NONLINEAR CLASSIFICATION
AND REGRESSION
Nonlinear Classification and Regression: Outline

- Multi-Layer Perceptrons
  - The Back-Propagation Learning Algorithm
- Generalized Linear Models
  - Radial Basis Function Networks
  - Sparse Kernel Machines
    - Nonlinear SVMs and the Kernel Trick
    - Relevance Vector Machines
Nonlinear Classification and Regression: Outline

- **Multi-Layer Perceptrons**
  - The Back-Propagation Learning Algorithm

- **Generalized Linear Models**
  - Radial Basis Function Networks
  - Sparse Kernel Machines
    - Nonlinear SVMs and the Kernel Trick
    - Relevance Vector Machines
AND and OR operations are linearly separable problems.
The XOR Problem

- XOR is not linearly separable.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>XOR</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>B</td>
</tr>
</tbody>
</table>

- How can we use linear classifiers to solve this problem?
Combining two linear classifiers

- Idea: use a logical combination of two linear classifiers.

\[ g_2(x) = x_1 + x_2 - \frac{3}{2} \]

\[ g_1(x) = x_1 + x_2 - \frac{1}{2} \]
Combining two linear classifiers

Let $f(x)$ be the unit step activation function:

$f(x) = 0, \ x < 0$
$f(x) = 1, \ x \geq 0$

Observe that the classification problem is then solved by

$$f\left(y_1 - y_2 - \frac{1}{2}\right)$$

where

$y_1 = f\left(g_1(x)\right)$ and $y_2 = f\left(g_2(x)\right)$
Combining two linear classifiers

- This calculation can be implemented sequentially:
  1. Compute \( y_1 \) and \( y_2 \) from \( x_1 \) and \( x_2 \).
  2. Compute the decision from \( y_1 \) and \( y_2 \).
- Each layer in the sequence consists of one or more linear classifications.
- This is therefore a two-layer perceptron.

\[
g_2(x) = x_1 + x_2 - \frac{3}{2} x_2
\]

where

\[
y_1 = f\left(g_1(x)\right) \text{ and } y_2 = f\left(g_2(x)\right)
\]

\[
g_1(x) = x_1 + x_2 - \frac{1}{2} x_2
\]
The Two-Layer Perceptron

\[
g_2(x) = x_1 + x_2 - \frac{3}{2}x_2
\]

\[
g_1(x) = x_1 + x_2 - \frac{1}{2}
\]

\[
f\left(y_1 - y_2 - \frac{1}{2}\right)
\]

where

\[y_1 = f\left(g_1(x)\right)\text{ and } y_2 = f\left(g_2(x)\right)\]
The Two-Layer Perceptron

- The first layer performs a nonlinear mapping that makes the data linearly separable.

\[ y_1 = f\left(g_1(x)\right) \text{ and } y_2 = f\left(g_2(x)\right) \]

\[ g_2(x) = x_1 + x_2 - \frac{3}{2} \]

\[ g_1(x) = x_1 + x_2 - \frac{1}{2} \]
The Two-Layer Perceptron Architecture

Input Layer

Hidden Layer

Output Layer

\[ g_1(x) = x_1 + x_2 - \frac{1}{2} \]

\[ g_2(x) = x_1 + x_2 - \frac{3}{2} \]

\[ y_1 - y_2 - \frac{1}{2} \]
Note that the hidden layer maps the plane onto the vertices of a unit square.

\[ y_1 = f(g_1(x)) \text{ and } y_2 = f(g_2(x)) \]

\[ g_1(x) = x_1 + x_2 - \frac{1}{2} \]

\[ g_2(x) = x_1 + x_2 - \frac{3}{2} \]
Higher Dimensions

- Each hidden unit realizes a hyperplane discriminant function.
- The output of each hidden unit is 0 or 1 depending upon the location of the input vector relative to the hyperplane.

\[ x \in \mathbb{R}^l \Rightarrow y = [y_1, \ldots, y_p]^T, \quad y_i \in \{0, 1\} \quad i = 1, 2, \ldots, p \]
Higher Dimensions

- Together, the hidden units map the input onto the vertices of a $p$-dimensional unit hypercube.

