Never Stand Still

Kernel Methods

COMP9417 Machine Learning & Data Mining

Term 1, 2020

Adapted from slides by Dr Michael Bain

Aims

This lecture will develop your understanding of kernel methods in machine learning. Following it you should be able to:

- describe perceptron learning
- describe learning with the dual perceptron
- outline the idea of learning in a dual space
- describe the concept of maximizing the margin in linear classification
- outline the typical loss function for maximizing the margin
- describe the method of support vector machines (SVMs)
- describe the concept of kernel functions
- outline the idea of using a kernel in a learning algorithm
- outline non-linear classification with kernel methods

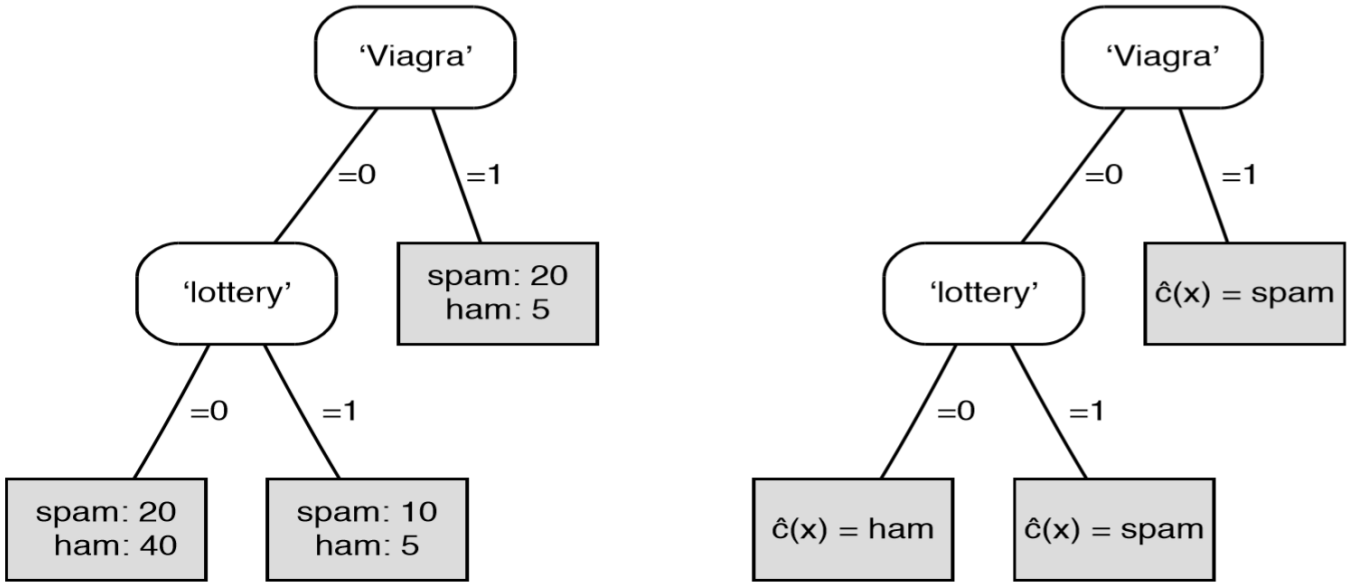
Predictive machine learning scenarios

Task	*Label space*	*Output space*	*Learning problem*
Classification	$\mathcal{L} = \mathcal{C}$	$\mathcal{Y} = \mathcal{C}$	learn an approximation $\hat{c} : \mathcal{X} \to \mathcal{C}$ to the true labelling function c
Scoring and ranking	$\mathcal{L} = \mathcal{C}$	$\mathcal{Y} = \mathbb{R}^{\lvert\mathcal{C}\rvert}$	learn a model that outputs a score vector over classes
Probability estimation	$\mathcal{L} = \mathcal{C}$	$\mathcal{Y} = [0,1]^{\lvert\mathcal{C}\rvert}$	learn a model that outputs a probability vector over classes
Regression	$\mathcal{L} = \mathbb{R}$	$\mathcal{Y} = \mathbb{R}$	learn an approximation $\hat{f} : \mathcal{X} \to \mathbb{R}$ to the true labelling function f

Classification

- A classifier is a mapping $\hat{c}: \mathcal{X} \rightarrow C$, where $C = \{C_1, C_2, \ldots, C_k\}$ is a finite and usually small set of class labels. We will sometimes also use C_i to indicate the set of examples of that class.
- We use the 'hat' to indicate that $\hat{c}(x)$is an estimate of the true but unknown function $c(x)$. Examples for a classifier take the form $(x, c(x))$, where $x \in \mathcal{X}$ is an instance and $c(x)$ is the true class of the instance (sometimes contaminated by noise).
- Learning a classifier involves constructing the function such that it matches c as closely as possible (and not just on the training set, but ideally on the entire instance space $\mathcal{X}$).

