then learn model parameters for this K.

A good choice for N is $N = M - 1$, where M is the number of data points. This is called **Leave-one-out cross-validation**.

Issues with Cross Validation. Cross validation is a very simple and empirical way of comparing models. However, there are a number of issues to keep in mind:

- The method can be very time-consuming, since many training runs may be needed. For models with more than a few parameters, cross validation may be too inefficient to be useful.
- Because a reduced dataset is used for training, there must be sufficient training data so that all relevant phenomena of the problem exist in both the training data and the test data.
- It is safest to use a random partition, to avoid the possibility that there are unmodeled correlations in the data. For example, if the data was collected over a period of a week, it is possible that data from the beginning of the week has a different structure than the data later in the week.
- Because cross-validation finds a minimum of an objective function, over- and under-fitting may still occur, although it is much less likely. For example, if the test set is very small, it may prefer a model that fits the random pattern in the test data.

Aside:
Testing machine learning algorithms is very much like testing scientific theories: scientific theories must be predictive, or, that is, falsifiable. Scientific theories must also describe plausible models of reality, whereas machine learning methods need only be useful for making decisions. However, statistical inference and learning first arose as theories of scientific hypothesis testing, and remain closely related today.

One of the most famous examples is the case of planetary motion. Prior to Newton, astronomers described the motion of the planets through onerous tabulation of measurements — essentially, big lookup tables. These tables were not especially predictive, and needed to updated constantly. Newton's equations of motion — which could describe the motion of the planets with only a few simple equations — were vastly simpler and yet also more effective at predicting motion, and became the accepted theories of motion.

However, there remained some anomolies. Two astronomers, John Couch Adams and Urbain Le Verier, thought that these discrepancies might be due to a new, as-yet-undiscovered planet. Using techniques similar to modern regression, but with laborious hand-calculation, they independently deduced the position, mass, and orbit of the new planet. By observing in the predicted directions, astronomers were

indeed able to observe a new planet, which was later named Neptune. This provided powerful validation for their models.

Incidentally, Adams was an undergraduate working alone when he began his investigations.

Reference: http://en.wikipedia.org/wiki/Discovery_of_Neptune

11 Bayesian Methods

So far, we have considered statistical methods which select a single "best" model given the data. This approach can have problems, such as over-fitting when there is not enough data to fully constrain the model fit. In contrast, in the "pure" Bayesian approach, as much as possible we only compute distributions over unknowns; we never maximize anything. For example, consider a model parameterized by some weight vector $\mathbf{w}$, and some training data $\mathcal{D}$ that comprises input-output pairs x_i, y_i, for $i = 1...N$. The posterior probability distribution over the parameters, conditioned on the data is, using Bayes' rule, given by

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})} \tag{205}$$

The reason we want to fit the model in the first place is to allow us to make predictions with future test data. That is, given some future input x_{new}, we want to use the model to predict y_{new}. To accomplish this task through estimation in previous chapters, we used optimization to find ML or MAP estimates of $\mathbf{w}$, e.g., by maximizing (205).

In a Bayesian approach, rather than estimation a single best value for $\mathbf{w}$, we computer (or approximate) the entire posterior distribution $p(\mathbf{w}|\mathcal{D})$. Given the entire distribution, we can still make predictions with the following integral:

$$\begin{aligned} p(y_{new}|\mathcal{D}, x_{new}) &= \int p(y_{new}, \mathbf{w}|\mathcal{D}, x_{new}) d\mathbf{w} \\ &= \int p(y_{new}|\mathbf{w}, \mathcal{D}, x_{new})\, p(\mathbf{w}|\mathcal{D}, x_{new}) d\mathbf{w} \end{aligned} \tag{206}$$

The first step in this equality follows from the Sum Rule. The second follows from the Product Rule. Additionally, the outputs y_{new} and training data $\mathcal{D}$ are independent conditioned on $\mathbf{w}$, so $p(y_{new}|\mathbf{w}, \mathcal{D}) = p(y_{new}|\mathbf{w})$. That is, given $\mathbf{w}$, we have all available information about making predictions that we could possibly get from the training data $\mathcal{D}$ (according to the model). Finally, given $\mathcal{D}$, it is safe to assume that x_{new}, in itself, provides no information about $\mathbf{W}$. With these assumptions we have the following expression for our predictions:

$$p(y_{new}|\mathcal{D}, x_{new}) = \int p(y_{new}|\mathbf{w}, x_{new})\, p(\mathbf{w}|\mathcal{D}) d\mathbf{w} \tag{207}$$

In the case of discrete parameters $\mathbf{w}$, the integral becomes a summation.

The posterior distribution $p(y_{new}|\mathcal{D}, x_{new})$ tells us everything there is to know about our beliefs about the new value y_{new}. There are many things we can do with this distribution. For example, we could pick the most likely prediction, i.e., $\arg\max_y p(y_{new}|\mathcal{D}, x_{new})$, or we could compute the variance of this distribution to get a sense of how much confidence we have in the prediction. We could sample from this distribution in order to visualize the range of models that are plausible for this data.

The integral in (207) is rarely easy to compute, often involving intractable integrals or exponentially large summations. Thus, Bayesian methods often rely on numerical approximations, such as Monte Carlo sampling; MAP estimation can also be viewed as an approximation. However, in a few cases, the Bayesian computations can be done exactly, as in the regression case discussed below.

11.1 Bayesian Regression

Recall the statistical model used in basis-function regression:

$$y \;=\; \mathbf{b}(x)^T\mathbf{w} + n, \quad n \sim \mathcal{N}(0, \sigma^2) \tag{208}$$

for a fixed set of basis functions $\mathbf{b}(x) = [b_1(x), ...b_M(x)]^T$.

To complete the model, we also need to define a "prior" distribution over the weights $\mathbf{w}$ (denoted $p(\mathbf{w})$) which expresses what we believe about $\mathbf{w}$, in absence of any training data. One might be tempted to assign a constant density over all possible weights. There are several problems with this. First, the result cannot be a valid probability distribution since no choice of the constant will give the density a finite integral. We could, instead, choose a uniform distribution with finite bounds, however, this will make the resulting computations more complex.

More importantly, a uniform prior is often inappropriate; we often find that smoother functions are more likely in practice (at least for functions that we have any hope in learning), and so we should employ a prior that prefers smooth functions. A choice of prior that does so is a Gaussian prior:

$$\mathbf{w} \sim N(0, \alpha^{-1}\mathbf{I}) \tag{209}$$

which expresses a prior belief that smooth functions are more likely. This prior also has the additional benefit that it will lead to tractable integrals later on. Note that this prior depends on a parameter α; we will see later in this chapter how this "hyperparameter" can be determined automatically as well.

As developed in previous chapters on regression, the data likelihood function that follows from the above model definition (with the input and output components of the training dataset denoted $x_{1:N}$ and $y_{1:N}$) is

$$p(y_{1:N}|x_{1:N}, \mathbf{w}) = \prod_{i=1}^{N} p(y_i|x_i, \mathbf{w}) \tag{210}$$

and so the posterior is:

$$p(\mathbf{w}|x_{1:N}, y_{1:N}) = \frac{\left(\prod_{i=1}^{N} p(y_i|x_i, \mathbf{w})\right) p(\mathbf{w})}{p(y_{1:N}|x_{1:N})} \tag{211}$$

In the negative log-domain, using Equations (208) and (209), the model is given by:

$$\begin{aligned} -\ln p(\mathbf{w}|x_{1:N}, y_{1:N}) &= -\sum_i \ln(p(y_i|x_i, \mathbf{w})) - \ln(p(\mathbf{w})) + \ln(p(y_{1:N}|x_{1:N})) \\ &= \frac{1}{2\sigma^2}\sum_i (y_i - f(x_i))^2 + \frac{\alpha}{2}||\mathbf{w}||^2 + \text{constants} \end{aligned}$$

As above in the regression notes, it is useful if we collect the training outputs into a single vector, i.e., $\mathbf{y} = [y_1, ..., y_N]^T$, and we collect the all basis functions evaluated at each of the inputs into a matrix $\mathbf{B}$ with elements $\mathbf{B}_{i,j} = b_j(x_i)$. In doing so we can simplify the log posterior as follows:

$$\begin{aligned} -\ln p(\mathbf{w}|x_{1:N}, y_{1:N}) &= \frac{1}{2\sigma^2}||\mathbf{y} - \mathbf{B}\mathbf{w}||^2 + \frac{\alpha}{2}||\mathbf{w}||^2 + \text{constants} \\ &= \frac{1}{2\sigma^2}(\mathbf{y} - \mathbf{B}\mathbf{w})^T(\mathbf{y} - \mathbf{B}\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} + \text{constants} \\ &= \frac{1}{2}\mathbf{w}^T(\mathbf{B}^T\mathbf{B}/\sigma^2 + \alpha\mathbf{I})\mathbf{w} - \frac{1}{2}\mathbf{y}^T\mathbf{B}\mathbf{w}/\sigma^2 - \frac{1}{2}\mathbf{w}^T\mathbf{B}^T\mathbf{y}/\sigma^2 + \text{constants} \\ &= \frac{1}{2}(\mathbf{w} - \bar{\mathbf{w}})^T\mathbf{K}^{-1}(\mathbf{w} - \bar{\mathbf{w}}) + \text{constants} \end{aligned} \tag{212}$$

where

$$\mathbf{K} = \left(\mathbf{B}^T\mathbf{B}/\sigma^2 + \alpha\mathbf{I}\right)^{-1} \tag{213}$$

$$\bar{\mathbf{w}} = \mathbf{K}\mathbf{B}^T\mathbf{y}/\sigma^2 \tag{214}$$

(The last step of the derivation uses the methods of *completing the square*. It is easiest to verify the last step by going backwards, that is by multiplying out $(\mathbf{w} - \bar{\mathbf{w}})^T\mathbf{K}^{-1}(\mathbf{w} - \bar{\mathbf{w}})$.)

The derivation above tells us that the posterior distribution over the weight vector is a multi-dimensional Gaussian with mean $\bar{\mathbf{w}}$ and covariance matrix $\mathbf{K}$, i.e.,

$$p(\mathbf{w}|x_{1:N}, y_{1:N}) = G(\mathbf{w}; \bar{\mathbf{w}}, \mathbf{K}) \tag{215}$$

In other words, our belief about $\mathbf{w}$ once we have seen the data is specified by a Gaussian density. We believe that $\bar{\mathbf{w}}$ is the most probable value for $\mathbf{w}$, but we have uncertainty about this estimate, as determined by the covariance $\mathbf{K}$. The covariance expresses our uncertainty about these parameters. If the covariance is very small, then we have a lot of confidence in the MAP estimate. The nature of the posterior distribution is illustrated visually in Figure 14. Note that $\bar{\mathbf{w}}$ is the MAP estimate for regression, since it maximizes the posterior.

Prediction. For a new data point x_{new}, the predictive distribution for y_{new} is given by:

$$\begin{aligned} p(y_{new}|x_{new}, \mathcal{D}) &= \int p(y_{new}|x_{new}, \mathcal{D}, \mathbf{w})p(\mathbf{w}|\mathcal{D})d\mathbf{w} \\ &= \mathcal{N}(y_{new}; \mathbf{b}(x_{new})^T\bar{\mathbf{w}}, \sigma^2 + \mathbf{b}(x_{new})^T\mathbf{K}\mathbf{b}(x_{new})) \end{aligned}$$

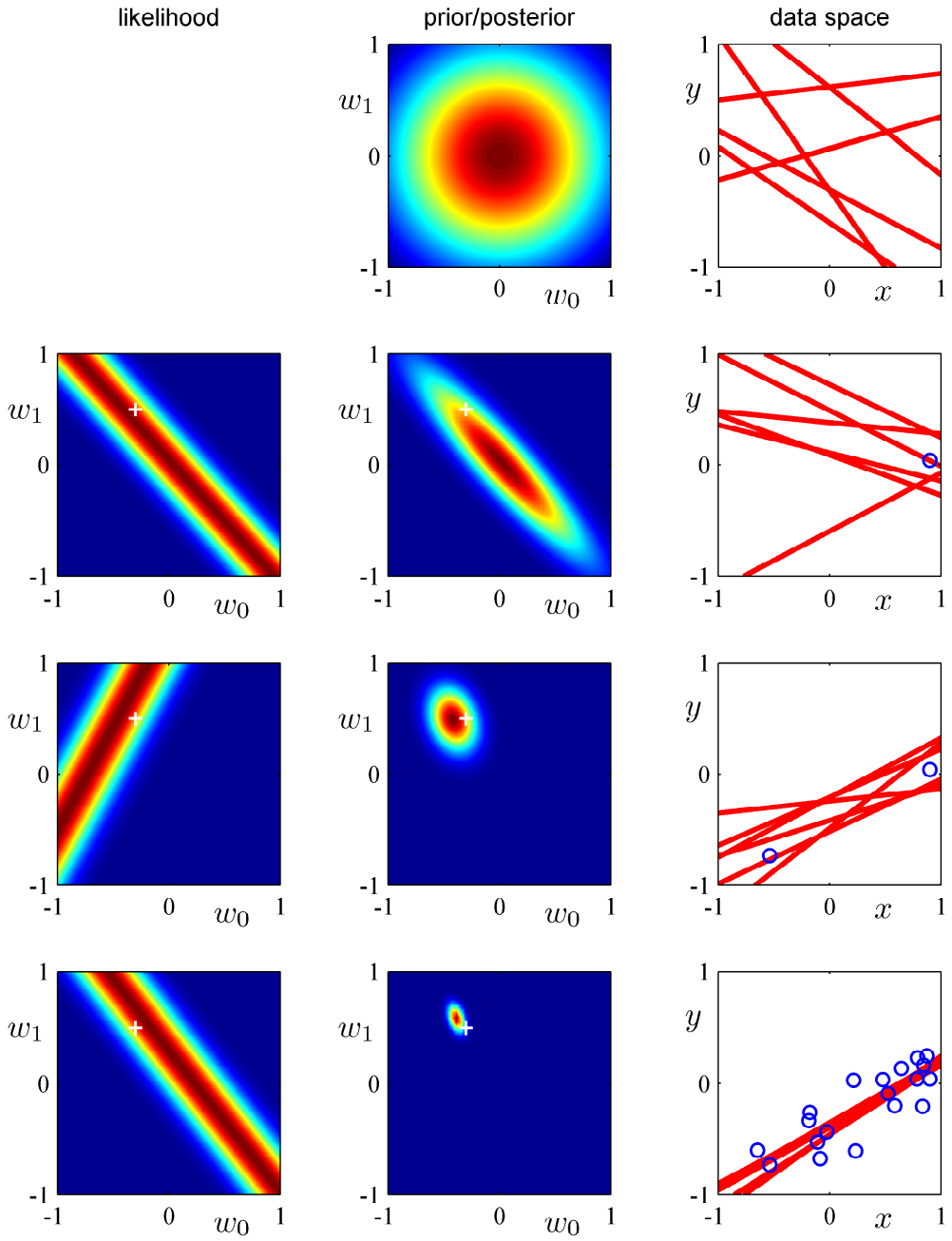

Figure 14: Iterative posterior computation for a linear regression model: $y = w_0 x + w_1$. The top row shows the prior distribution, and several fair samples from the prior distribution. The second row shows the likelihood over w after observing a single data point (i.e., an x, y pair), along with the resulting posterior (the normalized product of the likelihood and the prior), and then several fair samples from the posterior. The third row shows the liklihood when a new observation is added to the previous observation, followed by the corresponding posterior and random samples from the posterior. The final row shows the result of 20 observations.

The predictive distribution may be viewed as a function from x_{new} to a distribution over values of y_{new}. An example of this for an RBF model is given in Figure 15.

This is the Bayesian way to do regression. To predict a new value y_{new} for an input x_{new}, we don't estimate a single model $\mathbf{w}$. Instead we average over all possible models, weighting the different models according to their posterior probability.

11.2 Hyperparameters

There are often implicit parameters in our model that we hold fixed, such as the covariance constants in linear regression, or the parameters that govern the prior distribution over the weights. These are usually called "hyperparameters." For example, in the RBF model, the hyperparameters constitute the parameters α, σ^2, and the parameters of the basis functions (e.g., the width of the basis functions). Thus far we have assumed that the hyperparameters were "known" (which means that someone must set them by hand), or estimated by cross-validation (which has a number of pitfalls, including long computation times, especially for large numbers of hyperparameters). Instead of either of these approaches, we may apply the Bayesian approach in order to directly estimate these values as well.

To find a MAP estimate for the α parameter in the above linear regression example we compute:

$$\alpha^* = \arg\max \ln p(\alpha | x_{1:N}, y_{1:N}) \tag{216}$$

where

$$p(\alpha | x_{1:N}, y_{1:N}) = \frac{p(y_{1:N} | x_{1:N}, \alpha) p(\alpha)}{p(y_{1:N} | x_{1:N})} \tag{217}$$

and

$$\begin{aligned} p(y_{1:N} | x_{1:N}, \alpha) &= \int p(y_{1:N}, \mathbf{w} | x_{1:N}, \alpha) d\mathbf{w} \\ &= \int p(y_{1:N} | x_{1:N}, \mathbf{w}, \alpha) p(\mathbf{w} | \alpha) d\mathbf{w} \\ &= \int \left(\prod_i p(y_i | x_i, \mathbf{w}, \alpha) \right) p(\mathbf{w} | \alpha) d\mathbf{w} \end{aligned}$$

For RBF regression, this objective function can be computed in closed-form. However, depending on the form of the prior over the hyperparameters, it is often necessary to use some form of numerical optimization, such as gradient descent.

11.3 Bayesian Model Selection

How do we choose which model to use? For example, we might like to automatically choose the form of the basis functions or the number of basis functions. Cross-validation is one approach, but

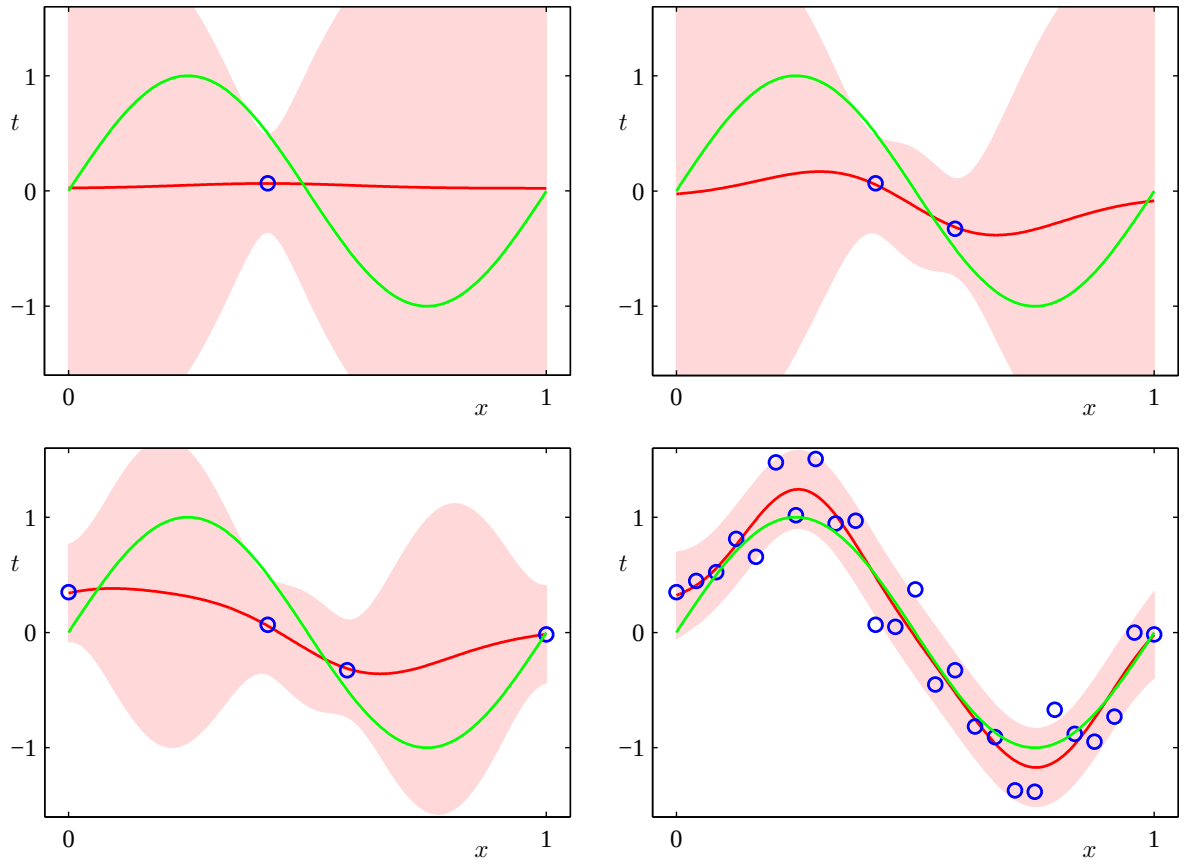

Figure 15: Predictive distribution for an RBF model (with 9 basis functions), trained on noisy sinusoidal data. The green curve is the true underlying sinusoidal function. The blue circles are data points. The red curve is the mean prediction as a function of the input. The pink region represents 1 standard deviation. Note how this region shrinks close to where more data points are observed. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

it can be expensive, and, more importantly, inaccurate if small amounts of data are available. In general one intuition is that we want to choose simple models over complex models to avoid overfitting,insofar as they provide equivalent fits to the data. Below we consider a Bayesian approach to model selection which provides just such a bias to simple models.

The goal of model selection is to choose the best model from some set of candidate models $\{\mathcal{M}_i\}_{i=1}^L$ based on some observed data $\mathcal{D}$. This may be done either with a maximum likelihood approach (picking the model that assigns the largest likelihood to the data) or a MAP approach (picking the model with the highest posterior probability). If we take a uniform prior over models (i.e. $p(\mathcal{M}_i)$ is a constant for all $i = 1...L$) then these approaches can be seen to be equivalent since:

$$\begin{aligned} p(\mathcal{M}_i|\mathcal{D}) &= \frac{p(\mathcal{D}|\mathcal{M}_i)p(\mathcal{M}_i)}{p(\mathcal{D})} \\ &\propto p(\mathcal{D}|\mathcal{M}_i) \end{aligned}$$

In practice a uniform prior over models may not be appropriate, but the design of suitable priors in these cases will depend significantly on one's knowledge of the application domain. So here we will assume a uniform prior over models and focus on $p(\mathcal{D}|\mathcal{M}_i)$.

In some sense, whenever we estimate a parameter in a model we are doing model selection where the family of models is indexed by the different values of that parameter. However the term "model selection" can also mean choosing the best model from some set of parametric models that are parameterized differently. A good example of this would be choosing the number of basis functions to use in an RBF regression model. Another simple example is choosing the polynomial degree for polynomial regression.