\[ x \in \mathbb{R}^l \quad \xrightarrow{\quad \rightarrow \quad} \quad x \rightarrow y = [y_1, \ldots, y_p]^T, \quad y_i \in \{0, 1\} \quad i = 1, 2, \ldots p \]
Two-Layer Perceptron

- These $p$ hyperplanes partition the $l$-dimensional input space into polyhedral regions.
- Each region corresponds to a different vertex of the $p$-dimensional hypercube represented by the outputs of the hidden layer.
In this example, the vertex \((0, 0, 1)\) corresponds to the region of the input space where:

- \(g_1(x) < 0\)
- \(g_2(x) < 0\)
- \(g_3(x) > 0\)
Limitations of a Two-Layer Perceptron

- The output neuron realizes a hyperplane in the transformed space that partitions the \( p \) vertices into two sets.

- Thus, the two layer perceptron has the capability to classify vectors into classes that consist of unions of polyhedral regions.

- But **NOT ANY** union. It depends on the relative position of the corresponding vertices.

- How can we solve this problem?
The Three-Layer Perceptron

- Suppose that Class A consists of the union of $K$ polyhedra in the input space.
- Use $K$ neurons in the 2$^{nd}$ hidden layer.
- Train each to classify one region as positive, the rest negative.
- Now use an output neuron that implements the OR function.
Thus the three-layer perceptron can separate classes resulting from any union of polyhedral regions in the input space.
The Three-Layer Perceptron

- The first layer of the network forms the hyperplanes in the input space.
- The second layer of the network forms the polyhedral regions of the input space.
- The third layer forms the appropriate unions of these regions and maps each to the appropriate class.
Learning Parameters
The training data consist of $N$ input-output pairs:

$$\left(y(i), x(i)\right), \quad i \in 1, \ldots, N$$

where

$$y(i) = \begin{bmatrix} y_1(i), \ldots, y_{k_l}(i) \end{bmatrix}^t$$

and

$$x(i) = \begin{bmatrix} x_1(i), \ldots, x_{k_0}(i) \end{bmatrix}^t$$
Choosing an Activation Function

- The unit step activation function means that the error rate of the network is a discontinuous function of the weights.
- This makes it difficult to learn optimal weights by minimizing the error.
- To fix this problem, we need to use a smooth activation function.
- A popular choice is the sigmoid function we used for logistic regression:
Smooth Activation Function

\[ f(a) = \frac{1}{1 + \exp(-a)} \]
Output: Two Classes

- For a binary classification problem, there is a single output node with activation function given by

\[ f(a) = \frac{1}{1 + \exp(-a)} \]

- Since the output is constrained to lie between 0 and 1, it can be interpreted as the probability of the input vector belonging to Class 1.
For a $K$-class problem, we use $K$ outputs, and the softmax function given by

$$y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

Since the outputs are constrained to lie between 0 and 1, and sum to 1, $y_k$ can be interpreted as the probability that the input vector belongs to Class $K$. 
Non-Convex

- Now each layer of our multi-layer perceptron is a logistic regressor.
- Recall that optimizing the weights in logistic regression results in a convex optimization problem.
- Unfortunately the cascading of logistic regressors in the multi-layer perceptron makes the problem non-convex.
- This makes it difficult to determine an exact solution.
- Instead, we typically use gradient descent to find a locally optimal solution to the weights.
- The specific learning algorithm is called the backpropagation algorithm.
End of Lecture
Nonlinear Classification and Regression: Outline

- Multi-Layer Perceptrons
  - The Back-Propagation Learning Algorithm

- Generalized Linear Models
  - Radial Basis Function Networks
  - Sparse Kernel Machines
    - Nonlinear SVMs and the Kernel Trick
    - Relevance Vector Machines
The Backpropagation Algorithm


Notation

- Assume a network with \( L \) layers
  - \( k_0 \) nodes in the input layer.
  - \( k_r \) nodes in the \( r \)'th layer.
Notation

Let $y_{k}^{r-1}$ be the output of the $k$th neuron of layer $r - 1$.

