4 Supervised Learning

4.1 Regression and maximum likelihood

An important requirement of natural or artificial agents is its ability to decide on an appropriate course of action given a specific situation. The specific circumstances are communicated to the agent by sensors that specify values of certain features. Let's represent these **feature values** as vector $\mathbf{x}$. The goal of the agent is then to determine an appropriate response based on this input,

$$y = f(\mathbf{x}). \tag{4.1}$$

This corresponds to the functional form of a controller as outlined in Chapter 2. In Chapter 3 we argued that a probabilistic framework is more appropriate to address uncertainties. The corresponding statement of the deterministic function approximation of equation (1) is then to find a probability density function

$$p(y|\mathbf{x}). \tag{4.2}$$

A common example is object recognition where the feature values might be RGB values of pixels in a digital image and the desired response might be the identification of a person in this image. A learning machines for such a task is a model that is given examples with specific feature vectors $\mathbf{x}$ and corresponding desired **labels** y. Learning in this circumstance is mainly about adjusting the model's parameters from the given examples. Since this type of learning is based on specific training examples with give labels, this type of learning is called **supervised** and is the subject of this chapter. Adjusting the parameters should there by guided by the best performance of generalize, that of predicting appropriate labels of previously unseen feature vectors.

More formally, in supervised learning we consider training data that consist of example inputs and corresponding labels, that is, pairs of values $(\mathbf{x}^{(i)}\mathbf{y}^{(i)})$, where the index $i = 1, ..., m$ labels each of m training example. As an example, let us consider the estimation of the motion model for the tribot. To automate the collection of data we can use the ultrasonic sensor to measure the distance to a wall while driving the tribot for different amount of time forward and backward. In Figure 4.1A we show several measurements of the distance traveled.

The data clearly reveal some systematic relation between the time of running the motor and the distance traveled, the general trend being that the traveled distance increases with increasing running time of the motors. While there seems to be some noise in the data, the outliers and the noise can not hide a linear trent for most of the data. This hypothesis can be quantified as a parameterized function,

$$\hat{h}(x;\theta) = \theta_0 + \theta_1 x. \tag{4.3}$$

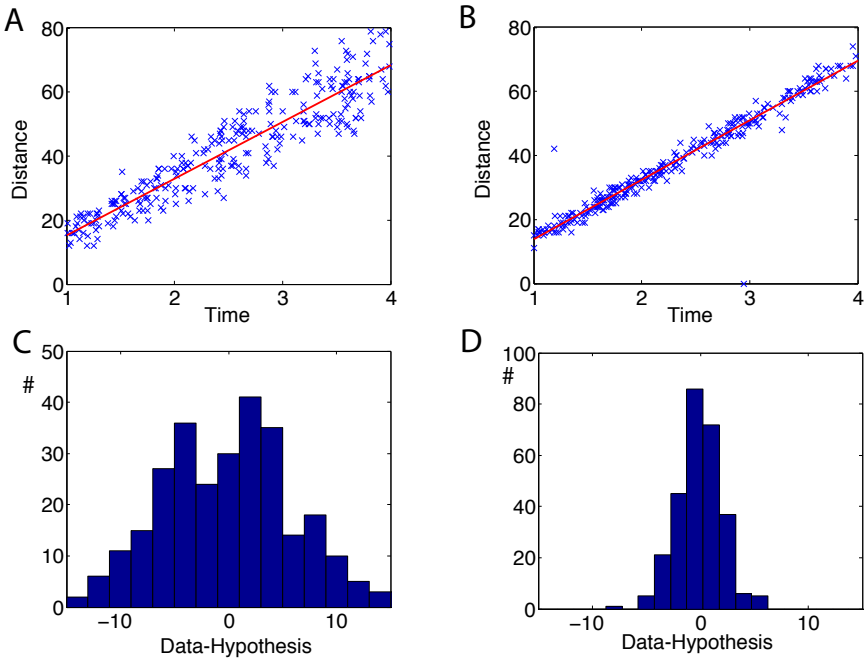


Fig. 4.1 (A) Measurements of distance travelled by the tribot when running the motor for different number of milliseconds with unknown power. (B) Same as (A) for constant motor power. (C,D) Corresponding histogram of differences between data and hypothesis.

This notation means that the hypothesis h is a function of the quantity x, and the hypothesis includes all possible straight lines, where each line can have a different offset θ_0 (intercept with the y-axis), and slope θ_1.

We typically collect parameters in a **parameter vector** θ. We only considered one input variable x above, but we can easily generalize this to higher dimensional problems where more input **attributes** are given. For example, we could not only vary the time the motor is running, let us label this attribute now with x_1, but also push the robot forward by hand for a certain time, labeled with x_2. As the effects of these manipulations are independent on the results, we can independently add the effects of higher dimensions to our hypothesis. To compress our notations further we also introduce here the convention that we consider a constant input, $x_0 = 1$ as the first component of the input vector, so that the corresponding parameter encodes the offset of the function. The state vector can then be written as,

$$\mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}. \tag{4.4}$$

With this convention we can write the hypothesis as

$$\hat{h}(\mathbf{x};\theta) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2. \tag{4.5}$$

It is then even easy to write a linear hypothesis with n attributes as

$$\hat{h}(\mathbf{x};\theta) = \theta_0 x_0 + ... + \theta_n x_n = \sum_i \theta_i x_i = \theta^T \mathbf{x}, \tag{4.6}$$

where the superscript T indicates the transpose of a vector.

Another factor that influences the distance traveled is the power setting of the motor. Of course, the distance traveled within a certain time does depend on the power and is not just an independent additive effect on the travelled distance. Results of the experiment for different power settings and different travel times are show in Figure 4.2. Figure 4.2A also includes a fit to equation 4.5. However, these data are better described by a bilinear hypothesis,

$$\hat{h}(\mathbf{x};\theta) = \theta_0 x_0 + \theta_1 x_1 x_2. \tag{4.7}$$

The corresponding fit of the same data is shown in Figure 4.2B.How to perform these fits is discussed further below.

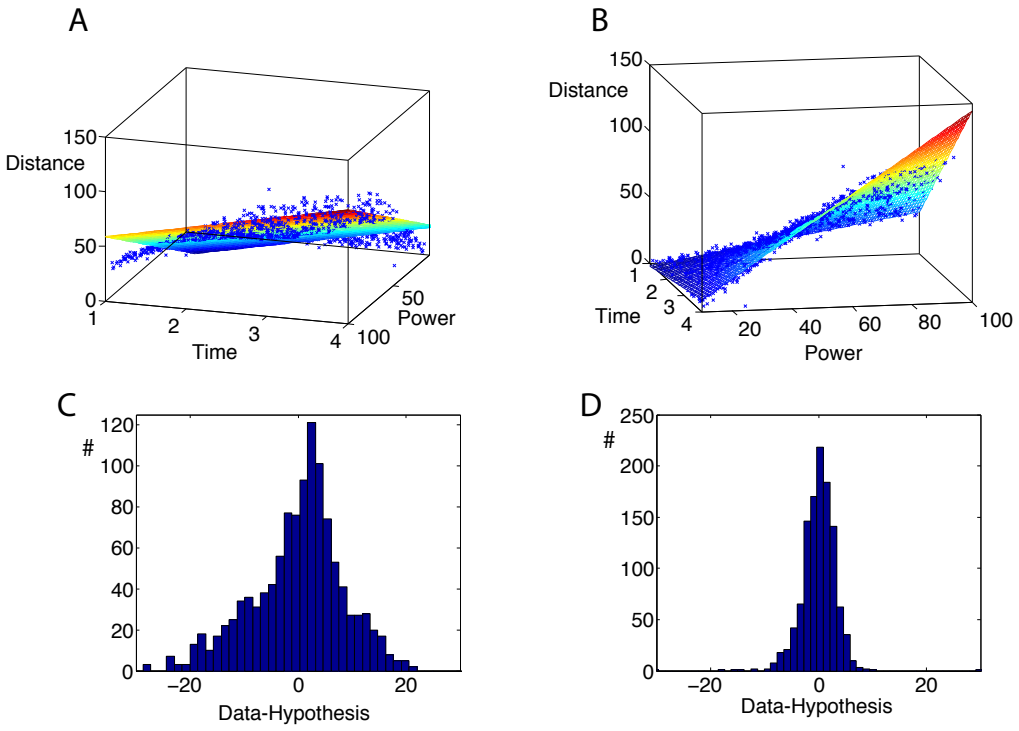


Fig. 4.2 (A) Measurements of distance travelled by the tribot when running the motor for different number of milliseconds and various power settings. Fit according to equation 4.5 (B) Same as (A) with fit according to equation 4.7. (C,D) Corresponding histogram of differences between data and hypothesis.