The key quantity for Bayesian model selection is $p(\mathcal{D}|\mathcal{M}_i)$, often called the *marginal data likelihood*. Given two models, $\mathcal{M}_1$ and $\mathcal{M}_2$, we will choose the model $\mathcal{M}_1$ when $p(\mathcal{D}|\mathcal{M}_1) > p(\mathcal{D}|\mathcal{M}_1)$. To specify these quantities in more detail we need to take the model parameters into account. Different models may have different numbers of parameters (e.g., polynomials of different degrees), or entirely different parameterizations (e.g., RBFs and neural networks). In what follows, let $\mathbf{w}_i$ be the vector of parameters for model $\mathcal{M}_i$. In the case of regression, for example, $\mathbf{w}_i$ might comprise the regression weights and hyper-parameters like the weight on the regularizer.

The extent to which a model explains (or fits) the data depends on the choice of the right parameters. Using the sum rule and Bayes' rule it follows we can write the marginal data likelihood as

$$p(\mathcal{D}|\mathcal{M}_i) = \int p(\mathcal{D}, \mathbf{w}_i|\mathcal{M}_i)d\mathbf{w}_i = \int p(\mathcal{D}|\mathbf{w}_i, \mathcal{M}_i)p(\mathbf{w}_i|\mathcal{M}_i)d\mathbf{w}_i \tag{218}$$

This tells us that the ideal model is one that assigns high prior probability $p(\mathbf{w}_i|\mathcal{M}_i)$ to every weight vector that also yields a high value of the likelihood $p(\mathcal{D}|\mathbf{w}_i, \mathcal{M}_i)$ (i.e., to parameter vectors that fit the data well). One can also recognize that the product of the data likelihood and the prior in the integrand is proportional to the posterior over the parameters that we previously maximized to find MAP estimates of the model parameters. [8]

[8] This is the same quantity we compute when optimizing hyper-parameters (which is a type of model selection) and

Typically, a "complex model" that assigns a significant posterior probability mass to complex data will be able to assign significantly less mass to simpler data than a simpler model would. This is because the integral of the probability mass must sum to 1 and so a complex model will have less mass to spend on simpler data. Also, since a complex model will require higher-dimensional parameterizations, mass must be spread over a higher-dimensional space and hence more thinly. This phenomenon is visualized in Figure 17.

As an aid to intuition to explain why this marginal data likelihood helps us choose good models, we consider a simple approximation to the marginal data likelihood $p(\mathcal{D}|\mathcal{M}_i)$ (depicted in Figure 16 for a scalar parameter w). First, as is common in many problems of interest, the posterior distribution over the model parameters $p(\mathbf{w}_i|\mathcal{D}, \mathcal{M}_i) \propto p(\mathcal{D}|\mathbf{w}_i, \mathcal{M}_i)p(\mathbf{w}_i|\mathcal{M}_i)$ to have a strong peak at the MAP parameter estimate $\mathbf{w}_i^{MAP}$. Accordingly we can approximate the integral in Equation (218) as the height of the peak, i.e., $p(\mathcal{D}|\mathbf{w}_i^{MAP}, \mathcal{M}_i)p(\mathbf{w}_i^{MAP}|\mathcal{M}_i)$, multiplied by its width $\Delta\mathbf{w}_i^{posterior}$.

$$\int p(\mathcal{D}|\mathbf{w}_i, \mathcal{M}_i)p(\mathbf{w}_i|\mathcal{M}_i)d\mathbf{w}_i \quad \approx \quad p(\mathcal{D}|\mathbf{w}_i^{MAP}, \mathcal{M}_i)\, p(\mathbf{w}_i^{MAP}|\mathcal{M}_i)\, \Delta\mathbf{w}_i^{posterior}$$

We then assume that the prior distribution over parameters $p(\mathbf{w}_i|\mathcal{M}_i)$ is a relatively broad uniform with width $\Delta\mathbf{w}_i^{prior}$, so $p(\mathbf{w}_i) \approx \frac{1}{\Delta\mathbf{w}_i^{prior}}$. This yields a further approximation:

$$\int p(\mathcal{D}|\mathbf{w}_i, \mathcal{M}_i)p(\mathbf{w}_i|\mathcal{M}_i)d\mathbf{w}_i \quad \approx \quad \frac{p(\mathcal{D}|\mathbf{w}_i^{MAP}, \mathcal{M}_i)\Delta\mathbf{w}_i^{posterior}}{\Delta\mathbf{w}_i^{prior}}$$

Taking the logarithm, this becomes

$$\ln p(\mathcal{D}|\mathbf{w}_i^{MAP}, \mathcal{M}_i) + \ln \frac{\Delta\mathbf{w}_i^{posterior}}{\Delta\mathbf{w}_i^{prior}}$$

Intuitively, this approximation tells us that models with wider prior distributions on the parameters will tend to assign less likelihood to the data because the wider prior captures a larger variety of data (so the density is spread thinner over the data-space). Similarly, models that have a very narrow peak around their modes are generally less preferable because they assign a lower probability mass to the surrounding area (and so a slightly perturbed setting of the parameters would provide a poor fit to the data, suggesting that over-fitting has occurred).

From another perspective, note that in most cases of interest we can assume that $\Delta\mathbf{w}_i^{posterior} < \Delta\mathbf{w}_i^{prior}$. I.e., the posterior width will be less than the width of the prior. The log ratio is maximal when the prior and posterior widths are equal. For example, a complex model with many parameters, or a a very broad prior over the parameters will necessarily assign a small probability to any single value (including those under the posterior peak). A simpler model will assign a higher prior

also corresponds to the denominator "$p(\mathcal{D})$" in Bayes' rule for finding the posterior probability of a particular setting of the parameters $\mathbf{w}_i$. Note that above we generally wrote $p(\mathcal{D})$ and not $p(\mathcal{D}|\mathcal{M}_i)$ because we were only considering a single model, and so it was not necessary to condition on it.

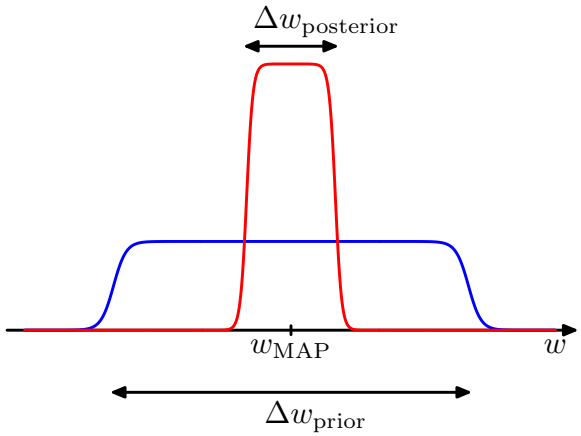

Figure 16: A visualization of the width-based evidence approximation. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

probability to the useful parameter values (ie those under the posterior peak). When the model is too simple, then the likelihood term in the integrand will be particularly high and therefore lowers the marginal data likelihood. So, as models become more complex the data likelihood increasingly fits the data better. But as the models become more and more complex the log ratio $\ln \frac{\Delta \mathbf{w}_i^{posterior}}{\Delta \mathbf{w}_i^{prior}}$ acts as a penalty on unnecessarily complex models.

By selecting a model that assigns the highest posterior probability to the data we are automatically balancing model complexity with the ability of the model to capture the data. This can be seen as the mathematical realization of Occam's Razor.

Model averaging. To be fully Bayesian, arguably, we shouldn't select a single "best" model but should instead combine estimates from all models according to their respective posterior probabilities:

$$p(y_{new}|\mathcal{D}, x_{new}) = \sum_i p(y_{new}|\mathcal{M}_i, \mathcal{D}, x_{new})\, p(\mathcal{M}_i|\mathcal{D}) \tag{219}$$

but this is often impractical and so we resort to model selection instead.

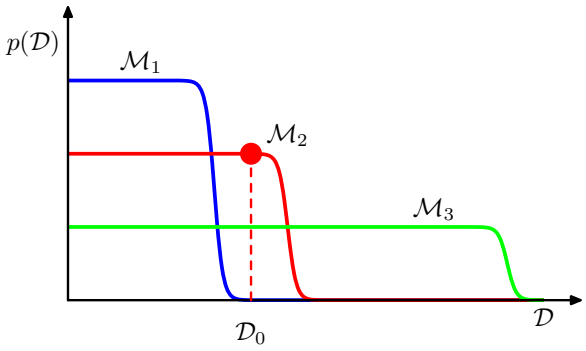

Figure 17: The x-axis is data complexity from simplest to most complex, and models $\mathcal{M}_i$ are indexed in order of increasing complexity. Note that in this example M_2 is the best model choice for data $\mathcal{D}_0$ since it simultaneously is complex enough to assign mass to $\mathcal{D}_0$ but not so complex that it must spread its mass too thinly. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

12 Monte Carlo Methods

Monte Carlo is an umbrella term referring to a set of numerical techniques for solving one or both of these problems:

1. Approximating expected values that cannot be solved in closed-form
2. Sampling from distributions for which is a simple sampling algorithm is not available.

Recall that expectation of a function $\phi(\mathbf{x})$ of a continuous variable $\mathbf{x}$ with respect to a distribution $p(\mathbf{x})$ is defined as:

$$E_{p(\mathbf{x})}[\phi(\mathbf{x})] \equiv \int p(\mathbf{x})\phi(\mathbf{x})d\mathbf{x} \tag{220}$$

Monte Carlo methods approximate this integral by drawing N samples from $p(\mathbf{x})$

$$\mathbf{x}_i \sim p(\mathbf{x}) \tag{221}$$

and then approximating the integral by the weighted average:

$$E_{p(\mathbf{x})}[\phi(\mathbf{x})] \approx \frac{1}{N}\sum_{i=1}^{N}\phi(\mathbf{x}_i) \tag{222}$$

Estimator properties. This estimate is unbiased:

$$E_{p(\mathbf{x}_{1:N})}\left[\frac{1}{N}\sum_i \phi(\mathbf{x}_i)\right] = \frac{1}{N}\sum_i E_{p(\mathbf{x}_i)}[\phi(\mathbf{x}_i)] = \frac{1}{N}NE_{p(\mathbf{x})}[\phi(\mathbf{x})] = E_{p(\mathbf{x})}[\phi(\mathbf{x})] \tag{223}$$

Furthermore, the variance of this estimate is inversely proportional to the number of samples:

$$\text{var}_{p(\mathbf{x}_{1:N})}\left[\frac{1}{N}\sum_i \phi(\mathbf{x}_i)\right] = \frac{1}{N^2}\sum_i \text{var}_{p(\mathbf{x}_{1:N})}[\phi(\mathbf{x}_i)] = \frac{1}{N^2}N\text{var}_{p(\mathbf{x}_i)}[\phi(x_i)] = \frac{1}{N}\text{var}_{p(\mathbf{x})}[\phi(\mathbf{x})] \tag{224}$$

Hence, the more samples we get, the better our estimate will be; in the limit, the estimator will converge to the true value.

Dealing with unnormalized distributions. We often wish to compute the expected value of a distribution for which evaluating the normalization constant is difficult. For example, the posterior distribution over parameters $\mathbf{w}$ given data D is:

$$p(\mathbf{w}|D) = \frac{p(D|\mathbf{w})p(\mathbf{w})}{p(D)} \tag{225}$$

The posterior mean and covariance ($\bar{\mathbf{w}} = E[\mathbf{w}]$ and $E[(\mathbf{w} - \bar{\mathbf{w}})(\mathbf{w} - \bar{\mathbf{w}})^T]$) can be useful to understand this posterior, i.e., what we believe the parameter values are "on average," and how much uncertainty there is in the parameters. The numerator of $p(\mathbf{w}|D)$ is typically easy to compute, but $p(D)$ entails an integral which is often intractable, and thus must be handled numerically.

Most generally, we can write the problem as computing the expected value with respect to a distribution $p(\mathbf{x})$ defined as

$$p(\mathbf{x}) \equiv \frac{1}{Z} P^*(\mathbf{x}), \qquad Z = \int P^*(\mathbf{x}) d\mathbf{x} \tag{226}$$

Monte Carlo methods will allow us to handle distributions of this form.

12.1 Sampling Gaussians

We begin with algorithms for sampling from a Gaussian distribution.

For the simple 1-dimensional case, $x \sim \mathcal{N}(0, 1)$, there is well known algorithm called the Box-Muller Method that is based on an approach called rejection sampling. It is implemented in Matlab in the command `randn`.

For a general 1D Gaussian, $x \sim \mathcal{N}(\mu, \sigma^2)$, we sample a variable $z \sim \mathcal{N}(0, 1)$, and then set $x = \sigma z + \mu$. You should be able to show that x has the desired mean and variance.

For the multi-dimensional case, $\mathbf{x} \sim \mathcal{N}(0, \mathbf{I})$, each element is independent and Gaussian: $x_i \sim \mathcal{N}(0, 1)$ and so each element can be sampled with `randn`.

To sample from a Gaussian with general mean vector $\boldsymbol{\mu}$ and variance matrix $\boldsymbol{\Sigma}$ we first sample $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$, and then set $\mathbf{x} = \mathbf{L}\mathbf{z} + \boldsymbol{\mu}$, where $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^T$. We can compute $\mathbf{L}$ from $\boldsymbol{\Sigma}$ by the Cholesky Factorization of $\boldsymbol{\Sigma}$, which must be positive definite. Then we have

$$E[\mathbf{x}] = E[\mathbf{L}\mathbf{z} + \boldsymbol{\mu}] = \mathbf{L}E[\mathbf{z}] + \boldsymbol{\mu} = \boldsymbol{\mu} \tag{227}$$

and

$$E[(\mathbf{z} - \boldsymbol{\mu})(\mathbf{z} - \boldsymbol{\mu})^T] = E[\mathbf{L}\mathbf{z}(\mathbf{L}\mathbf{z})^T] = \mathbf{L}E[\mathbf{z}\mathbf{z}^T]\mathbf{L}^T = \mathbf{L}\mathbf{L}^T = \boldsymbol{\Sigma} \tag{228}$$

12.2 Importance Sampling

In some situations, it may be difficult to sample from the desired distribution $p(\mathbf{x})$; however, we can sample from a similar distribution $q(\mathbf{x})$. Importance sampling is a technique that allows one to approximate expectation with respect to $p(\mathbf{x})$ by sampling from $q(\mathbf{x})$. The only requirement on q is that it have the same support as p, i.e., q is nonzero everywhere that p is nonzero.

Importance sampling is based on the following equality:

$$E_{q(\mathbf{x})}\left[\frac{p(\mathbf{x})}{q(\mathbf{x})}\phi(\mathbf{x})\right] = \int \frac{p(\mathbf{x})}{q(\mathbf{x})}\phi(\mathbf{x})q(\mathbf{x})d\mathbf{x} \tag{229}$$

$$= \int \phi(\mathbf{x})p(\mathbf{x})dx \tag{230}$$

$$= E_{p(\mathbf{x})}\left[\phi(\mathbf{x})\right] \tag{231}$$

In other words, we can compute the desired expectation by sampling values $\mathbf{x}_i$ from $q(\mathbf{x})$, and then computing

$$E_q\left[\frac{p(\mathbf{x})}{q(\mathbf{x})}\phi(\mathbf{x})\right] \approx \frac{1}{N}\sum_i \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}\phi(\mathbf{x}_i) \tag{232}$$

It often happens that p and/or q are known only up to multiplicative constants. That is,

$$p(\mathbf{x}) \equiv \frac{1}{Z_p}P^*(\mathbf{x}) \tag{233}$$

$$q(\mathbf{x}) \equiv \frac{1}{Z_q}Q^*(\mathbf{x}) \tag{234}$$

where P^* and Q^* are easy to evaluate but the constants Z_p and Z_q are not.

Then we have:

$$E_{p(\mathbf{x})}[\phi(\mathbf{x})] = \int \frac{\frac{1}{Z_p}P^*(\mathbf{x})}{\frac{1}{Z_q}Q^*(\mathbf{x})}\phi(\mathbf{x})q(\mathbf{x})dx = \frac{Z_q}{Z_p}E_{q(\mathbf{x})}\left[\frac{P^*(\mathbf{x})}{Q^*(\mathbf{x})}\phi(\mathbf{x})\right] \tag{235}$$

and so it remains to approximate $\frac{Z_q}{Z_p}$. If we substitute $\phi(\mathbf{x}) = 1$, the above formula states that

$$\frac{Z_q}{Z_p}E_{q(x)}\left[\frac{P^*(x)}{Q^*(x)}\right] = 1 \tag{236}$$

and so $\frac{Z_p}{Z_q} = E_{q(\mathbf{x})}[\frac{P^*(\mathbf{x})}{Q^*(\mathbf{x})}]$. Thus we have:

$$E_{p(\mathbf{x})}[\phi(\mathbf{x})] = \frac{E_{q(\mathbf{x})}\left[\frac{P^*(\mathbf{x})}{Q^*(\mathbf{x})}\phi(\mathbf{x})\right]}{E_{q(\mathbf{x})}\left[\frac{P^*(\mathbf{x})}{Q^*(\mathbf{x})}\right]} \tag{237}$$

Hence, the importance sampling algorithm is:

1. Sample N values $\mathbf{x}_i \sim q(\mathbf{x}_i)$
2. Compute
$$w_i = \frac{P^*(\mathbf{x}_i)}{Q^*(\mathbf{x}_i)} \tag{238}$$
3. Estimate the expected value
$$E[\phi(\mathbf{x})] \approx \frac{\sum_i w_i\phi(\mathbf{x}_i)}{\sum_i w_i} \tag{239}$$

The importance sampling algorithm will only work well when $q(\mathbf{x})$ is sufficiently similar to the function $p(\mathbf{x})|\phi(\mathbf{x})|$. Put more concretely, the variance of the estimator grows as the dissimilarity between $q(\mathbf{x})$ and $p(\mathbf{x})|\phi(\mathbf{x})|$ grows (and is minimized when they are equal). An alternative is to use the MCMC algorithm to draw samples directly from $p(\mathbf{x})$, as described below.

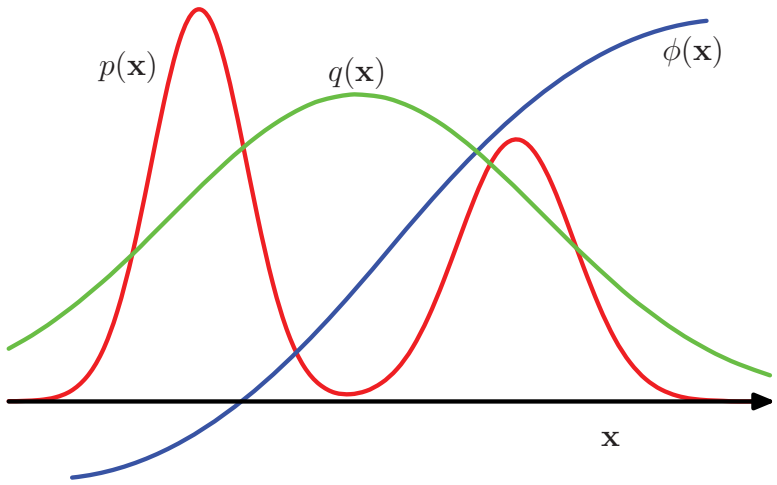

Figure 18: Importance sampling may be used to sample relatively complicated distributions like this bimodal $p(\mathbf{x})$ by instead sampling simpler distributions like this unimodal $q(\mathbf{x})$. Note that in this example, sampling from $q(\mathbf{x})$ will produce many samples that will be given a very low weight since $q(\mathbf{x})$ has a lot of mass where $p(\mathbf{x})$ is near zero (in the center of the plot). On the other hand, $q(\mathbf{x})$ has ample mass around the two modes of $p(\mathbf{x})$ and so it is a relatively good choice. If $q(\mathbf{x})$ had very little mass around one of the modes of $p(\mathbf{x})$, the estimate given by importance sampling would have a very high variance (unless $|\phi(\mathbf{x})|$ was small enough there to compensate for the difference). (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

12.3 Markov Chain Monte Carlo (MCMC)

MCMC is a very general algorithm for sampling from any distribution. For example, there is no simple method for sampling models $\mathbf{w}$ from the posterior distribution except in specialized cases (e.g., when the posterior is Gaussian).

MCMC is an iterative algorithm that, given a sample $\mathbf{x}_t \sim p(\mathbf{x})$, modifies that sample to produce a new sample $\mathbf{x}_{t+1} \sim p(\mathbf{x})$. This modification is done using a proposal distribution $q(\mathbf{x}'|\mathbf{x})$, that, given a $\mathbf{x}$, randomly selects a "mutation" to $\mathbf{x}$. This proposal distribution may be almost anything, and it is up to the user of the algorithm to choose this distribution; a common choice would be simply a Gaussian centered at $\mathbf{x}$: $q(\mathbf{x}'|\mathbf{x}) = \mathcal{N}(\mathbf{x}'|\mathbf{x}, \sigma^2\mathbf{I})$.

The entire algorithm is:

select initial point $\mathbf{x}_1$
$t \leftarrow 1$
loop
  Sample $\mathbf{x}' \sim q(\mathbf{x}'|\mathbf{x}_t)$
  $\alpha \leftarrow \frac{P^*(\mathbf{x}')}{P^*(\mathbf{x}_t)} \frac{q(\mathbf{x}_t|\mathbf{x}')}{q(\mathbf{x}'|\mathbf{x}_t)}$
  Sample $u \sim \text{Uniform}[0, 1]$
  if $u \leq \alpha$ **then**
    $\mathbf{x}_{t+1} \leftarrow \mathbf{x}'$
  else
    $\mathbf{x}_{t+1} \leftarrow \mathbf{x}_t$
  end if
  $t \leftarrow t + 1$
end loop

Amazingly, it can be shown that, if $\mathbf{x}_1$ is a sample from $p(\mathbf{x})$, then every subsequent $\mathbf{x}_t$ is also a sample from $p(\mathbf{x})$, if they are considered in isolation. The samples are correlated to each other via the Markov Chain, but the marginal distribution of any individual sample is $p(\mathbf{x})$.

So far we assumed that $\mathbf{x}_1$ is a sample from the target distribution, but, of course, obtaining this first sample is itself difficult. Instead, we must perform a process called **burn-in**: we initialize with any $\mathbf{x}_1$, and then discard the first T samples obtained by the algorithm; if we pick a large enough value of T, we are guaranteed that the remaining samples are valid samples from the target distribution. However, there is no exact method for determining a sufficient T, and so heuristics and/or experimentation must be used.