A decision tree



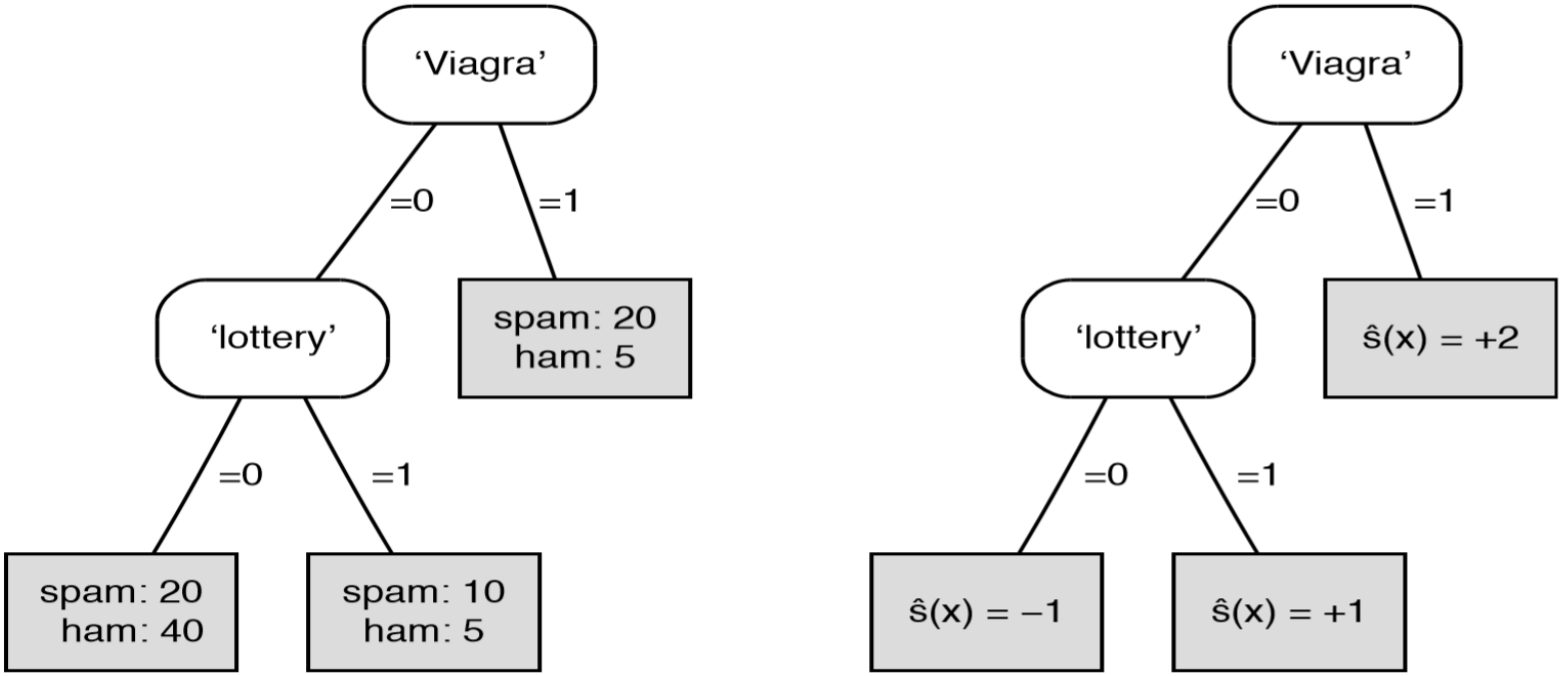
(left) A tree with the training set class distribution in the leaves.

(right) A tree with the majority class prediction rule in the leaves

Scoring classifier

- A scoring classifier is a mapping $\widehat{S}: \mathcal{X} \rightarrow \mathbb{R}^k$ i.e., a mapping from the instance space $\mathcal{X}$ to a k –vector of real numbers.
- The notation indicates that a scoring classifier outputs a vector $\hat{S}(x) = (\hat{s}_1(x), \ldots, \hat{s}_k(x))$ rather than a single number; $\hat{s}_i(x)$ is the score assigned to class C_i for instance x.
- This score indicates how likely it is that class label C_i applies.
- If we only have two classes, it usually suffices to consider the score for only one of the classes; in that case, we use $\hat{s}(x)$ to denote the score of the positive class for instance x.