Let $w_{jk}^{r}$ be the weight of the synapse on the $j$th neuron of layer $r$ from the $k$th neuron of layer $r - 1$. 
\[ y_k^0(i) = x_k(i), \quad k = 1, \ldots, k_0 \]
Notation

Let $\nu^r_j$ be the total input to the jth neuron of layer $r$:

$$\nu^r_j(i) = (w^r_j)^t y^{r-1}(i) = \sum_{k=0}^{k_{r-1}} w^r_{jk} y^{r-1}_k(i)$$

where we define $y^r_0(i) = +1$ to incorporate the bias term.

Then $y^r_j(i) = f\left(\nu^r_j(i)\right) = f\left(\sum_{k=0}^{k_{r-1}} w^r_{jk} y^{r-1}_k(i)\right)$
A common cost function is the squared error:

\[ J = \sum_{i=1}^{N} \varepsilon(i) \]

where \( \varepsilon(i) \triangleq \frac{1}{2} \sum_{m=1}^{k} (e_m(i))^2 = \frac{1}{2} \sum_{m=1}^{k} (y_m(i) - \hat{y}_m(i))^2 \)

and

\( \hat{y}_m(i) = y^r_k(i) \) is the output of the network.
To summarize, the error for input $i$ is given by

$$\varepsilon(i) = \frac{1}{2} \sum_{m=1}^{k_L} \left( e_m(i) \right)^2 = \frac{1}{2} \sum_{m=1}^{k_L} \left( \hat{y}_m(i) - y_m(i) \right)^2$$

where $\hat{y}_m(i) = y^r_k(i)$ is the output of the output layer and each layer is related to the previous layer through

$$y^r_j(i) = f \left( v^r_j(i) \right)$$

and

$$v^r_j(i) = \left( w^r_j \right)^t y^{r-1}(i)$$
Gradient Descent

\[ \varepsilon(i) = \frac{1}{2} \sum_{m=1}^{k_i} (e_m(i))^2 = \frac{1}{2} \sum_{m=1}^{k_i} (\hat{y}_m(i) - y_m(i))^2 \]

- Gradient descent starts with an initial guess at the weights over all layers of the network.
- We then use these weights to compute the network output \( \hat{y}(i) \) for each input vector \( \mathbf{x}(i) \) in the training data.
- This allows us to calculate the error \( \varepsilon(i) \) for each of these inputs.
- Then, in order to minimize this error, we incrementally update the weights in the negative gradient direction:

\[
\mathbf{w}_j^{r \text{(new)}} = \mathbf{w}_j^{r \text{(old)}} - \mu \frac{\partial J}{\partial \mathbf{w}_j^{r}} = \mathbf{w}_j^{r \text{(old)}} - \mu \sum_{i=1}^{N} \frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^{r}}
\]
Since $v_j^r(i) = (w_j^r)^t y^{r-1}(i)$,
the influence of the $j$th weight of the $r$th layer on the error can be expressed as:

$$\frac{\partial \varepsilon(i)}{\partial w_j^r} = \frac{\partial \varepsilon(i)}{\partial v_j^r(i)} \frac{\partial v_j^r(i)}{\partial w_j^r} = \delta_j^r(i) y^{r-1}(i)$$

where

$$\delta_j^r(i) \triangleq \frac{\partial \varepsilon(i)}{\partial v_j^r(i)}$$
Gradient Descent

\[
\frac{\partial \varepsilon(i)}{\partial w_j^r} = \delta_j^r(i)y^{r-1}(i),
\]

where

\[
\delta_j^r(i) \triangleq \frac{\partial \varepsilon(i)}{\partial v_j^r(i)}
\]

For an intermediate layer \( r \),

we cannot compute \( \delta_j^r(i) \) directly.

