



Never Stand Still

Kernel Methods

COMP9417 Machine Learning & Data Mining

Term 3, 2019

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 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

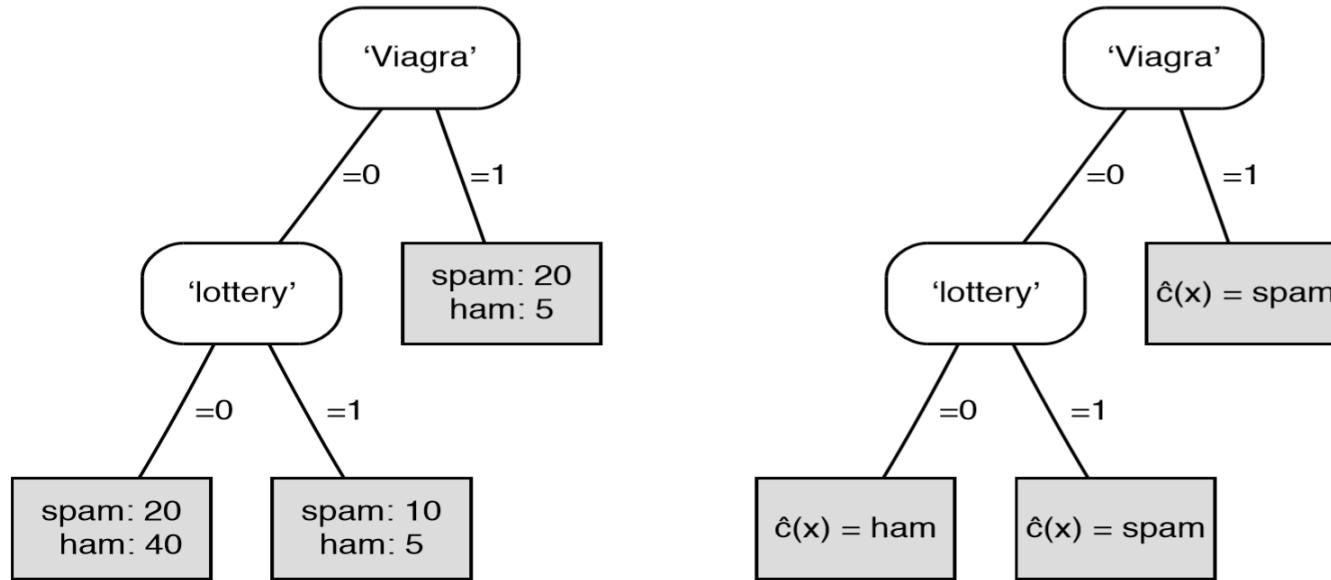
Predictive machine learning scenarios

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

Classification

- A classifier is a mapping $\hat{c}: X \rightarrow C$, where $C = \{C_1, C_2, \dots, C_k\}$ 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