A scoring tree



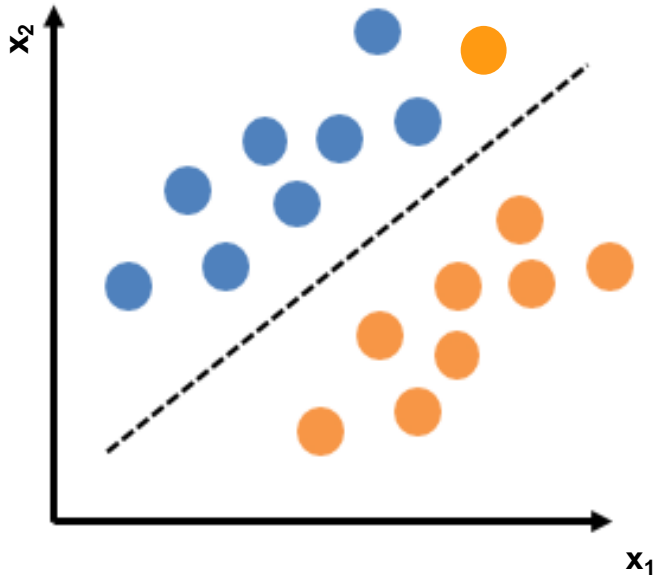
(left) A tree with the training set class distribution in the leaves.

(right) A tree using the logarithm of the class ratio as scores; spam is taken as the positive class.

Margins and loss functions

If we take the true class $c(x)$ as $+1$ for positive examples and -1 for negative examples, then the quantity $z(x) = c(x) \times \hat{s}(x)$ is positive for correct predictions and negative for incorrect predictions: this quantity is called the margin assigned by the scoring classifier to the example.

For example in a linear classifier, we can define the score to be the distance between the examples and the line

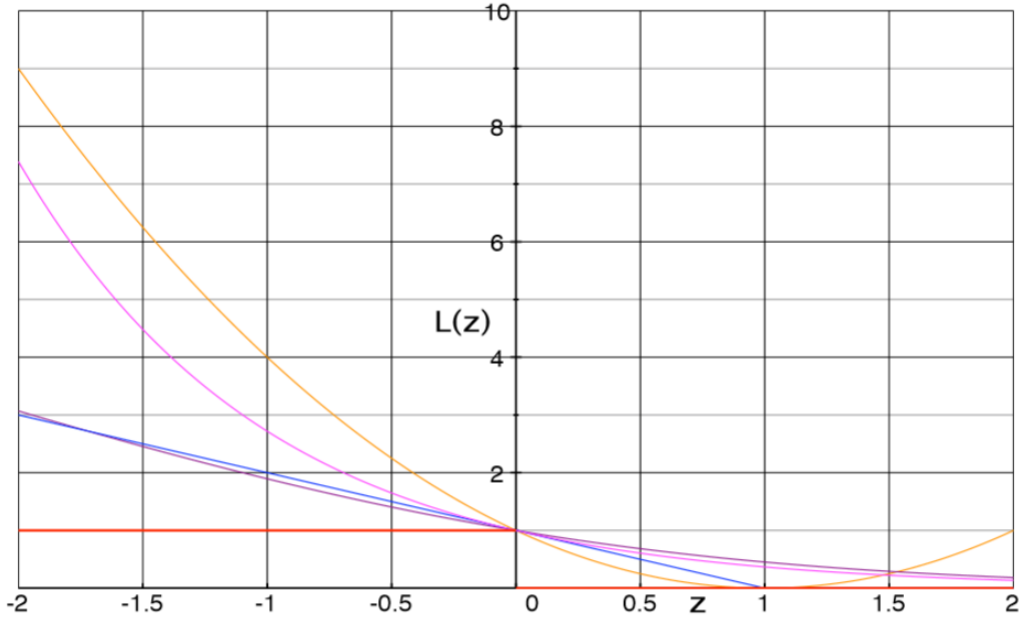


Margins and loss functions

We would like to reward large positive margins and penalize large negative values. This is achieved by means of a so-called **loss function** $L : R \rightarrow [0, \infty)$ which maps each example's margin $z(x)$ to an associated loss $L(z(x))$.