However, \( \delta_j^r(i) \) can be computed inductively,

starting from the output layer.
Thus at the output layer we have

\[
\delta^L_j(i) = \frac{\partial \epsilon(i)}{\partial v^L_j(i)} = \frac{\partial \epsilon(i)}{\partial e^L_j(i)} \frac{\partial e^L_j(i)}{\partial v^L_j(i)} = e^L_j(i)f'(v^L_j(i))
\]

Recall that \(\hat{y}_m(i) = y^L_j(i) = f(v^L_j(i))\)

\[
f(a) = \frac{1}{1 + \exp(-a)} \rightarrow f'(a) = f(a)(1 - f(a))
\]

\[
\delta^L_j(i) = e^L_j(i)f'(v^L_j(i)) \left(1 - f(v^L_j(i))\right)
\]
Observe that the dependence of the error on the total input to a neuron in a previous layer can be expressed in terms of the dependence on the total input of neurons in the following layer:

\[
\delta_{j}^{r-1}(i) = \frac{\partial \varepsilon(i)}{\partial v_{j}^{r-1}(i)} = \sum_{k=1}^{k} \frac{\partial \varepsilon(i)}{\partial v_{k}^{r}(i)} \frac{\partial v_{k}^{r}(i)}{\partial v_{j}^{r-1}(i)} = \sum_{k=1}^{k} \delta_{k}^{r}(i) \frac{\partial v_{k}^{r}(i)}{\partial v_{j}^{r-1}(i)}
\]

where

\[
v_{k}^{r}(i) = \sum_{m=0}^{k-1} w_{km}^{r} v_{m}^{r-1}(i) = \sum_{m=0}^{k-1} w_{km}^{r} f(v_{m}^{r-1}(i))
\]

Thus we have

\[
\frac{\partial v_{k}^{r}(i)}{\partial v_{j}^{r-1}(i)} = w_{kj}^{r} f'(v_{j}^{r-1}(i))
\]

and so

\[
\delta_{j}^{r-1}(i) = \frac{\partial \varepsilon(i)}{\partial v_{j}^{r-1}(i)} = f'(v_{j}^{r-1}(i)) \sum_{k=1}^{k} \delta_{k}^{r}(i) w_{kj}^{r} = f(v_{j}^{L}(i)) \left(1 - f(v_{j}^{L}(i))\right) \sum_{k=1}^{k} \delta_{k}^{r}(i) w_{kj}^{r}
\]

Thus once the \(\delta_{k}^{r}(i)\) are determined they can be propagated backward to calculate \(\delta_{j}^{r-1}(i)\) using this inductive formula.
Backpropagation: Summary of Algorithm

1. Initialization
   - Initialize all weights with small random values

2. Forward Pass
   - For each input vector, run the network in the forward direction, calculating:
     \[ v^r_j(i) = (w^r_j)^t y^{r-1}(i); \quad y^r_j(i) = f(v^r_j(i)) \]
     and finally \[ \epsilon(i) = \frac{1}{2} \sum_{m=1}^{k} (e_m(i))^2 = \frac{1}{2} \sum_{m=1}^{k} (\hat{y}_m(i) - y_m(i))^2 \]

3. Backward Pass
   - Starting with the output layer, use our inductive formula to compute the \( \delta_{j}^{r-1}(i) \):
     - Output Layer (Base Case): \( \delta_{j}^{L}(i) = e_{j}^{L}(i)f'\left(v_{j}^{L}(i)\right) \)
     - Hidden Layers (Inductive Case): \( \delta_{j}^{r-1}(i) = f'\left(v_{j}^{r-1}(i)\right) \sum_{k=1}^{k_r} \delta_{k}^{r}(i)w_{kj}^{r} \)

4. Update Weights
   \[ w_{j}^{r}(\text{new}) = w_{j}^{r}(\text{old}) - \mu \sum_{i=1}^{N} \frac{\partial \epsilon(i)}{\partial w_{j}^{r}} \quad \text{where} \quad \frac{\partial \epsilon(i)}{\partial w_{j}^{r}} = \delta_{j}^{r}(i)y^{r-1}(i) \]
Batch vs Online Learning

- As described, on each iteration backprop updates the weights based upon all of the training data. This is called **batch learning**.

\[
\begin{align*}
    w_{j}^{r}(\text{new}) &= w_{j}^{r}(\text{old}) - \mu \sum_{i=1}^{N} \frac{\partial \epsilon(i)}{\partial w_{j}^{r}} \\
    \text{where} \quad \frac{\partial \epsilon(i)}{\partial w_{j}^{r}} &= \delta_{j}^{r}(i)y^{r-1}(i)
\end{align*}
\]