While we had to make a good guess for the functional form of the trend in the data, the actual parameters have so far not been specified. Thus, we made a **hypothesis**

in the form of a parameterized function, $h(\mathbf{x};\theta)$, and the learning part boils down to determining appropriate values for the parameters from the sample data. After learning these parameters we can then use this function to predict specific reactions of a plant even for motor commands for which no training examples were given. The remaining question is how we find appropriate values for the parameters. However, before we do this we need to be more faithful to the data and acknowledge fluctuation around our initial hypothesis.

So far, we have only modelled the trend of the data, and we should investigate more the fluctuations around this trend. Of course, we expect several sources of noise such as the accuracy of the ultrasonic sensor and the tendency of the tribot to sometimes turn due to wheel slippage. Indeed, while gathering these data we have fixed the follower wheel to minimize turning when moving forward and backward. We also started new trials when the robot went too much off track. Thus, we already try to minimize error due to a careful setup of the experiment to gather the data. However, what we did not tell you is that we run the experiment in Figure 4.1A with different powers of the motor between 40 and 60. Such hidden information can also contribute to uncertainties in the environment. Data for doing the same experiment as before but with a fixed motor power of 50 is shown in Figure 4.1B. These data vary less but are still noisy due other sources of uncertainty.

Figures 4.2C and D are plots of the histogram of the differences between the actual data and the hypothesis regression line. The histogram of plots of Figures 4.2C and D look a bit Gaussian, which according to the central limit theorem is a likely finding for additive and independent noise sources. In any case, we should revise our hypothesis by acknowledging the stochastic nature of the data and writing a down a specific functional form of a conditional density function for the quantity y given some input values $\mathbf{x}$. Similar to before, we also allow this probabilistic hypothesis to depend on some parameters,

$$h(y|\mathbf{x};\theta). \tag{4.8}$$

For our specific example of the tribot we assume here that the data follow our previous deterministic hypothesis $\hat{h}(\mathbf{x};\theta)$ with **additive Gaussian noise**, or with other words, that the data in Figure 4.1B are Gaussian distributed with a mean $\mu = \hat{h}(x)$ depends linearly on the value of x,

$$h(y|x;\theta,\sigma) = N(\mu = \hat{h}(x),\sigma) \tag{4.9}$$

$$= \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y-\theta^T\mathbf{x})^2}{2\sigma^2}\right) \tag{4.10}$$

This functions specifies the probability of values for y, given an input x and the parameters $\theta = (\mu,\sigma)^T$.

Specifying a model with a density function is an important step in modern modelling and machine learning. In this type of thinking, we treat data from the outset as fundamentally stochastic, that is, data can be different even in situations that we deem identical. This randomness may come from an **irreducible indeterminacy**, that is, true randomness in the world that can not be penetrated by further knowledge, or this noise might represent **epistemological limitations** such as the lack of knowledge of hidden processes or limitations in observing states directly. The only important fact for

us is that we have to live with these limitations. This acknowledgement together with the corresponding language of probability theory has helped to make large progress in the machine learning area.

We will now turn to the important principle that will guide our learning process which corresponds here to estimating the parameters of the model. While the parameterized hypothesis so far describes the form of the data, we need to estimate values for the parameters to make real predictions. We therefore consider now the examples for the input-output pairs, our training set $\{(x^{(i)}, y^{(i)}); i = 1...m\}$. The important principle that we will now follow is to choose the parameters so that the examples we have are most likely. This is called **maximum likelihood estimation**. To formalize this principle, we need to think about how to combine probabilities for several observations. If the observations are independent, then the joint probability of several observations is the product of the individual probabilities,

$$p(y_1, y_2, ..., y_m|x_1, x_2, ..., x_m; \theta) = \Pi_i^m p(y_i|x_i; \theta). \tag{4.11}$$

Note that y_i are still random variables at this point. But we now use our training examples as specific observations for each of these random variables, and introduce the **Likelihood function**

$$L(\theta) = \Pi_i^m h(\theta; y^{(i)}, x^{(i)}). \tag{4.12}$$

The h on the right hand side is now not a density function, but it is a regular function (with the same form as our parameterized hypothesis) of the parameters θ for the given values $y^{(i)}$ and $x^{(i)}$. Instead of evaluating this large product, it is common to use the logarithm of the likelihood function, so that we can use the sum over the training examples,

$$l(\theta) = \log L(\theta) = \sum_i^m \log(h(\theta; y^{(i)}, x^{(i)})). \tag{4.13}$$

Since the log function increases monotonically, the maximum of L is also the maximum of l. The maximum (log-)likelihood can thus be calculated from the examples as

$$\theta^{\mathrm{MLE}} = arg \max_\theta l(\theta). \tag{4.14}$$

We might be able to calculate this analytically or use one of the search algorithms to find a maximum from this function.

Let us apply this to the linear regression discussed above. The log-likelihood function for this example is

$$l(\theta) = \log \Pi_{i=1}^m \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T \mathbf{x^{(i)}})^2}{2\sigma^2}\right) \tag{4.15}$$

$$= \sum_{i=1}^m \left(\log \frac{1}{\sqrt{2\pi}\sigma} - \frac{(y^{(i)} - \theta^T \mathbf{x^{(i)}})^2}{2\sigma^2}\right) \tag{4.16}$$

$$= -\frac{m}{2} \log 2\pi\sigma - \sum_{i=1}^m \frac{(y^{(i)} - \theta^T \mathbf{x}^{(i)})^2}{2\sigma^2}. \tag{4.17}$$

Thus, the log was chosen so that we can use the sum in the estimate instead of dealing with big numbers based on the product of the examples.

Let us now consider the special case in which we assume that the constant σ, the variance of the data, is the same for all x and thus has a fixed value given to us. We can thus concentrate on the estimation of the other parameters θ. Since the first term in the expression 4.17, $-\frac{m}{2}\log 2\pi\sigma$, is independent of θ, maximizing the log-likelihood function is equivalent to minimizing a quadratic error term

$$E = \frac{1}{2}(y - h((x;(\theta))^2 \Longleftrightarrow p(y|\mathbf{x};\theta) = \frac{1}{\sqrt{2\pi}}\exp(-\frac{((y - h((x;\theta)^2}{2}) \quad (4.18)$$

This **error function** or **cost function** was a frequently used criteria called **Least Mean Square (LSM)** regression for parameters estimation when considering deterministic hypothesis. I terms of our probabilistic view, the LSM regression is equivalent to MLE for gaussian data with constant variance. When the variance is a free parameter, then we need to minimize equation 4.17. instead.

