6 Generalized Learning Machines

In this chapter we discuss some methods that are widely used for machine learning applications. In the previous discussions we always assumed specific hypothesis functions for particular problems. However, finding an appropriate hypothesis function requires considerable domain knowledge and it would be much more practical if we would have more general machines that can learn without making very specific functional assumptions. But how can we do this? The approach taken in this chapter is to provide a very general function 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 sizes. This chapter starts with a brief historical introduction to general learning machines and neural networks. We then discuss support vector machines, which are very powerful for many classification examples. SVMs also allow us to outline some more rigorous learning theory. Finally, we will discuss deep convolutional learning that are currently the best known solutions to many large data problems.

6.1 The Perceptron

There was always a strong interest of AI researchers in **real intelligence**, that is, to understand the human mind. For example, both Alan Turing and John von Neumann worked directly on biological systems in their last years before their early passing. Indeed, many AI researchers have been interested, or are continue to be interested, in human behaviour and the brain. Understand how the brain works is an important scientific quest in its own right, but studying the brain as an example of a successful learning machine has often inspire AI approaches or the development of cognitive and learning machines. We will now discuss so called **artificial neural networks** that are often said to be inspired by the brain, and which have a history that dates back to early AI approaches.

A seminal paper, which has greatly influenced the development of early learning machines, is the 1943 paper by Warren McCulloch and Walter Pitts. In this paper, McColloch and Pitts proposed a simple model of a neuron which they called the **threshold logical unit** and which is now often called the **McCulloch–Pitts neuron**. Such a unit is shown in Fig. 6.1A. This model neuron has typically many input channels, although we only depicted three input channels in the figure. The input values are denoted by x with a subscript for each channel. Each channel has a **weight parameter**, w_i. With these parameters, the McCulloch–Pitts neuron operates in the following way. Each input value is multiplied with the corresponding weight value, and these weighted values are then summed. If the weighted and summed input is larger

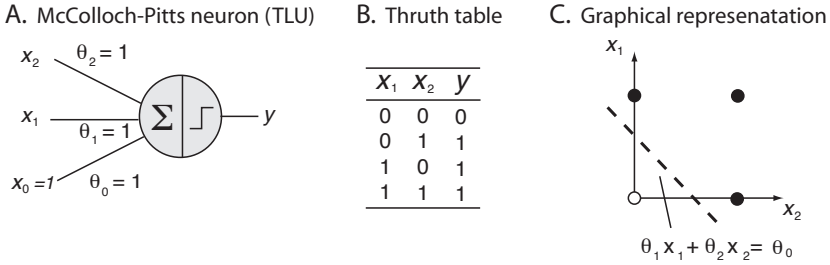


Fig. 6.1 (A) A McCulloch-Pitts neuron model with three input channels. One input is always equal to 1 so that a variable threshold can be represented as the associated weight value. (B) The truth table of a Boolean OR function. (C) Graphical representation of the Boolean OR function. The dotted line represents a decision line that can be implemented with a threshold perceptron (McCulloch-Pitts neuron or threshold linear unit;TLU).

than a certain threshold value, θ, then the output is set to one, and zero otherwise, that is,

$$h(\mathbf{x};\theta) = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} w_i x_i = \mathbf{w}^T\mathbf{x} > \theta \\ 0 \qquad\quad \text{otherwise} \end{cases} . \tag{6.1}$$

The McCulloch-Pitts model resembles in a simplistic way the basic operation of a neuron. That is, a neuron sums synaptic inputs and fires (has a spike in its membrane potential) when the membrane potential reaches a certain level that opens special voltage-gated ion channels. McCulloch and Pitts introduced this unit as a simple neuron model, and they argued that such a unit can perform computational tasks resembling boolean logic. We use here the symbol h to denote the output values since the output of this neuron represents the **hypothesis** that this neuron implements given the parameters $\mathbf{w}$.

A convenient trick to remove the threshold value from the right hand site, which will make our following discussions easier, is to replace it with an additional weight value, w_0 and a corresponding fixed inout value of $x_0 = 1$ as shown in Fig. 6.1A. We will also denote the **net weighted input** of the neuron, before applying the threshold function, as u,

$$u(\mathbf{x};\mathbf{w}) = \sum_{i=0}^{n} w_i x_i = \mathbf{w}^T\mathbf{x}. \tag{6.2}$$

The output of a perceptron is then given by transferring this net input through a gain function $g(u)$,

$$h(\mathbf{x};\theta) = g(\mathbf{u}) \tag{6.3}$$

where the McCulloch-Pitts neurone model uses the Heaviside function and gain function

$$g(u) = \begin{cases} 1 \;\text{ if } u > 0 \\ 0 \text{ otherwise} \end{cases} . \tag{6.4}$$

The use of the non-linear step-function used in this neuron model is appropriate when applying these models to classification. This is demonstrated in Fig. 6.1 for a threshold unit that implements the Boolean OR function. However, other grain functions are more appropriate when applying the perceptron to regression problems. For example,

the **linear perceptron**, which is a perceptron with a linear gain function $g(u) = u$ is equivalent to linear regression. We will discuss other gain functions further below.

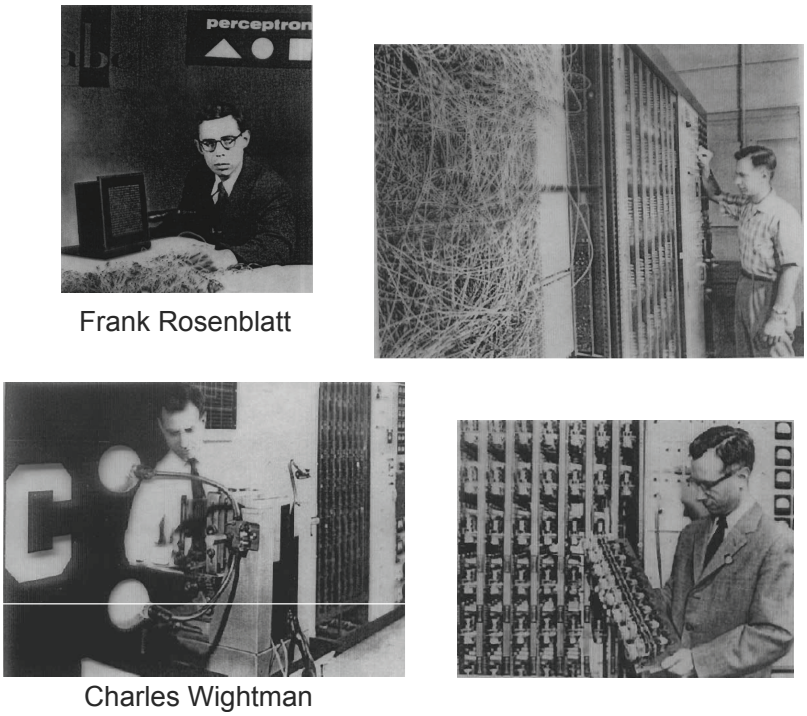


Fig. 6.2 Neural Network computers in the late 1950s.