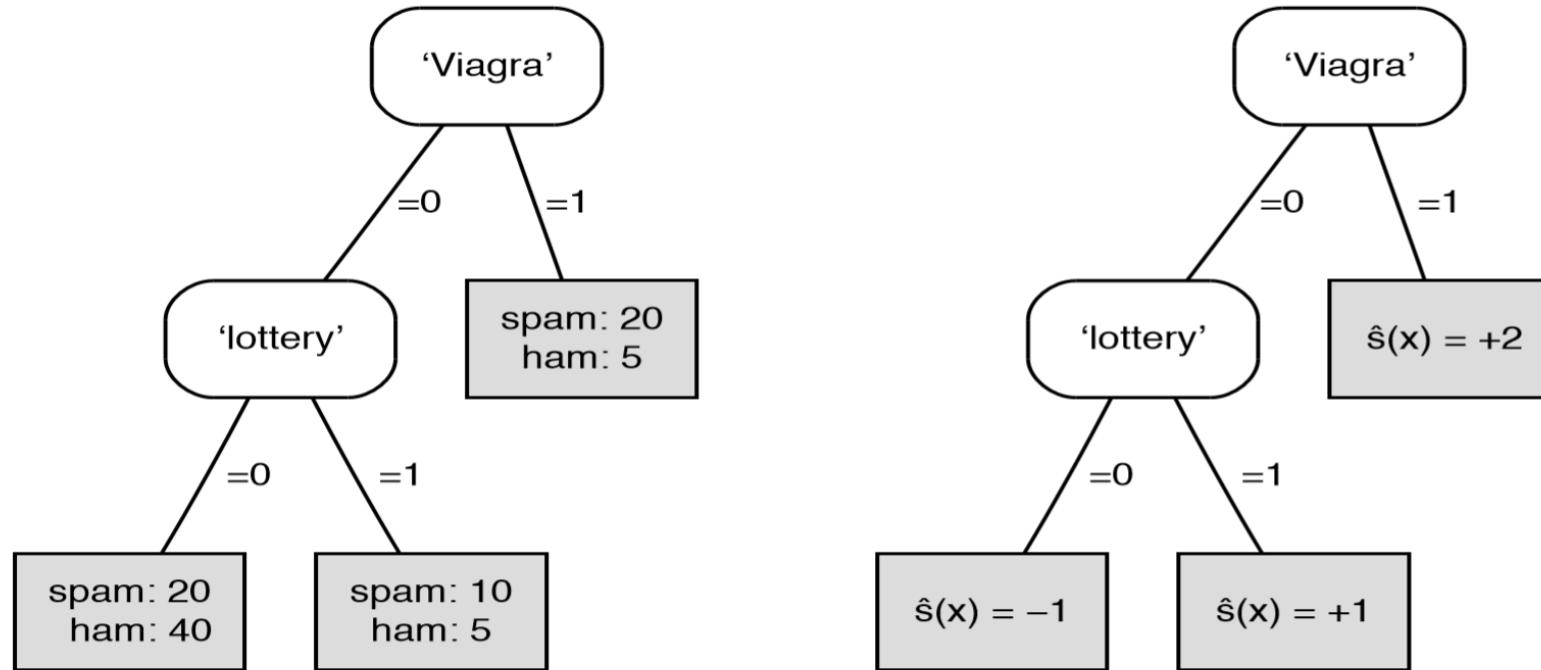
(left) A tree with the training set class distribution in the leaves.

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

Scoring classifier

- A scoring classifier is a mapping $\hat{S}: \mathcal{X} \rightarrow \mathbb{R}^k$ i.e., a mapping from the instance space to a k –vector of real numbers.
- The boldface notation indicates that a scoring classifier outputs a vector $\hat{S}(x) = (\hat{s}_1(x), \dots, \hat{s}_k(x))$ 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.

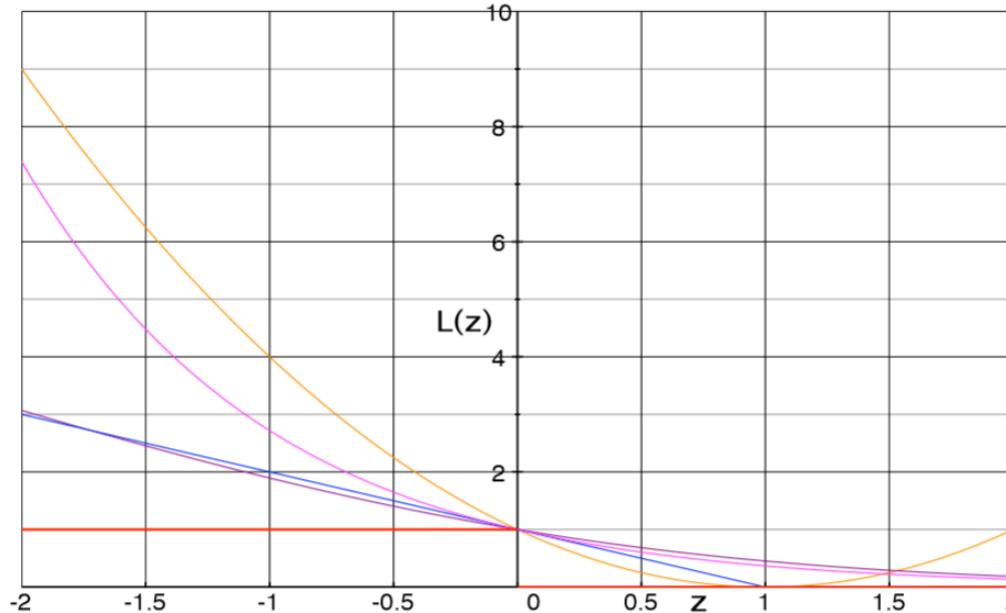
Margins and loss functions

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

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 T_e is $\frac{1}{|T_e|} \sum_{x \in T_e} L(z(x))$