We have discussed Gaussian distributed data in most of this section, but one can similarly find corresponding error functions for other distributions. For example, a **polynomial error function** correspond more generally to a density model of the form

$$E = \frac{1}{p}||y - h((x;(\theta)||^p \Longleftrightarrow p(y|\mathbf{x};\theta) = \frac{1}{2\Gamma(1/p)}\exp(-||y - h((x;\theta||^p). \quad (4.19)$$

Later we will specifically discuss and use the ϵ**-insensitive error function**, where errors less than a constant ϵ do not contribute to the error measure, only errors above this value,

$$E = ||y - h((x;(\theta)||_\epsilon \Longleftrightarrow p(y|\mathbf{x};\theta) = \frac{p}{2(1-\epsilon)}\exp(-||y - h((x;\theta||_\epsilon). \quad (4.20)$$

Since we already acknowledged that we do expect that data are noisy, it is somewhat logical to not count some deviations form the expectation as errors. It also turns out that this error function is much more robust than other error functions.

In the maximum likelihood estimation we assumed that we have no prior knowledge of the parameters θ. However, we sometimes might know which values of the parameters are impossible or less likely. This prior knowledge can be summarized in the prior distribution $p(\theta)$, and the next question is how to combine this prior knowledge in the maximum likelihood scheme. Combining prior knowledge with some evidence is described by Bayes' theorem. Thus, let us consider again that we have some observations (x, y) from specific realizations of the parameters, which is given by $(p(x, y|\theta)$, and the prior about the possible values of the parameters, given by $p(\theta)$. The prior is in this situation sometimes called the **regularizer**, restricting possible values in a specific domain. We want to know the distribution of parameters given the observation, $p(\theta|x, y)$, which can be calculated from Bayes's theorem,

$$p(\theta|x, y) = \frac{p(x, y|\theta)p(\theta)}{\int_{\theta' \in \Theta} p(x, y|\theta')p(\theta')\mathrm{d}\theta'}, \quad (4.21)$$

where Θ is the domain of the possible parameter values. We can now use this expression to estimate the most likely values for the parameters. For this we should notice that the denominator, which is called the **partition function**, does not depend on the parameters

θ. The most likely values for the parameters can thus be calculated without this term and is given by the **maximum a posteriori (MAP)** estimate,

$$\theta^{\mathrm{MAP}} = arg \max_{\theta} p(x, y|\theta)p(\theta). \tag{4.22}$$

This is, in a Bayesian sense, the most likely value for the parameters, where, of course, we now treat the probability function as a function of the parameters (e.g., a likelihood function).

A final caution: ML and MAP estimates give us a **point estimate**, a single answer of the most likely values of the parameters. This is often useful as a first guess and is commonly used to **make decisions** about which actions to take. However, it is possible that other sets of parameters values might have only a little smaller likelihood value, and should therefore also be considered. Thus, one limit of the estimation methods discussed here is that they do not take distribution of answers into account, which is more common in more advanced Bayesian methods.

4.2 Minimization and gradient descent

We have discussed the important principle of maximum likelihood estimation to learn parameters from data so that the data are most likely under our hypothesis. This principle tells us how to use the training examples to come-up with some reasonable values for the parameters. Tho execute these principles we have to find the maximize of a (log)likelihood function.

Let us for now concentrate again on the maximum likelihood estimation of the parameters θ for the mean of the Gaussian data. The maximum likelihood estimate can the be found by minimizing the **mean square error** (MSE) function

$$E(\theta) = \frac{1}{2}(y - h(x;\theta))^2 \tag{4.23}$$

$$\approx \frac{1}{2m}\sum_i (y^{(i)} - h(x^{(i)};\theta))^2. \tag{4.24}$$

Note that the objective function is a function of the parameters. Also, note that while we wrote the general form of the objective function in line 4.23, we consider its maximum likelihood estimation from independent data in line 4.24. With this objective function, we reduced the learning problem to a search problem of finding the parameter values that minimize this objective function,

$$\theta = \arg\min_{\theta} E(\theta) \tag{4.25}$$

We will demonstrate practical solutions to this search problem with three important methods. The first method is an analytical one. You might remember from basic mathematics classes that finding a minimum of a function $f(x)$ is characterized by $\frac{\mathrm{d}f}{\mathrm{d}x} = 0$, and $\frac{\mathrm{d}^2 f}{\mathrm{d}x^2} > 0$. Here we have a vector function since the cost function depends an several parameters. The derivative then becomes a gradient

$$\nabla_\theta E(\theta) = \begin{pmatrix} \frac{\partial E}{\partial \theta_0} \\ . \\ . \\ . \\ \frac{\partial E}{\partial \theta_n} \end{pmatrix}. \tag{4.26}$$

It is useful to collect the training data in a large matrix for the x values, and a vector for the y values,

$$X = \left(\mathbf{x}^{(1)} ... \mathbf{x}^{(m)}\right) \qquad Y = \left(y^{(1)} ... y^{(m)}\right). \tag{4.27}$$

Not let us use again the specific linear hypothesis of the data above. We can then write the cost function as

$$E(\theta) = \frac{1}{2m}(Y - X\theta)(Y - X\theta)^T. \tag{4.28}$$

Some straight forward calculation will then provide the parameters for which the gradient is zero,

$$\theta = (XX^T)^{-1}XY^T, \tag{4.29}$$

which is also known as the **normal equation**. We have still to make sure these parameters are the minimum and not a maximum value, but this can easily be done and is also obvious when plotting the result. This analytic methods is optimal in the requirement of computational time. However, it requires that we can analytically solve an equation system. This was easy in the linear case but fails in general for more complex functions. The next methods are more widely applicable.

The second method we mainly mention for illustrative reasons is random search. This is a very simple algorithm, but worthwhile considering to compare them to the other solutions. In this algorithm, a new random values for the parameters are tried, and these new parameters replace the old ones if the new values result in a smaller error value (see Matlab code in Tab.4.1). The benefit of this method is that it can be applied to any function, but in practice it takes much too long to find a solution.

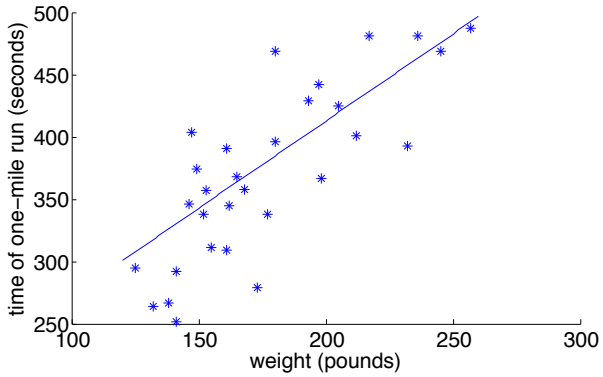


Fig. 4.3 Health data with linear least-mean-square (LMS) regression from random search.

The final method discussed here for finding a minimum of a function $E(\theta)$ is **Gradient Descent**. This method will often be used in the following and it will thus be reviewed here in more detail.

Table 4.1 Program randomSearch.m

```
%% Linear regression with random search
clear; clf; hold on;

load healthData;
E=1000000;
for trial=1:100
    thetaNew=[100*rand()+50, 3*rand()];
    Enew=0.5*sum((y-(thetaNew(1)+thetaNew(2)*x)).^2);
    if Enew<E; E=Enew; theta=thetaNew; end
end

plot(x,y,'*')
plot(120:260,theta(1)+theta(2)*(120:260))
xlabel('weight (pounds)')
ylabel('time of one-mile run (seconds)')
```

Gradient Descent starts at some initial value for the parameters, θ, and improves the values iteratively by making changes to the parameters along the negative gradient of the cost function,

$$\theta \leftarrow \theta - \alpha \nabla_\theta E(\theta). \tag{4.30}$$

The constant α is called a **learning rate**. The principle idea behind this method is illustrated for a general cost function with one parameter in Fig.4.4.