A further major developments in the use of perceptrons were done by Frank Rosenblatt and his engineering colleague Charles Wightman (Fig. 6.2). It was actually Rosenblatt who called these elements a **perceptron**. As can be seen in the figures, they worked on a machine that can perform letter recognition, and that the machine consisted of a lot of cables, forming a network of simple, neuron-like elements.

The most important challenge for the team was to find a way how to adjust the parameters of the model, the connection weights w_i, so that the perceptron would perform a task correctly. The procedure was to provide to the system a number of examples, let's say m input data, $\mathbf{x}^{(i)}$ and the corresponding desired outputs, $y^{(i)}$. The procedure they used thus resembles supervised learning. The learning rule they used is called the **perceptron learning rule**,

$$w_j := w_j + \alpha \left(y^{(i)} - h(x_i; \mathbf{w}) \right) x_j^{(i)}, \tag{6.5}$$

which is also related to the Widrow-Hoff learning rule, the Adaline rule, and the delta rule. These learning rules are nearly identical, but are sometimes used in slightly different contexts. It is often called the delta rule because the difference between the desired and actual output (difference between actual (training) data and hypothesis) to guide the learning. When multiplying out the difference with the inputs we end up

with the product of the activity between the desired output and the inputs values for each synaptic channel, minus the product between the desired output times the input,

$$\Delta w_j \propto (yx_j - hx_j). \tag{6.6}$$

Such a learning rule reinforces the correlation between the input and the desired output and reduces the correlation between input and the actual output. The above learning rule also resembles the effect of synaptic plasticity in the brain as first conjectured by the famous Nova Scotian Donald Hebb, and such rules are therefore also called **Hebb rule**.

Note that this learning rule is equivalent to a gradient descent rule for a linear hypothesis function which we will use further below. Although this rule is not ideal for a perceptron with non-linear functions, it turned out that it still works in same cases since it corresponds to taking a step towards minimizing MSE, albeit with a wrong gradient.

There was a lot of excitement during the 1960s in the AI and psychology community about such learning system that resemble some brain functions. However, Minsky and Peppert showed in 1968 that such perceptrons can not represent all possible boolean functions (sometimes called the XOR problem). While it was known at this time that this problem could be overcome by a layered structure of such perceptrons (called **multilayer perceptrons**), a learning algorithms was not widely known at this time. This nearly abolished the field of learning machines, and the AI community concentrated on rule-based systems in the following years.

6.2 Multilayer perceptron (MLP)

So far we have only considered one model neuron which maps an input vector $\mathbf{x}$ into a scalar output y. Of course, one could use several output neurone to implement a mapping function from an input vector to an output vector $\mathbf{y}$. However, the type of mapping functions that could be implemented with such a perceptron a rather limited since any nonlinearity would have to be provided by the grain function $g(u)$. However, if we allow nodes between the input nodes and output nodes, then we could build more elaborate networks. A typical example of the networks studied in the mid 1990 is shown in Fig. 6.3. We could also include connections between different hidden layers, not just between consecutive layers, but the basic layered structure is more suitable for an introductory discussion.

The operation of such a network is a direct generalization of the simple perceptron as the output of a hidden nodes simply becomes the input for a note in the next layer. A multilayer perceptron with a layer of m input nodes, a layer of h hidden nodes, and a layer of n output nodes, is shown in Figure 6.3. The input layer is merely just relaying the inputs, while the hidden and output layer do active calculations. Such a network is thus called a 2-layer network. The term hidden nodes comes from the fact that these nodes do not have connections to the external world such as the input and output nodes. Instead of the step function used in the McCulloch-Pitts model above, most such networks use a sigmoidal non-linearity,

$$g(x) = tanh(\beta x) = 2\frac{1}{1 + e^{-\beta}x} - 1, \tag{6.7}$$

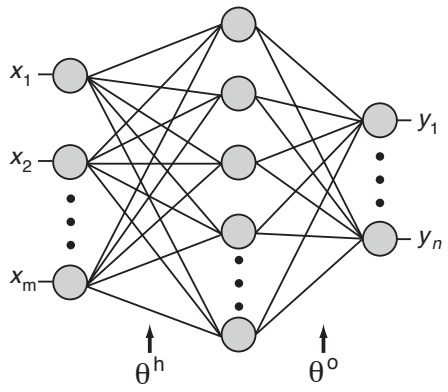


Fig. 6.3 Example of a 2-layer multilayer perceptron (MLP).

to allow for continuous values of the nodes. The network is thus a graphical representation of a nonlinear function of the form

$$h_\theta = g(\mathbf{w}^{\mathrm{o}} g(\mathbf{w}^{\mathrm{h}} \mathbf{x})). \tag{6.8}$$

It is easy to include more hidden layers in this formula. For example, the operation rule for a four-layer network with three hidden layers and one output layer can be written as

$$h_\theta = g(\mathbf{w}^{\mathrm{o}} g(\mathbf{w}^{\mathrm{h}3} g(\mathbf{w}^{\mathrm{h}2} g(\mathbf{w}^{\mathrm{h}1} \mathbf{x})))). \tag{6.9}$$

Let us discuss a special case of a multilayer mapping network where all the nodes in all hidden layers have linear activation functions ($g(x) = x$). Eqn 6.9 then simplifies to

$$\begin{aligned} \mathbf{h} &= \mathbf{w}^{\mathrm{o}} \mathbf{w}^{\mathrm{h}3} \mathbf{w}^{\mathrm{h}2} \mathbf{w}^{\mathrm{h}1} \mathbf{x} \\ &= \tilde{\mathbf{w}} \mathbf{x}. \end{aligned} \tag{6.10}$$

In the last step we have used the fact that the multiplication of a series of matrices simply yields another matrix, which we labelled $\tilde{\mathbf{w}}$. Eqn 6.10 represents a single-layer network as discussed before. It is therefore essential to include non-linear activation functions, at least in the hidden layers, to make possible the advantages of hidden layers that we are about to discuss.