- An alternative is to update the weights after each training input has been processed by the network, based only upon the error for that input. This is called **online learning**.

\[
\begin{align*}
    w_{j}^{r}(\text{new}) &= w_{j}^{r}(\text{old}) - \mu \frac{\partial \epsilon(i)}{\partial w_{j}^{r}} \\
    \text{where} \quad \frac{\partial \epsilon(i)}{\partial w_{j}^{r}} &= \delta_{j}^{r}(i)y^{r-1}(i)
\end{align*}
\]
Batch vs Online Learning

- One advantage of batch learning is that averaging over all inputs when updating the weights should lead to smoother convergence.
- On the other hand, the randomness associated with online learning might help to prevent convergence toward a local minimum.
- Changing the order of presentation of the inputs from epoch to epoch may also improve results.
Remarks

□ Local Minima
   ▣ The objective function is in general non-convex, and so the solution may not be globally optimal.

□ Stopping Criterion
   ▣ Typically stop when the change in weights or the change in the error function falls below a threshold.

□ Learning Rate
   ▣ The speed and reliability of convergence depends on the learning rate $\mu$. 
Nonlinear Classification and Regression: Outline

- **Multi-Layer Perceptrons**
  - The Back-Propagation Learning Algorithm

- **Generalized Linear Models**
  - Radial Basis Function Networks
  - Sparse Kernel Machines
    - Nonlinear SVMs and the Kernel Trick
    - Relevance Vector Machines
Generalizing Linear Classifiers

- One way of tackling problems that are not linearly separable is to transform the input in a nonlinear fashion prior to applying a linear classifier.

- The result is that decision boundaries that are linear in the resulting feature space may be highly nonlinear in the original input space.
Nonlinear Basis Function Models

- Generally
  \[ y(x, w) = \sum_{j=0}^{M-1} w_j \phi_j(x) = w^T \phi(x) \]

- where \( \phi_j(x) \) are known as basis functions.

- Typically, \( \phi_0(x) = 1 \), so that \( w_0 \) acts as a bias.
Nonlinear basis functions for classification

- In the context of classification, the discriminant function in the feature space becomes:

\[ g(y(x)) = w_0 + \sum_{i=1}^{M} w_i y_i(x) = w_0 + \sum_{i=1}^{M} w_i \phi_i(x) \]

- This formulation can be thought of as an input space approximation of the true separating discriminant function \( g(x) \) using a set of interpolation functions \( \phi_i(x) \).
The dimensionality $M$ of the feature space may be less than, equal to, or greater than the dimensionality $D$ of the original input space.

- **$M < D$:** This may result in a factoring out of irrelevant dimensions, reduction in the number of model parameters, and resulting improvement in generalization (reduced overlearning).

- **$M > D$:** Problems that are not linearly separable in the input space may become separable in the feature space, and the probability of linear separability generally increases with the dimensionality of the feature space. Thus choosing $M >> D$ helps to make the problem linearly separable.
Cover’s Theorem

“A complex pattern-classification problem, cast in a high-dimensional space non-linearly, is more likely to be linearly separable than in a low-dimensional space, provided that the space is not densely populated.”

— Cover, T.M., Geometrical and Statistical properties of systems of linear inequalities with applications in pattern recognition., 1965

Example
Nonlinear Classification and Regression: Outline

- Multi-Layer Perceptrons
  - The Back-Propagation Learning Algorithm
- Generalized Linear Models
  - Radial Basis Function Networks
  - Sparse Kernel Machines
    - Nonlinear SVMs and the Kernel Trick
    - Relevance Vector Machines
Radial Basis Functions

- Consider interpolation functions (kernels) of the form
  \[ \phi_i \left( \| x - \mu_i \| \right) \]

- In other words, the feature value depends only upon the Euclidean distance to a ‘centre point’ in the input space.