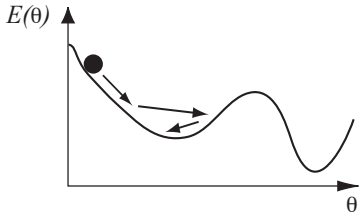


Fig. 4.4 Illustration of error minimization with a gradient descent method on a one-dimensional error surface $E(\theta)$.

The gradient is simple the slope (local derivative) for a function with one variable, but with functions in higher dimensions (more variables), the gradient is the local slope along the direction of the steepest accent, and since we are interested here in minimizing the cost function we make changes along the negative gradient. For large gradients, this method takes large steps, whereas the effective step-width becomes smaller near a minimum. Gradient descent works often well for local optimization, but it can get stuck in local minima. The corresponding update rule in case of the LMS error function,

$$E(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(y^{(i)} - h(x^{(i)};\theta)\right)^2, \tag{4.31}$$

with a general hypothesis function $h(x^{(i)};\theta)$ is given by

$$\theta_k \leftarrow \theta_k - \frac{\alpha}{m}\sum_{i=1}^{m}(y^{(i)} - h(x^{(i)};\theta))\frac{\partial h(\theta)}{\partial \theta_k}. \tag{4.32}$$

For a linear regression function, the update rule for the two parameters θ_0 and θ_1 are therefore:

$$h(x^{(i)};\theta) = \theta_0 + \theta_1 x^{(i)} \tag{4.33}$$

$$\theta_0 \leftarrow \theta_0 - \alpha\frac{\partial E(\theta)}{\partial \theta_0} \tag{4.34}$$

$$\theta_1 \leftarrow \theta_1 - \alpha\frac{\partial E(\theta)}{\partial \theta_1} \tag{4.35}$$

$$\frac{\partial E(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \theta_0 - \theta_1 x^{(i)})(-1) \tag{4.36}$$

$$\frac{\partial E(\theta)}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \theta_0 - \theta_1 x^{(i)})(-x^{(i)}), \tag{4.37}$$

which lead to the final rule:

$$\theta_0 \leftarrow \theta_0 + \frac{\alpha}{m}\sum_{i=1}^{m}(y_i - \theta_0 - \theta_1 x_i) \tag{4.38}$$

$$\theta_1 \leftarrow \theta_1 + \frac{\alpha}{m}\sum_{i=1}^{m}(y_i - \theta_0 - \theta_1 x_i)x_i. \tag{4.39}$$

Note that the learning rate α has to be chosen small enough for the algorithm to converge. An example is show in Fig.4.5, where the dashed line shows the initial hypothesis, and the solid line the solution after 100 updates.

In the algorithm above we calculate the average gradient over all examples before updating the parameters. This is called a **batch algorithm** or **synchronous update** since the whole batch of training data is used for each updating step and the update is only made after seeing all training data. This might be problematic in some applications as the training examples have to be stored somewhere and have to be recalled continuously. A much more applicable methods, also thought to be more biological realistic, is to use each training example when it comes in and disregards it right after. In this way we do not have to store all the data. Such algorithms are called **online algorithms** or **asynchronous update**. Specifically, in the example above, we calculate the change for each training examples and update the parameters for this training example before moving to the next example. While we might have to run through the short list of training examples in this specific example, it can still be considered online since we need only one training example at each training step and a list could be supplied to us externally. There are also variations of this algorithms depending on the order we use

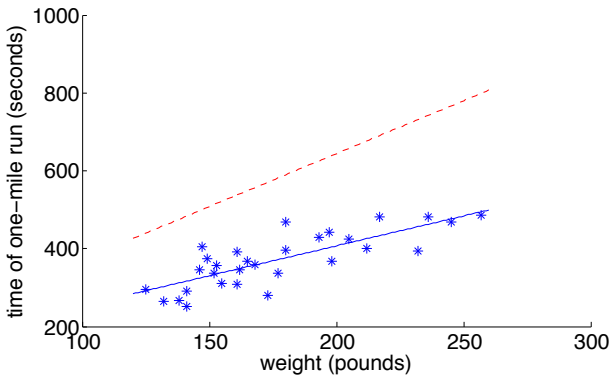


Fig. 4.5 Health data with linear least-mean-square (LMS) regression with gradient descent. The dashed line shows the initial hypothesis, and the solid line the solution after 100 updates.

the training examples (e.g. random or sequential), although this should not be crucial for the examples discussed here.

The simple gradient descent as outlined above should highlight the general idea of gradient-based minimization. There are several problems with this simple approach when the function to be minimized is strongly non-linear. To illustrate this better, let us consider the Taylor expansion of a function

$$f(x+\epsilon) = f(x) + \epsilon^T \nabla_x f + O(\epsilon^2) \tag{4.40}$$

The simple gradient term is a good approximation if the higher order terms are small. However, if these are large than we make considerable errors and the method becomes unstable. You can experience this when using a large learning rate. Of course, we could then consider higher order approximations like

$$f(x+\epsilon) = f(x) + \epsilon^T \nabla_x f + \frac{1}{2}\epsilon^T \mathbf{H}\epsilon + O(\epsilon^3), \tag{4.41}$$

where $\mathbf{H}$ is the Hessian matrix. Minimization based on this second-order term is called **Newton method**. While the Newton method converges after much faster than the simple gradient method, it is often difficult or time consuming to calculate or approximate the Hessian. There are further methods that are mainly used in practice such as the **Levenberg-Marquardt algorithm** or the **Natural Gradient**. We will not discuss this here in more detail, but you should consider these algorithms in practical applications.

4.2.1 LinearRegressionExampleCode

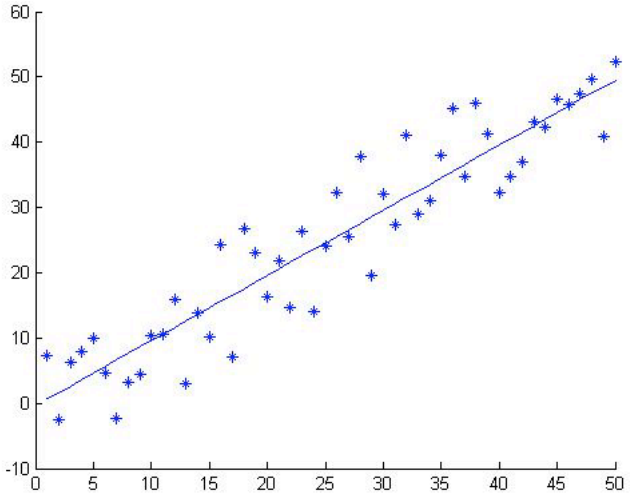
```
%% Linear regression with gradient descent
clear all; clc; hold on;

load SampleRegressionData; m=50; alpha=0.001;
theta1=rand*100*((rand<0.5)*2-1);
```

```
theta2=rand*100*((rand<0.5)*2-1);

for trial=1:50000
    sum1 = sum(y - theta1 - theta2 *x);
    sum2 = sum((y - theta1 - theta2 *x).* x);
    theta1 = theta1 + (2*alpha/m) * sum1;
    theta2 = theta2 + (2*alpha/m) * sum2;
    sum1 = 0; sum2 = 0;
end

plot(x, y, '*');
plot(x, x*theta2+theta1);
hold off;
```



4.3 Classification as logistic regression

We have grounded supervised learning in probabilistic function regression and maximum likelihood estimation. An important special case of supervised learning is **classification**, The simplest example is that of binary classification which are data that have only two possible labels such as $y = (0, 1)$. For example, let us consider a one dimensional case where data tend to fall into one class when the feature value is below a threshold θ, or into the other class when above. The most difficult situation for such classification is around the threshold value since small changes in this value might trigger one versus the other class. It is then appropriate to make a hypothesis for the probability of $p(y = 1|x)$ which is small for x ($x << \theta$), around 0.5 for $x \approx \theta$, and approaching one for large x ($x >> \theta$).