Which functions can be approximated by multilayer perceptrons? The answer is, in principle, any. A multilayer feedforward network is a **universal function approximator**. This means that, given enough hidden nodes, any mapping functions can be approximated with arbitrary precision by these networks. 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. The remaining problems are to know how many hidden nodes we need, and to find the right weight values. Also, the general approximator characteristics does not tell us if it is better to use more hidden layers or just to increase the number of nodes in one hidden layer. Also, learning can be slow in such networks and there are a few other problems that we will address later. These are important concerns for practical engineering applications of those networks.

The main question in the 1970s was how to train the network since it we don't have a supervised signal (desired output) for the hidden nodes. It was hence only by mid 1980s that neural networks became popular again with the introduction of the **error-backpropagation learning rule** in a paper by Rumelhart, Hinton and Williams in 1986 (Rumelhart et al. 1986). In this rule the error or delta term, that is the difference between the actual and desired output, are multiplied by the weight values from a hidden node to the output node to provide this error signal, which looks a lot like propagating a signal back though the network. Such algorithms have been used before, in particular by Sunichi Amari, and it was also proposed in the doctoral thesis of Paul Werbos. However, the renewed discovery by Rumelhart and colleagues let to an explosion of the field of **Artificial Neural Networks** in the 1990s.

The error-backpropagation algorithm is actually just an application of a gradient descent rule of minimizing the MSE for the MLP. Such a rule, as pointed out before, is appropriate for Gaussian data around the mean described by the model. Specifically, the gradient of the MSE error function with respect to the output weights is given by

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{\mathrm{out}}} &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{out}}}\sum_i (r_i^{\mathrm{out}} - y_i)^2 \\ &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{out}}}\sum_i (g^{\mathrm{out}}(\sum_j w_{ij}^{\mathrm{out}} r_j^{\mathrm{h}}) - y_i)^2 \\ &= g^{\mathrm{out}\prime}(h_i^{\mathrm{h}})(\sum_j w_{ij}^{\mathrm{out}} r_j^{\mathrm{h}} - y_i) r_j^{\mathrm{h}} \\ &= \delta_i^{\mathrm{out}} r_j^{\mathrm{h}}, \end{aligned} \tag{6.11}$$

where we have defined the delta factor

$$\begin{aligned}\delta_i^{\mathrm{out}} &= g^{\mathrm{out}\prime}(h_i^{\mathrm{h}})(\sum_j w_{ij}^{\mathrm{out}} r_j^{\mathrm{h}} - y_i) \\ &= g^{\mathrm{out}\prime}(h_i^{\mathrm{h}})(r_i^{\mathrm{out}} - y_i). \end{aligned} \tag{6.12}$$

Eqn 6.11 is just the delta rule as before because we have only considered the output layer. The calculation of the gradients with respect to the weights to the hidden layer again requires the chain rule as they are more embedded in the error function. Thus we have to calculate the derivative

$$\frac{\partial E}{\partial w_{ij}^{\mathrm{h}}} = \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{h}}}\sum_i (r_i^{\mathrm{out}} - y_i)^2$$

Table 6.1 Summary of error-back-propagation algorithm

Initialize weights arbitrarily
Repeat until error is sufficiently small
Apply a sample pattern to the input nodes: $r_i^0 := r_i^{\rm in} = \xi_i^{\rm in}$
Propagate input through the network by calculating the rates of nodes in successive layers l: $r_i^l = g(h_i^l) = g(\sum_j w_{ij}^l r_j^{l-1})$
Compute the delta term for the output layer: $\delta_i^{\rm out} = g'(h_i^{\rm out})(\xi_i^{\rm out} - r_i^{\rm out})$
Back-propagate delta terms through the network: $\delta_i^{l-1} = g'(h_i^{l-1}) \sum_j w_{ji}^l \delta_j^l$
Update weight matrix by adding the term: $\Delta w_{ij}^l = \epsilon \delta_i^l r_j^{l-1}$

$$= \frac{1}{2} \frac{\partial}{\partial w_{ij}^{\rm h}} \sum_i (g^{\rm out}(\sum_j w_{ij}^{\rm out} g^{\rm h}(\sum_k w_{jk}^{\rm h} r_k^{\rm in})) - y_i)^2. \tag{6.13}$$

After some battle with indices (which can easily be avoided with analytical calculation programs such as MAPLE or MATHEMATICA), we can write the derivative in a form similar to that of the derivative of the output layer, namely

$$\frac{\partial E}{\partial w_{ij}^{\rm h}} = \delta_i^{\rm h} r_j^{\rm in}, \tag{6.14}$$

when we define the delta term of the hidden term as

$$\delta_i^{\rm h} = g^{\rm h\prime}(h_i^{\rm in}) \sum_k w_{ik}^{\rm out} \delta_k^{\rm out}. \tag{6.15}$$

The error term $\delta_i^{\rm h}$ is calculated from the error term of the output layer with a formula that looks similar to the general update formula of the network, except that a signal is propagating from the output layer to the previous layer. This is the reason that the algorithm is called the **error-back-propagation algorithm**. The algorithm is summarized in Table 6.1.

Before leaving this area it is useful to point out some more general observations. Artificial neural networks have certainly been one of the first successful methods for nonlinear regression, implementing nonlinear hypothesis of the form $h(\mathbf{x};\theta) = g(\theta^T x)$. The corresponding mean square loss function,

$$L \propto \left(y - g(\theta^T x)\right)^2 \tag{6.16}$$

is then also a general nonlinear function of the parameters. Minimizing such a function is generally difficult. However, we could consider instead hypothesis that are linear in the parameters, $h(\mathbf{x};\theta) = \theta^T \phi(\mathbf{x};$, so that the MSE loss function is quadratic in the parameters,

$$L \propto \left(y - \theta^T \phi(\mathbf{x})\right)^2. \tag{6.17}$$

The corresponding quadratic optimization problem can be solved much more efficiently. This line of ideas are further developed in support vector machines discussed next. An interesting and central further issue is how to chose the non-linear function

ϕ. This will be an important ingredient for nonlinear support vector machines and unsupervised learning discussed below.