Loss functions

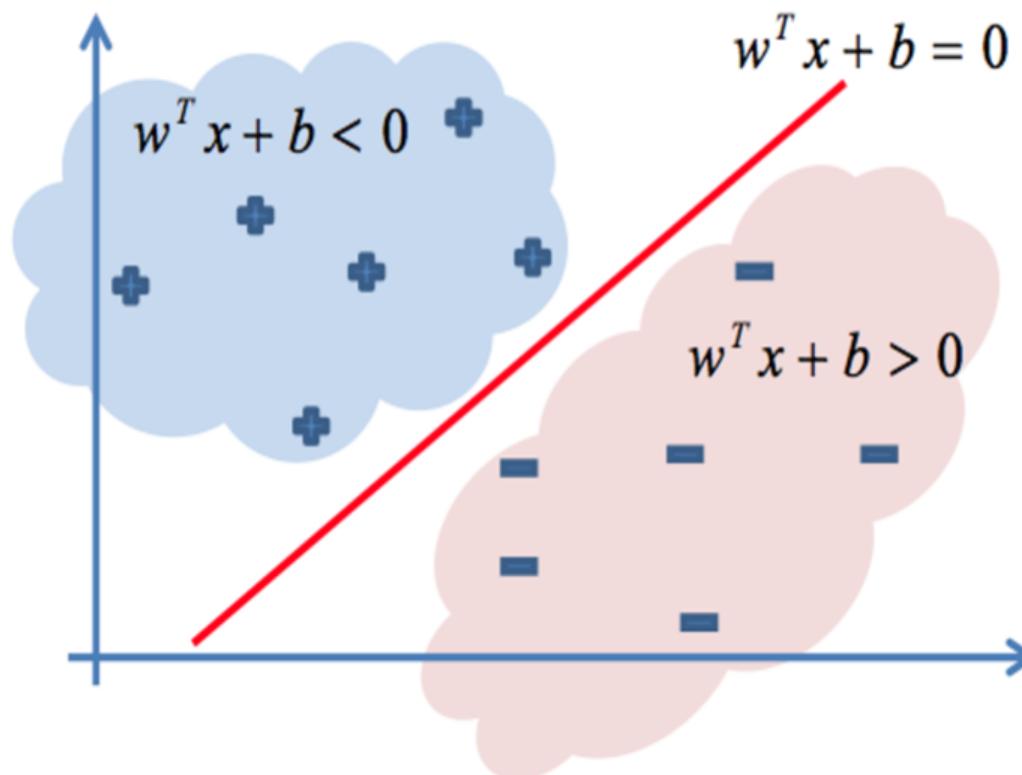


From bottom-left: (i) 0–1 loss $L_{01}(z) = 1$ if $z \leq 0$, and $L_{01}(z) = 0$ if $z > 0$; (ii) hinge loss $L_h(z) = (1 - z)$ if $z \leq 1$, and $L_h(z) = 0$ if $z > 1$; (iii) logistic loss $L_{log}(z) = \log_2(1 + \exp(-z))$; (iv) exponential loss $L_{exp}(z) = \exp(-z)$; (v) squared loss $L_{sq}(z)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