More formally, let us consider a random number which takes the value of 1 with probability ϕ and the value 0 with probability $1 - \phi$ (the probability of being either of

the two choices has to be 1.) Such a random variable is called Bernoulli distributed. Tossing a coin is a good example of a process that generates a Bernoulli random variable and we can use maximum likelihood estimation to estimate the parameter ϕ from such trials. That is, let us consider m tosses in which h heads have been found. The log-likelihood of having h heads ($y = 1$) and $m - h$ tails ($y = 0$) is

$$l(\phi) = \log(\phi^h(1-\phi)^{m-h}) \tag{4.42}$$

$$= h\log(\phi) + (m-h)\log(1-\phi). \tag{4.43}$$

To find the maximum with respect to ϕ we set the derivative of l to zero,

$$\frac{\mathrm{d}l}{\mathrm{d}\phi} = \frac{h}{\phi} - \frac{m-h}{1-\phi} = 0 \tag{4.44}$$

$$\rightarrow \quad \phi = \frac{h}{m} \tag{4.45}$$

As you might have expected, the maximum likelihood estimate of the parameter ϕ is the fraction of heads in m trials.

Now let us discuss the case when the probability of observing a head or tail, the parameter ϕ, depends on an attribute x, as usual in a stochastic (noisy) way. An example is illustrated in Fig.4.6 with 100 examples plotted with star symbols. The data suggest that it is far more likely that the class is $y = 0$ for small values of x and that the class is $y = 1$ for large values of x, and the probabilities are more similar in-between. We put forward the hypothesis that the transition between the low and high probability region is smooth and qualify this hypothesis as parameterized density function known as a **logistic** (sigmoidal) function

$$p(y=1) = \frac{1}{1+\exp(-\theta^T\mathbf{x})}. \tag{4.46}$$

As before, we can then treat this density function as a function of the parameters θ for the given data values (likelihood function), and use maximum likelihood estimation to estimate values for the parameters so that the data are most likely.

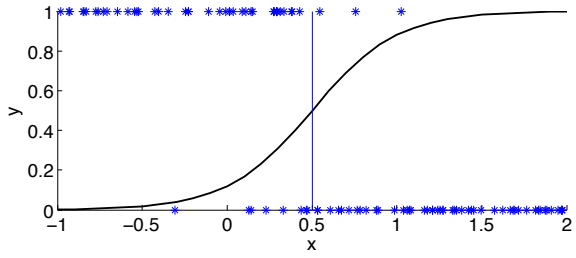


Fig. 4.6 Binary random numbers (stars) drawn from the density $p(y=1) = \frac{1}{1+\exp(-\theta_1 x-\theta_0)}$ (solid line) with offset $\theta_0 = 2$ and slope $\theta_1 = 4$.

How can we use the knowledge (estimate) of the density function to do classification? The obvious choice is to predict the class with the higher probability, given the

input attribute. This **bayesian decision point**, x_d, or **dividing hyperplane** in higher dimensions, is given by

$$p(y = 1|x_d) = p(y = 0|x_d) = 0.5 \rightarrow x_d \theta^T \mathbf{x_d} = 0. \tag{4.47}$$

We have here considered binary classification with linear decision boundaries as logistic regression, and we can also generalize this method to problems with non-linear decision boundaries by considering hypothesis with different functional forms of the decision boundary. However, coming up with specific functions for boundaries is often difficult in practice, and we will discuss much more practical methods for binary classification later in this chapter.

4.4 Multivariate generative models and probabilistic reasoning

4.4.1 Graphical models

We have so far only considered very simple hypothesis appropriate for the low dimensional data given in the above examples. An important issues that has to be considered in machine learning is that of generalizing to more complex non-linear data in high dimension, that is, when many factors interact in a complicated way. This topic is probably one of the most important when applying ML to real world data. This section discusses a useful way of formulating more complicated stochastic models with causal relations and how to use such models to argue (inference). Parameters of such models must often be learned with supervised techniques such as maximum likelihood estimation.

Let us consider high dimensional data and the corresponding supervised learning. In the probabilistic framework, this means making a hypothesis for joint density function of the problem,

$$p(y, \mathbf{x}) = p(y, x_1, x_2, ...|\theta), \tag{4.48}$$

where $y, x_1, ...$ are random variables and θ represents the parameters of the model. With this joint density function we could argue about every possible situation in the environment. For example, we could again ask for classification or object recognition by calculating the conditional density function

$$p(y|\mathbf{x}) = p(y|x_1, x_2, ...; \theta). \tag{4.49}$$

Of course, the general joint density function and even this conditional density function for high dimensional problems have typically many free parameters that we need to estimate. It is then useful to make more careful assumptions of causal relations that restrict the density functions. The object recognition formulation above is sometimes called a **discriminative approach** to object recognition because it tries to discriminate labels give the feature values. Another approach is to consider modelling the inverse

$$p(\mathbf{x}|y) = p(x_1, x_2, ...|y; \theta), \tag{4.50}$$

This is called a **generative model** as it can generate examples from a class give a label. To use generative models in classification or object recognition we can use Bayes'

rule to calculate a discriminative model. That is, we use **class priors** (the relative frequencies of the classes) to calculate the probability that an item with features $\mathbf{x}$ belong to a class y,

$$p(y|\mathbf{x};\theta) = \frac{p(\mathbf{x}|y;\theta)p(y)}{p(x)}. \tag{4.51}$$

While using generative models for classification seem to be much more elaborate, there are several reasons that make generative models attractive for machine learning. For example, in many cases features might be conditionally independent given a label, that is

$$p(x_1, x_2, ...|y) = p(x_1|y) * p(x_2|y) * .... \tag{4.52}$$

where I have dropped the indication of the parameter vector to make the formula less cluttered. Even if the independence does not hold strictly, this **naive Bayes assumption** it is often useful and drastically reduces the number of parameters that have to be estimated. This can be seen from factorizing the full joint density function with the chain rule

$$\begin{aligned} p(x_1, x_2, ..., x_n|y) &= p(x_n|y, x_1, ...x_{n-1})p(x_1, ..., x_{n-1}|y) && (4.53)\\ &= p(x_n|y, x_1, ..., x_{n-1}) * ... * p(x_2|y, x_1) * p(x_1|y) && (4.54)\\ &= \prod_{i=1}^{n} p(x_i|y, x_{i-1}, ...x_1). && (4.55) \end{aligned}$$

But what if the naive Bayes assumption is not appropriate? Then we need to build more elaborate models. Building and using such models have been greatly simplified with **graphical methods** to build **causal models** that specify the conditional dependencies between random variables (Pearl 2000). A well known example of one of the inventors of graphical models, Judea Pearl, is shown in Fig.4.7. In such graphical models, the nodes represent random variables, and the links between them represent causal relations with conditional probabilities. In this case there are arrows on the links. This is thus an example of a **directed acyclic graph** (DAG). The RBM discussed above is an example of an undirected Bayesian network.