As an example of applying such neural networks, consider the problem of letter recognition. In order to present a printed letter to a network, we must first digitize them as could be done with a digital camera. This process is illustrated in Fig.6.4. The result is a unary feature vector that is specific to each letter. Of course, many examples with different digitized binary feature vectors exists for each letter of the alphabet. We are interested in recognizing the meaning of such a printed letter. We could encode the meaning of each letter with the corresponding ASCII representation. Therefore, the letter recognition tasks boils down to the challenge of mapping many examples of binary letter vectors to the vector representing the corresponding ASCII code. The following exercise explores this example application.

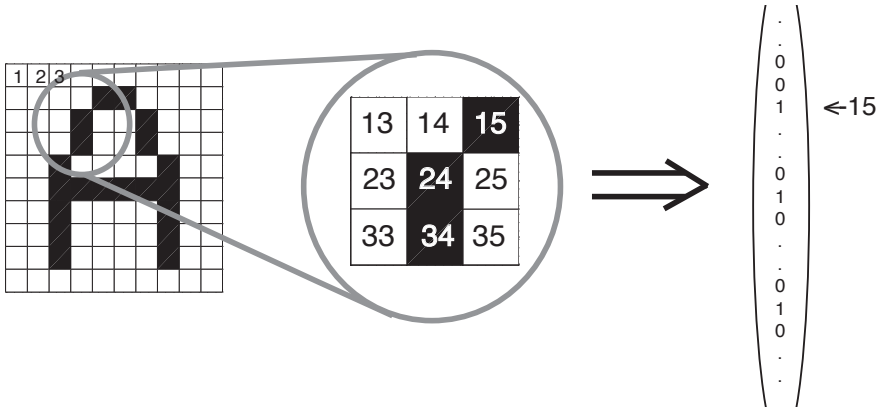


Fig. 6.4 Example of a 2-layer multilayer perceptron (MLP).

Exercise

Write a MLP program that can recognize the letters in file `letter.txt` and test the performance of the network when trying to recognize noisy versions of the letters.

6.3 Support Vector Machine

6.3.1 Basic example

The the basic multilayer perceptrons have been popular in the 1990s, applications to many real world problems turned out to be problematic for several reasons. When applying neural networks to more complex problems, larger networks had to be used. This in turn increased considerable the number of parameters, and the machines hence

required more training examples to be trained. Overfitting and flow learning was a common problem.

Meanwhile, Vladimir Vapnik has been working on statistical learning machines since the 1960, and after moving to the Bell laboratories in the mid 1995, led the development of support vector machine (SVM) that are quite powerful and practical for many applications. The theory behind them does also illuminate the basics behind statistical learning. SVMs are based on 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.6.7. 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.

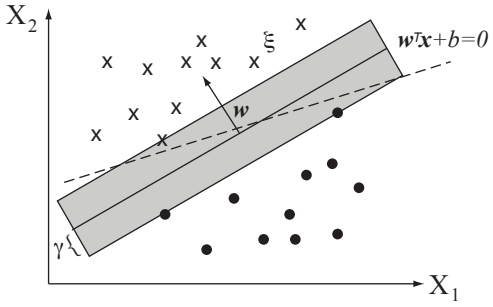


Fig. 6.5 Illustration of linear support vector classification.

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 corresponding dot products as a **Kernel function**