Review: Linear classification

- Two-class classifier “separates” instances in feature space:

$$f(x) = \text{sign}(w^T x + b)$$

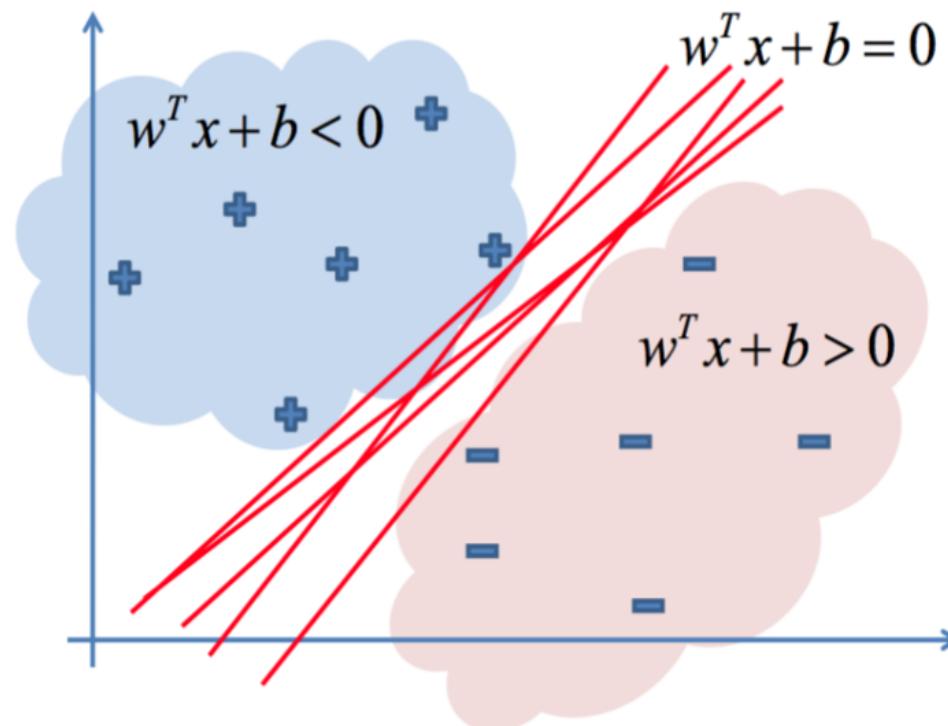


Issues in linear classification

- 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?

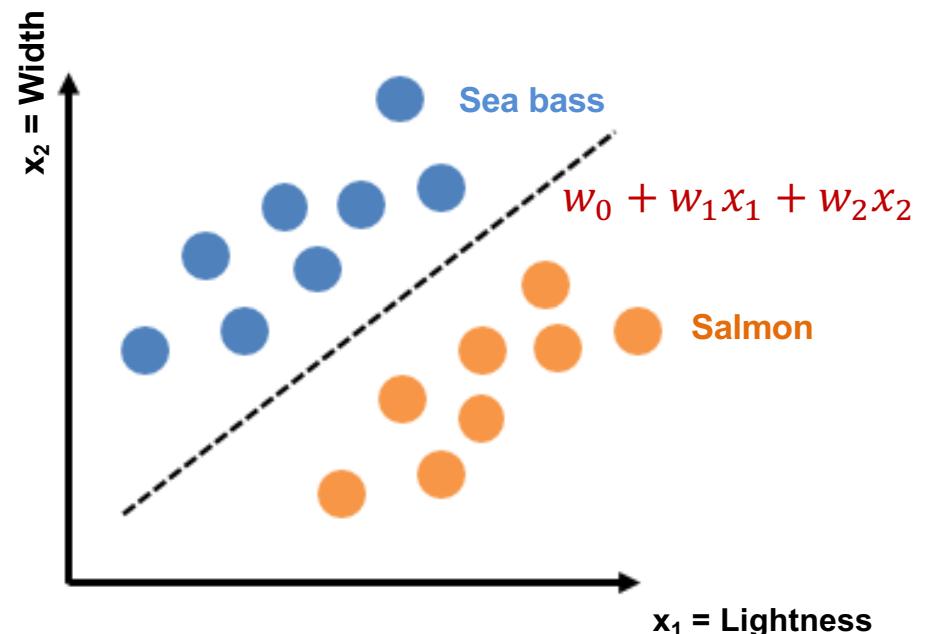


Linear classification: Perceptron

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:

$$f(x) = \begin{cases} +1 & \text{if } w_0 + w_1x_1 + w_2x_2 > 0 \\ -1 & \text{otherwise.} \end{cases}$$



Linear classification: Perceptron

For a general case with n attributes:

$$f(\mathbf{x}) = \begin{cases} +1 & \text{if } w_0 + w_1x_1 + \cdots + w_nx_n \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

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

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

$$\sum_{i=0}^n w_i x_i = \mathbf{w} \cdot \mathbf{x}$$

Dot product: $\mathbf{a} \cdot \mathbf{b} = a_1b_1 + a_2b_2 + \cdots + a_nb_n$

Perceptron learning

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

$$\hat{y} = f(\mathbf{x}) = \operatorname{sgn}(\mathbf{w} \cdot \mathbf{x})$$

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

Please note that, here, the subscripts correspond to the sample number and not features.