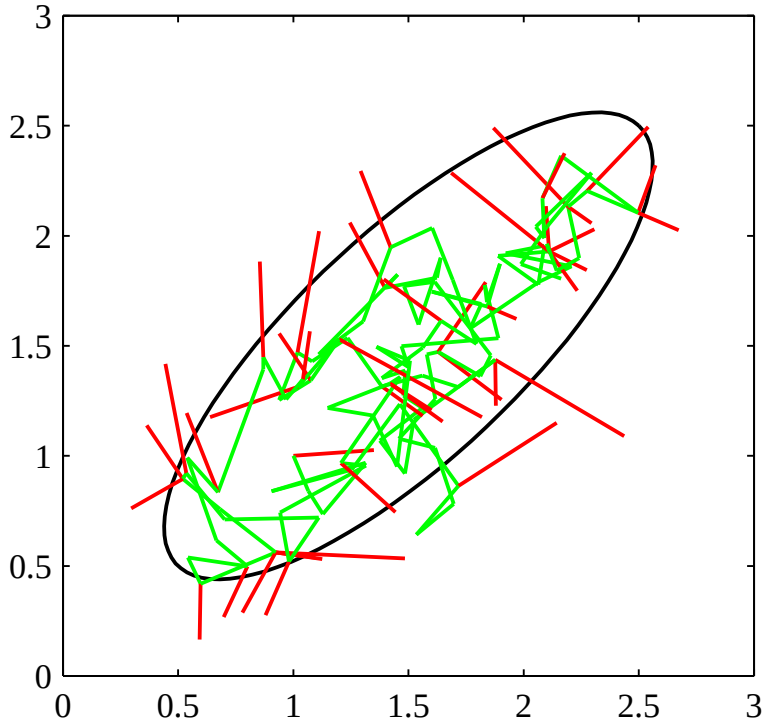

Figure 19: MCMC applied to a 2D elliptical Gaussian with a proposal distribution consisting of a circular Gaussian centered on the previous sample. Green lines indicate accepted proposals while red lines indicate rejected ones. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

13 Principal Components Analysis

We now discuss an **unsupervised** learning algorithm, called Principal Components Analysis, or PCA. The method is unsupervised because we are learning a mapping without any examples of what the mapping looks like; all we see are the outputs, and we want to estimate both the mapping *and* the inputs.

PCA is primarily a tool for dealing with high-dimensional data. If our measurements are 17-dimensional, or 30-dimensional, or 10,000-dimensional, manipulating the data can be extremely difficult. Quite often, the actual data can be described by a much lower-dimensional representation that captures all of the structure of the data. PCA is perhaps the simplest approach for finding such a representation, and yet is it also very fast and effective, resulting in it being very widely used.

There are several ways in which PCA can help:

- **Visualization:** PCA provides a way to visualize the data, by projecting the data down to two or three dimensions that you can plot, in order to get a better sense of the data. Furthermore, the principal component vectors sometimes provide insight as to the nature of the data as well.
- **Preprocessing:** Learning complex models of high-dimensional data is often very slow, and also prone to overfitting — the number of parameters in a model is usually exponential in the number of dimensions, meaning that very large data sets are required for higher-dimensional models. This problem is generally called **the curse of dimensionality.** PCA can be used to first map the data to a low-dimensional representation before applying a more sophisticated algorithm to it. With PCA one can also *whiten* the representation, which rebalances the weights of the data to give better performance in some cases.
- **Modeling:** PCA learns a representation that is sometimes used as an entire model, e.g., a prior distribution for new data.
- **Compression:** PCA can be used to compress data, by replacing data with its low-dimensional representation.

13.1 The model and learning

In PCA, we assume we are given N data vectors $\{\mathbf{y}_i\}$, where each vector is D-dimensional: $\mathbf{y}_i \in \mathbb{R}^D$. Our goal is to replace these vectors with lower-dimensional vectors $\{\mathbf{x}_i\}$ with dimensionality C, where $C < D$. We assume that they are related by a linear transformation:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} = \sum_{j=1}^{C} \mathbf{w}_j x_j + \mathbf{b} \tag{240}$$

The matrix $\mathbf{W}$ can be viewed as a containing a set of C basis vectors $\mathbf{W} = [\mathbf{w}_1, ..., \mathbf{w}_C]$. If we also assume Gaussian noise in the measurements, this model is the same as the linear regression model studied earlier, but now the $\mathbf{x}$'s are unknown in addition to the linear parameters.

To learn the model, we solve the following constrained least-squares problem:

$$\arg \min_{\mathbf{W},\mathbf{b},\{\mathbf{x}_i\}} \quad \sum_i ||\mathbf{y}_i - (\mathbf{W}\mathbf{x}_i + \mathbf{b})||^2 \tag{241}$$

$$\text{subject to } \mathbf{W}^T\mathbf{W} = \mathbf{I} \tag{242}$$

The constraint $\mathbf{W}^T\mathbf{W} = \mathbf{I}$ requires that we obtain an orthonormal mapping $\mathbf{W}$; it is equivalent to saying that

$$\mathbf{w}_i^T\mathbf{w}_j \;=\; \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{243}$$

This constraint is required to resolve an ambiguity in the mapping: if we did not require $\mathbf{W}$ to be orthonormal, then the objective function is underconstrained (why?). Note that an ambiguity remains in the learning even with this constraint (which one?), but this ambiguity is not very important.

The $\mathbf{x}$ coordinates are often called **latent** coordinates.

The algorithm for minimizing this objective function is as follows:

1. Let $\mathbf{b} = \frac{1}{N}\sum_i \mathbf{y}_i$
2. Let $\mathbf{K} = \frac{1}{N}\sum_i (\mathbf{y}_i - \mathbf{b})(\mathbf{y}_i - \mathbf{b})^T$
3. Let $\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T = \mathbf{K}$ be the eigenvector decomposition of $\mathbf{K}$. $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues ($\mathbf{\Lambda} = \text{diag}(\lambda_1, ...\lambda_D)$). The matrix $\mathbf{V}$ contains the eigenvectors: $\mathbf{V} = [\mathbf{V}_1, ...\mathbf{V}_D]$ and is orthonormal $\mathbf{V}^T\mathbf{V} = \mathbf{I}$.
4. Assume that the eigenvalues are sorted from largest to smallest ($\lambda_i \geq \lambda_{i+1}$). If this is not the case, sort them (and their corresponding eigenvectors).
5. Let $\mathbf{W}$ be a matrix of the first C eigenvectors: $\mathbf{W} = [\mathbf{V}_1, ...\mathbf{V}_C]$.
6. Let $\mathbf{x}_i = \mathbf{W}^T(\mathbf{y}_i - \mathbf{b})$, for all i.

13.2 Reconstruction

Suppose we have learned a PCA model, and are given a new $\mathbf{y}_{new}$ value; how do we estimate its corresponding $\mathbf{x}_{new}$? This can be done by minimizing

$$||\mathbf{y}_{new} - (\mathbf{W}\mathbf{x}_{new} + \mathbf{b})||^2 \tag{244}$$

This is a linear least-squares problem, and can be solved with standard methods (in MATLAB, implemented by the backslash operator). However $\mathbf{W}$ is orthonormal, and thus its transpose is the pseudoinverse, so the solution is given simply by:

$$\mathbf{x}^*_{new} = \mathbf{W}^T(\mathbf{y}_{new} - \mathbf{b}) \tag{245}$$

13.3 Properties of PCA

Mean zero coefficients. One can show that the PCA coefficients that represent the training data, i.e., $\{\mathbf{x}_i\}_{i=1}^N$, are mean zero.

$$\begin{aligned} \text{mean}(\mathbf{x}) \equiv \frac{1}{N}\sum_i \mathbf{x}_i &= \frac{1}{N}\sum_i \mathbf{W}^T(\mathbf{y}_i - \mathbf{b}) && (246)\\ &= \frac{1}{N}\mathbf{W}^T\left(\sum_i \mathbf{y}_i - N\mathbf{b}\right) && (247)\\ &= 0 && (248)\end{aligned}$$

Variance maximization. PCA can also be defined in the following way; in fact, this is the original definition of PCA, and the one that is often meant when people discuss PCA. However, this formulation is exactly equivalent to the one discussed above. In this goal, we wish to find the first principal component $\mathbf{w}_1$ to maximize the variance of the first coordinate of the data:

$$\text{var}(x_1) = \frac{1}{N}\sum_i x_{1,i}^2 = \frac{1}{N}\sum_i (\mathbf{w}_1^T(\mathbf{y}_i - \mathbf{b}))^2 \quad (249)$$

such that $||\mathbf{w}_1||^2 = 1$. Then, we wish to choose the second principal component to be a unit vector and orthogonal to the first component, while maximizing the variance of $\mathbf{x}_2$. The remaining principle components are also defined in this recursive way, so that each component $\mathbf{w}_i$ is a unit vector, orthogonal to all previous basis vectors.

Uncorrelated coefficients. It is straightforward to show that the covariance matrix of the PCA coefficients is the just the upper left $C \times C$ submatrix of $\mathbf{\Lambda}$ (i.e., the diagonal matrix containing the C leading eigenvalues of $\mathbf{K}$.

$$\begin{aligned} \text{cov}(\mathbf{x}) &\equiv \frac{1}{N}\sum_i (\mathbf{W}^T(\mathbf{y}_i - \mathbf{b}))(\mathbf{W}^T(\mathbf{y}_i - \mathbf{b}))^T && (250)\\ &= \frac{1}{N}\mathbf{W}^T\left(\sum_i (\mathbf{y}_i - \mathbf{b})(\mathbf{y}_i - \mathbf{b})^T\right)\mathbf{W} && (251)\\ &= \mathbf{W}^T\mathbf{K}\mathbf{W} && (252)\\ &= \mathbf{W}^T\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T\mathbf{W} && (253)\\ &= \tilde{\mathbf{\Lambda}} && (254)\end{aligned}$$

where $\tilde{\mathbf{\Lambda}}$ is the diagonal matrix containing the C leading eigenvalues in $\mathbf{\Lambda}$. This simple derivation also shows that the marginal variances of the PCA coefficients are given by the eigenvalues; i.e., $\text{var}(x_j) = \lambda_j$.

Out of Subspace Error. The total variance in the data is given by the sum of the eigenvalues of the sample covariance matrix $\mathbf{K}$. The variance captured by the PCA subspace representation is the sum of the first C eigenvalues. The total amount of variance *lost* in the representation is given by the sum of the remaining eigenvalues. In fact, one can show that the least-squares error in the approximation to the original data provided by the optimal (ML) model parameters, $\mathbf{W}^*$, $\{\mathbf{x}_i^*\}$, and $\mathbf{b}^*$, is given by

$$\sum_i ||\mathbf{y}_i - (\mathbf{W}^*\mathbf{x}_i^* + \mathbf{b}^*)||^2 = \sum_{j=C+1}^{D} \lambda_j \,. \tag{255}$$

When learning a PCA model it is common to use the ratio of the total LS error and the total variance in the training data (i.e., the sum of all eigenvalues). One needs to choose C to be large enough that this ratio is small (often 0.1 or less).

13.4 Whitening

Whitening is a preprocess that replaces the data with a representation that has zero-mean and unit covariance, and is often useful as a data preprocessing step. Given measurements $\{\mathbf{y}_i\}$, we replace them with $\{\mathbf{z}_i\}$ given by

$$\mathbf{z}_i = \tilde{\mathbf{\Lambda}}^{-\frac{1}{2}}\mathbf{W}^T(\mathbf{y}_i - \mathbf{b}) = \tilde{\mathbf{\Lambda}}^{-\frac{1}{2}}\mathbf{x}_i \tag{256}$$

where $\tilde{\mathbf{\Lambda}}$ is a diagonal matrix of the first C eigenvalues.

Then, the sample mean of the $\mathbf{z}$'s is equal to 0:

$$\text{mean}(\mathbf{z}) = \text{mean}(\tilde{\mathbf{\Lambda}}^{-\frac{1}{2}}\mathbf{x}_i) = \tilde{\mathbf{\Lambda}}^{-\frac{1}{2}}\text{mean}(\mathbf{x}) = 0 \tag{257}$$

To derive the sample covariance, we will first compute the covariance of the untruncated values: $\tilde{\mathbf{z}} \equiv \mathbf{\Lambda}^{-\frac{1}{2}}\mathbf{V}^T(\mathbf{y} - \mathbf{b})$:

$$\begin{aligned}
\text{cov}(\tilde{\mathbf{z}}) &\equiv \frac{1}{N}\sum_i \mathbf{\Lambda}^{-\frac{1}{2}}\mathbf{V}^T(\mathbf{y}_i - \mathbf{b})(\mathbf{y}_i - \mathbf{b})^T\mathbf{V}\mathbf{\Lambda}^{-\frac{1}{2}} && (258)\\
&= \mathbf{\Lambda}^{-\frac{1}{2}}\mathbf{V}^T\left(\frac{1}{N}\sum_i (\mathbf{y}_i - \mathbf{b})(\mathbf{y}_i - \mathbf{b})^T\right)\mathbf{V}\mathbf{\Lambda}^{-\frac{1}{2}} && (259)\\
&= \mathbf{\Lambda}^{-\frac{1}{2}}\mathbf{V}^T\mathbf{K}\mathbf{V}\mathbf{\Lambda}^{-\frac{1}{2}} && (260)\\
&= \mathbf{\Lambda}^{-\frac{1}{2}}\mathbf{V}^T\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T\mathbf{V}\mathbf{\Lambda}^{-\frac{1}{2}} && (261)\\
&= \mathbf{I} && (262)
\end{aligned}$$

Since $\mathbf{z}$ is just the first C elements of $\tilde{\mathbf{z}}$, $\mathbf{z}$ also has sample covariance $\mathbf{I}$.

13.5 Modeling

PCA is sometimes used to model data likelihood, e.g., we can use it as a form of a "prior". For example, suppose we have noisy measurements of some $\mathbf{y}$ values and wish to estimate their true values. If we parameterize the unknown $\mathbf{y}$ values by their corresponding $\mathbf{x}$ values instead, then we constrain the estimated values to lie in the low-dimensional subspace of the original data. However, this approach implies a uniform prior over $\mathbf{x}$ values, which may be inadequate, while being intolerant to deviations from the subspace. A better approach with an inherently probabilistic model is described below.

13.6 Probabilistic PCA

Probabilistic PCA is a way to estimate a probability distribution $p(\mathbf{y})$; in fact, it is a form of Gaussian distribution. In particular, we assume the following probability distribution:

$$\mathbf{x} \sim \mathcal{N}(0, \mathbf{I}) \tag{263}$$

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} + \mathbf{n}, \quad \mathbf{n} \sim \mathcal{N}(0, \sigma^2\mathbf{I}) \tag{264}$$

where $\mathbf{x}$ and $\mathbf{n}$ are assumed to be statistically independent. The model says that the low-dimensional coordinates $\mathbf{x}$ (i.e., the underlying causes) come from a unit Gaussian distribution, and the $\mathbf{y}$ measurements are a linear function of these low-dimensioanl causes, plus Gaussian noise. Note that we do **not** require that $\mathbf{W}$ be orthonormal anymore (in part because we now constrain the magnitude of the $\mathbf{x}$ variables).

Since any linear transformation of a Gaussian variable is itself Gaussian, $\mathbf{y}$ must also be Gaussian. This distribution is:

$$p(\mathbf{y}) = \int p(\mathbf{x}, \mathbf{y}) d\mathbf{x} = \int p(\mathbf{y}|\mathbf{x}) p(\mathbf{x}) d\mathbf{x} = \int G(\mathbf{y}; \mathbf{W}\mathbf{x} + \mathbf{b},\, \sigma^2\mathbf{I})\, G(\mathbf{x}; 0,\, \mathbf{I})\, d\mathbf{x} \tag{265}$$

Evaluating this integral will give us $p(\mathbf{y})$, however, there is a simpler way to solve for the Gaussian distribution.

Since we know that $\mathbf{y}$ is Gaussian, all we need to do is derive its mean and covariance, which can be done as follows (using the fact that mathematical expectation is linear):

$$\begin{aligned}
\text{mean}(\mathbf{y}) &= E[\mathbf{y}] = E[\mathbf{W}\mathbf{x} + \mathbf{b} + \mathbf{n}] && (266)\\
&= \mathbf{W}E[\mathbf{x}] + \mathbf{b} + E[\mathbf{n}] && (267)\\
&= \mathbf{b} && (268)\\
\text{cov}(\mathbf{y}) &= E[(\mathbf{y} - \mathbf{b})(\mathbf{y} - \mathbf{b})^T] && (269)\\
&= E[(\mathbf{W}\mathbf{x} + \mathbf{b} + \mathbf{n} - \mathbf{b})(\mathbf{W}\mathbf{x} + \mathbf{b} + \mathbf{n} - \mathbf{b})^T] && (270)\\
&= E[(\mathbf{W}\mathbf{x} + \mathbf{n})(\mathbf{W}\mathbf{x} + \mathbf{n})^T] && (271)\\
&= E[\mathbf{W}\mathbf{x}\mathbf{x}^T\mathbf{W}^T] + E[\mathbf{W}\mathbf{x}\mathbf{n}^T] + E[\mathbf{n}\mathbf{x}^T\mathbf{W}^T] + E[\mathbf{n}\mathbf{n}^T] && (272)\\
&= \mathbf{W}E[\mathbf{x}\mathbf{x}^T]\mathbf{W}^T + \mathbf{W}E[\mathbf{x}]E[\mathbf{n}^T] + E[\mathbf{n}]E[\mathbf{x}^T]\mathbf{W}^T + \sigma^2\mathbf{I} && (273)\\
&= \mathbf{W}\mathbf{W}^T + \sigma^2\mathbf{I} && (274)
\end{aligned}$$

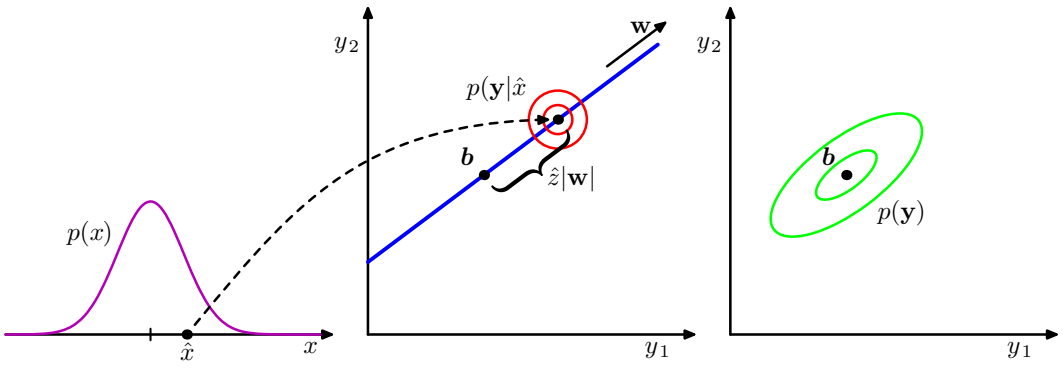

Figure 20: Visualization of PPCA mapping for a 1D to 2D model. A Gaussian in 1D is mapped to a line, and then blurred with 2D noise. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

Hence

$$\mathbf{y} \sim \mathcal{N}(\mathbf{b}, \mathbf{W}\mathbf{W}^T + \sigma^2\mathbf{I}) \tag{275}$$

In other words, learning a PPCA model is equivalent to learning a particular form of a Gaussian distribution. This is illustrated in Figure 20. The PPCA model is not as general as learning a full Gaussian model with a $D \times D$ covariance matrix; however, it uses fewer numbers to represent the Gaussian ($CD+1$ versus $D^2/2+D/2$; why?). Because the representation is more compact, it can be estimated from smaller datasets, and requires less memory to store the model.

These differences will be significant when D is large; e.g., if $D = 100$, the full covariance matrix would require 5050 parameters and thus require hundreds of thousands of data points to estimate reliably. However, if the effective dimensionality is, say, 2 or 3, then the PPCA representation will only have a few hundred parameters and many fewer measurements.

Learning. The PPCA model can be learned by Maximum Likelihood, i.e., by minimizing:

$$L(\mathbf{W}, \mathbf{b}, \sigma^2) = -\ln \prod_{i=1}^{N} G(\mathbf{y}_i;\, \mathbf{b},\, \mathbf{W}\mathbf{W}^T + \sigma^2\mathbf{I}) \tag{276}$$

$$= \frac{1}{2}\sum_i (\mathbf{y}_i - \mathbf{b})^T(\mathbf{W}\mathbf{W}^T + \sigma^2\mathbf{I})^{-1}(\mathbf{y}_i - \mathbf{b}) + \frac{N}{2}\ln(2\pi)^D|\mathbf{W}\mathbf{W}^T + \sigma^2\mathbf{I}| \tag{277}$$

This can be optimized in closed form. The solution is very similar to the conventional PCA case:

1. Let $\mathbf{b} = \frac{1}{N}\sum_i \mathbf{y}_i$
2. Let $\mathbf{K} = \frac{1}{N}\sum_i (\mathbf{y}_i - \mathbf{b})(\mathbf{y}_i - \mathbf{b})^T$

3. Let $\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T = \mathbf{K}$ be the eigenvector decomposition of $\mathbf{K}$. $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues ($\mathbf{\Lambda} = \mathrm{diag}(\lambda_1, ...\lambda_D)$). The matrix $\mathbf{V}$ contains the eigenvectors: $\mathbf{V} = [\mathbf{V}_1, ...\mathbf{V}_D]$ and is orthonormal $\mathbf{V}^T\mathbf{V} = \mathbf{I}$.

4. Assume that the eigenvalues are sorted from largest to smallest ($\lambda_i \geq \lambda_{i+1}$). If this is not the case, sort them (and their corresponding eigenvectors).

5. Let $\sigma^2 = \frac{1}{D-C}\sum_{j=C+1}^{D} \lambda_j$. In words, the estimated noise variance is equal to the average marginal data variance over all directions that are orthogonal to the C principal directions (i.e., this is the average variance (per dimension) of the data that is lost in the approximation of the data in the C dimensional subspace).

6. Let $\tilde{\mathbf{V}}$ be the matrix comprising the first C eigenvectors: $\tilde{\mathbf{V}} = [\mathbf{V}_1, ...\mathbf{V}_C]$, and let $\tilde{\mathbf{\Lambda}}$ be the diagonal matrix with the C leading eigenvalues: $\tilde{\mathbf{\Lambda}} = [\lambda_1, ...\lambda_C]$.

7. $\mathbf{W} = \tilde{\mathbf{V}}(\tilde{\mathbf{\Lambda}} - \sigma^2\mathbf{I})^{\frac{1}{2}}$.

8. Let $\mathbf{x}_i = \mathbf{W}^T(\mathbf{y}_i - \mathbf{b})$, for all i.

Note that this solution is similar to that in the conventional PCA case with whitening, except that (a) the noise variance is estimated, and (b) the noise is removed from the variances of the remaining eigenvalues.