Table 6.2 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()
```

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z}). \tag{6.18}$$

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.6.8 is given in Table 6.2, though there are other implementations such as in the scikit-learn toolbox. The left graph in in Fig.6.8 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).

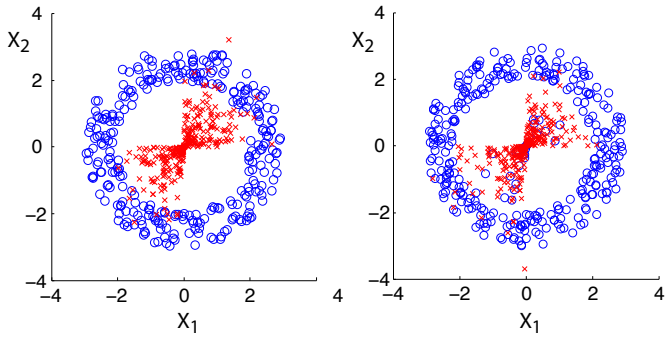


Fig. 6.6 Example of using training data on the left to predict the labels of the test data on the right.

6.3.2 Linear classifiers with large margins

In this section we briefly outline the basic idea behind Support Vector Machines (SVM) that are currently thought to be the best general purpose supervised classifier algorithm. SVMs, and the underlying statistical learning theory, has been worked out by Vladimir Vapnik since the early 1960, but some breakthroughs were also made in the late 1990 with some collaborators like Corinna Cortes, Chris Burges, Alex Smola, and Bernhard Schölkopf to name but a few, and SVM have since become very popular and hard to beat. While we outline some of the underlying formulas, we do not derive all the steps but will try to give some intuition. A more thorough treatment can be found in the references on the web page. The here we just want to provide the big picture, but need to show some formulas to highlight some of the discussion.

The basic SVMs are concerned with binary classification. Figure 6.7 shows an example of two classes, depicted by different symbols, in a two dimensional attribute space. We distinguish here attributes from features as follows. Attributes are the raw measurements, where as features can be made up by combining attributes. For example, the attributes x_1 and x_2 could be combines in a feature vector $(x_1, x_1x_2, x_2, x_1^2, x_2^2)^T$. This will become important later, but it is important to introduce the notation here. Our training set consists again of m data with attribute values $\mathbf{x}^{(i)}$ and labels $y^{(i)}$. We chose here the labels of the two classes as $y \in \{-1, 1\}$, as this will nicely simplify some equations.

The two classes in the figure 6.7 can be separated by a line, which can be parameterized by

$$w_1x_1 + w_2x_2 - b = \mathbf{w}^T\mathbf{x} + b = 0. \tag{6.19}$$

While the first equation shows the lines equation with its components in two dimensions, the next expression is the same in any dimension. Of course, in three dimension

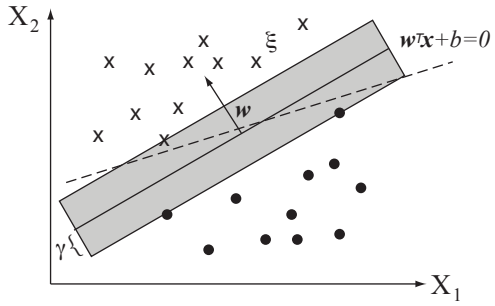


Fig. 6.7 Illustration of linear support vector classification.

we would talk about a plane. In general, we will talk about a **hyperplane** in any dimensions. The particular hyperplane is the dividing or separating hyperplane between the two classes. We also introduce what the **margin** γ, which is the perpendicular distance between the dividing hyperplane and the closest point.

The main point to realize now is that the dividing hyperplane that maximizes the the margin, the so called **maximum margin classifier**, is the best solution we can find. Why is that? We should assume that the training data, shown in the figure, are some unbiased examples of the true underlying density function describing the distribution of points within each class and thus representative of the most likely data. It is then likely that new data points, which we want to classify, are close to the already existing data points. Hence, if we make the separating hyperplane as far as possible from each point, than it is most likely to not make wrong classification. Or, with other words, a separating hyperplane like the one shown as dashed line in the figure, is likely to generalize much worse than the maximum margin hyperplane. So the maximum margin hyperplane is the best generalizer for binary classification for the training data.

What is if we can not divide the data with a hyperplane and we have to consider non-linear separators. Don't we then run into the same problems as outlined before, specifically the bias-variance tradeoff? Yes, indeed, this will still be the challenge, and our aim is really to work on this problem. But before going there it is useful to formalize the linear separable case in some detail as the representation of the optimization problem will be a key in applying some tricks later.

Learning a linear maximum margin classifier on labeled data means finding the parameters (w) and b that maximizes the margin. For this we could computer the distances of each point from the hyperplane, which is simply a geometric exercise,

$$\gamma^{(i)} = y^{(i)} \left((\frac{\mathbf{w}}{||\mathbf{w}||})^T \mathbf{x}^{(i)} + \frac{b}{||\mathbf{w}||} \right). \tag{6.20}$$

The vector $\mathbf{w}/||\mathbf{w}||$ is the normal vector of the hyperplane, a vector of unit length perpendicular to the hyperplane ($||\mathbf{w}||$ is the Euclidean length of the vector $\mathbf{w}$. We overall margin we want to maximize is the distance to the closest point,

$$\gamma = \min_i \gamma^{(i)}. \tag{6.21}$$

By looking at equation 6.20 we see that maximizing γ is equivalent to minimizing $||\mathbf{w}||$, or, equivalently, of minimizing

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2. \tag{6.22}$$

More precisely, we want to maximize this margin under the constraint that no training data lies within the margin,

$$\mathbf{w}^T\mathbf{x}^{(i)} + b \geq 1 \quad \text{for} \quad y^{(i)} = 1 \tag{6.23}$$
$$\mathbf{w}^T\mathbf{x}^{(i)} + b \leq -1 \quad \text{for} \quad y^{(i)} = -1, \tag{6.24}$$

which can nicely be combines with our choice of class representation as

$$-(y^{(i)}(\mathbf{w}^T\mathbf{x} + b) - 1) \leq 0. \tag{6.25}$$

Thus we have a quadratic minimization problem with linear inequalities as constraint. Taking a constrain into account can be done with a **Lagrange formalism**. For this we simply add the constraints to the main objective function with parameters α_i called Lagrange multipliers,

$$Ł^P(\mathbf{w}, b, \alpha_i) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{m} \alpha_i[y^{(i)}(\mathbf{w}^T\mathbf{x} + b) - 1]. \tag{6.26}$$

The Lagrange multipliers determine how well the constrain are observed. In the case of $\alpha_i = 0$, the constrains do not matter. In order conserve the constrains, we should thus make these values as big as we can. Finding the maximum margin classifier is given by

$$p^* = \min_{\mathbf{w},b} \max_{\alpha_i} Ł^P(\mathbf{w}, b, \alpha_i) \leq p^* = \max_{\alpha_i} \min_{\mathbf{w},b} Ł^D(\mathbf{w}, b, \alpha_i) = d^*. \tag{6.27}$$

In this formula we also added the formula when interchanging the min and max operations, and the reason for this is the following. It is straight forward to solve the optimization problem on the left hand side, but we can also solve the related problem on the right hand side which turns out to be essential when generalizing the method to nonlinear cases below. Moreover, the equality hold when the optimization function and the constraints are convex[6]. So, if we minimize Ł by looking for solutions of the derivatives $\frac{\partial Ł}{\partial \mathbf{w}}$ and $\frac{\partial Ł}{\partial b}$, we get

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y^{(i)} \mathbf{x}^{(i)} \tag{6.28}$$
$$0 = \sum_{i=1}^{m} \alpha_i y^{(i)} \tag{6.29}$$

Substituting this into the optimization problem we get

[6]Under these assumptions there are other conditions that hold, called the **Karush-Kuhn-Tucker** conditions, that are useful in providing proof in the convergence of these the methods outlined here.

$$\max_{\alpha_i} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} y^{(i)} y^{(y)} \alpha_i \alpha_j \mathbf{x}^{(i)T} \mathbf{x}^{(j)}, \tag{6.30}$$

subject to the constrains

$$\alpha_i \geq 0 \tag{6.31}$$

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \tag{6.32}$$

From this optimization problem it turns out that the α_i's of only a few examples, those ones that are lying on the margin, are the only ones with have $\alpha_i \neq 0$. The corresponding training examples are called **support vectors**. The actual optimization can be done with several algorithms. In particular, John Platt developed the sequential minimal optimization (SMO) algorithm that is very efficient for this optimization problem. Please note that the optimization problem is convex and can thus be solved very efficiently without the danger of getting stuck in local minima.