We will assume that $L(0) = 1$, which is the loss incurred by having an example on the decision boundary. We furthermore have $L(z) \geq 1$ for $z < 0$, and usually also $0 \leq L(z) < 1$ for $z > 0$. The average loss over a test set Te is $\frac{1}{|Te|} \sum_{x \in Te} L(z(x))$

Loss functions

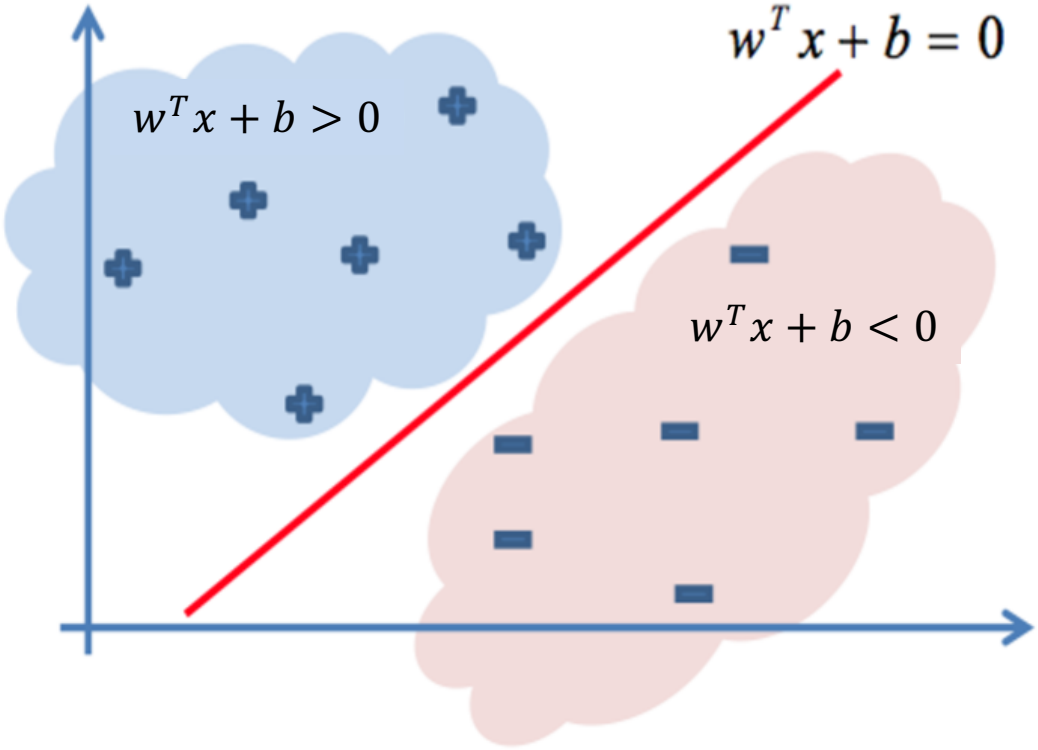


From bottom-left: (i) 0–1 loss $L_{01}(z) = 1$ if $z \leq 0$, and $L_{01}(z) = 0$ if $z > 0$; (ii) hinge loss $L_h(z) = (1 - z)$ if $z \leq 1$, and $L_h(z) = 0$ if $z > 1$; (iii) logistic loss $L_{log}(z) = log_2(1 + \exp(-z))$; (iv) exponential loss $L_{exp}(z) = \exp(-z)$; (v) squared loss $L_{sq}(z) = (1 - z)^2$ (can be set to 0 for $z > 1$, just like hinge loss).

Review: Linear classification

- Two-class classifier "separates" instances in feature space:

$$f(x) = sign(w^T x + b)$$

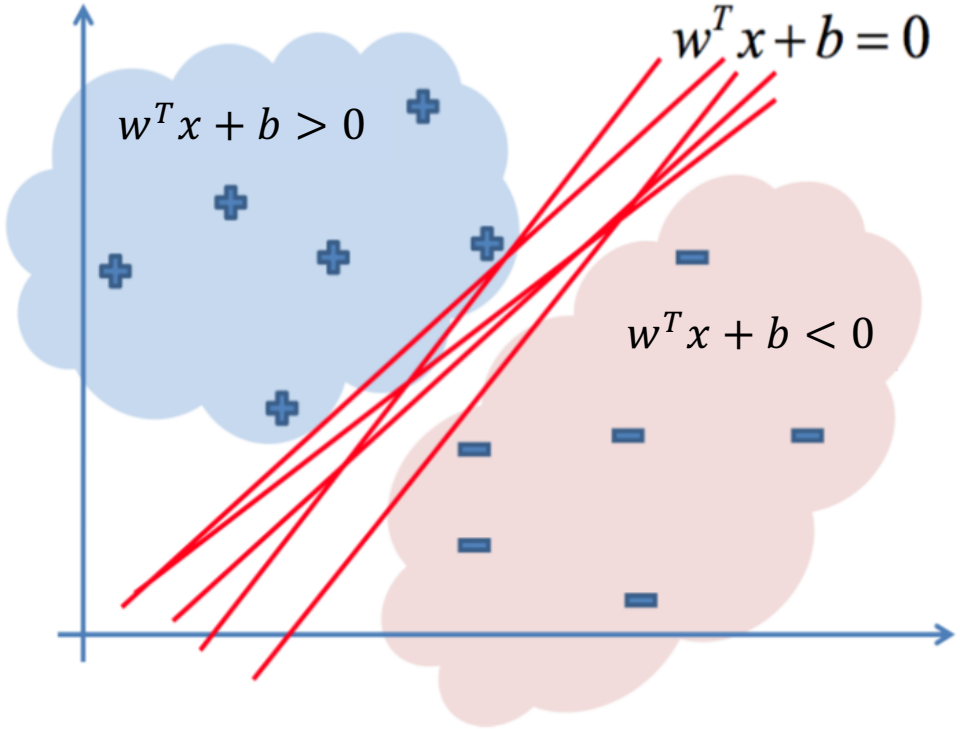


Issues in linear classification

- Define a decision boundary by a hyperplane in feature space
- A linear model can be used for classification

Issues in linear classification

- Many possible linear decision boundaries: which one to choose?

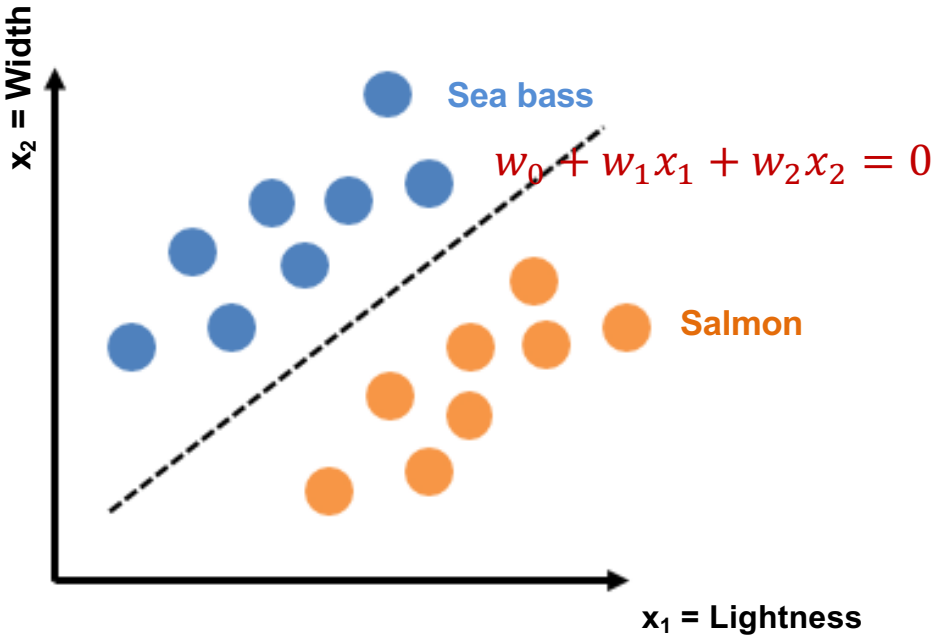


Linear classification: Perceptron

Perceptron: is an algorithm for binary classification that uses a linear prediction function

If we have two attributes/features of x_1 and x_2 then we can predict the target function $f(x)$ with:

$$f(\mathrm{x}) = \begin{cases} +1 & if \;\; w_0 + w_1 x_1 + w_2 x_2 > 0 \\ -1 & otherwise. \end{cases}$$



Linear classification: Perceptron

For a general case with n attributes:

$$f(\mathrm{x}) = \begin{cases} +1 & if \;\; w_0 + w_1 x_1 + \cdots + w_n x_n \geq 0 \\ -1 & otherwise. \end{cases}$$

If we add $x_0 = 1$ to the feature vector:

$$f(\mathrm{x}) = \begin{cases} +1 & if \;\; \sum_{i=0}^{n} w_i x_i \geq 0 \\ -1 & otherwise. \end{cases}$$

$\sum_{i=0}^{n} w_i x_i = w.\mathrm{x}$

Dot product: $a.b = a_1 b_1 + a_1 b_1 + \cdots + a_n b_n$

Perceptron learning

$$f(\mathrm{x}) = \begin{cases} +1 & if \;\; w.\mathrm{x} \geq 0 \\ -1 & otherwise. \end{cases}$$

$$\hat{y} = f(\mathrm{x}) = \mathrm{sgn}(w.\mathrm{x})$$

sgn is the sign function.