An alternative optimization. In the above learning algorithm, we "marginalized out" $\mathbf{x}$ when estimating PPCA. In other words, we maximized

$$p(\mathbf{y}_{1:N}|\mathbf{W}, \mathbf{b}, \sigma^2) = \int p(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|\mathbf{W}, \mathbf{b}, \sigma^2) d\mathbf{x}_{1:N} \quad (278)$$

$$= \int p(\mathbf{y}_{1:N}|\mathbf{x}_{1:N}, \mathbf{W}, \mathbf{b}, \sigma^2) p(\mathbf{x}_{1:N}) d\mathbf{x}_{1:N} \quad (279)$$

$$= \prod_i \int p(\mathbf{y}_i|\mathbf{x}_i, \mathbf{W}, \mathbf{b}, \sigma^2) p(\mathbf{x}_i) d\mathbf{x}_i \quad (280)$$

instead of maximizing

$$p(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|\mathbf{W}, \mathbf{b}, \sigma^2) = \prod_i p(\mathbf{y}_i, \mathbf{x}_i|\mathbf{W}, \mathbf{b}, \sigma^2) \quad (281)$$

$$= \prod_i p(\mathbf{y}_i|\mathbf{x}_i, \mathbf{W}, \mathbf{b}, \sigma^2) p(\mathbf{x}_i) \quad (282)$$

By integrating out $\mathbf{x}$, we are estimating fewer parameters and thus can get better estimates. Loosely speaking, doing so might be viewed as being "more Bayesian." Suppose we did instead try to

estimate the $\mathbf{x}$'s together with the model parameters:

$$\begin{aligned} L(\mathbf{x}_{1:N}, \mathbf{W}, \mathbf{b}, \sigma^2) &= -\ln p(\mathbf{y}_{1:N}, \mathbf{x}_{1:N} | \mathbf{W}, \mathbf{b}, \sigma^2) && (283) \\ &= \sum_i \left(\frac{1}{2\sigma^2} ||\mathbf{y}_i - (\mathbf{W}\mathbf{x}_i + \mathbf{b})||^2 + \frac{1}{2} ||\mathbf{x}_i||^2 \right) \\ &\quad + \frac{ND}{2} \ln \sigma^2 + ND \ln 2\pi && (284) \end{aligned}$$

Now, suppose we are optimizing this objective function, and we have some estimates for $\mathbf{W}$ and $\mathbf{x}$. We can always reduce the objective function by replacing

$$\begin{aligned} \mathbf{W} &\leftarrow 2\mathbf{W} && (285) \\ \mathbf{x} &\leftarrow \mathbf{x}/2 && (286) \end{aligned}$$

By doing this replacement arbitrarily many times, we can get infinitesimal values for $\mathbf{x}$. This indicates that the objective function is degenerate; using it will yield to very poor results.

Note that, however, this arises using *the same model* as before, but without marginalizing out $\mathbf{x}$. This illustrates a general principle: the more parameters you estimate (instead of marginalizing out), the greater the danger of biased and/or degenerate solutions.

14 Lagrange Multipliers

The Method of Lagrange Multipliers is a powerful technique for constrained optimization. While it has applications far beyond machine learning (it was originally developed to solve physics equations), it is used for several key derivations in machine learning.

The problem set-up is as follows: we wish to find extrema (i.e., maxima or minima) of a differentiable objective function

$$E(\mathbf{x}) = E(x_1, x_2, ...x_D) \tag{287}$$

If we have no constraints on the problem, then the extrema are solutions to the following system of equations:

$$\nabla E = 0 \tag{288}$$

which is equivalent to writing $\frac{dE}{dx_i} = 0$ for all i. This equation says that there is no way to infinitesimally perturb $\mathbf{x}$ to get a different value for E; the objective function is locally flat.

Now, however, our goal will be to find extrema subject to a single constraint:

$$g(\mathbf{x}) = 0 \tag{289}$$

In other words, we want to find the extrema among the set of points $\mathbf{x}$ that satisfy $g(\mathbf{x}) = 0$.

It is sometimes possible to reparameterize the problem in order to eliminate the constraints (i.e., so that the new parameterization includes all possible solutions to $g(\mathbf{x}) = 0$), however, this can be awkward in some cases, and impossible in others.

Given the constraint $g(\mathbf{x}) = 0$, we are no longer looking for a point where no perturbation in any direction changes E. Instead, we need to find a point at which perturbations that satisfy the constraints do not change E. This can be expressed by the following condition:

$$\nabla E + \lambda \nabla g \;\; = \;\; 0 \tag{290}$$

for some arbitrary scalar value λ. The expression $\nabla E = -\lambda g$ says that any perturbation to $\mathbf{x}$ that changes E also makes the constraint become violated. Hence, perturbations that do not change g do not change E either. Hence, our goal is to find a point $\mathbf{x}$ that satisfies this condition and also $g(\mathbf{x}) = 0$

In the Method of Lagrange Multipliers, we define a new objective function, called the **Lagrangian**:

$$L(\mathbf{x}, \lambda) = E(\mathbf{x}) + \lambda g(\mathbf{x}) \tag{291}$$

Now we will instead find the extrema of L with respect to both $\mathbf{x}$ and λ. The key fact is that **extrema of the unconstrained objective L are the extrema of the original constrained problem.** So we have eliminated the nasty constraints by changing the objective function and also introducing new unknowns.

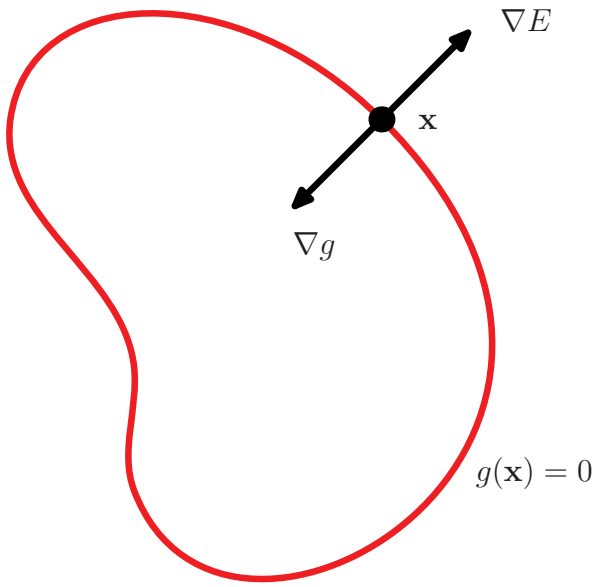

Figure 21: The set of solutions to $g(\mathbf{x}) = 0$ visualized as a curve. The gradient ∇g is always normal to the curve. At an extremal point, ∇E points is parallel to ∇g. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

To see why, let's look at the extrema of L. The extrema to L occur when

$$\frac{dL}{d\lambda} = g(\mathbf{x}) = 0 \tag{292}$$

$$\frac{dL}{d\mathbf{x}} = \nabla E + \lambda \nabla g = 0 \tag{293}$$

which are exactly the conditions given above. Using the Lagrangian is just a convenient way of combining these two constraints into one unconstrained optimization.

14.1 Examples

Minimizing on a circle. We begin with a simple geometric example. We have the following constrained optimization problem:

$$\arg\min_{x,y} x + y \tag{294}$$

$$\text{subject to } x^2 + y^2 = 1 \tag{295}$$

In other words, we want to find the point on a circle that minimizes $x+y$; the problem is visualized in Figure 22. Here, $E(x, y) = x + y$ and $g(x, y) = x^2 + y^2 - 1$. The Lagrangian for this problem is:

$$L(x, y, \lambda) = x + y + \lambda(x^2 + y^2 - 1) \tag{296}$$

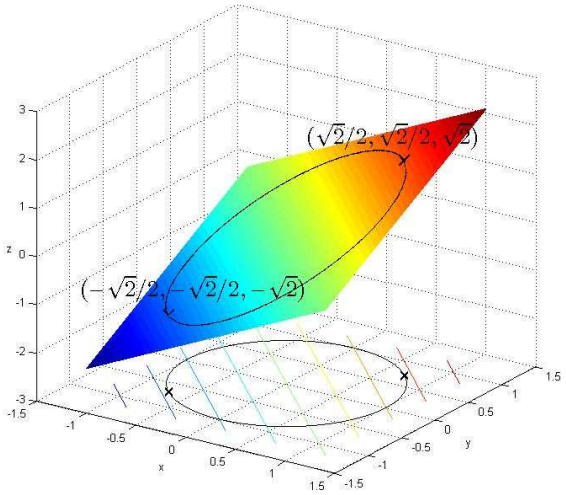

Figure 22: Illustration of the maximization on a circle problem. (Image from Wikipedia.)

Setting the gradient to zero gives this system of equations:

$$\frac{dL}{dx} = 1 + 2\lambda x = 0 \tag{297}$$

$$\frac{dL}{dy} = 1 + 2\lambda y = 0 \tag{298}$$

$$\frac{dL}{d\lambda} = x^2 + y^2 - 1 = 0 \tag{299}$$

From the first two lines, we can see that $x = y$. Substituting this into the constraint and solving gives two solutions $x = y = \pm\frac{1}{\sqrt{2}}$. Substituting these two solutions into the objective, we see that the minimum is at $x = y = -\frac{1}{\sqrt{2}}$.

Estimating a multinomial distribution. In a multinomial distribution, we have an event e with K possible discrete, disjoint outcomes, where

$$P(e = k) = p_k \tag{300}$$

For example, coin-flipping is a binomial distribution where $N = 2$ and $e = 1$ might indicate that the coin lands heads.

Suppose we observe N events; the likelihood of the data is:

$$\prod_{i=1}^{K} P(e_i|p) = \prod_k p_k^{N_k} \tag{301}$$

where N_k is the number of times that $e = k$, i.e., the number of occurrences of the k-th event. To estimate this distribution, we can minimize the negative log-likelihood:

$$\begin{aligned} \text{arg min} \quad & -\textstyle\sum_k N_k \ln p_k & (302) \\ \text{subject to} \quad & \textstyle\sum_k p_k = 1, p_k \geq 0, \text{ for all } k & (303) \end{aligned}$$

The constraints are required in order to ensure that the p's form a valid probability distribution. One way to optimize this problem is to reparameterize: set $p_K = 1 - \sum_{k=1}^{K-1} p_k$, substitute in, and then optimize the unconstrained problem in closed-form. While this method does work in this case, it breaks the natural symmetry of the problem, resulting in some messy calculations. Moreover, this method often cannot be generalized to other problems.

The Lagrangian for this problem is:

$$L(p, \lambda) = -\sum_k N_k \ln p_k + \lambda \left(\sum_k p_k - 1 \right) \tag{304}$$

Here we omit the constraint that $p_k \geq 0$ and hope that this constraint will be satisfied by the solution (it will). Setting the gradient to zero gives:

$$\begin{aligned} \frac{dL}{dp_k} &= -\frac{N_k}{p_k} + \lambda = 0 \text{ for all } k & (305) \\ \frac{dL}{d\lambda} &= \sum_k p_k - 1 = 0 & (306) \end{aligned}$$

Multiplying $dL/dp_k = 0$ by p_k and summing over k gives:

$$0 = -\sum_{k=1}^{K} N_k + \lambda \sum_k p_k = -N + \lambda \tag{307}$$

since $\sum_k N_k = N$ and $\sum_k p_k = 1$. Hence, the optimal $\lambda = N$. Substituting this into dL/dp_k and solving gives:

$$p_k = \frac{N_k}{N} \tag{308}$$

which is the familiar maximum-likelihood estimator for a multinomial distribution.

Maximum variance PCA. In the original formulation of PCA, the goal is to find a low-dimensional projection of N data points $\mathbf{y}$

$$x = \mathbf{w}^T(\mathbf{y} - \mathbf{b}) \tag{309}$$

such that the variance of the $x_i's$ is maximized, subject to the constraint that $\mathbf{w}^T\mathbf{w} = 1$. The Lagrangian is:

$$
\begin{aligned}
L(\mathbf{w}, \mathbf{b}, \lambda) &= \frac{1}{N}\sum_i \left(x_i - \frac{1}{N}\sum_i x_i \right)^2 + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (310)\\
&= \frac{1}{N}\sum_i \left(\mathbf{w}^T(\mathbf{y}_i - \mathbf{b}) - \frac{1}{N}\sum_i \mathbf{w}^T(\mathbf{y}_i - \mathbf{b}) \right)^2 + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (311)\\
&= \frac{1}{N}\sum_i \left(\mathbf{w}^T \left((\mathbf{y}_i - \mathbf{b}) - \frac{1}{N}\sum_i (\mathbf{y}_i - \mathbf{b}) \right) \right)^2 + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (312)\\
&= \frac{1}{N}\sum_i \left(\mathbf{w}^T(\mathbf{y}_i - \bar{\mathbf{y}}) \right)^2 + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (313)\\
&= \frac{1}{N}\sum_i \mathbf{w}^T(\mathbf{y}_i - \bar{\mathbf{y}})(\mathbf{y}_i - \bar{\mathbf{y}})^T\mathbf{w} + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (314)\\
&= \mathbf{w}^T \left(\frac{1}{N}\sum_i (\mathbf{y}_i - \bar{\mathbf{y}})(\mathbf{y}_i - \bar{\mathbf{y}})^T \right) \mathbf{w} + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (315)
\end{aligned}
$$

where $\bar{\mathbf{y}} = \sum_i \mathbf{y}_i/N$. Solving $dL/d\mathbf{w} = 0$ gives:

$$
\left(\frac{1}{N}\sum_i (\mathbf{y}_i - \bar{\mathbf{y}})(\mathbf{y}_i - \bar{\mathbf{y}})^T \right) \mathbf{w} = \lambda\mathbf{w} \qquad (316)
$$

This is just the eigenvector equation: in other words, $\mathbf{w}$ must be an eigenvector of the sample covariance of the $\mathbf{y}'s$, and λ must be the corresponding eigenvalue. In order to determine which one, we can substitute this equality into the Lagrangian to get:

$$
\begin{aligned}
L &= \mathbf{w}^T\lambda\mathbf{w} + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (317)\\
&= \lambda && (318)
\end{aligned}
$$

since $\mathbf{w}^T\mathbf{w} = 1$. Since our goal is to maximize the variance, we choose the eigenvector $\mathbf{w}$ which has the largest eigenvalue λ.

We have not yet selected $\mathbf{b}$, but it is clear that the value of the objective function does not depend on $\mathbf{b}$, so we might as well set it to be the mean of the data $\mathbf{b} = \sum_i \mathbf{y}_i/N$, which results in the $x's$ having zero mean: $\sum_i x_i/N = 0$.

14.2 Least-Squares PCA in one-dimension

We now derive PCA for the case of a one-dimensional projection, in terms of minimizing squared error. Specifically, we are given a collection of data vectors $\mathbf{y}_{1:N}$, and wish to find a bias $\mathbf{b}$, a single

unit vector $\mathbf{w}$, and one-dimensional coordinates $x_{1:N}$, to minimize:

$$\arg \min_{\mathbf{w}, x_{1:N}, \mathbf{b}} \sum_i ||\mathbf{y}_i - (\mathbf{w}x_i + \mathbf{b})||^2 \tag{319}$$

$$\text{subject to } \mathbf{w}^T\mathbf{w} = 1 \tag{320}$$

The vector $\mathbf{w}$ is called the first principal component. The Lagrangian is:

$$L(\mathbf{w}, x_{1:N}, \mathbf{b}, \lambda) = \sum_i ||\mathbf{y}_i - (\mathbf{w}x_i + \mathbf{b})||^2 + \lambda(||\mathbf{w}||^2 - 1) \tag{321}$$

There are several sets of unknowns, and we derive their optimal values each in turn.

Projections (x_i). We first derive the projections:

$$\frac{dL}{dx_i} = -2\mathbf{w}^T(\mathbf{y}_i - (\mathbf{w}x_i + \mathbf{b})) = 0 \tag{322}$$

Using $\mathbf{w}^T\mathbf{w} = 1$ and solving for x_i gives:

$$x_i = \mathbf{w}^T(\mathbf{y}_i - \mathbf{b}) \tag{323}$$

Bias ($\mathbf{b}$). We begin by differentiating:

$$\frac{dL}{d\mathbf{b}} = -2\sum_i(\mathbf{y}_i - (\mathbf{w}x_i + \mathbf{b})) \tag{324}$$

Substituting in Equation 323 gives

$$\frac{dL}{d\mathbf{b}} = -2\sum_i(\mathbf{y}_i - (\mathbf{w}\mathbf{w}^T(\mathbf{y}_i - \mathbf{b}) + \mathbf{b})) \tag{325}$$

$$= -2\sum_i \mathbf{y}_i + 2\mathbf{w}\mathbf{w}^T\sum_i \mathbf{y}_i - 2N\mathbf{w}\mathbf{w}^T\mathbf{b} + 2N\mathbf{b} \tag{326}$$

$$= -2(\mathbf{I} - \mathbf{w}\mathbf{w}^T)\sum_i \mathbf{y}_i + 2(\mathbf{I} - \mathbf{w}\mathbf{w}^T)N\mathbf{b} = 0 \tag{327}$$

Dividing both sides by $2(\mathbf{I} - \mathbf{w}\mathbf{w}^T)N$ and rearranging terms gives:

$$\mathbf{b} = \frac{1}{N}\sum_i y_i \tag{328}$$

Basis vector (w). To make things simpler, we will define $\tilde{\mathbf{y}}_i = (\mathbf{y}_i - \mathbf{b})$ as the mean-subtracted data points, and the reconstructions are then $x_i = \mathbf{w}^T\tilde{\mathbf{y}}_i$, and the objective function is:

$$\begin{aligned} L &= \sum_i ||\tilde{\mathbf{y}}_i - \mathbf{w}x_i||^2 + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (329)\\ &= \sum_i ||\tilde{\mathbf{y}}_i - \mathbf{w}\mathbf{w}^T\tilde{\mathbf{y}}_i||^2 + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (330)\\ &= \sum_i (\tilde{\mathbf{y}}_i - \mathbf{w}\mathbf{w}^T\tilde{\mathbf{y}}_i)^T(\tilde{\mathbf{y}}_i - \mathbf{w}\mathbf{w}^T\tilde{\mathbf{y}}_i) + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (331)\\ &= \sum_i (\tilde{\mathbf{y}}_i^T\tilde{\mathbf{y}}_i - 2\tilde{\mathbf{y}}_i^T\mathbf{w}\mathbf{w}^T\tilde{\mathbf{y}}_i + \tilde{\mathbf{y}}_i^T\mathbf{w}\mathbf{w}^T\mathbf{w}\mathbf{w}^T\tilde{\mathbf{y}}_i) + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (332)\\ &= \sum_i \tilde{\mathbf{y}}_i^T\tilde{\mathbf{y}}_i - \sum_i (\tilde{\mathbf{y}}_i^T\mathbf{w})^2 + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (333) \end{aligned}$$

where we have used $\mathbf{w}^T\mathbf{w} = 1$. We then differentiate and simplify:

$$\frac{dL}{d\mathbf{w}} = -2\sum_i \tilde{\mathbf{y}}_i\tilde{\mathbf{y}}_i^T\mathbf{w} + 2\lambda\mathbf{w} = 0 \quad (334)$$

We can rearrange this to get:

$$\left(\sum_i \tilde{\mathbf{y}}_i\tilde{\mathbf{y}}_i^T\right)\mathbf{w} = \lambda\mathbf{w} \quad (335)$$

This is exactly the eigenvector equation, meaning that extrema for L occur when $\mathbf{w}$ is an eigenvector of the matrix $\sum_i \tilde{\mathbf{y}}_i\tilde{\mathbf{y}}_i^T$, and λ is the corresponding eigenvalue. Multiplying both sides by $1/N$, we see this matrix has the same eigenvectors as the data covariance:

$$\left(\frac{1}{N}\sum_i (\mathbf{y}_i - \mathbf{b})(\mathbf{y}_i - \mathbf{b})^T\right)\mathbf{w} = \frac{\lambda}{N}\mathbf{w} \quad (336)$$

Now we must determine which eigenvector to use. We rewrite Equation 333 as:

$$\begin{aligned} L &= \sum_i \tilde{\mathbf{y}}_i^T\tilde{\mathbf{y}}_i - \sum_i \mathbf{w}^T\tilde{\mathbf{y}}_i\tilde{\mathbf{y}}_i^T\mathbf{w} + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (337)\\ &= \sum_i \tilde{\mathbf{y}}_i^T\tilde{\mathbf{y}}_i - \mathbf{w}^T\left(\sum_i \tilde{\mathbf{y}}_i\tilde{\mathbf{y}}_i^T\right)\mathbf{w} + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (338)\\ & && (339) \end{aligned}$$

and substitute in Equation 335:

$$\begin{aligned} L &= \sum_i \tilde{\mathbf{y}}_i^T\tilde{\mathbf{y}}_i - \lambda\mathbf{w}^T\mathbf{w} + \lambda(\mathbf{w}^T\mathbf{w} - 1) && (340)\\ &= \sum_i \tilde{\mathbf{y}}_i^T\tilde{\mathbf{y}}_i - \lambda && (341) \end{aligned}$$

again using $\mathbf{w}^T\mathbf{w} = 1$. We must pick the eigenvalue λ that gives the smallest value of L. Hence, we pick the largest eigenvalue, and set $\mathbf{w}$ to be the corresponding eigenvector.

14.3 Multiple constraints

When we wish to optimize with respect to multiple constraints $\{g_k(\mathbf{x})\}$, i.e.,

$$\arg\min_{\mathbf{x}} E(\mathbf{x}) \tag{342}$$

$$\text{subject to } g_k(\mathbf{x}) = 0 \text{ for } k = 1...K \tag{343}$$

Extrema occur when:

$$\nabla E + \sum_k \lambda_k \nabla g_k \;\; = \;\; 0 \tag{344}$$

where we have introduced K Lagrange multipliers λ_k. The constraints can be combined into a single Lagrangian:

$$L(\mathbf{x}, \lambda_{1:K}) = E(\mathbf{x}) + \sum_k \lambda_k g_k(\mathbf{x}) \tag{345}$$

14.4 Inequality constraints

The method can be extended to inequality constraints of the form $g(\mathbf{x}) \geq 0$. For a solution to be valid and maximal, there two possible cases:

- The optimal solution is inside the constraint region, and, hence $\nabla E = 0$ and $g(\mathbf{x}) > 0$. In this region, the constraint is "inactive," meaning that λ can be set to zero.
- The optimal solution lies on the boundary $g(\mathbf{x}) = 0$. In this case, the gradient ∇E must point in the *opposite* direction of the gradient of g; otherwise, following the gradient of E would cause g to become positive while also modifying E. Hence, we must have $\nabla E = -\lambda \nabla g$ for $\lambda \geq 0$.

Note that, in both cases, we have $\lambda g(\mathbf{x}) = 0$. Hence, we can enforce that one of these cases is found with the following optimization problem:

$$\max_{\mathbf{w},\lambda} \quad E(\mathbf{x}) + \lambda g(\mathbf{x}) \tag{346}$$

$$\text{such that} \quad g(\mathbf{x}) \geq 0 \tag{347}$$

$$\lambda \geq 0 \tag{348}$$

$$\lambda g(\mathbf{x}) = 0 \tag{349}$$

These are called the Karush-Kuhn-Tucker (KKT) conditions, which generalize the Method of Lagrange Multipliers.

When minimizing, we want ∇E to point in the same direction as ∇g when on the boundary, and so we minimize $E - \lambda g$ instead of $E + \lambda g$.