- A commonly used RBF is the isotropic Gaussian:
  \[ \phi_i(x) = \exp \left( - \frac{1}{2\sigma_i^2} \| x - \mu_i \|^2 \right) \]
Relation to KDE

- We can use Gaussian RBFs to approximate the discriminant function $g(x)$:

$$g(y(x)) = w_0 + \sum_{i=1}^{M} w_i y_i(x) = w_0 + \sum_{i=1}^{M} w_i \phi_i(x)$$

- where

$$\phi_i(x) = \exp\left( -\frac{1}{2\sigma_i^2} \|x - \mu_i\|^2 \right)$$

- This is reminiscent of kernel density estimation, where we approximated probability densities as a normalized sum of Gaussian kernels.
Relation to KDE

- For KDE we planted a kernel at each data point. Thus there were $N$ kernels.
- For RBF networks, we generally use far fewer kernels than the number of data points: $M << N$.
- This leads to greater efficiency and generalization.
The Linear Classifier with nonlinear radial basis functions can be considered an artificial neural network where:
- The hidden nodes are nonlinear (e.g., Gaussian).
- The output node is linear.

RBF Network for 2 Classes
Recall that for a perceptron, the output of a hidden unit is invariant on a hyperplane.

For an RBF, the output of a hidden unit is invariant on a circle centred on $\mu_i$.

Thus hidden units are global in a perceptron, but local in an RBF network.
This difference has consequences:

- Multilayer perceptrons tend to learn slower than RBFs.
- However, multilayer perceptrons tend to have better generalization properties, especially in regions of the input space where training data are sparse.
- Typically, more neurons are needed for an RBF than for a multilayer perceptron to solve a given problem.
Parameters

- There are two options for choosing the parameters (centres and scales) of the RBFs:
  1. **Fixed.**
     - For example, randomly select a subset of $M$ of the input vectors and use these as centres. Use a common scale based upon your judgement.
  2. **Learned.**
     - Note that when the RBF parameters are fixed, the weights could be learned using linear classifier techniques (e.g., least squares).
     - Thus the RBF parameters could be learned in an outer loop, by gradient descent.
Nonlinear Classification and Regression: Outline

- Multi-Layer Perceptrons
  - The Back-Propagation Learning Algorithm
- Generalized Linear Models
  - Radial Basis Function Networks
- Sparse Kernel Machines
  - Nonlinear SVMs and the Kernel Trick
  - Relevance Vector Machines
The Kernel Function

Recall that an SVM is the solution to the problem

Maximize $\tilde{L}(a) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m k(x_n, x_m)$

subject to $0 \leq a_n \leq C$ and $\sum_{n=1}^{N} a_n t_n = 0$

A new input $x$ is classified by computing

$y(x) = \sum_{n \in S} a_n t_n k(x, x_n) + b$

Where $S$ is the set of support vectors.

Here we introduced the kernel function $k(x, x')$, defined as

$k(x, x') = \phi(x)^t \phi(x')$

This is more than a notational convenience!!
The Kernel Trick

Maximize \( \tilde{L}(\mathbf{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m) \)

subject to \( 0 \leq a_n \leq C \) and \( \sum_{n=1}^{N} a_n t_n = 0 \)

where \( k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^t \phi(\mathbf{x}') \)

- Note that the basis functions and individual training vectors are no longer part of the objective function.
- Instead all we need is the kernel value (like a distance measure) for all pairs of training vectors.
The Kernel Function

The kernel function $k(\mathbf{x}, \mathbf{x}')$ measures the 'similarity' of input vectors $\mathbf{x}$ and $\mathbf{x}'$ as an inner product in a feature space defined by the feature space mapping $\phi(\mathbf{x})$:

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^t \phi(\mathbf{x}')$$

If $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$ we say that the kernel is stationary.