Now, we have to find a good set of weights using our training set $\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_m$ with labels $y_1, y_2, \ldots, y_m$

Please note that, here, x_j corresponds to observation j and is a vector of n features: $\mathrm{x}_j = [x_{j1}, \ldots, x_{jn}]$

Perceptron learning

The perceptron algorithm initializes all weights w_i to zero, and learns the weights using the following update rule:

$$w := w + \frac{1}{2}\left(y_j - f\left(\mathrm{x}_j\right)\right)\mathrm{x}_j$$

There are 4 cases:

$$y = +1, f(\mathrm{x}) = +1 \implies \left(y - f(\mathrm{x})\right) = 0$$
$$y = +1, f(\mathrm{x}) = -1 \implies \left(y - f(\mathrm{x})\right) = +2$$
$$y = -1, f(\mathrm{x}) = +1 \implies \left(y - f(\mathrm{x})\right) = -2$$
$$y = -1, f(\mathrm{x}) = -1 \implies \left(y - f(\mathrm{x})\right) = 0$$

Perceptron learning

w gets updated only if the prediction mismatches the actual class label (misclassification) and otherwise remains the same. Therefore, for misclassified instances we can write:

$$w := w + y_j x_j$$

Perceptron algorithm:

1. Initialize all weights w to zero
2. Iterate through the training data. For each training sample, classify the sample:
   a) If the prediction was correct, don't do anything
   b) If the prediction was wrong, modify the weights by using the update rules
3. Repeat step 2 some number of times

Perceptron training algorithm

Algorithm Perceptron(D) / perceptron training for linear classification

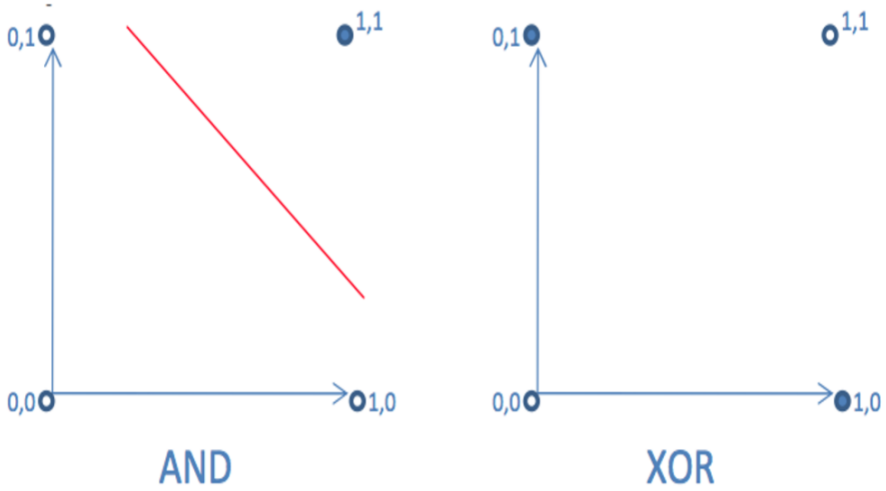
Input: labelled training data D in homogeneous coordinates

Output: weight vector w

$w \leftarrow 0$
$converged \leftarrow false$
while $converged = false$ **do**
    $converged \leftarrow true$
    for $i = 1$ to $|D|$ **do**
        if $y_i w.\mathrm{x}_i \leq 0$ **then**
            $w \leftarrow w + y_i \mathrm{x}_i$
            $converged \leftarrow false$
        end
    end
end

Issues in linear classification

- May not be possible to find a linear separating hyperplane



- Filled / empty circles are in / out of the target concept
- AND is linearly separable – but not XOR

Extending linear classification

- Linear classifiers can't model nonlinear class boundaries
- Simple trick to allow them to do that:
  - Nonlinear mapping: map attributes into new space consisting of combinations of attribute values
  - For example: all products with n factors that can be constructed from the attributes (feature construction or basis expansion)
- e.g., for 2 attributes, all products with $n = 3$ factors

$$y = w_1 x_1^3 + w_2 x_1^2 x_2 + w_3 x_1 x_2^2 + w_4 x_2^3$$

- y is predicted output for instances with two attributes x_1 and x_2

Two main problems

- Efficiency:
  - With 10 attributes and $n = 5$ more than 2000 coefficients (weights) have to be learned
  - Linear regression (with attribute selection) running time is cubic in the number of attributes
- Overfitting:
  - "Too nonlinear" – number of coefficients large relative to number of training instances
  - Curse of dimensionality applies ...