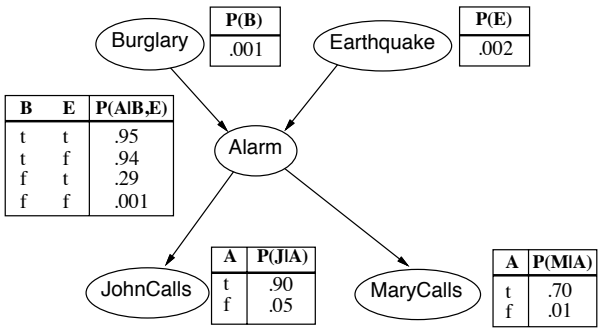


Fig. 4.7 Example of causal model a two-dimensional probability density function (pdf) and some examples of marginal pdfs.

In this specific example, each of the five nodes stands for a random binary variable (Burglary B={yes,no}, Earthquake E={yes,no}, Alarm A={yes,no}, JohnCalls J={yes,no}, MaryCalls M={yes,no}). In general, a joint distribution of several variables can be factories in various ways following the chain rule mentioned before (equations 4.53), for example as

$$p(B, E, A, J, M) = P(B|E, A, J, M)P(E|A, J, M)P(A|J, M)P(J|M)P(M). \tag{4.56}$$

In case of binary random variables we need $2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 31$ parameters to specify the full joint density function. However, the specific model illustrated in Fig.4.7 specifies restricted causal relations between the random variables which represents a specific factorization of the joint probability functions, namely

$$p(B, E, A, J, M) = P(B)P(E)P(A|B, E)P(J|A)P(M|A). \tag{4.57}$$

In this case we only need $1+1+2^2+2+2 = 10$ parameters to specify all the knowledge in the system. Example parameters for a specific case are include in **conditional probability tables (CPTs)** that specify the conditional probabilities represented by the links between the nodes. The graphical representation makes is very convenient to specify a specific hypothesis about causal relations.

The graph structure of the graphical models make it also easy to do inference (draw conclusions), for specific questions. For example, say we want to know the probability that there was no earthquake or burglary when the alarm rings and both John and Mary call. This is given by

$$\begin{aligned}
&P(B = f, E = f, A = t, J = t, M = t) = \\
&= P(B = f)P(E = f,)P(A = t|B = f, E = f)P(J = t|A = t)P(M = t|A = t) \\
&= 0.998 * 0.999 * 0.001 * 0.7 * 0.9 \\
&= 0.00062
\end{aligned}$$

Although we have a causal model where parents variables influence the outcome of child variables, we can also use evidence from child variables to infer some possible values of parent variables. For example, let us calculate the probability that the alarm rings given that John calls, $P(A = t|J = t)$. For this we should first calculate the probability that the alarm rings as we need this later. This is given by

$$\begin{aligned}
P(A = t) = &P(A = t|B = t, E = t)P(B = t)P(E = t) + ... \\
&P(A = t|B = t, E = f)P(B = t)P(E = f) + ... \\
&P(A = t|B = f, E = t)P(B = f)P(E = t) + ... \\
&P(A = t|B = f, E = f)P(B = f)P(E = f) \\
= &0.95 * 0.001 * 0.002 + 0.94 * 0.001 * 0.998 + ... \\
&0.29 * 0.999 * 0.002 + 0.001 * 0.999 * 0.998 \\
= &0.0025
\end{aligned}$$

We can then use Bayes' rule to calculate the required probability,

$$P(A = t|J = t) = \frac{P(J = t|A = t)P(A = t)}{P(J = t|A = t)P(A = t) + P(J = t|A = f)P(A = f)}$$

$$= \frac{0.90.0025}{0.90.0025 + 0.050.9975}$$
$$= 0.0434$$

We can similarly apply the rules of probability theory to calculate other quantities, but these calculations can get cumbersome with larger graphs. It is therefore useful to use numerical tools to perform such inference. For example, a useful Matlab toolbox for Bayesian networks can be downloaded at `http://code.google.com/p/bnt/`. The Matlab implementation of the model in Fig.4.7 is implemented with this toolbox in file `www.cs.dal.ca/~tt/repository/MLintro2012/PearlBurglary.m`

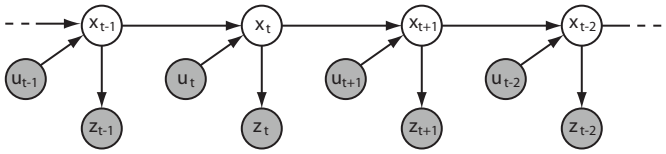


Fig. 4.8 A temporal Bayesian model called the Hidden Markov Model with hidden states x_t observations z_t and external influences u_t.

I mentioned already the importance of learning about temporal sequences (anticipatory systems), and Bayesian Networks are easily extended to this domain. An important example of such a Dynamic Bayesian Network (DBN) is a Hidden Markov Model (HMM) as shown in Fig.4.8. In this model a state variable, x_t is not directly observed. This is therefore a **hidden** or **latent** random variable. The Markov condition in this model means that the states only depend on situations in the previous state, which can include external influences such described here as u_t. T typical example is a robot localization where we drive the robot with some motor command u_t and want to know the new state of the robot. We can use some knowledge about the influence of the motor command on the system to calculate a new expected location, and we can also combine this in a Baysian optima way with sensor measurement depicted as z_t. Such Bayesian models are essential in many robotics applications.

4.4.2 Naive Bayes

One of the simplest Bayesian models is shown in Figure **??**. As this model illustrates, the random variables causing the state of the child process is assumed to be conditionally independent of each other. This is often the case at least to a first approximation calculating the joined distribution in this case is computationally feasible. We will illustrate this method on an data mining example.

In the following example we want to make a spam filter that classifies email messages as either spam ($y = 1$) or non-spam ($y = 0$) emails. To do this we need first a method to represent the problem in a suitable way. We chose here to represent a text (email in this situation) as **vocabulary** vector. A vocabulary is simply the list of all possible words that we consider, and the text is represented by this vector with entries 1 if the word can be found in the list or an entry 0 if not, e.g.

$$\mathbf{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ . \\ . \\ . \\ 1 \\ . \\ . \\ 0 \end{pmatrix} \begin{matrix} \text{a} \\ \text{aardvark} \\ \text{aardwolf} \\ . \\ . \\ . \\ \text{buy} \\ . \\ . \\ \text{zygmurgy} \end{matrix} \tag{4.58}$$

We are here only considering values 0 and 1 instead, for example, counting how often the corresponding word appears. The difference is that each entry is a binomial random variable and would be a multinomial in the other example, though the methods generalize directly to the other case. Note that this feature vector is typically very high dimensional. Let us consider here that our vocabulary has 50.000 word, which is a typical size of common languages.