If $k(\mathbf{x}, \mathbf{x}') = k(\|\mathbf{x} - \mathbf{x}'\|)$ we call it a radial basis function.
We can construct a kernel by selecting a feature space mapping \( \phi(x) \) and then defining

\[
k(x, x') = \phi(x)^t \phi(x')
\]
Alternatively, we can construct the kernel function directly, ensuring that it corresponds to an inner product in some (possibly infinite-dimensional) feature space.
Constructing Kernels

\[ k(x) = \phi(x)^t \phi(x') \]

Example 1: \( k(x, z) = x^t z \)

Example 2: \( k(x, z) = x^t z + c, \ c > 0 \)

Example 3: \( k(x, z) = (x^t z)^2 \)
Kernel Properties

- Kernels obey certain properties that make it easy to construct complex kernels from simpler ones.
Kernel Properties

Given valid kernels $k_1(x, x')$ and $k_2(x, x')$ the following kernels will also be valid:

$$k(x, x') = ck_1(x, x')$$  \hspace{1cm} (6.13)
$$k(x, x') = f(x)k_1(x, x')f(x')$$  \hspace{1cm} (6.14)
$$k(x, x') = q(k_1(x, x'))$$  \hspace{1cm} (6.15)
$$k(x, x') = \exp(k_1(x, x'))$$  \hspace{1cm} (6.16)
$$k(x, x') = k_1(x, x') + k_2(x, x')$$  \hspace{1cm} (6.17)
$$k(x, x') = k_1(x, x')k_2(x, x')$$  \hspace{1cm} (6.18)
$$k(x, x') = k_3(\phi(x), \phi(x'))$$  \hspace{1cm} (6.19)
$$k(x, x') = x^T A x'$$  \hspace{1cm} (6.20)
$$k(x, x') = k_a(x_a, x'_a) + k_b(x_b, x'_b)$$  \hspace{1cm} (6.21)
$$k(x, x') = k_a(x_a, x'_a)k_b(x_b, x'_b)$$  \hspace{1cm} (6.22)

where $c > 0$, $f(\cdot)$ is any function, $q(\cdot)$ is a polynomial with nonnegative coefficients, $\phi(x)$ is a mapping from $x \to \mathbb{R}^M$, $k_3$ is a valid kernel on $\mathbb{R}^M$, $A$ is a symmetric positive semidefinite matrix, $x_a$ and $x_b$ are variables such that $x = (x_a, x_b)$ and $k_a, k_b$ are valid kernels over their respective spaces.
Constructing Kernels

- **Examples:**

\[ k(x, x') = \left( x^t x' + c \right)^M, c > 0 \]  \hspace{1cm} (Use 6.18)

\[ k(x, x') = \exp \left( - \frac{\|x - x'\|^2}{2\sigma^2} \right) \]  \hspace{1cm} (Use 6.14 and 6.16.)

Corresponds to infinite-dimensional feature vector
Nonlinear SVM Example (Gaussian Kernel)
SVMs for Regression

In standard linear regression, we minimize
\[
\frac{1}{2} \sum_{n=1}^{N} (y_n - t_n)^2 + \frac{\lambda}{2} \|w\|^2
\]

This penalizes all deviations from the model.

To obtain sparse solutions, we replace the quadratic error function by an \(\varepsilon\)-insensitive error function, e.g.,

\[
E_{\varepsilon}(y(x) - t) = \begin{cases} 
0, & \text{if } |y(x) - t| < \varepsilon \\
|y(x) - t| - \varepsilon, & \text{otherwise}
\end{cases}
\]

See text for details of solution.
Example
Nonlinear Classification and Regression: Outline

- Multi-Layer Perceptrons
  - The Back-Propagation Learning Algorithm
- Generalized Linear Models
  - Radial Basis Function Networks
  - Sparse Kernel Machines
    - Nonlinear SVMs and the Kernel Trick
    - Relevance Vector Machines
Relevance Vector Machines

- Some drawbacks of SVMs:
  - Do not provide posterior probabilities.
  - Not easily generalized to $K > 2$ classes.
  - Parameters $(C, \varepsilon)$ must be learned by cross-validation.

- The Relevance Vector Machine is a sparse Bayesian kernel technique that avoids these drawbacks.