Once we found the support vectors with corresponding α_i's, we can calculate (w) from equation 6.28 and b from a similar equation. Then, if we are given a new input vector to be classified, this can then be calculated with the hyperplane equation 6.19 as

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{m} \alpha_i y^{(i)} \mathbf{x}^{(i)T} \mathbf{x} > 0 \\ -1 & \text{otherwise} \end{cases} \tag{6.33}$$

Since this is only a sum over the support vectors, which should be only a few data points from the training set, classification becomes very efficient after training.

6.3.3 Soft margin classifier

So far we only discussed the linear separable case. But how about the case when there are overlapping classes? It is possible to extend the optimization problem by allowing some data points to be in the margin while penalizing these points somewhat. We include therefore some **slag variables** ξ_i that reduce the effective margin for each data point, but we add to the optimization a penalty term that penalizes if the sum of these slag variables are large,

$$\min_{\mathbf{w},b} \frac{1}{2} ||\mathbf{w}||^2 + C \sum_i \xi_i, \tag{6.34}$$

subject to the constrains

$$y^{(i)}(\mathbf{w}^T \mathbf{x} + b) \geq 1 - \xi_i \tag{6.35}$$

$$\xi_i \geq 0 \tag{6.36}$$

The constant C is a free parameter in this algorithm. Making this constant large means allowing less points to be in the margin. This parameter must be tuned and it is advisable to at least try to vary this parameter to verify that the results do not dramatically depend on a initial choice.

6.3.4 Nonlinear Support Vector Machines

We have treated the case of overlapping classes while assuming that the best we can do is still a linear separation. But what if the underlying problem is separable, f only with a more complex function. We will now look into the non-linear generalization of the SVM.

When discussing regression we started with the linear case and then discussed non-linear extensions such as regressing with polynomial functions. For example, a linear function in two dimensions (two attribute values) is given by

$$y = w_0 + w_1 x_1 + w_2 x_2, \tag{6.37}$$

and an example of a non-linear function, that of an polynomial of 3rd order, is given by

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2. \tag{6.38}$$

The first case is a linear regression of a feature vector

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}. \tag{6.39}$$

We can also view the second equation as that of linear regression on a feature vector

$$\mathbf{x} \rightarrow \phi(\mathbf{x}) = \begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \\ x_1^2 \\ x_2^2 \end{pmatrix}, \tag{6.40}$$

which can be seen as a mapping $\phi(\mathbf{x})$ of the original attribute vector. We call this mapping a **feature map**. Thus, we can use the above maximum margin classification method in non-linear cases if we replace all occurrences of the attribute vector $\mathbf{x}$ with the mapped feature vector $\phi(\mathbf{x})$. There are only two problems remaining. One is that we have again the problem of overfitting as we might use too many feature dimensions and corresponding free parameters w_i. The second is also that with an increased number of dimensions, the evaluation of the equations becomes more computational intensive. However, there is a great trick to alleviate the later problem in the case when the methods only rely on dot products, like in the case of the formulation in the dual problem. In this, the function to be minimized, equation 6.30 with the feature maps, only depends on the dot products $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$. Also, when predicting the class for a new input vector $\mathbf{x}$ from equation 6.28 when adding the feature maps, we only need the resulting values for the dot products $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x})$ which can sometimes be represented as function called **Kernel function**,

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z}). \tag{6.41}$$

Instead of actually specifying a feature map, which is often a guess to start, we could actually specify a Kernel function. For example, let us consider a quadratic feature map

$$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z} + c)^2. \tag{6.42}$$

We can then try to write this in the form of equation 6.41 to find the corresponding feature map. That is

$$K(\mathbf{x},\mathbf{z}) = (\mathbf{x}^T\mathbf{z})^2 + 2c\mathbf{x}^T\mathbf{z} + c^2 \tag{6.43}$$

$$= (\sum_i x_i z_i)^2 + 2c\sum_i x_i z_i + c^2 \tag{6.44}$$

$$= \sum_j \sum_i (x_i x_j)(z_i z_j) + \sum_i (\sqrt{(2c)}x_i)(\sqrt{(2c)}z_i) + cc \tag{6.45}$$

$$= \phi(\mathbf{x})^T\phi(\mathbf{z}), \tag{6.46}$$

with

$$\phi(\mathbf{x}) = \begin{pmatrix} x_1x_1 \\ x_1x_2 \\ \dots \\ x_nx_1 \\ x_nx_2 \\ \dots \\ \sqrt{2c}x_1 \\ \sqrt{2c}x_2 \\ \dots \\ c \end{pmatrix}, \tag{6.47}$$

The dimension of this feature vector is $O(n^2)$ for n original attributes. Thus, evaluating the dot product in the mapped feature space is much more time consuming then calculating the Kernel function which is just the square of the dot product of the original attribute vector. The dimensionality Kernels with higher polynomials is quickly rising, making the benefit of the Kernel method even more impressive.

While we have derived the corresponding feature map for a specific Kernel function, this task is not always easy and not all functions are valid Kernel functions. We have also to be careful that the Kernel functions still lead to convex optimization problems. In practice, only a small number of Kernel functions is used. Besides the polynomial Kernel mention before, one of the most popular is the Gaussian Kernel,

$$K(\mathbf{x},\mathbf{z}) = \exp -\frac{||\mathbf{x}-\mathbf{z}||^2}{2\gamma^2}, \tag{6.48}$$

which corresponds to an infinitely large feature map.

As mentioned above, a large feature space corresponds to a complex model that is likely to be prone to overfitting. We must therefore finally look into this problem. The key insight here is that we are already minimizing the sum of the components of the parameters, or more precisely the square of the norm $||\mathbf{w}||^2$. This term can be viewed as **regularization** which favours a smooth decision hyperplane. Moreover, we have discussed two extremes in classifying complicated data, one was to use Kernel functions to create high-dimensional non-linear mappings and hence have a high-dimensional separating hyperplane, the other method was to consider a low-dimensional separating hyperplane and interpret the data as overlapping. The last method includes a parameter C that can be used to tune the number of data points

that we allow to be within the margin. Thus, we can combine these two approaches to classify non-linear data with overlaps where the soft margins will in addition allow us to favour more smooth dividing hyperplanes.

6.3.5 Regularization and parameter tuning

In practice we have to consider several free parameters when applying support vector machines. First, we have to decide which Kernel function to use. Most packages have a number of choices implemented. We will use for the following discussion the Gaussian Kernel function with width parameter γ. Setting a small value for γ and allowing for a large number of support vectors (small C), corresponds to a complex model. In contrast, larger width values and regularization constant C will increase the stiffness of the model and lower the complexity. In practice we have to tune these parameters to get good results. To do this we need to use some form of validation set, as discussed in section **??**, and k-times cross validation is often implemented in the software packages. An example of the SVM performance (accuracy) on some examples (Iris Data set from the UCI repository; From Broadman and Trappenberg, 2006) is shown in figure 6.8 for several values of γ and C. It is often typical that there is a large area where the SVM works well and has only little variations in terms of performance. This robustness has helped to make SVMs practical methods that often outperform other methods. However, there is often also an abrupt onset of the region where the SVM fails, and some parameter tuning is hence required. While just trying a few settings might be sufficient, some more systematic methods such as grid search or simulated annealing also work well.