We now want to build a discriminative model from some training examples. That is, we want to model

$$p(\mathbf{x}|y) = p(x_1, x_2, ..., x_{50000}|y). \tag{4.59}$$

This is a very high dimensional density function which has $2^{50.000} - 1$ parameters (the -1 comes from the normalization condition). We can factorize this conditional density function with the chain rule

$$p(x_1, x_2, ..., x_{50000}|y) = p(x_1|y)p(x_2|y, y_1)...p(x_{50000}|y, x_1, ..., x_{49999}). \tag{4.60}$$

While the right hand side has only 50.000 factors, there are still $2^{50.000} - 1$ parameters we have to learn. However, we no make a strong assumption namely that all the words are conditionally independent in each text, that is,

$$p(x_1|y)p(x_2|y, y_1)...p(x_{50000}|y, x_1, ..., x_{49999}) = p(x_1|y)p(x_2|y)...p(x_{50000}|y). \tag{4.61}$$

This is called the **Naive Bayes (NB) assumption**. Hence, we can write the conditional probability as a factor of terms with 50.000 parameters

$$p(\mathbf{x}|y) = \prod_{i=1}^{50000} p(x_i|y). \tag{4.62}$$

To estimate these parameters we can apply again maximum likelihood estimation, which gives

$$\phi_{j,y=1} = \frac{\sum_{i=1}^{m} \text{true}(x_j^{(i)} = 1 \wedge y^{(i)} = 1)}{\sum_{i=1}^{m} \text{true}(y^{(i)} = 1)} \tag{4.63}$$

$$\phi_{j,y=0} = \frac{\sum_{i=1}^{m} \text{true}(x_j^{(i)} = 1 \wedge y^{(i)} = 0)}{\sum_{i=1}^{m} \text{true}(y^{(i)} = 0)} \tag{4.64}$$

$$\phi_{y=1} = \frac{\sum_{i=1}^{m} \text{true}(y^{(i)} = 1)}{m}. \tag{4.65}$$

The function $\text{true}(a)$ returns a 1 if the expression a is true, and zero otherwise. Thus, $sum_i \text{true}(a^{(i)})$ counts how many times the expression a is true for each training example.

With these parameters we can now calculate the probability that email $\mathbf{x}$ is spam as

$$p(y = 1|\mathbf{x}) = \frac{\prod_{i=1}^{m} \phi_{i,y=1}\phi_{y=1}}{\prod_{i=1}^{m} \phi_{i,y=1}\phi_{y=1} + \prod_{i=1}^{m} \phi_{i,y=0}\phi_{y=0}}. \tag{4.66}$$

In practice this often works well when the Naive Bayes assumption is appropriate, that is, if there are no strong correlation between the features. Finally, note that there is a slight problem if some of the words, say x_{100}, are not part of the training set. In this case we get an estimate that the probability of this work every occurring is zero, $\phi_{100,y=1} = 0$ and $\phi_{100,y=0} = 0$, and hence $p(y = 1|x) = \frac{0}{0}$. A common trick, called **Laplace smoothing** is to add one occurrence of this word in every case, which will insert a small probability proportional to your training examples to the estimates,

$$\phi_{j,y=1} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} + 2} \tag{4.67}$$

$$\phi_{j,y=0} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} + 2}. \tag{4.68}$$

We will later compare the Naive Bayes classification with other classification methods.

4.5 Non-linear regression and the bias-variance tradeoff

While graphical models are great to argue about situations (doing inference), the role of supervised learning is to determine the parameters of the model. We have only considered binary models where each Bernoulli variable is characterized by a single parameter ϕ. However, the density function can be much more complicated and introduce many more parameters. A major problem in practice is thus to have enough training examples with labels to restrict useful learning appropriately. This is one important reason for unsupervised learning as we have usually many unlabelled data that can be used to represent the problem appropriately. But we still need to understand the relations between free parameters and the number of training data.

We already discussed the bias-variance tradeoff in the first section. Finding the right function that describe nonlinear data is one of the most difficult tasks in modelling, and there is not a simple algorithm that can give us the answer. This is why more general learning machines, which we will discuss in the next section, are quite popular. To evaluate the generalization performance of a specific model it is useful to split the training data into a **training set**, which is used to estimate the parameters of the model, and a **validation set**, which is used to study the **generalization performance** on data that have not been used during training the model.

A important question is then how many data we should keep to validate versus train the model. If we use too many data for validation, than we might have too less

data for accurate learning in the first place. On the other hand, if we have to few data for validation than this might not be very representative. In practice we are often using some **cross-validation** techniques to minimize the tradeoff. That is, we use the majority of the data for training, but we repeat the selection of the validation data several times to make sure that the validation was not just a result of outliers. The repeated division of the data into a training set and validation set can be done in different ways. For example, in **random subsampling** we just use random subsample for each set and repeat the procedure with other random samples. More common is k**-fold cross-validation**. In this technique we divide the data set into k-subsamples and use $k-1$ subsamples for training and one subsample for validation. In the next round we use another subsample for validating the training. A common choice for the number of subsamples is $k = 10$. By combining the results for the different runs we can often reduce the variance of our prediction while utilizing most data for learning.

We can sometimes help the learning process further. In many learning examples it turns out that some data are easy to learn while others are much harder. In some techniques called **boosting**, data which are hard to learn are over-sampled in the learning set so that the learning machine has more opportunities to learn these examples. A popular implementation of such an algorithm is **AdaBoost** (adaptive Boosting).

Before proceeding to general non-linear learning machines, I would like to outline a point that was recently made very eloquently by Doug Tweet in a course module that we shared last summer in a computational neuroscience course in Kingston, Canada. As discussed above, supervised learning is best phrased in terms of regression and that many applications are nonlinear in nature. It is common to make a nonlinear hypothesis in form of $y = h(\theta^T \mathbf{x})$, where θ is a parameter vector and h is a nonlinear hypothesis function. A common example of such a model is an artificial perceptrons with a sigmoidal transfer function such as $h(x) = \tanh(\theta x)$. However, as nicely stressed by Doug, there is no reason to make the functions nonlinear in the parameters which then result in a non-linear optimization problem. Support vector machines that are reviewed next are a good example where the optimization problem is only quadratic in the parameters. The corresponding convex optimization has no local minima that plagued multilayer perceptrons. The different strategies might be summarized with the following optimization functions:

$$\text{Linear Perceptron } E \propto \left(y - \theta^T \mathbf{x}\right)^2 \tag{4.69}$$

$$\text{Nonlinear Perceptron } E \propto \left(y - h(\mathbf{x};\theta)\right)^2 \tag{4.70}$$

$$\text{Linear in Parameter (LIP) } E \propto \left(y - \theta^T \phi(\mathbf{x})\right)^2 \tag{4.71}$$

$$\text{Linear SVM } E \propto \alpha_i \alpha_j y_i y_j \mathbf{x}^T \mathbf{x} + \text{constraints} \tag{4.72}$$

$$\text{nonlinear SVM } E \propto \alpha_i \alpha_j y_i y_j \phi(\mathbf{x})^T \phi(\mathbf{x}) + \text{constraints} \tag{4.73}$$

The LIP (linear in parameters) model is more general than a linear model in that it considers functions of the form $y = \theta^T \phi(\mathbf{x})$ with some mapping function $\phi(\mathbf{x})$. In light of this review, the transformation $\phi(\mathbf{x})$ can be seen as re-coding a sensory signal into a more appropriate form with unsupervised learning methods as discussed above.

4.6 General Learning Machines

Before we leave this discussion of basic supervised learning, I would like to mention some methods which are very popular and often used for machine learning applications. In the previous section we discussed the formulation of specific hypothesis functions. However, finding an appropriate hypothesis function requires considerable domain knowledge. This is why universal learning machines have been popular with computer scientists and have a long history. A good example are artificial neural networks, specifically multilayer perceptrons, which became popular in the 1980s although they have been introduced much earlier. The general idea behind these general learning machines is to provide a very general functions with many parameters that will be adjusted through learning. Of course, the real problem is then not to over-fit the model by using appropriate restrictions and also to make the learning efficient so that it can be used to large problem size. There has been much progress in this area, specifically though the introduction of Support Vector Machines (SVMs)that I will briefly describe in this section.

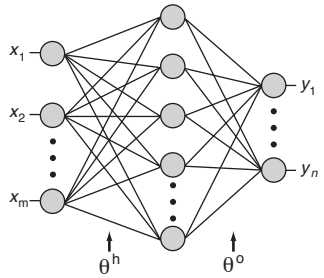


Fig. 4.9 Multilayer perceptron with one hidden layer. The parameters are called weights $\mathbf{w}$.