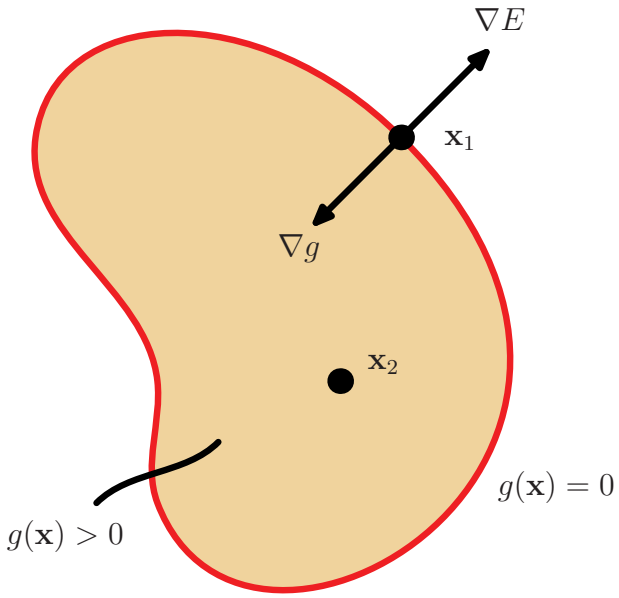

Figure 23: Illustration of the condition for inequality constraints: the solution may lie on the boundary of the constraint region, or in the interior. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

15 Clustering

Clustering is an **unsupervised** learning problem in which our goal is to discover "clusters" in the data. A cluster is a collection of data that are similar in some way.

Clustering is often used for several different problems. For example, a market researcher might want to identify distinct groups of the population with similar preferences and desires. When working with documents you might want to find clusters of documents based on the occurrence frequency of certain words. For example, this might allow one to discover financial documents, legal documents, or email from friends. Working with image collections you might find clusters of images which are images of people versus images of buildings. Often when we are given large amounts of complicated data we want to look for some underlying structure in the data, which might reflect certain *natural kinds* within the training data. Clustering can also be used to compress data, by replacing all of the elements in a cluster with a single representative element.

15.1 K-means Clustering

We begin with a simple method called K-means. Given N input data vectors $\{\mathbf{y}_i\}_{i=1}^N$, we wish to label each vector as belonging to one of K clusters. This labeling will be done via a binary matrix $\mathbf{L}$, the elements of which are given by

$$L_{i,j} = \begin{cases} 1 & \text{if data point } i \text{ belongs to cluster } j \\ 0 & \text{otherwise} \end{cases} \tag{350}$$

The clustering is mutually exclusive. Each data vector i can only be assigned to only cluster: $\sum_{j=1}^K L_{i,j} = 1$. Along the way, we will also be estimating a center $\mathbf{c}_j$ for each cluster.

The full objective function for K-means clustering is:

$$E(\mathbf{c}, \mathbf{L}) = \sum_{i,j} L_{i,j} ||\mathbf{y}_i - \mathbf{c}_j||^2 \tag{351}$$

This objective function penalizes the distance between each data point and the center of the cluster to which it is assigned. Hence, to minimize this error, we want to bring the cluster centers close to the data it has been assigned, and we also want to assign the data to nearby centers.

This objective function cannot be optimized in closed-form, and so an iterative method is required. It includes discrete variables (the labels $\mathbf{L}$), and so gradient-based methods aren't directly applicable. Instead, we use a strategy called **coordinate descent**, in which we alternate between closed-form optimization of one set of variables holding the other variables fixed. That is, we first pick initial values, then we alternate between updating the labels for the current centers, and then updating the centers for the current labels.

Here is the K-means algorithm:

pick initial values for $\mathbf{L}$ and $\mathbf{c}_{1:K}$
loop
  // Labeling update: set $\mathbf{L} \leftarrow \arg\min_{\mathbf{L}} E(\mathbf{c}, \mathbf{L})$
  for each data point i **do**
    $j \leftarrow \arg\min_j ||\mathbf{y}_i - \mathbf{c}_j||^2$
    $L_{i,j} = 1$
    $L_{i,a} = 0$ for all $a \neq j$
  end for
  // Centers update: set $\mathbf{c} \leftarrow \arg\min_{\mathbf{c}} E(\mathbf{c}, \mathbf{L})$
  for each center j **do**
    $\mathbf{c}_j \leftarrow \frac{\sum_i L_{i,j} \mathbf{y}_i}{\sum_i L_{i,j}}$
  end for end loop

Each step of the optimization is guaranteed to lower the objective function until the algorithm converges (you should be able to show that each step is optimal.) However, there is no guarantee that the algorithm will find the global optimum and indeed it may easily get trapped in a poor local minima.

Initialization. The algorithm is sensitive to initialization, and poor initialization can sometimes lead to very poor results. Here are a few strategies that can be used to initialize the algorithm:

1. **Random labeling.** Initialize the labeling $\mathbf{L}$ randomly, and then run the center-update step to determine the initial centers. This approach is not recommended because the initial centers will likely end up just being very close to the mean of the data.

2. **Random initial centers.** We could try to place initial center locations randomly, e.g., by random sampling in the bounding box of the data. However, it is very likely that some of the centers will fall into empty regions of the feature space, and will therefore be assigned no data. Getting a good initialization this way can be difficult.

3. **Random data points as centers.** This method works much better: use a random subset of the data as the initial center locations.

4. K**-medoids clustering**. This will be described below.

5. **Multiple restarts.** In multiple restarts, we run K-means multiple times, each time with a different random initialization (using one of the above methods). We then take the best clustering out of all of the runs, based on the value of the objective function above in Equation (351).

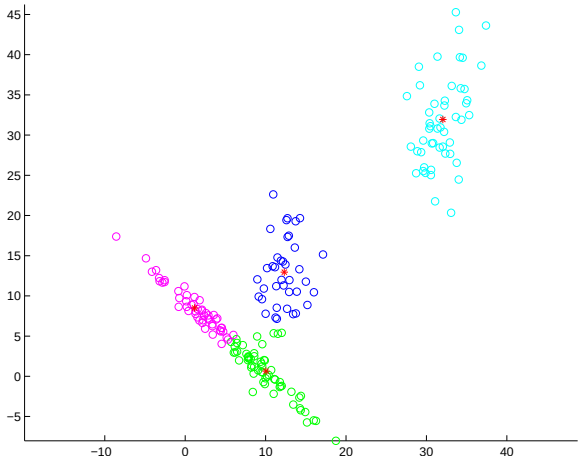

Figure 24: K-means applied to a dataset sampled from three Gaussian distributions. Each data assigned to each cluster are drawn as circles with distinct colours. The cluster centers are shown as red stars.

Another key question is how one chooses the number of clusters, i.e., K. A common approach is to fix K based on some prior knowledge or computational constraints. One can also try different values of K, adding another term to to the objective function to penalize model complexity.

15.2 K-medoids Clustering

(The material in this section is not required for this course.)

K-medoids clustering is a variant of K-means with the additional constraint that the cluster centers must be drawn from the data. The following algorithm, called Farthest First Traversal, or Hochbaum-Shmoys, is simple and effective:

Randomly select a data point $\mathbf{y}_i$ as the first cluster center: $\mathbf{c}_1 \leftarrow \mathbf{y}_i$
for $j = 2$ to K
  Find the data point furthest from all existing centers:
  $i \leftarrow \arg\max_i \min_{k<j} ||\mathbf{y}_i - \mathbf{c}_k||^2$
  $\mathbf{c}_j \leftarrow \mathbf{y}_i$
end for
Label all remaining data points according to their nearest centers (as in k-means)

This algorithm provides a quality guarantee: it gives a clustering that is no worse than twice the error of the optimal clustering.

K-medoids clustering can also be improved by coordinate descent. The labeling step is the same as in K-means. However, the cluster updates must be done by brute-force search for each candidate cluster center update.

15.3 Mixtures of Gaussians

The Mixtures-of-Gaussians (MoG) model is a generalization of K-means clustering. Whereas K-means clustering works for clusters that are more or less spherical, the MoG model can handle oblong clusters and overlapping clusters. The K-means algorithm does an excellent job when clusters are well separated, but not when the clusters overlap. MoG algorithms compute a "soft," probabilistic clustering which allows the algorithm to better handle overlapping clusters. Finally, the MoG model is probabilistic, and so it can be used to learn probability distributions from data.

The MoG model consists of K Gaussian distributions, each with their own means and covariances $\{(\mu_j, \mathbf{K}_j)\}$. Each Gaussian also has an associated (prior) probability a_j, such that $\sum_j a_j = 1$. That is, the probabilities a_j will represent the fraction of the data that are assigned to (or generated by) the different Gaussian components. As a shorthand, we will write all the model parameters with a single variable, i.e., $\theta = \{a_{1:K}, \mu_{1:K}, \mathbf{K}_{1:K}\}$. When used for clustering, the idea is that each Gaussian component in the mixture should correspond to a single cluster.

The complete probabilistic model comprises the prior probabilities of each Gaussian component, and Gaussian likelihood over the data (or feature) space for each component:

$$P(L = j|\theta) = a_j \tag{352}$$

$$p(\mathbf{y}|\theta, L = j) = G(\mathbf{y};\, \mu_j, \mathbf{K}_j) \tag{353}$$

To sample a single data point from this (generative) model, we first randomly select a Gaussian component according to their prior probabilities $\{a_j\}$, and then we randomly sample from the corresponding Gaussian component. The likelihood of a single data point can be derived by the product rule and the sum rule as follows:

$$p(\mathbf{y}|\theta) = \sum_{j=1}^{K} p(\mathbf{y}, L = j|\theta) \tag{354}$$

$$= \sum_{j=1}^{K} p(\mathbf{y}|L = j, \theta)\, P(L = j|\theta) \tag{355}$$

$$= \sum_{j=1}^{K} a_j \frac{1}{\sqrt{(2\pi)^D |K_j|}} e^{-\frac{1}{2}(\mathbf{y}-\mu_j)^T \mathbf{K}_j^{-1}(\mathbf{y}-\mu_j)} \tag{356}$$

where D is the dimension of data vectors. This model can be interpreted as a linear combination (or blend) of Gaussians: we get a multimodal distribution by adding together unimodal Gaussians. Interestingly, the MoG model is similar to the Gaussian Class-Conditional model that we used for classification; the difference is that the class labels will no longer be included in the training set.

In general, the approach of building models by mixtures is quite general and can be used for many other types of distributions as well, for example, we could build a mixture of Student-t distributions, or a mixture of a Gaussian and a uniform, and so on.

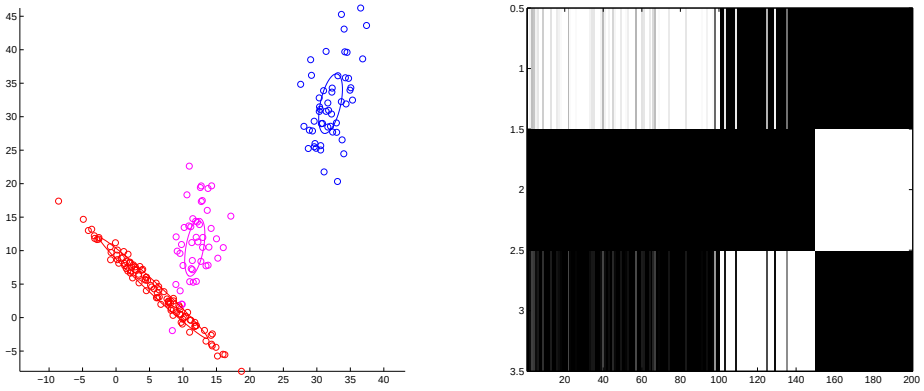

Figure 25: Mixture of Gaussians model applied to a dataset generated from three Gaussians. The resulting γ is visualized on the right. The data points are shown as colored circles. The color is determined by the cluster with the highest posterior assignment probability γ_{ij}. One standard deviation ellipses are shown for each Gaussian. Note that the blue points are well isolated and there is little ambiguity in their assignments. The other two distributions overlap, and one can see how the orientation and eccentricity of the covariance structure (the ellipses) influence the assignment probabilities.

15.3.1 Learning

Given a data set $\mathbf{y}_{1:N}$, where each data point is assumed to be drawn independently from the model, we learn the model parameters, θ, by minimizing the negative log-likelihood of the data:

$$\begin{aligned} \mathcal{L}(\theta) &= -\ln p(\mathbf{y}_{1:N}|\theta) && (357) \\ &= -\sum_i \ln p(\mathbf{y}_i|\theta) && (358) \end{aligned}$$

Note that this is a constrained optimization, since we require $a_j \geq 0$ and $\sum_j a_j = 1$. Furthermore, $\mathbf{K}_j$ must be symmetric, positive-definite matrix to be a covariance matrix. Unfortunately, this optimization cannot be performed in closed-form.

One approach is to use gradient descent to optimization by gradient descent. There are a few issues associated with doing so. First, some care is required to avoid numerical issues, as discussed below. Second, this learning is a constrained optimization, due to constraints on the values of the a's. One solution is to project onto the constraints during optimization: at each gradient descent step (and inside the line search loop), we clamp all negative a values to zero and renormalize the a's so that they sum to one. Another option is to reparameterize the problem to be unconstrained. Specifically, we define new variables β_j, and define the a's as functions of the βs, e.g.,

$$a_j(\beta) = \frac{e^{\beta_j}}{\sum_{j=1}^K e^{\beta_j}} \quad (359)$$

This definition ensures that, for any choice of the βs, the as will satisfy the constraints. We substitute this expression into the model definition and then optimize for the βs instead of the as with gradient descent. Similarly, we can enforce the constraints on the covariance matrix by reparameterization; this is normally done using a upper-triangular matrix $\mathbf{U}$ such that $\mathbf{K} = \mathbf{U}^T\mathbf{U}$.

An alternative to gradient descent is the **Expectation-Maximization** algorithm, or EM. EM is a quite general algorithm for "hidden variable" problems; in this case, the labels L are "hidden" (or "unobserved"). In EM, we define a probabilistic labeling variable $\gamma_{i,j}$. The variable $\gamma_{i,j}$ corresponds to the probability that data point i came from cluster j: $\gamma_{i,j}$ is meant to estimate $P(L = j|\mathbf{y}_i)$. In EM, we optimize both θ and γ together. The algorithm alternates between the "E-step" which updates the γs, and the "M-step" which updates the model parameters θ.

pick initial values for γ and θ
loop
  E-step:
  for each data point i **do**
    $\gamma_{i,j} \leftarrow P(L = j|\mathbf{y}_i, \theta)$
  end for
  M-step:
  for each cluster j **do**
    $a_j \leftarrow \frac{\sum_i \gamma_{i,j}}{N}$
    $\mu_j \leftarrow \frac{\sum_i \gamma_{i,j}\mathbf{y}_i}{\sum_i \gamma_{i,j}}$
    $\mathbf{K}_j \leftarrow \frac{\sum_i \gamma_{i,j}(\mathbf{y}_i-\mu_j)(\mathbf{y}_i-\mu_j)^T}{\sum_i \gamma_{i,j}}$
  end for
end loop

Note that the E-step is the same as classification in the Gaussian Class-Conditional model.

The EM algorithm is a local optimization algorithm, and so the results will depend on initialization. Initialization strategies similar to those used for K-means above can be used.

15.3.2 Numerical issues

Exponentiating very small negative numbers can often lead to underflow when implemented in floating-point arithmetic, e.g., e^{-A} will give zero for large A, and $\ln e^{-A}$ will give an error (or `-Inf`) whereas it should return $-A$. These issues will often cause machine learning algorithms to fail; MoG has several steps which are susceptible. Fortunately, there are some simple tricks that can be used.

1. Many computations can be performed directly in the log domain. For example, it may be

more stable to compute

$$ae^b \tag{360}$$

as

$$e^{\ln a + b} \tag{361}$$

This avoids issues where b is so small that e^b evaluates to zero in floating point, but ae^b is much greater than zero.

2. When computing an expression of the form:

$$\frac{e^{-\beta_j}}{\sum_j e^{-\beta_j}} \tag{362}$$

large values of β could lead to the above expression being zero for all j, even though the expression must sum to one. This may arise, for example, when computing the γ updates, which have the above form. The solution is to make use of the identity:

$$\frac{e^{-\beta_j}}{\sum_j e^{-\beta_j}} = \frac{e^{-\beta_j + C}}{\sum_j e^{-\beta_j + C}} \tag{363}$$

for any value of C. We can choose C to prevent underflow; a suitable choice is $C = \min_j \beta_j$.

3. Underflow can also occur when evaluating

$$\ln \sum_i e^{-\beta_j} \tag{364}$$

which can be fixed by using the identity

$$\ln \sum_i e^{-\beta_j} = \left(\ln \sum_i e^{-\beta_j + C} \right) - C \tag{365}$$

15.3.3 The Free Energy

Amazingly, EM optimizes the log-likelihood, which doesn't even have a γ parameter. In order to understand the EM algorithm and why it works, it is helpful to introduce a quantity called the **Free Energy**:

$$\begin{aligned}
\mathcal{F}(\theta, \gamma) &= -\sum_{i,j} \gamma_{i,j} \ln p(\mathbf{y}_i, L = j | \theta) + \sum_{i,j} \gamma_{i,j} \ln \gamma_{i,j} && (366)\\
&= \frac{1}{2} \sum_{i,j} \gamma_{i,j} (\mathbf{y}_i - \mu_j)^T \mathbf{K}_j^{-1} (\mathbf{y}_i - \mu_j) && (367)\\
&\quad + \frac{1}{2} \sum_{i,j} \gamma_{i,j} \ln (2\pi)^D |\mathbf{K}_j| - \sum_{i,j} \gamma_{i,j} \ln a_j && (368)\\
&\quad + \sum_{i,j} \gamma_{i,j} \ln \gamma_{i,j} && (369)
\end{aligned}$$

The EM algorithm is a coordinate descent algorithm for optimizing the free energy, subject to the constraint that $\sum_j \gamma_{i,j} = 1$ and the constraints on a. In other words, EM can be written compactly as:

pick initial values for γ and θ
loop
E-step:
$\gamma \leftarrow \arg\min_\gamma \mathcal{F}(\theta, \gamma)$
M-step:
$\theta \leftarrow \arg\min_\theta \mathcal{F}(\theta, \gamma)$
end loop

However, the free energy is different from the negative log-likelihood $\mathcal{L}(\theta)$ that we initially set out to minimize. Fortunately, the free energy has the following important properties:

- When the value of γ is optimal, the Free Energy is equal to the negative log-likelihood:

$$\mathcal{L}(\theta) = \min_\gamma \mathcal{F}(\theta, \gamma) \tag{370}$$

  We can use this fact to evaluate the negative log-likelihood simply by running an E-step and then computing the free energy. In fact, this is often more efficient than directly computing the negative log-likelihood. The proof is given in the next section.

- The minima of the free energy are also minima of the negative log-likelihood:

$$\min_\theta \mathcal{L}(\theta) = \min_{\theta,\gamma} \mathcal{F}(\theta, \gamma) \tag{371}$$

  This follows from the previous property. Hence, **optimizing the free energy is the same as optimizing the negative log-likelihood.**

- The Free Energy is an upper-bound on the negative log-likelihood:

$$\mathcal{F}(\theta, \gamma) \geq \mathcal{L}(\theta) \tag{372}$$

  for all values of γ. This observation gives a sanity check for debugging the free energy computation.

The Free Energy also provides a very helpful tool for debugging: any step of an implementation that increases the free energy must be incorrect. The term Free Energy arises from its original definition in statistical physics.

15.3.4 Proofs

This content of this section is not required material for this course and you may skip it. Here we outline proofs for the key features of the free energy.

EM updates. The steps of the EM algorithm may be derived by solving $\arg\min_\gamma \mathcal{F}(\theta, \gamma)$ and $\arg\min_\theta \mathcal{F}(\theta, \gamma)$. In most cases, the derivations generalize familiar ones, e.g., weighted least-squares. The a and γ parameters are multinomial distributions, and optimization of them requires Lagrange multipliers or reparameterization. One may ignore the positivity constraint, as it turns out to be automatically satisfied. The details will be skipped here.

Equality after the E-step. The E-step computes the optimal value for γ:

$$\gamma^* \leftarrow \arg\min_\gamma \mathcal{F}(\theta, \gamma) \tag{373}$$

which is given by:

$$\gamma^*_{i,j} = P(L = j|\mathbf{y}_i) \tag{374}$$

Substituting this into the Free Energy gives:

$$\begin{aligned}
\mathcal{F}(\theta, \gamma^*) &= -\sum_{i,j} P(L = j|\mathbf{y}_i) \ln \frac{p(\mathbf{y}_i, L = j)}{P(L = j|\mathbf{y}_i)} && (375)\\
&= -\sum_{i,j} P(L = j|\mathbf{y}_i) \ln p(\mathbf{y}_i) && (376)\\
&= -\sum_i \left(\ln p(\mathbf{y}_i) \sum_j P(L = j|\mathbf{y}_i) \right) && (377)\\
&= -\sum_i \ln p(\mathbf{y}_i) && (378)\\
&= \mathcal{L}(\theta) && (379)
\end{aligned}$$

Hence,

$$\mathcal{L}(\theta) = \min_\gamma \mathcal{F}(\theta, \gamma) \tag{380}$$

Bound. An important building block in proving that $\mathcal{F}(\theta, \gamma) \geq \mathcal{L}(\theta)$ is **Jensen's Inequality**, which applies since $\ln$ is a "concave" function and $\sum_j b_j = 1$, $b_j \geq 0$.

$$\begin{aligned}
\ln \sum_j b_j x_j &\geq \sum_j b_j \ln x_j, \quad \text{or} && (381)\\
-\ln \sum_j b_j x_j &\leq -\sum_j b_j \ln x_j && (382)
\end{aligned}$$

We will not prove this here.

We can then derive the bound as follows, dropping the dependence on θ for brevity:

$$\begin{aligned}
\mathcal{L}(\theta) &= -\sum_i \ln \sum_j p(\mathbf{y}_i, L=j) && (383)\\
&= -\sum_i \ln \sum_j \frac{\gamma_{i,j}}{\gamma_{i,j}} p(\mathbf{y}_i, L=j) && (384)\\
&\leq -\sum_{i,j} \gamma_{i,j} \ln \frac{p(\mathbf{y}_i, L=j)}{\gamma_{i,j}} && (385)\\
&= \mathcal{F}(\theta, \gamma) && (386)
\end{aligned}$$

15.3.5 Relation to K-means

It should be clear that the K-means algorithm is very closely related to EM. In fact, EM reduces to K-means if we make the following restrictions on the model:

- The class probabilities are equal: $a_j = \frac{1}{K}$.
- The Gaussians are spherical with identical variances: $\mathbf{K}_j = \sigma^2 \mathbf{I}$ for all j.
- The Gaussian variances are infinitesimal, i.e., we consider the algorithm in the limit as $\sigma^2 \to 0$. This causes the optimal values for γ to be binary, since, if j is the nearest class, $\lim_{\sigma^2 \to 0} P(L=j|\mathbf{y}_i) = 1$.