Duality (optimization)

- Essentially, an optimization approach has to be used to find the discriminative line
- Duality concept in optimization can help to view the problem from another perspective and make it simpler
- Sometimes solving the dual form is much easier. (Computational advantage)
- We will see, how using *dual form* and *kernel trick* simplify the computations.

Duality (optimization)

One way of thinking about duality is that when you have an optimization problem, we can construct another optimization problem which is called *dual problem* and is related to our original problem (*primal problem*) and can be useful in solving the primal problem.

In convex optimization problems, the optimal values of the primal and dual problems are equal under a constraint qualification condition.

Perceptron classifiers in dual form

Every time an example x_i is misclassified, add $y_i \mathrm{x}_i$ to the weight vector.

- After training has completed, each example has been misclassified zero or more times. Denoting this number as α_i for example x_i, the weight vector for m observations can be expressed as

$$w = \sum_{i=1}^{m} \alpha_i y_i \mathrm{x}_i$$

- In the dual instance-based view of linear classification we are learning instance weights α_i rather than feature weights w_j. An instance x is classified as

$$\hat{y} = f(\mathrm{x}) = \mathrm{sgn}(w.\mathrm{x})$$

$$\hat{y} = \mathrm{sgn}\left(\sum_{i=1}^{m} \alpha_i y_i (\mathrm{x}_i.\mathrm{x})\right)$$

Perceptron training in dual form

Algorithm Dual-Perceptron(D) / perceptron training in dual form

Input: labelled training data D in homogeneous coordinates

Output: coefficients α_i defining weight vector $W = \sum_{i=1}^{|D|} \alpha_i y_i x_i$

$\alpha_i \leftarrow 0$
$converged \leftarrow false$
while $converged = false$ **do**
    $converged \leftarrow true$
    for $i = 1\ to\ |D|$ **do**
        if $y_i \sum_{j=1}^{|D|} \alpha_j y_j \mathrm{x}_j \,.\, \mathrm{x}_i \leq 0$ **then**
            $\alpha_i \leftarrow \alpha_i + 1$
            $converged \leftarrow false$
        end
    end
end

Perceptron classifiers in dual form

- Using the dual form of perceptron, we estimate values of α_i instead of w

- During training, the only information needed about the training data is all pairwise dot products: the m-by-m matrix $G = XX^T$ containing these dot products is called the Gram matrix.

$$G(\mathrm{x}_1, \ldots, \mathrm{x}_n) = \begin{bmatrix} \mathrm{x}_1 . \mathrm{x}_1, & \mathrm{x}_1 . \mathrm{x}_2, & \ldots, & \mathrm{x}_1 . \mathrm{x}_m \\ \mathrm{x}_2 . \mathrm{x}_1, & \mathrm{x}_2 . \mathrm{x}_2, & \ldots, & \mathrm{x}_2 . \mathrm{x}_m \\ & & \cdot & \\ & & \cdot & \\ & & \cdot & \\ \mathrm{x}_m . \mathrm{x}_1, & \mathrm{x}_m . \mathrm{x}_2, & \ldots, & \mathrm{x}_m . \mathrm{x}_m \end{bmatrix}$$

Nonlinear dual perceptron

- We can use nonlinear mapping to map attributes into new space consisting of combinations of attribute values

$$\mathrm{x} \rightarrow \varphi(\mathrm{x})$$

- The perceptron decision will be:

$$\hat{y} = \mathrm{sign}\left(\sum_{i=1}^{m} \alpha_i y_i (\varphi(\mathrm{x}_i).\varphi(\mathrm{x}))\right)$$

- So the only thing we need is the dot product in the new feature space ($(\varphi(\mathrm{x}_i).\varphi(\mathrm{x}))$ or $\langle \varphi(\mathrm{x}_i), \varphi(\mathrm{x}) \rangle$)
- Why this matters?