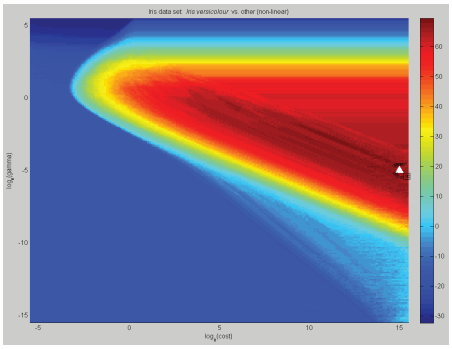


Fig. 6.8 Illustration of SVM accuracy for different values of paraneters C abd γ.

6.3.6 Statistical learning theory and VC dimension

SVMs are good and practical classification algorithms for several reasons, including the advantage of being convex optimization problem that than can be solved with quadratic programming, have the advantage of being able to utilize the Kernel trick, have a compact representation of the decision hyperplane with support vectors, and turn out to be fairly robust with respect to the hyper parameters. However, in order to

be good learners, they need to moderate the variance-bias tradeoff dicussed in section **??**. A great theoretical contributions of Vapnik and colleagues was the embedding of supervised learning into statistical learning theory and to derive some bounds that make statements on the average ability to learn form data. We outline here briefly the ideas and state some of the results. We discuss this issue here in the context of binary classification, although similar observations can be made in the case of multiclass classification and regression.

We start again by stating our objective, which is to find a hypothesis which minimized the generalization error. To state this a bit more differentiated and to use the nomenclature common in these discussions, we call the error function here the **risk function** R. In particular, the **expected risk** for a binary classification problem is the probability of misclassification,

$$R(h) = P(h(x) \neq y) \tag{6.49}$$

Of course, we generally do not know this density function, though we need to approximate this with our validation data. We assume thereby again that the samples are iid (independent and identical distributed) data, and can then estimate what is called the **empirical risk**,

$$\hat{R}(h) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}(h(\mathbf{x}^{(i)};\theta) = y^{(i)}). \tag{6.50}$$

We use here again m as the number of examples, but note that this is here the number of examples in the validations set, which is the number of all training data minus the ones used for training. Also, we will discuss this empirical risk further, but note that it is better to use the regularized version that incorporates a smoothness constrain such as

$$\hat{R}_{rmreg}(h) = \frac{1}{m}\sum_{i} \mathbb{1}(h(\mathbf{x}^{(i)};\theta) = y^{(i)}) - \lambda||\mathbf{w}||^2 \tag{6.51}$$

in the case of SVM, where λ is a regularization constant. Thus, wherever $\hat{R}(h)$ is used in the following, we can replace this with $\hat{R}_{rmreg}(h)$. **Empirical risk minimization** is the process of finding the hypothesis $\hat{h}$ that minimizes the empirical risk,

$$\hat{h} = arg\min_{h} \hat{R}(h). \tag{6.52}$$

The empirical risk is the MLE of the mean of a Bernoulli-distributed random variable with true mean $R(h)$. Thus, the empirical risk is itself a random variable for each possible hypothesis h. Let us first assume that we have k possible hypothesis h_i. We now draw on a theorem by Hoeffding called the **Hoeffding inequality** that provides and upper bound for the sum of random numbers to its mean,

$$P(|R(h_i) - \hat{R}(h_i)| > \gamma) \leq 2\exp(-2\gamma^2 m). \tag{6.53}$$

This formula states that there is a certain probability that we make an error larger than γ for each hypothesis of the empirical risk compared to the expected risk, although the good news is that this probability is bounded and that the bound itself becomes exponentially smaller with the number of validation examples. This is already an

interesting results, but we now want to know the probability that some, out of all possible hypothesis, are less than γ. Using the fact that the probability of the union of several events is always less or equal to the sum of the probabilities, one can show that with probability $1 - \delta$ the error of a hypothesis is bounded by

$$|R(h) - \hat{R}(h)| \leq \sqrt{\frac{1}{2m} \log \frac{2k}{\delta}}. \tag{6.54}$$

This is a great results since it shows how the error of using an estimate the risk, the empirical risk that we can evaluate from the validation data, is getting smaller with training examples and with the number of possible hypothesis.

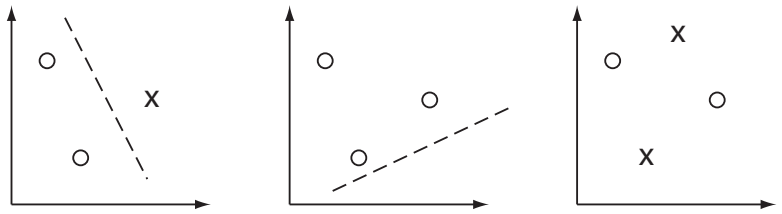


Fig. 6.9 Illustration of VC dimensions for the class of linear functions in two dimensions.

While the error scales only with the log of the number of possible hypothesis, the values goes still to infinite when the number of possible hypothesis goes to infinite, which much more resembles the situation when we have parameterized hypothesis. However, Vapnik was able to show the following generalization in the infinite case, which is that given a hypothesis space with **Vapnic-Chervonencis** dimension VC($\{\mathrm{h}\}$), then, with probability $1 - \delta$, the error of the empirical risk compared to the expected risk (true generalization error) is

$$|R(h) - \hat{R}(h)| \leq O\left(\sqrt{\frac{VC}{m} \log \frac{m}{VC} + \frac{1}{m} \log \frac{1}{\delta}}\right). \tag{6.55}$$

The VC dimensions is thereby a measure of how many points can be divided by a member of the hypothesis set for all possible label combinations of the point. For example, consider three arbitrary points in two dimensions as shown in figure 6.9, and let us consider the hypothesis class of all possible lines in two dimensions. I can always divide the three points under any class membership condition, of which two examples are also shown in the figure. In contrast, it is possible to easily find examples with four points that can not be divided by a line in two dimensions. The VC dimension of lines in two dimensions is hence $VC = 3$.[7]

[7] Three points of different classes can not be separated by a single line, but these are singular points that are not effective in the definition of VC dimension.

6.4 SV-Regression and implementation

6.4.1 Support Vector Regression

While we have mainly discussed classification in the last few sections, it is time to consider the more general case of regression and to connect these methods to the general principle of maximum likelihood estimation outlined in the previous chapter. It is again easy to illustrate the method for the linear case before generalizing it to the non-linear case similar to the strategy followed for SVMs.

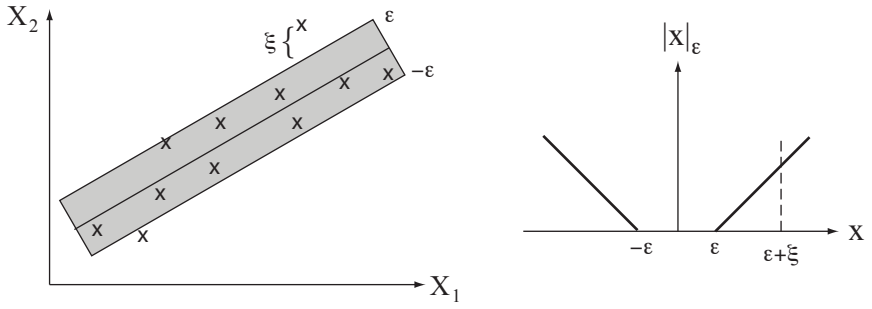


Fig. 6.10 Illustration of support vector regression and the ϵ-insensitive cost function.

We have already mentioned in section 4.1 the ϵ-insensitive error function which does not count deviations of data from the hypothesis that are less than ϵ form the hypothesis, This is illustrated in figure 6.10. The corresponding optimization problem is

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_i(\xi_i + \xi^*), \tag{6.56}$$

subject to the constrains

$$y^{(i)} - \mathbf{w}^T\mathbf{x} - b \leq \xi_i \tag{6.57}$$

$$y^{(i)} - \mathbf{w}^T\mathbf{x} - b \geq \xi_i^* \tag{6.58}$$

$$\xi_i, \xi_i^* \geq 0 \tag{6.59}$$

The dual formulations does again only depend on scalar products between the training examples, and the regression line can be also be expressed by a scalar product between the support vectors and the prediction vector,

$$h(\mathbf{x}; \alpha_i, \alpha_i^*) = \sum_{i=1}^{m}(\alpha_i - \alpha_i^*)\mathbf{x}_i^T\mathbf{x}. \tag{6.60}$$

This, we can again use Kernels to generalize the method to non-linear cases.

6.4.2 Implementation

There are several SVM implementations available, and SVMs are finally becoming a standard component of data mining tools. We will use the implementation

called LIBSVM which was written by Chih-Chung Chang and Chih-Jen Lin and has interfaces to many computer languages including Matlab. There are basically two functions that you need to use, namely `model=svmtrain(y,x,options)` and `svmpredict(y,x,model,options)`. The vectors `x` and `y` are the training data or the data to be tested. `svmtrain` uses k-fold cross validation to train and evaluate the SVM and returns the trained machine in the structure `model`. The function `svmpredict` uses this model to evaluate the new data points. Below is a list of options that shows the implemented SVM variants. We have mainly discussed C-SVC for the basic soft support vector classification, and epsilonSVR for support vector regression.