With these modifications, the Free Energy becomes equivalent to the K-means objective function, up to constant values, and the EM algorithm becomes identical to K-means.

15.3.6 Degeneracy

There is a degeneracy in the MoG objective function. Suppose we center one Gaussian at one of the data points, so that $\mathbf{c}_j = \mathbf{y}_i$. The error for this data point will be zero, and by reducing the variance of this Gaussian, we can always increase the likelihood of the data. In the limit as this Gaussian's variance goes to zero, the data likelihood goes to infinity. Hence, some effort may be required to avoid this situation. This degeneracy can also be avoided by using a more Bayesian form of the algorithm, e.g., marginalizing out the cluster centers rather than estimating them.

15.4 Determining the number of clusters

Determining the value of K is a model selection problem: we want to determine the most-likely value of K given the data. Cross validation is not appropriate here, since we do not have any supervision (e.g., correct labels from a subset of the data). Bayesian model selection can be employed, e.g., by maximizing

$$K^* = \arg\max_K P(K|\mathbf{y}_{1:N}) = \arg\max_K \int p(K, \theta|\mathbf{y}_{1:N}) d\theta \qquad (387)$$

where θ are the model parameters. This evaluation is somewhat mathematically-involved. A very coarse approximation to this computation is Bayesian Information Criterion (BIC).

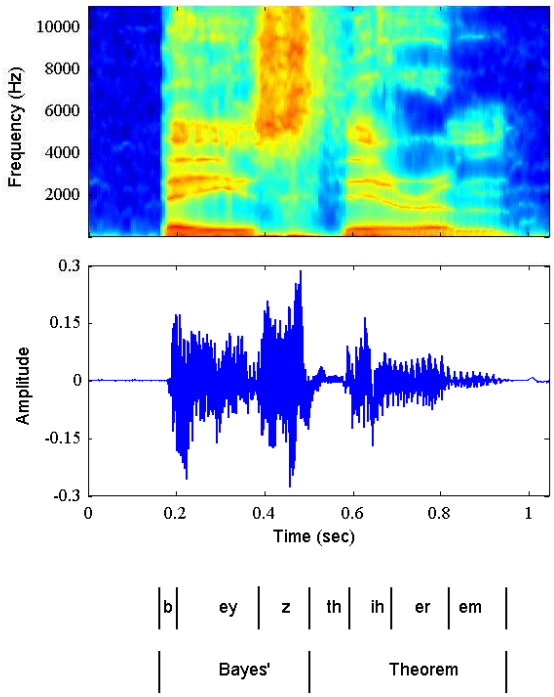

Figure 26: Examples of time-series data: speech and language.

16 Hidden Markov Models

Until now, we have exclusively dealt with data sets in which each individual measurement is independent and identically distributed (IID). That is, for two points $\mathbf{y}_1$ and $\mathbf{y}_2$ in our data set, we have $p(\mathbf{y}_1) = p(\mathbf{y}_2)$ and $p(\mathbf{y}_1, \mathbf{y}_2) = p(\mathbf{y}_1)p(\mathbf{y}_2)$ (for a fixed model). Time-series data consist of sequences of data that are not IID: they arise from a process that varies over time, and modeling the dynamics of this process is important.

16.1 Markov Models

Markov models are time series that have the Markov property:

$$P(s_t|s_{t-1}, s_{n-2}, ..., s_1) = P(s_t|s_{t-1}) \tag{388}$$

where s_t is the state of the system at time t. Intuitively, this property says the probability of a state at time t is competely determined by the system state at the previous time step. More generally, for any set A of indices less than t and set of indices B greater than t we have:

$$P(s_t|\{s_i\}_{i\in A\cup B}) = P(s_t|s_{\max(A)}, s_{\min(B)}) \tag{389}$$

which follows from the Markov property.

Another useful identity which also follows directly from the Markov property is:

$$P(s_{t-1}, s_{t+1}|s_t) = P(s_{t-1}|s_t)P(s_{t+1}|s_t) \tag{390}$$

Discrete Markov Models. A important example of Markov chains are discrete Markov models. Each state s_t can take on one of a discrete set of states, and the probability of transitioning from one state to another is governed by a probability table for the whole sequence of states. More concretely, $s_t \in \{1, ..., K\}$ for some finite K and, for all times t, $P(s_t = j|s_{t-1} = i) = A_{ij}$ where A is parameter of the model that is a fixed matrix of valid probabilities (so that $A_{ij} \geq 0$ and $\sum_{j=1}^{K} A_{ij} = 1$). To fully characterize the model, we also require a distribution over states for the first time-step: $P(s_1 = i) = a_i$.

16.2 Hidden Markov Models

A Hidden Markov model (HMM) models a time-series of observations $\mathbf{y}_{1:T}$ as being determined by a "hidden" discrete Markov chain $s_{1:T}$. In particular, the measurement $\mathbf{y}_t$ is assumed to be determined by an "emission" distribution that depends on the hidden state at time t: $p(\mathbf{y}_t|s_t = i)$. The Markov chain is called "hidden" because we do not measure it, but must reason about it indirectly. Typically, s_t encodes underlying structure of the time-series, where as the $\mathbf{y}_t$ correspond to the measurements that are actually observed. For example, in speech modeling applications, the measurements $\mathbf{y}$ might be the waveforms measured from a microphone, and the hidden states might be the corresponding word that the speaker is uttering. In language modeling, the measurements might be discrete words, and the hidden states their underlying parts of speech.

HMMs can be used for discrete or continuous data; in this course, we will focus solely on the continuous case, with Gaussian emission distributions.

The joint distribution over observed and hidden is:

$$p(s_{1:T}, \mathbf{y}_{1:T}) = p(\mathbf{y}_{1:T}|s_{1:T})P(s_{1:T}) \tag{391}$$

where

$$P(s_{1:T}) = P(s_1) \prod_{t=2}^{T} P(s_t|s_{t-1}) \tag{392}$$

and

$$P(\mathbf{y}_{1:T}|s_{1:T}) = \prod_{t=1}^{T} p(\mathbf{y}_t|s_t) \tag{393}$$

The Gaussian model says:

$$p(\mathbf{y}_t|s_t = i) = \mathcal{N}(\mathbf{y}_t; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \tag{394}$$

for some mean and covariance parameters $\boldsymbol{\mu}_i$ and $\boldsymbol{\Sigma}_i$. In other words, each state i has its own Gaussian with its own parameters. A complete HMM consists of the following parameters: $a, A, \boldsymbol{\mu}_{1:K}$,

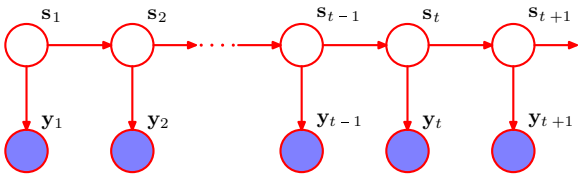

Figure 27: Illustration of the variables in an HMM, and their conditional dependencies. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

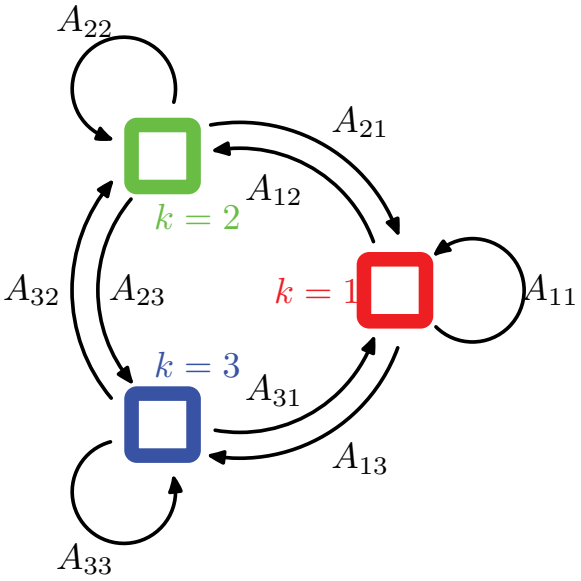

Figure 28: The hidden states of an HMM correspond to a state machine. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

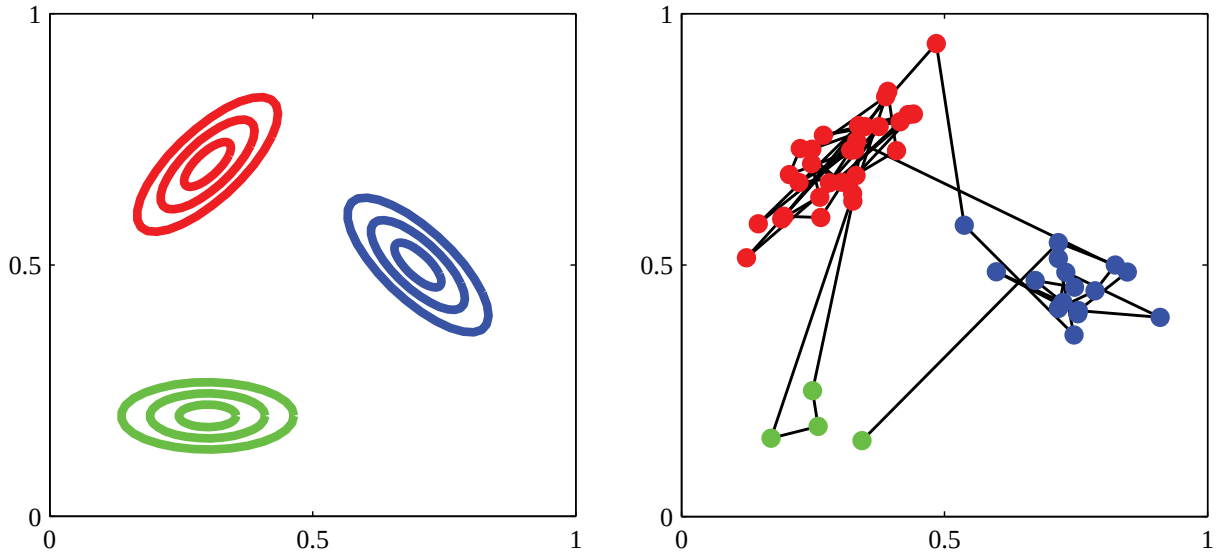

Figure 29: Illustration of sampling a sequence of datapoints from an HMM. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

and $\boldsymbol{\Sigma}_{1:K}$. As a short-hand, we will denote these parameters by a variable $\theta = \{a, A, \boldsymbol{\mu}_{1:K}, \boldsymbol{\Sigma}_{1:K}\}$.

Note that, if $A_{ij} = a_j$ for all i, then this model is equivalent to a Mixtures-of-Gaussian model with mixing proportions given by the a_i's, since the distribution over states at any instant does not depend on the previous state.

In the remainder of this chapter, we will discuss algorithms for computing with HMMs.

16.3 Viterbi Algorithm

We begin by considering the problem of computing the most-likely sequence of states given a data set $\mathbf{y}_{1:T}$ and a known HMM model. That is, we wish to compute

$$s^*_{1:T} = \arg\max_{s_{1:T}} P(s_{1:T}|\theta, \mathbf{y}_{1:T}) \tag{395}$$

The naive approach is to simply enumerate every possible state sequence and choose the one that maximizes the above conditional probability. Since there are K^T possible state-sequences, this approach is clearly infeasible for sequences of more than a few steps.

Fortunately, we can take advantage of the Markov property to perform this computation much more efficiently. The Viterbi algorithm is a dynamic programming approach to finding the most likely sequence of states $s_{1:T}$ given θ and a sequence of observations $\mathbf{y}_{1:T}$.

We begin by defining the following quantity for each state and each time-step:

$$\delta_t(i) \equiv \max_{s_{1:t-1}} p(s_{1:t-1}, s_t = i, \mathbf{y}_{1:t}) \tag{396}$$

(Henceforth, we omit θ from these equations for brevity.) This quantity tells us the likelihood that the most-likely sequence up to time t ends at state i, given the data up to time t. We will compute this quantity recursively. The base case is simply:

$$\delta_1(i) = p(s_1 = i, \mathbf{y}_1) = p(\mathbf{y}_1|s_1 = i)P(s_1 = i) \tag{397}$$

for all i. The recursive case is:

$$\begin{aligned}
\delta_t(i) &= \max_{s_{1:t-1}} p(s_{1:t-1}, s_t = i, \mathbf{y}_{1:t}) \\
&= \max_{s_{1:t-2}, j} p(s_t = i|s_{t-1} = j)p(\mathbf{y}_t|s_t = i)p(s_{1:t-2}, s_{t-1} = j, \mathbf{y}_{1:t-1}) \\
&= p(\mathbf{y}_t|s_t = i) \max_j \left[p(s_t = i|s_{t-1} = j) \max_{s_{1:t-2}} p(s_{1:t-2}, s_{t-1} = j, \mathbf{y}_{1:t-1}) \right] \\
&= p(\mathbf{y}_t|s_t = i) \max_j A_{ji}\delta_{t-1}(j)
\end{aligned}$$

Once we have computed δ for all time-steps and all states, we can determine the final state of the most-likely sequence as:

$$\begin{aligned}
s_T^* &= \arg\max_i P(s_T = i|\mathbf{y}_{1:T}) && (398)\\
&= \arg\max_i P(s_T = i, \mathbf{y}_{1:T}) && (399)\\
&= \arg\max_i \delta_T(i) && (400)
\end{aligned}$$

since $p(\mathbf{y}_{1:T})$ does not depend on the state sequence. We can then backtrack through δ to determine the states of each previous time-step, by finding which state j was used to compute each maximum in the recursive step above. These states would normally be stored during the recursive process so that they can be looked-up later.

16.4 The Forward-Backward Algorithm

We may be interested in computing quantities such as $p(\mathbf{y}_{1:T}|\theta)$ or $P(s_t|\mathbf{y}_{1:T}, \theta)$; these distributions are useful for learning and analyzing models. Again, the naive approach to computing these quantities involves summing over all possible sequences of hidden states, and thus is intractable to compute. The Forward-Backward Algorithm allows us to compute these quantities in polynomial time, using dynamic programming.

In the **Forward Recursion**, we compute:

$$\alpha_t(i) \equiv p(\mathbf{y}_{1:t}, s_t = i) \tag{401}$$

The base case is:

$$\alpha_1(i) = p(\mathbf{y}_1|s_1 = i)p(s_1 = i) \tag{402}$$

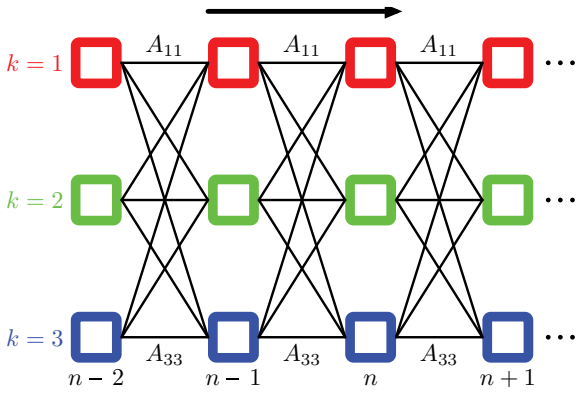

Figure 30: Illustration of the $\alpha_t(i)$ values computed during the Forward Recursion. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

and the recursive case is:

$$\alpha_t(i) = \sum_j p(\mathbf{y}_{1:t}, s_t = i, s_{t-1} = j) \tag{403}$$

$$= \sum_j p(\mathbf{y}_t|s_t = i)P(s_t = i|s_{t-1} = j)p(\mathbf{y}_{1:t-1}, s_{t-1} = j) \tag{404}$$

$$= p(\mathbf{y}_t|s_t = i) \sum_{j=1}^{K} A_{ji}\alpha_{t-1}(j) \tag{405}$$

Note that this is identical to the Viterbi algorithm, except that maximization over j has been replaced by summation.

In the **Backward Recursion** we compute:

$$\beta_t(i) \equiv p(\mathbf{y}_{t+1:T}|s_t = i) \tag{406}$$

The base case is:

$$\beta_T(i) = 1 \tag{407}$$

The recursive case is:

$$\beta_t(i) = \sum_{j=1}^{K} A_{ij}p(\mathbf{y}_{t+1}|s_{t+1} = j)\beta_{t+1}(j) \tag{408}$$

From these quantities, we can easily the following useful quantities.

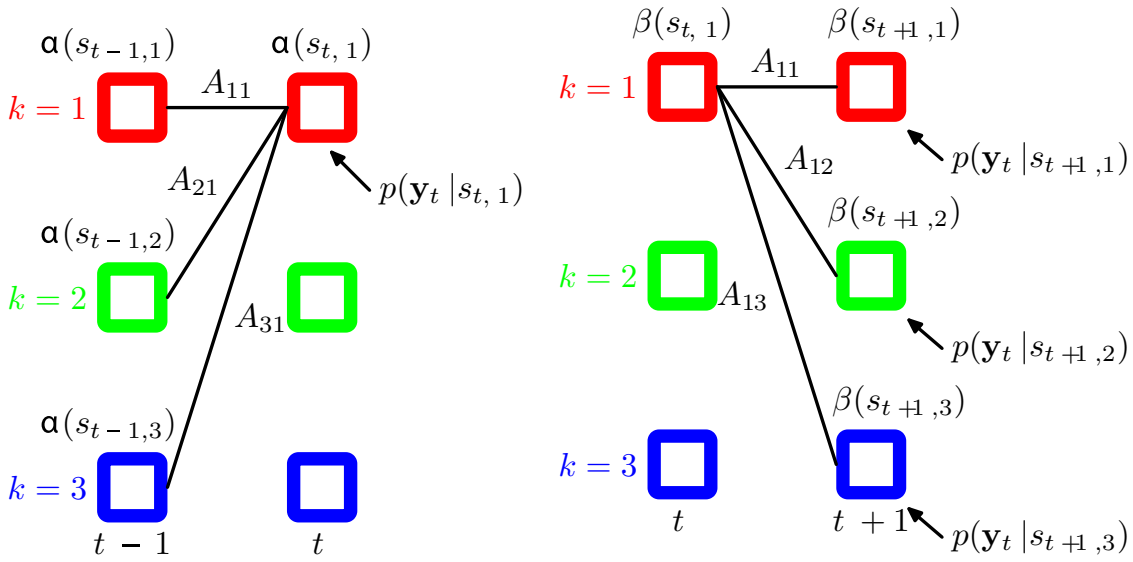

Figure 31: Illustration of the steps of the Forward Recursion and the Backward Recursion (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

The probability that the hidden sequence had state i at time t is:

$$\gamma_t(i) \equiv p(s_t = i|\mathbf{y}_{1:T}) \tag{409}$$

$$= \frac{p(\mathbf{y}_{1:T}|s_t = i)p(s_t = i)}{p(\mathbf{y}_{1:T})} \tag{410}$$

$$= \frac{p(\mathbf{y}_{1:t}|s_t = i)p(\mathbf{y}_{t+1:T}|s_t = i)p(s_t = i)}{p(\mathbf{y}_{1:T})} \tag{411}$$

$$= \frac{\alpha_t(i)\beta_t(i)}{p(\mathbf{y}_{1:T})} \tag{412}$$

The normalizing constant — which is also the likelihood of the entire sequence, $p(\mathbf{y}_{1:T})$ — can be computed by the following formula:

$$p(\mathbf{y}_{1:T}) = \sum_i p(s_t = i, \mathbf{y}_{1:T}) \tag{413}$$

$$= \sum_i \alpha_t(i)\beta_t(i) \tag{414}$$

The result of this summation will be the same regardless of which time-step t we choose to do the summation over.

The probability that the hidden sequence transitioned from state i at time t to state j at time

$t+1$ is:

$$\xi_t(i,j) \equiv P(s_t = i, s_{t+1} = j|\mathbf{y}_{1:T}) \quad (415)$$

$$= \frac{\alpha_t(i)A_{ij}p(\mathbf{y}_{t+1}|s_{t+1}=j)\beta_{t+1}(j)}{p(\mathbf{y}_{1:T})} \quad (416)$$

$$= \frac{\alpha_t(i)A_{ij}p(\mathbf{y}_{t+1}|s_{t+1}=j)\beta_{t+1}(j)}{\sum_i \sum_j \alpha_t(i)A_{ij}p(\mathbf{y}_{t+1}|s_{t+1}=j)\beta_{t+1}(j)} \quad (417)$$

Note that the denominator gives an expression for $p(\mathbf{y}_{1:T})$, which can be computed for any value of t.

16.5 EM: The Baum-Welch Algorithm

Learning in HMMs is normally done by maximum likelihood, i.e., we wish to find the model parameters such that:

$$\theta^* = \arg\max_\theta p(\mathbf{y}_{1:T}|\theta) \quad (418)$$

As before, even evaluating this objective function requires K^T steps, and methods like gradient descent will be impractical. Instead, we can use the EM algorithm. Note that, since an HMM is a generalization of a Mixture-of-Gaussians, EM for HMMs will be a generalization of EM for MoGs. The EM algorithm applied to HMMs is also known as the Baum-Welch Algorithm.

The algorithm alternates between the following two steps:

- **The E-Step:** The Forward-Backward Algorithm is performed, in order to compute γ and ξ.
- **The M-Step:** The parameters θ are updated as follows:

$$a_i = \gamma_1(i) \quad (419)$$

$$\boldsymbol{\mu}_i = \frac{\sum_t \gamma_t(i)\mathbf{y}_t}{\sum_t \gamma_t(i)} \quad (420)$$

$$\boldsymbol{\Sigma}_i = \frac{\sum_t \gamma_t(i)(\mathbf{y}_t - \boldsymbol{\mu}_i)(\mathbf{y}_t - \boldsymbol{\mu}_i)^T}{\sum_t \gamma_t(i)} \quad (421)$$

$$A_{ij} = \frac{\sum_t \xi_t(i,j)}{\sum_k \sum_t \xi_t(i,k)} \quad (422)$$

16.5.1 Numerical issues: renormalization

In practice, numerical issues are a problem for the straightforward implementation of the Forward-Backward Algorithm. Since the α_t's involve joint probabilities over the entire sequence up to time t, they will be very small. In fact, as t grows, the values of α_t tend to shrink exponentially

towards 0. Thus the limit of machine precision will quickly be reached and the computed values will underflow (evaluate to zero).

The solution is to compute a normalized terms in the Forward-Backward recursions:

$$\hat{\alpha}_t(i) = \frac{\alpha_i(t)}{\prod_{m=1}^{T} c_m} \tag{423}$$

$$\hat{\beta}_t(i) = \left(\prod_{m=t+1}^{T} c_m \right) \beta_i(t) \tag{424}$$

Specifically, we use $c_t = p(\mathbf{y}_t|\mathbf{y}_{1:t-1})$. It can then be seen that, if we use $\hat{\alpha}$ and $\hat{\beta}$ in the M-step instead of α and β, the c_t terms will cancel out (you can see this by substituting the formulas for γ and ξ into the M-step). We then must choose c_t to keep the scaling of $\hat{\alpha}$ and $\hat{\beta}$ within machine precision.