The kernel trick

Let $\mathrm{x} = (x_1, x_2)$ and $\mathrm{x}' = (x_1', x_2')$ be two data points, and consider the following mapping to a three-dimensional feature space:

$$(x_1, x_2) \rightarrow (x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

(original feature space) $\mathcal{X} \longrightarrow \mathcal{Z}$ (new feature space)

The points in feature space corresponding to x and x' are

$$\mathrm{z} = \left(x_1^2, x_2^2, \sqrt{2}x_1x_2\right) \text{ and } \mathrm{z}' = (x_1'^2, x_2'^2, \sqrt{2}x_1'x_2')$$

The dot product of these two feature vectors is

$$\mathrm{z}.\mathrm{z}' = x_1^2\, x_1'^2 + x_2^2\, x_2'^2 + 2x_1x_1'x_2x_2' = (x_1x_1' + x_2x_2')^2 = (\mathrm{x}_1.\mathrm{x}_2)^2$$

The kernel trick

- By squaring the dot product in the original space we obtain the dot product in the new space without actually constructing the feature vectors! A function that calculates the dot product in feature space directly from the vectors in the original space is called a kernel – here the kernel is $K(\mathrm{x}_1, \mathrm{x}_2) = (\mathrm{x}_1 . \mathrm{x}_2)^2$.
- In this example order is 2, so the computational gain may not be very obvious, but if we aim for higher orders, for example 20, then we see more clearly the computational advantage

The kernel trick

- A valid kernel function is equivalent to a dot product in some space.
- The kernel trick helps to go to a high dimensional space without paying the price (!!), because if we find the right kernel, we do not need to explicitly map the features into the other high dimensional feature space.

Kernel function

Example: if $\mathrm{x} = (x_1, x_2)$ and $\mathrm{x}' = (x_1', x_2')$, is the following function a valid kernel? If so, what is the feature map function?

$$K(\mathrm{x}, \mathrm{x}') = (1 + \mathrm{x}.\mathrm{x}')^2$$

$$\mathrm{x}.\mathrm{x}' = x_1 x_1' + x_2 x_2'$$

$$(1 + \mathrm{x}.\mathrm{x}')^2 = 1 + x_1^2 x_1'^2 + x_2^2 x_2'^2 + 2x_1 x_1' + 2x_2 x'_2 + 2x_1 x'_1 x_2 x'_2$$

This is the dot product of:

$$\left(1, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x_2\right) \quad \text{and} \quad (1, x'^2_1, x'^2_2, \sqrt{2}x'_1, \sqrt{2}x'_2, \sqrt{2}x'_1 x'_2)$$

So this is a valid kernel.

The kernel trick

- The kernel trick avoids the explicit mapping from original feature space ($\mathcal{X}$) to another space ($\mathcal{Z}$).
- For $\{x,x'\} \in \mathcal{X}$, certain function $K(x,x')$ can be expressed as a dot product in another space and K is called *kernel* or *kernel function*.
- If $\varphi(x)$ is the input x in the new (higher dimensional) feature space, then computation becomes much simpler, if we can have kernel function that:

$$K(x,x') = \varphi(x).\varphi(x')$$

The kernel trick

- A kernel function is a *similarity* function that corresponds to a dot product in some expanded feature space
- Some very useful kernels in machine learning are polynomial kernel and radial basis function kernel (RBF kernel)
- Polynomial kernel is defined as:

$$K(\mathrm{x},\mathrm{x}') = (\mathrm{x}.\mathrm{x}' + c)^q$$

- RBF kernel is defined as:

$$K(\mathrm{x},\mathrm{x}') = \exp\left(-\frac{||\mathrm{x}-\mathrm{x}'||^2}{2\sigma^2}\right)$$

Using Taylor expansion, it can be shown that RBF kernel is equivalent of mapping features into infinite dimensions

Nonlinear dual perceptron

Using kernel trick, the nonlinear perceptron:

$$\hat{y} = \text{sign}\left(\sum_{i=1}^{m} \alpha_i y_i (\varphi(\mathrm{x}_i).\varphi(\mathrm{x}))\right)$$

can be solved using the dual form and the kernel as follow:

$$\hat{y} = \text{sign}\left(\sum_{i=1}^{m} \alpha_i y_i K(\mathrm{x}_i, \mathrm{x})\right)$$

The same algorithm as in linear perceptron can be used, but the $\mathrm{x}_j.\mathrm{x}_i$ has to be replaced with $K(\mathrm{x}_j, \mathrm{x}_i)$