```
-s svm_type : set type of SVM (default 0)
0 -- C-SVC
1 -- nu-SVC
2 -- one-class SVM
3 -- epsilon-SVR
4 -- nu-SVR
-t kernel_type : set type of kernel function (default 2)
0 -- linear: u'*v
1 -- polynomial: (gamma*u'*v + coef0)^degree
2 -- radial basis function: exp(-gamma*|u-v|^2)
3 -- sigmoid: tanh(gamma*u'*v + coef0)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
-m cachesize : set cache memory size in MB (default 100)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking: whether to use the shrinking heuristics, 0 or 1 (default 1)
-b probability_estimates: whether to train a SVC or SVR model for probability estimates, 0 o
-wi weight: set the parameter C of class i to weight*C, for C-SVC (default 1)

The k in the -g option means the number of attributes in the input data.
```

Exercise: Supervised Line-following

6.4.3 Objective

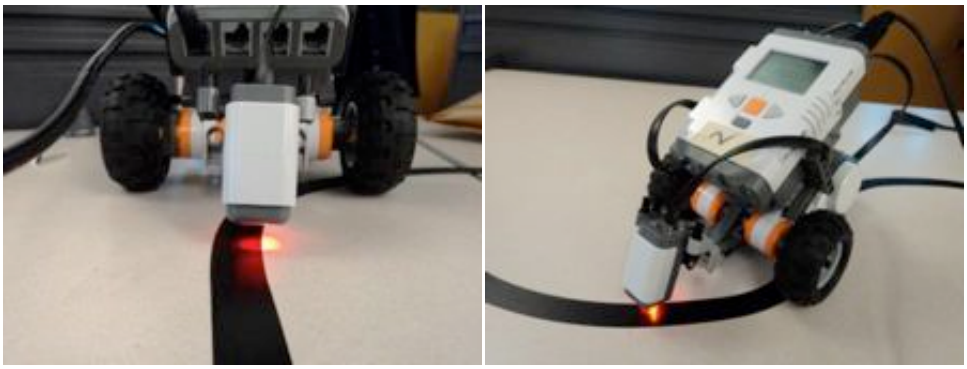
The objective of this experiment is to investigate Supervised Learning through teaching a robot how to follow a line using a Support Vector Machine (SVM).

6.4.4 Setup

- Mount light sensor on front of NXT, plugged into Port 1
- Use a piece of dark tape (i.e. electrical tape) to mark a track on a flat surface. Make sure the tape and the surface are coloured differently enough that the light

sensor returns reasonably different values between the two surfaces.

- This program requires a MATLAB extension that can use Support Vector Machines. Download:



6.4.5 Program

Data collection requires the user to manually move the wheels of the NXT. When the training begins, start the NXT so the light sensor's beam is either on the tape or the surface. Zig zag the NXT so the beam travels on and off the tape by moving either the right or the left wheel, one at a time. Record the positions of the left and right wheels, as well as the light sensor's reading during frequent intervals. It is important to make sure the wheel not in motion stays as stationary as possible to obtain the optimal training set of data.
After data collection, find the difference between the right and the left wheel positions for each time sample taken, and use the SVM to create a model between these differences and the light sensor readings. For instance:

```
model = svmtrain(delta,lightReading,'-s 8 -g 0 -b 1 -e 0.1 -q');
```

To implement the model, place the NXT on the line and use SVMPredict to input a light sensor reading and drive the robot left or right depending on the returned value of the SVMPredict.

```
lightVal=GetLight(SENSOR_1);
if svmpredict(0,lightVal,model,'0')>0
    left.Stop();
    right.SendToNXT();
else
    right.Stop();
    left.SendToNXT();
end
```