In the base case for the forward recursion, we set:

$$c_1 = \sum_{i=1}^{K} p(\mathbf{y}_1|s_1 = i) a_i \tag{425}$$

$$\hat{\alpha}_1(i) = \frac{p(\mathbf{y}_1|s_1 = i) a_i}{c_1} \tag{426}$$

(This may be implemented by first computing the numerator of $\hat{\alpha}$, and then summing it to get c_1). The recursion for computing $\hat{\alpha}$ is

$$c_t = \sum_i p(\mathbf{y}_t|s_t = i) \sum_{j=1}^{K} A_{ji} \hat{\alpha}_{t-1}(j) \tag{427}$$

$$\hat{\alpha}_t(i) = p(\mathbf{y}_t|s_t = i) \frac{\sum_{j=1}^{K} A_{ji} \hat{\alpha}_{t-1}(j)}{c_t} \tag{428}$$

In the backward step, the base case is:

$$\hat{\beta}_T(i) = 1 \tag{429}$$

and the recursive case is

$$\hat{\beta}_t(i) = \frac{\sum_{j=1}^{K} A_{ij} p(\mathbf{y}_{t+1}|s_{t+1} = j) \hat{\beta}_{t+1}(j)}{c_{t+1}} \tag{430}$$

using the same c_t values computed in the forward recursion.

The γ and ξ variables can then be computed as

$$\gamma_t(i) = \hat{\alpha}_t(i) \hat{\beta}_t(i) \tag{431}$$

$$\xi_t(i,j) = \frac{\hat{\alpha}_t(i) p(\mathbf{y}_{t+1}|s_{t+1} = j) A_{ij} \hat{\beta}_{t+1}(j)}{c_{t+1}} \tag{432}$$

It can be shown that $c_t = p(\mathbf{y}_t|\mathbf{y}_{1:t-1})$. Hence, once the recursion is complete, we can compute the data likelihood as

$$p(\mathbf{y}_{1:T}) = \prod_t c_t \tag{433}$$

or, in the log-domain (which is more stable),

$$\ln p(\mathbf{y}_{1:T}) = \sum \ln c_t \tag{434}$$

This quantity is decreased after ever EM-step until convergence of EM.

16.5.2 Free Energy

EM can be viewed as optimizing the model parameters θ together with the distribution ξ.

The Free Energy for a Hidden Markov Model is:

$$\begin{aligned} F(\theta,\xi) &= -\sum_i \gamma_1(i)\ln a_i - \sum_{i,j}\sum_{t=1}^{T-1}\xi_t(i,j)\ln A_{ij} - \sum_i\sum_{t=1}^{T}\gamma_t(i)\ln p(\mathbf{y}_t|s_t=i) \\ &\quad + \sum_{i,j}\sum_{t=1}^{T-1}\xi_t(i,j)\ln\xi_t(i,j) - \sum_i\sum_{t=2}^{T-2}\gamma_t(i)\ln\gamma_t(i) \end{aligned} \tag{435}$$

where γ is defined as a function of ξ as:

$$\gamma_t(i) = \sum_k \xi_t(i,k) = \sum_k \xi_{t-1}(k,i) \tag{436}$$

Warning! Since we weren't able to find any formula for the free energy, we derived it from scratch (see below). In our tests, it didn't precisely match the negative log-likelihood. So there might be a mistake here, although the free energy did decrease as expected.

Derivation. This material is very advanced and not required for the course. It is mainly here because we couldn't find it elsewhere.

As a short-hand, we define $\mathbf{s} = s_{1:T}$ to be a variable representing an entire state sequence. The likelihood of a data sequence is:

$$p(\mathbf{y}_{1:T}) = \sum_{\mathbf{s}} p(\mathbf{y}_{1:T},\mathbf{s}) \tag{437}$$

where the summation is over all possible state sequences.

In EM, we're really optimizing θ and a distribution $q(\mathbf{s})$ over the possible state sequences. The variable ξ is just one way of representing this distribution by its marginals; the variable γ are the

marginals of ξ:

$$\gamma_t(i) = q(s_t = i) = \sum_{\mathbf{s}\setminus\{i\}} q(\mathbf{s}) \tag{438}$$

$$\xi_t(i,j) = q(s_t = i, s_{t+1} = j) = \sum_{\mathbf{s}\setminus\{i,j\}} q(\mathbf{s}) \tag{439}$$

We can also compute the full distribution from ξ and γ:

$$q(\mathbf{s}) = q(s_1) \prod_{t=1}^{T-1} q(s_{t+1}|s_t) \tag{440}$$

$$= \gamma_1(i) \prod_{t=1}^{T-1} \frac{\xi_t(i,j)}{\gamma_t(i)} \tag{441}$$

$$= \frac{\prod_{t=1}^{T-1} \xi_t(i,j)}{\prod_{t=2}^{T-1} \gamma_t(i)} \tag{442}$$

The Free Energy is then:

$$F(\theta, q) = -\sum_{\mathbf{s}} q(\mathbf{s}) \ln p(\mathbf{s}, \mathbf{y}_{1:T}) + \sum_{\mathbf{s}} q(\mathbf{s}) \ln q(\mathbf{s}) \tag{443}$$

$$= F_1(q) + F_2(q) \tag{444}$$

The first term can be decomposed as:

$$F_1(q) = -\sum_{\mathbf{s}} q(\mathbf{s}) \ln p(\mathbf{s}, \mathbf{y}_{1:T}) \tag{445}$$

$$= \sum_{\mathbf{s}} q(\mathbf{s}) \ln \left(P(s_1) \prod_{t=1}^{T-1} P(s_{t+1}|s_t) \prod_{t=1}^{T} p(\mathbf{y}_t|s_t) \right) \tag{446}$$

$$= -\sum_{\mathbf{s}} q(\mathbf{s}) \ln P(s_1) - \sum_{\mathbf{s}} \sum_{t} q(\mathbf{s}) \ln P(s_{t+1}|s_t) - \sum_{\mathbf{s}} \sum_{t} q(\mathbf{s}) \ln p(\mathbf{y}_t|s_t) \tag{447}$$

$$= -\sum_{i} \gamma_1(i) \ln P(s_1 = i) - \sum_{i,j,t} \xi_t(i,j) \ln P(s_{t+1} = j | s_t = i)$$
$$- \sum_{i,t} \gamma_t(i) \ln p(\mathbf{y}_t | s_t = i) \tag{448}$$

The second term can be simplified as:

$$
\begin{aligned}
F_2(q) &= \sum_{\mathbf{s}} q(\mathbf{s}) \ln q(\mathbf{s}) && (449)\\
&= \sum_{\mathbf{s}} q(\mathbf{s}) \ln \frac{\prod_{t=1}^{T-1} \xi_t(i,j)}{\prod_{t=2}^{T-1} \gamma_t(i)} && (450)\\
&= \sum_{\mathbf{s}} \sum_{t=1}^{T-1} q(\mathbf{s}) \ln \xi_t(i,j) - \sum_{\mathbf{s}} \sum_{t=2}^{T-1} q(\mathbf{s}) \ln \gamma_t(i) && (451)\\
&= \sum_{i,j} \sum_{t=1}^{T-1} \xi_t(i,j) \ln \xi_t(i,j) - \sum_{i} \sum_{t=2}^{T-1} \gamma_t(i) \ln \gamma_t(i) && (452)
\end{aligned}
$$

16.6 Most likely state sequences

Suppose we wanted to computed the most likely states s_t for each time in a sequence. There are two ways that we might do it: we could take the **most likely state sequence**:

$$s^*_{1:T} = \arg\max_{s_{1:T}} p(s_{1:T}|\mathbf{y}_{1:T}) \tag{453}$$

or we could take **the sequence of most-likely states**:

$$s^*_t = \arg\max_{s_t} p(s_t|\mathbf{y}_{1:T}) \tag{454}$$

While these sequences may often be similar, they can be different as well. For example, it is possible that the most likely states for two consecutive time-steps do not have a valid transition between them, i.e., if $s^*_t = i$ and $s^*_{t+1} = j$, it is possible (though unlikely) that $A_{ij} = 0$. This illustrates that these two ways to create sequences of states answer two different questions: what sequence is jointly most likely? And, for each time-step, what is the most likely state just for that time-step?

17 Support Vector Machines

We now discuss an influential and effective classification algorithm called Support Vector Machines (SVMs). In addition to their successes in many classification problems, SVMs are responsible for introducing and/or popularizing several important ideas to machine learning, namely, *kernel methods, maximum margin methods, convex optimization,* and *sparsity/support vectors*. Unlike the mostly-Bayesian treatment that we have given in this course, SVMs are based on some very sophisticated Frequentist arguments (based on a theory called Structural Risk Minimization and VC-Dimension) which we will not discuss here, although there are many close connections to Bayesian formulations.

17.1 Maximizing the margin

Suppose we are given N training vectors $\{(\mathbf{x}_i, y_i)\}$, where $\mathbf{x} \in \mathbb{R}^D$, $y \in \{-1, 1\}$. We want to learn a classifier

$$f(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b \tag{455}$$

so that the classifier's output for a new $\mathbf{x}$ is $\text{sign}(f(\mathbf{x}))$.

Suppose that our training data are linearly-separable in the feature space $\phi(\mathbf{x})$, i.e., as illustrated in Figure 32, the two classes of training exemplars are sufficiently well separated in the feature space that one can draw a hyperplane between them (e.g., a line in 2D, or plane in 3D). If they are linearly separable then in almost all cases there will be many possible choices for the linear decision boundary, each one of which will produce no classification errors on the training data. Which one should we choose? If we place the boundary very close to some of the data, there seems to be a greater danger that we will misclassify some data, especially when the training data are alsmot certainy noisy.

This motivates the idea of placing the boundary to maximize the **margin**, that is, the distance from the hyperplane to the closest data point in either class. This can be thought of having the largest "margin for error" — if you are driving a fast car between a scattered set of obstacles, it's safest to find a path that stays as far from them as possible.

More precisely, in a *maximum margin method*, we want to optimize the following objective function:

$$\max_{\mathbf{w},b} \min_i \text{dist}(\mathbf{x}_i, \mathbf{w}, b) \tag{456}$$

$$\text{such that, for all } i,\ y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b) \geq 0 \tag{457}$$

where $\text{dist}(\mathbf{x}, \mathbf{w}, b)$ is the Euclidean distance from the feature point $\phi(\mathbf{x})$ to the hyperplane defined by $\mathbf{w}$ and b. With this objective function we are maximizing the distance from the decision boundary $\mathbf{w}^T \phi(\mathbf{x}) + b = 0$ to the nearest point i. The constraints force us to find a decision boundary that classifies all training data correctly. That is, for the classifier a training point correctly y_i and $\mathbf{w}^T \phi(\mathbf{x}_i) + b$ should have the same sign, in which case their product must be positive.

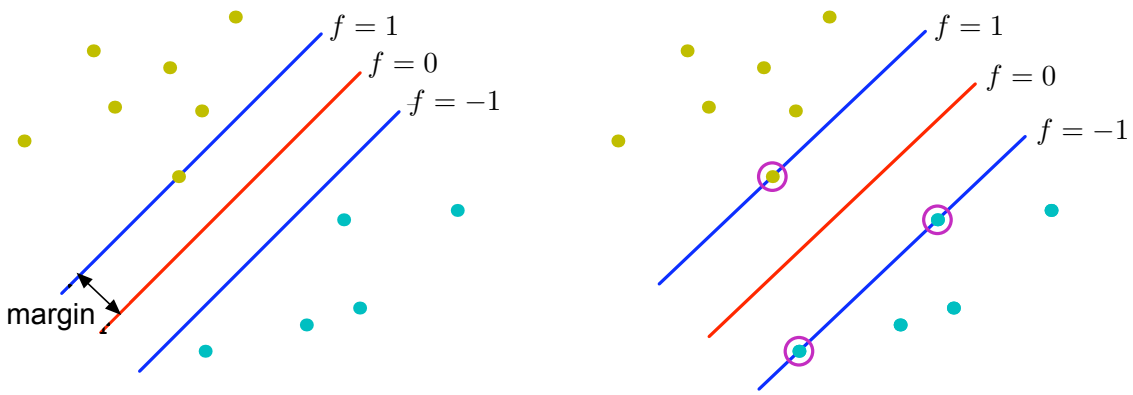

Figure 32: Left: the margin for a decision boundary is the distance to the nearest data point. Right: In SVMs, we find the boundary with maximum margin. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

It can be shown that the distance from a point $\phi(\mathbf{x}_i)$ to a hyperplane $\mathbf{w}^T\phi(\mathbf{x}) + b = 0$ is given by $\frac{|\mathbf{w}^T\phi(\mathbf{x}_i)+b|}{||\mathbf{w}||}$, or, since y_i tells us the sign of $f(\mathbf{x}_i)$, $\frac{y_i(\mathbf{w}^T\phi(\mathbf{x}_i)+b)}{||\mathbf{w}||}$. This can be seen intuitively by writing the hyperplane in the form $f(\mathbf{x}) = \mathbf{w}^T(\phi(\mathbf{x}_i) - \mathbf{p})$, where $\mathbf{p}$ is a point on the hyperplane such that $\mathbf{w}^T\mathbf{p} = b$. The vector from $\phi(\mathbf{x}_i)$ to the hyperplane projected onto $\mathbf{w}/||\mathbf{w}||$ gives a vector from the hyperplane to the the point; the length of this vector is the desired distance.

Substituting this expression for the distance function into the above objective function, we get:

$$\max_{\mathbf{w},b} \min_i \frac{y_i(\mathbf{w}^T\phi(\mathbf{x}_i)+b)}{||\mathbf{w}||} \tag{458}$$

$$\text{such that, for all } i,\ y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 0 \tag{459}$$

Note that, because of the normalization by $||\mathbf{w}||$ in (458), the scale of $\mathbf{w}$ is arbitrary in this objective function. That is, if we were to multiply $\mathbf{w}$ and b by some real scalar α, the factors of α in the numerator and denominator will cancel one another. Now, suppose that we choose the scale so that the *nearest point* to the hyperplane, $\mathbf{x}_i$, satisfies $y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) = 1$. With this assumption the $\min_i$ in Eqn (458) becomes redundant and can be removed. Thus we can rewrite the objective function and the constraint as

$$\max_{\mathbf{w},b} \frac{1}{||\mathbf{w}||} \tag{460}$$

$$\text{such that, for all } i,\ y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 \tag{461}$$

Finally, as a last step, since maximizing $1/||\mathbf{w}||$ is the same as minimizing $||\mathbf{w}||^2/2$, we can re-express the optimization problem as

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 \tag{462}$$

$$\text{such that, for all } i,\ y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 \tag{463}$$

This objective function is a **quadratic program**, or QP, because the objective function and the constraints are both quadratic in the unknowns. A QP has a single global minima, which can be found efficiently with current optimization packages.

In order to understand this optimization problem, we can see that the constraints will be "active" for only a few datapoints. That is, only a few datapoints will be close to the margin, thereby constraining the solution. These points are called the **support vectors**. Small movements of the other data points have no effect on the decision boundary. Indeed, the decision boundary is determined only by the support vectors. Of course, moving points to within the margin of the decision boundary will change which points are support vectors, and thus change the decision boundary. This is in constrast to the probabilistic methods we have seen earlier in the course, in which the positions of all data points affect the location of the decision boundary.

17.2 Slack Variables for Non-Separable Datasets

Many datasets will not be linearly separable. As a result, there will be no way to satisfy all the constraints in Eqn. (463). One way to cope with such datasets and still learn useful classifiers is to loosen some of the constraints by introducing **slack variables**.

Slack variables are introduced to allow certain constraint to be violated. That is, certain training points will be allowed to be within the margin. We want the number of points within the margin to be as small as possible, and of course we want their penetration of the margin to be as small as possible. To this end, we introduce a slack variable ξ_i, one for each datapoint i. (ξ is the Greek letter xi, pronounced "ksi."). The slack variable is introduced into the optmization problem in two ways. First, the slack variable ξ_i dictates the degree to which the constraint on the ith datapoint can be violated. Second, by adding the slack variable to the energy function we are aiming to simultaneously minimize the use of the slack variables.

Mathematically, the new optimization problem can be expressed as

$$\min_{\mathbf{w},b,\xi_{1:N}} \sum_i \xi_i + \lambda \frac{1}{2}||\mathbf{w}||^2 \tag{464}$$

$$\text{such that, for all } i,\ y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i \text{ and } \xi_i \geq 0 \tag{465}$$

As discussed above, we aim to both maximize the margin and minimize violation of the margin constraints. This objective function is still a QP, and so can be optimized with a QP library. However, it does have a much larger number of optimization variables, namely, one ξ must now be optimized for each datapoint. In practice, SVMs are normally optimized with special-purpose optimization procedures designed specifically for SVMs.

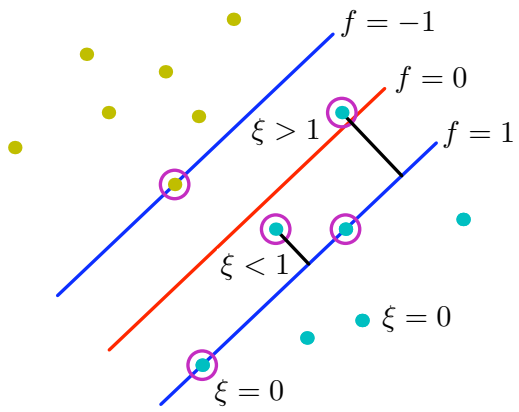

Figure 33: The slack variables $\xi_i \geq 1$ for misclassified points, and $0 < \xi_i < 1$ for points close to the decision boundary. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)++

17.3 Loss Functions

In order to better understand the behavior of SVMs, and how they compare to other methods, we will analyze them in terms of their **loss functions.**[9] In some cases, this loss function might come from the problem being solved: for example, we might pay a certain dollar amount if we incorrectly classify a vector, and the penalty for a false positive might be very different for the penalty for a false negative. The rewards and losses due to correct and incorrect classification depend on the particular problem being optimized. Here, we will simply attempt to minimize the total number of classification errors, using a penalty is called the **0-1 Loss:**

$$L_{0-1}(\mathbf{x}, y) = \begin{cases} 1 & yf(\mathbf{x}) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{466}$$

(Note that $yf(\mathbf{x}) > 0$ is the same as requiring that y and $f(\mathbf{x})$ have the same sign.) This loss function says that we pay a penalty of 1 when we misclassify a new input, and a penalty of zero if we classify it correctly.

Ideally, we would choose the classifier to minimize the loss over the new test data that we are given; of course, we don't know the true labels, and instead we optimize the following surrogate objective function over the training data:

$$E(\mathbf{w}) = \sum_i L(\mathbf{x}_i, y_i) + \lambda R(\mathbf{w}) \tag{467}$$

[9] A loss function specifies a measure of the quality of a solution to an optimization problem. It is the penalty function that tell us how badly we want to penalize errors in a models ability to fit the data. In probabilistic methods it is typically the negative log likelihood or the negative log posterior.

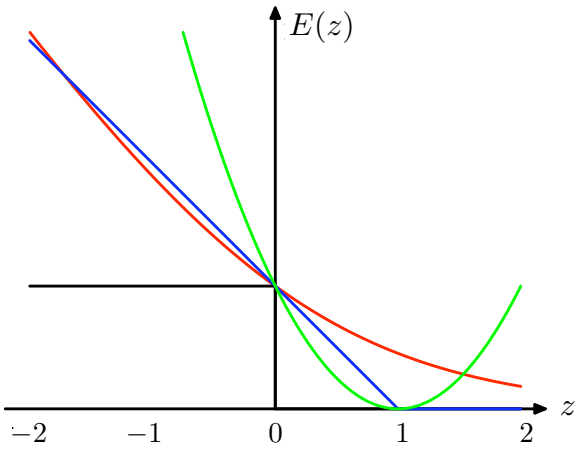

Figure 34: Loss functions, $E(z)$, for learning, for $z = y\,f(x)$. Black: 0-1 loss. Red: LR loss. Green: Quadratic loss ($(z-1)^2$). Blue: Hinge loss. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

where $R(\mathbf{w})$ is a regularizer meant to prevent overfitting (and thus improve performance on future data). The basic assumption is that loss on the training set should correspond to loss on the test set. If we can get the classifier to have small loss on the training data, while also being smooth, then the loss we pay on new data ought to not be too big either. This optimization framework is equivalent to MAP estimation as discussed previously[10]; however, here we are not at all concerned with probabilities. We only care about whether the classifier gets the right answers or not.

Unfortunately, optimizing a classifier for the 0-1 loss is very difficult: it is not differentiable everywhere, and, where it is differentiable, the gradient is zero everywhere. There are a set of algorithms called Perceptron Learning which attempt to do this; of these, the Voted Perceptron algorithm is considered one of the best. However, these methods are somewhat complex to analyze and we will not discuss them further. Instead, we will use other loss functions that approximate 0-1 loss.

We can see that maximum likelihood logistic regression is equivalent to optimization with the following loss function:

$$L_{\text{LR}} = \ln\left(1 + e^{-yf(\mathbf{x})}\right) \tag{468}$$

which is the negative log-likelihood of a single data vector. This function is a poor approximation to the 0-1 loss, and, if all we care about is getting the labels right (and not the class probabilities), then we ought to search for a better approximation.

SVMs minimize the slack variables, which, from the constraints, can be seen to give the **hinge loss**:

$$L_{hinge} = \begin{cases} 1 - yf(\mathbf{x}) & 1 - yf(\mathbf{x}) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{469}$$

[10]However, not all loss functions can be viewed as the negative log of a valid likelihood function, although all negative-log likelihoods can be viewed as loss functions for learning.

This loss function is zero for points that are classified correctly (with distance to the decision boundary at least 1); hence, it is insensitive to correctly-classified points far from the boundary. It increases linearly for misclassified points, not nearly as quickly as the LR loss.

17.4 The Lagrangian and the Kernel Trick

We now use the Lagrangian in order to transform the SVM problem in a way that will lead to a powerful generalization. For simplicity here we assume that the dataset is linearly separable, and so we drop the slack variables.

The Langrangian allows us to take the constrained optimization problem above in Eqn. (463) and re-express it as an unconstrained problem. The Lagrangian for the SVM objective function in Eqn. (463), with Lagrange multipliers $a_i \geq 0$, is:

$$L(\mathbf{w}, b, a_{1:N}) \;=\; \frac{1}{2}||\mathbf{w}||^2 - \sum_i a_i \left(y_i \left(\mathbf{w}^T \phi(\mathbf{x}_i) + b\right) - 1\right) \tag{470}$$

The minus sign with the secon term is used because we are minimizing with respect to the first term, but maximizing the second.