- RVMs also typically lead to sparser models.
RVMs for Regression

\[ p(t \mid x, w, \beta) = \mathcal{N}(t \mid y(x), \beta^{-1}) \]

where \( y(x) = w^t \phi(x) \)

In an RVM, the basis functions \( \phi(x) \) are kernels \( k(x, x_n) \):

\[ y(x) = \sum_{n=1}^{N} w_n k(x, x_n) + b \]

However, unlike in SVMs, the kernels need not be positive definite, and the \( x_n \) need not be the training data points.
RVMs for Regression

Likelihood:

\[ p(t | X, w, \beta) = \prod_{n=1}^{N} p(t_n | x_n, w, \beta) \]

where the \( n^{th} \) row of \( X \) is \( x_n^t \).

Prior:

\[ p(w | \alpha) = \prod_{i=1}^{M} N(w_i | 0, \alpha_i^{-1}) \]

- Note that each weight parameter has its own precision hyperparameter.
RVMs for Regression

\[ p(w_i | \alpha_i) = N(w_i | 0, \alpha_i^{-1}) \]
\[ p(\alpha_i) = \text{Gam}(\alpha_i | a, b) \]
\[ p(w_i) = \text{St}(w_i | 2a) \]

- The conjugate prior for the precision of a Gaussian is a gamma distribution.
- Integrating out the precision parameter leads to a Student’s t distribution over \( w_i \).
- Thus the distribution over \( w \) is a product of Student’s t distributions.
- As a result, maximizing the evidence will yield a sparse \( w \).
- Note that to achieve sparsity it is critical that each parameter \( w_i \) has a separate precision \( \alpha_i \).
RVMs for Regression

\[ p(w_i | \alpha_i) = N(w_i | 0, \alpha_i^{-1}) \]

\[ p(\alpha_i) = \text{Gam}(\alpha_i | a, b) \]

\[ p(w_i) = \text{St}(w_i | 2a) \]

Gaussian prior

Marginal prior: single \( \alpha \)

Independent \( \alpha \)

Gamma Distribution: \( p(x | a,b) = \frac{b^a}{\Gamma(a)} x^{a-1} e^{-bx} \).

Student's t Distribution: \( p(x | \nu) = \frac{\Gamma\left(\frac{\nu + 1}{2}\right)}{\sqrt{\nu \pi} \Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu + 1}{2}} \).

Also recall the rule for transforming densities:

If \( y \) is a monotonic function of \( x \), then

\[ p_y(y) = p_x(x) \left| \frac{dx}{dy} \right| \]

Thus if we let \( a \to 0, b \to 0 \), then

\[ p(\log \alpha_i) \to \text{uniform} \text{ and } p(w_i) \to \left| w_i \right|^{-1}. \]

Very sparse!!
RVMs for Regression

Likelihood:

\[ p(t | X, w, \beta) = \prod_{n=1}^{N} p(t_n | x_{n}, w, \beta) \]

where the \( n^{th} \) row of \( X \) is \( x_n \).

Prior:

\[ p(w | \alpha) = \prod_{i=1}^{M} N(w_i | 0, \alpha_i^{-1}) \]

- In practice, it is difficult to integrate \( \alpha \) out exactly.
- Instead, we use an approximate maximum likelihood method, finding ML values for each \( \alpha_i \).
- When we maximize the evidence with respect to these hyperparameters, many will \( \to \infty \).
- As a result, the corresponding weights will \( \to 0 \), yielding a sparse solution.
Since both the likelihood and prior are normal, the posterior over $w$ will also be normal:

Posterior:
$$p(w | t, X, \alpha, \beta) = N(w | m, \Sigma)$$

where
$$m = \beta \Sigma \Phi^t t$$
$$\Sigma = (A + \beta \Phi^t \Phi)^{-1}$$

and
$$\Phi_{ni} = \phi_i(x_n)$$
$$A = \text{diag}(\alpha_i)$$

Note that when $\alpha_i \to \infty$, the $i^{th}$ row and column of $\Sigma \to 0$, and
$$p(w_i | t, X, \alpha, \beta) = N(w_i | 0, 0)$$
The values for $\alpha$ and $\beta$ are determined using the evidence approximation, where we maximize

$$p(t | X, \alpha, \beta) = \int p(t | X, w, \beta) p(w | \alpha) dw$$

In general, this results in many of the precision parameters $\alpha_i \rightarrow \infty$, so that $w_i \rightarrow 0$.

Unfortunately, this is a non-convex problem.
Example