Let us start with a multilayer perceptron as shown in Fig.4.9. Each node represents a simple calculation. The input layer simply relaying the inputs, while the hidden and output layer multiplay each input channel with an associated weight w_i, sum this net input, and transfer it through a generally non-linear transfer function, often chosen as a sigmoid function such as the logistic function. Such networks are thus a graphical representation of a nested nonlinear functions with parameters $\mathbf{w}$. Applying an input results in a specific output $\mathbf{y}$ that can be compared to a desired output $\mathbf{y}_{des}$ in supervised learning. The parameters can then be adjusted, as usual in LMS regression, by minimizing the least square error $E = (\mathbf{y} - \mathbf{y}_{des})^2$, typically with gradient descent $w \leftarrow w + \alpha \frac{\partial E}{\partial w_i}$, where α is a lerning rate. Since $\mathbf{y}$ is a nested function of the parameters, this requires the applivation of the chain rule. The resulting equations look like propagating back an error term $\mathbf{y} - \mathbf{y}_{des}$ from the output to earlier layers, and this algorithm has thus been termed error-backpropagation (Rumelhart et al. 1986).

It is easy to see that such networks are universal approximators (Hornik 1991), that is, the error of the training examples can be made as small as desired by increasing the number of parameters. This can be achieved by adding hidden nodes. However, the aim of supervised learning is to make predictions, that is to minimize the generalization error and not the training error. Thus, choosing a smaller number of hidden nodes

might be more appropriate for this. The bias-variance tradeoff reappears here in this specific graphical model, and years of research have been investigated in solving this puzzle. There have been some good practical methods and research directions such as early stopping (Weigend & Rumelhart 1991), weight decay (Caruana et al. 2000) or Bayesian regularization (MacKay 1992) to counter overfitting, and transfer learning (Silver & Bennett 2008; **?**) can be seen as biasing models beyond the current data set.

Most prominent are currently support vector machines (SVMs) that start by minimizing the estimated generalization (called the empirical error in this community). The main idea behind support vector machines (SVM) for binary classification is that the best linear classifier for a separable binary classification problem is the one that maximizes the margin (Vapnik 1995; Cortes & Vapnik 1995). That is, there are many lines that separate the data as shown in Fig.4.10. The one that can be expected most robust is the one that tries to be as far from any data as possible since we can expect new data to be more likely close to the clusters of the training data if the training data are representative of the general distribution. Also, the separating line (hyperplane in higher dimensions) is determined only by a few close points that are called **support vectors**. And Vapnik's important contributions did not stop there. He also formulated the margin maximization problem in a form so that the formulas are quadratic in the parameters and only contain dot products of training vectors, $\mathbf{x}^T\mathbf{x}$ by solving the dual problem in a Lagrange formalism (Vapnik 1995). This has several important benefits. The problem becomes a convex optimization problem that avoids local minima which have crippled MLPs. Furthermore, since only dot products between example vectors appear in these formulations, it is possible to apply of Kernel trick to efficiently generalize these approaches to non-linear functions.

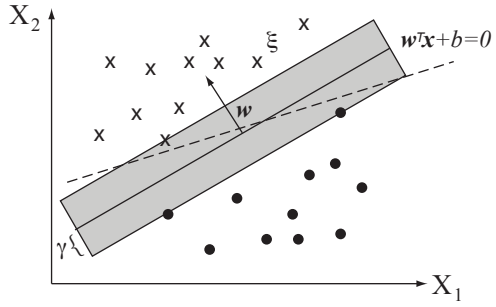


Fig. 4.10 Illustration of linear support vector classification.

Let me illustrate the idea behind using Kernel functions for dot products. To do this it is important to distinguish attributes from features as follows. Attributes are the raw measurements, whereas features can be made up by combining attributes. For example, the attributes x_1 and x_2 could be combines in a feature vector $(x_1, x_2, x_1x_2, x_1^2, x_2^2)^T$. This is a bit like trying to guess a better representation of the problem which should be useful as discussed above with structural learning. So let us now write this transformation as function $\phi(\mathbf{x})$. The interesting part of Vapnik's formulation is that we actually do not even have to calculate this transformation explicitly but can replace the

Table 4.2 Using the SVM implementation from scikit-learn for classification

```
from pylab import *
from sklearn.svm import SVC

N = 300; seed(12345)
#Generate and plot N random samples
r1, r2 = 2 +  rand(N), randn(N)
a1, a2 = 2*pi*rand(N), 0.5*pi*rand(N)
figure("Predict the label")
polar(a1,r1,'.' , a2, r2,'.')

#randomly order classes, convert data to cartesion, split train/test
order = permutation(len(r1)+len(r2))
r = append(r1,r2)[order];  a = append(a1,a2)[order];
labels= append(zeros(N), ones(N))[order];
xy = array([r * cos(a), r * sin(a)]).T
train_data, test_data = xy[:-50], xy[-50:]
train_lbls, test_lbls = labels[:-50], labels[-50:]

#Train and test SVM, plot predictions
svc = SVC() #Radial basis function kernel is default
svc.fit(train_data, train_lbls)
p = svc.predict(test_data).astype(bool)

figure("Predictions");
polar((a[-50:])[-p], (r[-50:])[-p], '.',
      (a[-50:])[p] , (r[-50:])[p] , '.')
show()
```

corresponding dot products as a **Kernel function**

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z}). \tag{4.74}$$

Such Kernel functions are sometimes much easier to calculate. For example, a Gaussian Kernel function corresponds formally to an infinite dimensional feature transformation ϕ. There are some arguments from structural learning (Vapnik 1995; Burges 1998) why SVMs are less prone to overfitting, and extensions have also been made to problems with overlapping data in form of soft margin classification (Cortes & Vapnik 1995) and to more general regression problems (Smola & Schölkopf 2004). We will not dwell more into the theory of Suport Vector Machine but show instead an example using the SVM implementation of the `scikit-learn` library. SVMs are likely currently the most successful general learning machines and should definitely be considered in practical applications. We will discuss later methods that can augment SVMs for even better performances and also discuss methods that go beyond it.

An example of using the LIBSVM library on data shown in Fig.4.11 is given in Table 4.2, though there are other implementations such as in the scikit-learn toolbox. The left graph in in Fig.4.11 shows training data. These data are produced from sampling two distributions. The data of the first class, shown as circles, are chosen within a ring of radius 2 to 3, while the second class, shown as crosses, are Gaussian distributed in two quadrants. These data are given with their corresponding lables to the training function `svmtrain`. The data on the right are test data. The corresponding class labels are given to the function `svmpredict` only to calculate cross validation

error. For true predictions, this vector can be set to arbitrary values. The performance of this classification is around 97% with the standard parameters of the LIBSVM package. However, it is advisable to tune these parameters, for example with some search methods (Boardman & Trappenberg 2006).

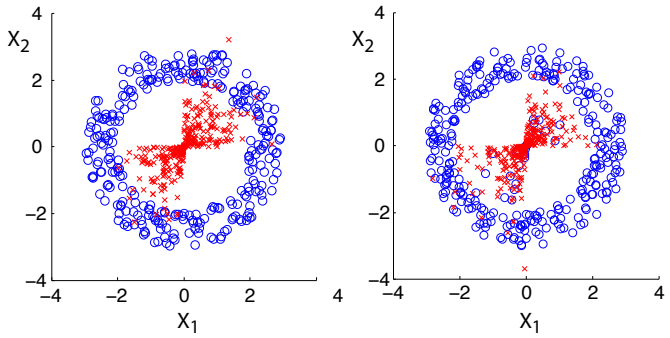


Fig. 4.11 Example of using training data on the left to predict the labels of the test data on the right.