Setting the derivative of $\frac{dL}{d\mathbf{w}} = 0$ and $\frac{dL}{db} = 0$ gives the following constraints on the solution:

$$\mathbf{w} \;=\; \sum_i a_i y_i \phi(\mathbf{x}_i) \tag{471}$$

$$\sum_i y_i a_i \;=\; 0 \tag{472}$$

Using (471) we can substitute for $\mathbf{w}$ in 470. Then simplifying the result, and making use of the next constraint (471), one can derive what is often called the **dual Lagrangian**:

$$L(a_{1:N}) = \sum a_i - \frac{1}{2} \sum_i \sum_j a_i a_j y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \tag{473}$$

While this objective function is actually more expensive to evaluate than the primal Lagrangian (i.e., 470), it does lead to the following modified form

$$L(a_{1:N}) = \sum a_i - \frac{1}{2} \sum_i \sum_j a_i a_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) \tag{474}$$

where $k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ is called a **kernel function.** For example, if we used the basic linear features, i.e., $\phi(\mathbf{x}) = \mathbf{x}$, then $k(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$.

The advantage of the kernel function representation is that it frees us from thinking about the features directly; the classifier can be specified solely in terms of the kernel. Any kernel that

satisfies a specific technical condition[11] is a valid kernel. For example, one of the most commonly-used kernels is the "RBF kernel":

$$k(\mathbf{x}, \mathbf{z}) = e^{-\gamma||\mathbf{x}-\mathbf{z}||^2} \tag{475}$$

which corresponds to a vector of features $\phi(\mathbf{x})$ with infinite dimensionality! (Specifically, each element of ϕ is a Gaussian basis function with vanishing variance).

Note that, just as most constraints in the Eq. (463) are not "active", the same will be true here. That is, only some constraints will be active (ie the support vectors), and for all other constraints, $a_i = 0$. Hence, once the model is learned, most of the training data can be discarded; only the support vectors and their a values matter.

The one final thing we need to do is estimate the bias b. We now know the values for a_i for all support vectors (i.e., for data constraints that are considered active), and hence we know $\mathbf{w}$. Accordingly, for all support vectors we know, by assumption above, that

$$f(\mathbf{x}_i) = \mathbf{w}^T \phi(\mathbf{x}_i) + b = 1 \, . \tag{476}$$

From this one can easily solve for b.

Applying the SVM to new data. For the kernel representation to be useful, we need to be able to classify new data without needing to evaluate the weights. This can be done as follows:

$$\begin{aligned} f(\mathbf{x}_{new}) &= \mathbf{w}^T \phi(\mathbf{x}_{new}) + b & (477) \\ &= \left(\sum_i a_i y_i \phi(\mathbf{x}_i) \right)^T \phi(\mathbf{x}_{new}) + b & (478) \\ &= \sum_i a_i y_i k(\mathbf{x}_i, \mathbf{x}_{new}) + b & (479) \end{aligned}$$

Generalizing the kernel representation to non-separable datasets (i.e., with slack variables) is straightforward, but will not be covered in this course.

17.5 Choosing parameters

To determine an SVM classifier, one must select:

- The regularization weight λ
- The parameters to the kernel function
- The type of kernel function

These values are typically selected either by hand-tuning or cross-validation.

[11] Specifically, suppose one is given N input points $\mathbf{x}_{1:N}$, and forms a matrix $\mathbf{K}$ such that $\mathbf{K}_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$. This matrix must be positive semidefinite (i.e., all eigenvalues non-negative) for all possible input sets for k to be a valid kernel.

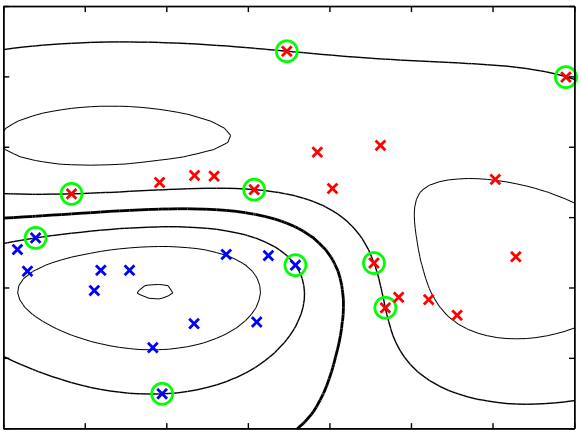

Figure 35: Nonlinear classification boundary learned using kernel SVM (with an RBF kernel). The circled points are the support vectors; curves are isocontours of the decision function (e.g., the decision boundary $f(x) = 0$, etc.) (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

17.6 Software

Like many methods in machine learning there is freely available software on the web. For SVM classification and regression there is well-known software developed by Thorsten Joachims, called SVMlight, (`URL: http://svmlight.joachims.org/`).

18 AdaBoost

Boosting is a general strategy for learning classifiers by combining simpler ones. The idea of boosting is to take a "weak classifier" — that is, any classifier that will do at least slightly better than chance — and use it to build a much better classifier, thereby boosting the performance of the weak classification algorithm. This boosting is done by averaging the outputs of a collection of weak classifiers. The most popular boosting algorithm is **AdaBoost**, so-called because it is "adaptive."[12] AdaBoost is extremely simple to use and implement (far simpler than SVMs), and often gives very effective results. There is tremendous flexibility in the choice of weak classifier as well. Boosting is a specific example of a general class of learning algorithms called **ensemble methods,** which attempt to build better learning algorithms by combining multiple simpler algorithms.

Suppose we are given training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $\mathbf{x}_i \in \mathbb{R}^K$ and $y_i \in \{-1, 1\}$. And suppose we are given a (potentially large) number of weak classifiers, denoted $f_m(\mathbf{x}) \in \{-1, 1\}$, and a **0-1** loss function I, defined as

$$I(f_m(\mathbf{x}), y) \;=\; \begin{cases} 0 & \text{if} f_m(\mathbf{x}_i) = y_i \\ 1 & \text{if} f_m(\mathbf{x}_i) \neq y_i \end{cases} \tag{480}$$

Then, the pseudocode of the AdaBoost algorithm is as follows:

for i from 1 to N, $w_i^{(1)} = 1$
for $m = 1$ to M **do**
  Fit weak classifier m to minimize the objective function:
    $\epsilon_m = \frac{\sum_{i=1}^N w_i^{(m)} I(f_m(\mathbf{x}_i) \neq y_i)}{\sum_i w_i^{(m)}}$
  where $I(f_m(\mathbf{x}_i) \neq y_i) = 1$ if $f_m(\mathbf{x}_i) \neq y_i$ and 0 otherwise
  $\alpha_m = \ln \frac{1-\epsilon_m}{\epsilon_m}$
  for all i **do**
    $w_i^{(m+1)} = w_i^{(m)} e^{\alpha_m I(f_m(\mathbf{x}_i) \neq y_i)}$
  end for
end for

After learning, the final classifier is based on a linear combination of the weak classifiers:

$$g(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^M \alpha_m f_m(\mathbf{x})\right) \tag{481}$$

Essentially, AdaBoost is a greedy algorithm that builds up a "strong classifier", i.e., $g(\mathbf{x})$, incrementally, by optimizing the weights for, and adding, one weak classifier at a time.

[12] AdaBoost was called adaptive because, unlike previous boosting algorithms, it does not need to know error bounds on the weak classifiers, nor does it need to know the number of classifiers in advance.

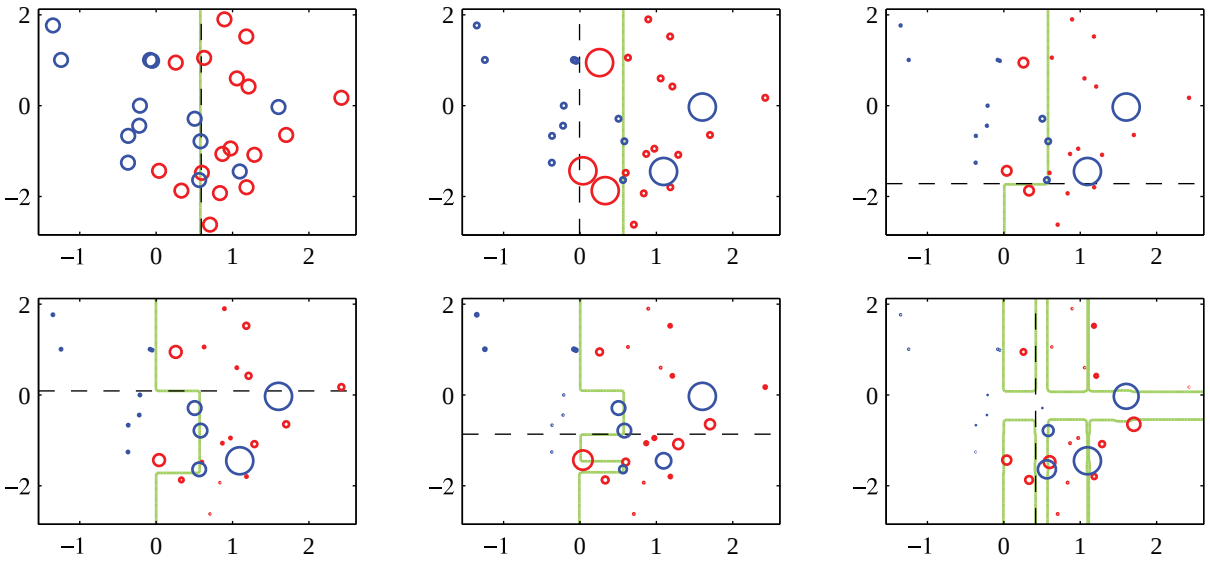

Figure 36: Illustration of the steps of AdaBoost. The decision boundary is shown in green for each step, and the decision stump for each step shown as a dashed line. The results are shown after 1, 2, 3, 6, 10, and 150 steps of AdaBoost. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

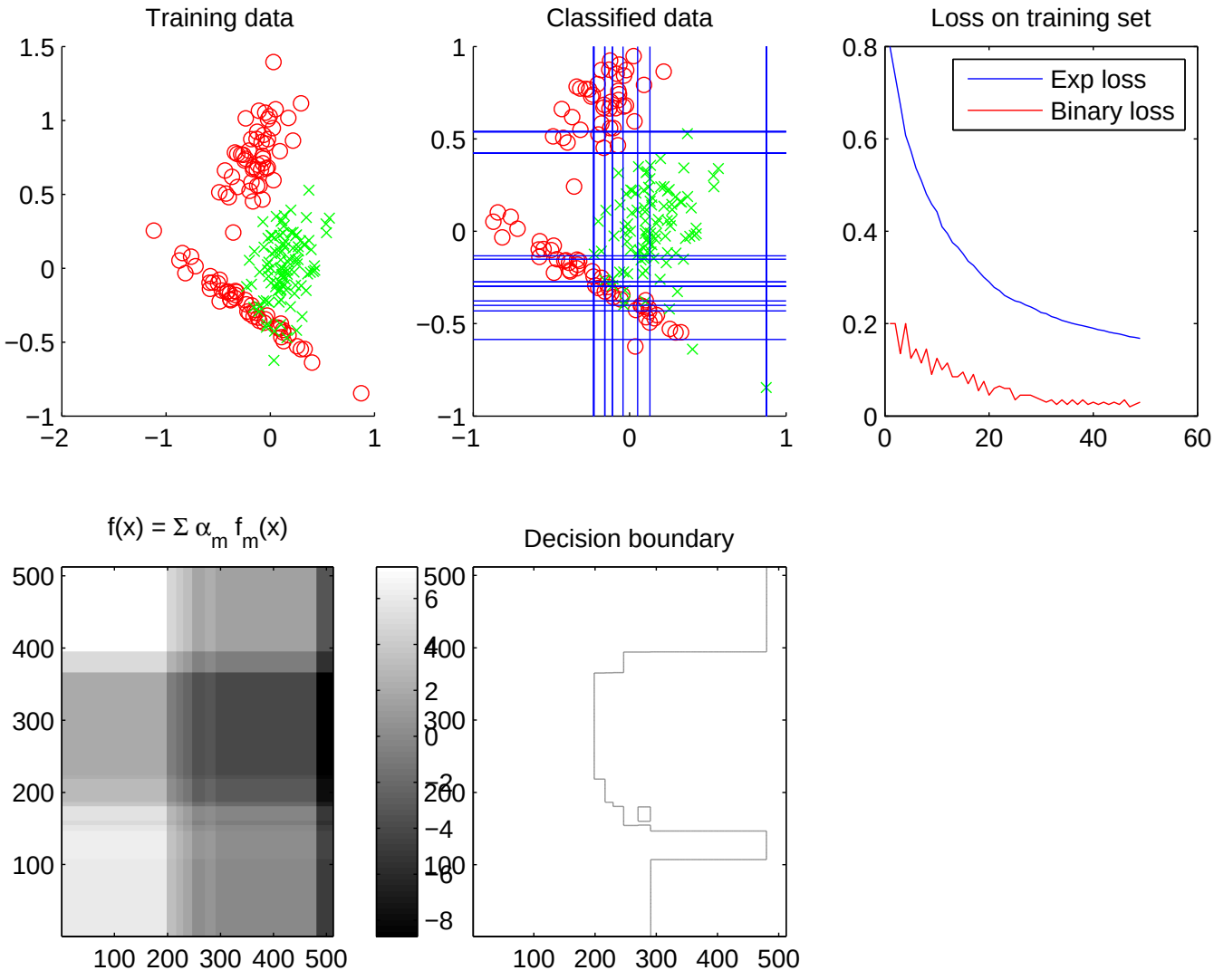

Figure 37: 50 steps of AdaBoost used to learn a classifier with decision stumps.

18.1 Decision stumps

As an example of a weak classifier, we consider "decision stumps," which are a trivial special case of decision trees. A decision stump has the following form:

$$f(\mathbf{x}) = s(x_k > c) \tag{482}$$

where the value in the parentheses is 1 if the k-th element of the vector $\mathbf{x}$ is greater than c, and -1 otherwise. The scalar s is either -1 or 1 which allows one the classifier to respond with class 1 when $x_k \leq c$. Accordingly, there are three parameters to a decision stump:

- $c \in \mathbb{R}$
- $k \in \{1, ...K\}$, where K is the dimension of $\mathbf{x}$, and
- $s \in \{-1, 1\}$

Because the number of possible parameter settings is relatively small, a decision stump is often trained by brute force: discretize the real numbers from the smallest to the largest value in the training set, enumerate all possible classifiers, and pick the one with the lowest training error. One can be more clever in the discretization: between each pair of data points, only one classifier must be tested (since any stump in this range will give the same value). More sophisticated methods, for example, based on binning the data, or building CDFs of the data, may also be possible.

18.2 Why does it work?

There are many different ways to analyze AdaBoost; none of them alone gives a full picture of why AdaBoost works so well. AdaBoost was first invented based on optimization of certain bounds on training, and, since then, a number of new theoretical properties have been discovered.

Loss function view. Here we discuss the loss function interpretation of AdaBoost. As was shown (decades after AdaBoost was first invented), AdaBoost can be viewed as greedy optimization of a particular loss function. We define $f(\mathbf{x}) = \frac{1}{2}\sum_m \alpha_m f_m(\mathbf{x})$, and rewrite the classifier as $g(\mathbf{x}) = \text{sign}(f(\mathbf{x}))$ (the factor of 1/2 has no effect on the classifier output). AdaBoost can then be viewed as optimizing the **exponential loss:**

$$L_{exp}(\mathbf{x}, y) = e^{-yf(\mathbf{x})} \tag{483}$$

so that the full learning objective function is

$$E = \sum_i e^{-\frac{1}{2}y_i \sum_{m=1}^M \alpha_m f_m(\mathbf{x})} \tag{484}$$

which must be optimized with respect to the weights α and the parameters of the weak classifiers. The optimization process is greedy and sequential: we add one weak classifier at a time, choosing it

and its α to be optimal with respect to E, and then never change it again. Note that the exponential loss is an upper-bound on the 0-1 loss:

$$L_{exp}(\mathbf{x}, y) \geq L_{0-1}(\mathbf{x}, y) \tag{485}$$

Hence, if exponential loss of zero is achieved, then the 0-1 loss is zero as well, and all training points are correctly classified.

Consider the weak classifier f_m to be added at step m. The entire objective function can be written to separate out the contribution of this classifier:

$$E = \sum_i e^{-\frac{1}{2} y_i \sum_{j=1}^{m-1} \alpha_j f_j(\mathbf{x}) - \frac{1}{2} y_i \alpha_m f_m(\mathbf{x})} \tag{486}$$

$$= \sum_i e^{-\frac{1}{2} y_i \sum_{j=1}^{m-1} \alpha_j f_j(\mathbf{x})} \; e^{-\frac{1}{2} y_i \alpha_m f_m(\mathbf{x})} \tag{487}$$

Since we are holding constant the first $m-1$ terms, we can replace them with a single constant $w_i^{(m)} = e^{-\frac{1}{2} y_i \sum_{j=1}^{m-1} \alpha_j f_j(\mathbf{x})}$. Note that these are the same weights computed by the recursion used by AdaBoost: $w_i^{(m)} \propto w_i^{(m-1)} e^{-\frac{1}{2} y_i \alpha_j f_{m-1}(\mathbf{x})}$. (There is a proportionality constant that can be ignored). Hence, we have

$$E = \sum_i w_i^{(m)} e^{-\frac{1}{2} y_m \alpha_m f_m(\mathbf{x})} \tag{488}$$

We can split this into two summations, one for data correctly classified by f_m, and one for those misclassified:

$$E = \sum_{i: f_m(\mathbf{x}_i) = y_i} w_i^{(m)} e^{-\frac{\alpha_m}{2}} + \sum_{i: f_m(\mathbf{x}_i) \neq y_i} w_i^{(m)} e^{\frac{\alpha_m}{2}} \tag{489}$$

Rearranging terms, we have

$$E = (e^{\frac{\alpha_m}{2}} - e^{-\frac{\alpha_m}{2}}) \sum_i w_i^{(m)} I(f_m(\mathbf{x}_i) \neq y_i) + e^{-\frac{\alpha_m}{2}} \sum_i w_i^{(m)} \tag{490}$$

Optimizing this with respect to f_m is equivalent to optimizing $\sum_i w_i^{(m)} I(f_m(\mathbf{x}_i) \neq y_i)$, which is what AdaBoost does. The optimal value for α_m can be derived by solving $\frac{dE}{d\alpha_m} = 0$:

$$\frac{dE}{d\alpha_m} = \frac{\alpha_m}{2} \left(e^{\frac{\alpha_m}{2}} + e^{-\frac{\alpha_m}{2}} \right) \sum_i w_i^{(m)} I(f_m(\mathbf{x}_i) \neq y_i) - \frac{\alpha_m}{2} e^{-\frac{\alpha_m}{2}} \sum_i w_i^{(m)} = 0 \tag{491}$$

Dividing both sides by $\frac{\alpha_m}{2 \sum_i w_i^{(m)}}$, we have

$$0 = e^{\frac{\alpha_m}{2}} \epsilon_m + e^{-\frac{\alpha_m}{2}} \epsilon_m - e^{-\frac{\alpha_m}{2}} \tag{492}$$

$$e^{\frac{\alpha_m}{2}} \epsilon_m = e^{-\frac{\alpha_m}{2}} (1 - \epsilon_m) \tag{493}$$

$$\frac{\alpha_m}{2} + \ln \epsilon_m = -\frac{\alpha_m}{2} + \ln(1 - \epsilon_m) \tag{494}$$

$$\alpha_m = \ln \frac{1 - \epsilon_m}{\epsilon_m} \tag{495}$$

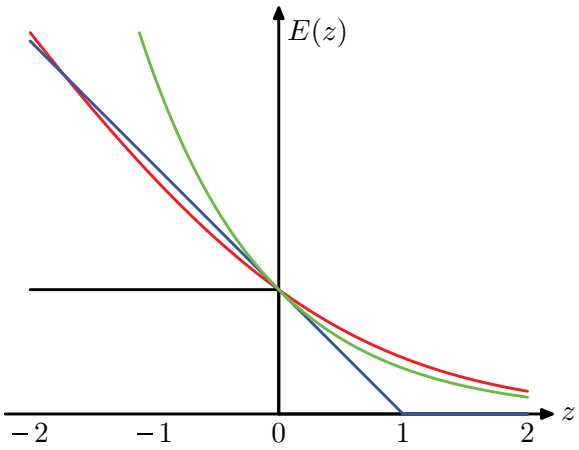

Figure 38: Loss functions for learning: Black: 0-1 loss. Blue: Hinge Loss. Red: Logistic regression. Green: Exponential loss. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

Problems with the loss function view. The exponential loss is not a very good loss function to use in general. For example, if we directly optimize the exponential loss over all variables in the classifier (e.g., with gradient descent), we will often get terrible performance. So the loss-function interpretation of AdaBoost does not tell the whole story.

Margin view. One might expect that, when AdaBoost reaches zero training set error, adding any new weak classifiers would cause overfitting. In practice, the opposite often occurs: continuing to add weak classifiers actually improves test set performance in many situations. One explanation comes from looking at the margins: adding classifiers tends to increase the margin size. The formal details of this will not be discussed here.

18.3 Early stopping

It is nonetheless possible to overfit with AdaBoost, by adding too many classifiers. The solution that is normally used practice is a procedure called **early stopping**. The idea is as follows. We partition our data set into two pieces, a training set and a test set. The training set is used to train the algorithm normally. However, at each step of the algorithm, we also compute the 0-1 binary loss on the test set. During the course of the algorithm, the exponential loss on the training set is guaranteed to decrease, and the 0-1 binary loss will generally decrease as well. The errors on the testing set will also generally decrease in the first steps of the algorithm, however, at some point, the testing error will begin to get noticeably worse. When this happens, we revert the classifier to the form that gave the best test error, and discard any subsequent changes (i.e., additional weak classifiers).

The intuition for the algorithm is as follows. When we begin learning, our initial classifier is

extremely simple and smooth. During the learning process, we add more and more complexity to the model to improve the fit to the data. At some point, adding additional complexity to the model overfits: we are no longer modeling the decision boundary we wish to fit, but are fitting the noise in the data instead. We use the test set to determine when overfitting begins, and stop learning at that point.

Early stopping can be used for most iterative learning algorithms. For example, suppose we use gradient descent to learn a regression algorithm. If we begin with weights $\mathbf{w} = 0$, we are beginning with a very smooth curve. Each step of gradient descent will make the curve less smooth, as the entries of $\mathbf{w}$ get larger and larger; stopping early can prevent $\mathbf{w}$ from getting too large (and thus too non-smooth).

Early stopping is very simple and very general; however, it is heuristic, as the final result one gets will depend on the particulars in the optimization algorithm being used, and not just on the objective function. However, AdaBoost's procedure is suboptimal anyway (once a weak classifier is added, it is never updated).

An even more aggressive form of early stopping is to simply stop learning at a fixed number of iterations, or by some other criteria unrelated to test set error (e.g., when the result "looks good.") In fact, pracitioners often using early stopping to regularize **unintentionally,** simply because they halt the optimizer before it has converged, e.g., because the convergence threshold is set too high, or because they are too impatient to wait.