Perceptron learning

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

$$w := w + \frac{1}{2} (y_j - f(x_j)) x_j$$

There are 4 cases:

$$y = +1, f(x) = +1 \Rightarrow (y - f(x)) = 0$$

$$y = +1, f(x) = -1 \Rightarrow (y - f(x)) = +2$$

$$y = -1, f(x) = +1 \Rightarrow (y - f(x)) = -2$$

$$y = -1, f(x) = -1 \Rightarrow (y - f(x)) = 0$$

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 ← 0
converged ← false
while converged = false do
    converged ← true
    for i = 1 to |D| do
        if  $y_i w \cdot x_i \leq 0$  then
            w ← w +  $y_i x_i$ 
            converged ← false
        end
    end
end
```

Issues in linear classification

Different classification learning algorithms use different criteria:

- Basic linear classifier finds class means (centroids), joins them by a straight line, and its perpendicular bisector is the separating hyperplane
- Perceptron training uses iterative reweighting (gradient descent)
- Perceptron may find different models depending on starting conditions

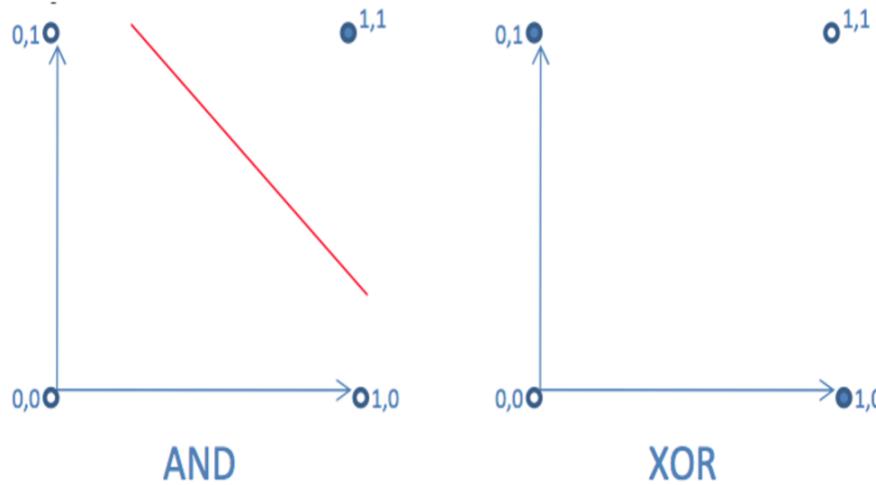
Issues in linear classification

Is there an optimal linear classification learning method ?

- one approach is to define the empirical risk, or error on the data
- the maximum margin separating hyperplane
- unique solution
- minimizes empirical risk
- is there a way to trade-off minimizing risk with model complexity?
- answer: yes, under Vapnik's framework for statistical learning
 - structural risk minimization

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 $n = 5$ have to learn more than 2000 coefficients (weights)
 - 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 $y_i x_i$ 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 can be expressed as

$$w = \sum_{i=1}^m \alpha_i y_i x_i$$

- 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

$$\hat{y} = \text{sign} \left(\sum_{i=1}^m \alpha_i y_i (x_i \cdot x) \right)$$

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  $y_i \sum_{j=1}^{|D|} \alpha_j y_j x_j \cdot x_i \leq 0$  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 n -by- n matrix $G = XX^T$ containing these dot products is called the **Gram matrix**.

$$G(x_1, \dots, x_n) = \begin{bmatrix} x_1 \cdot x_1, & x_1 \cdot x_2, & \dots, & x_1 \cdot x_n \\ x_2 \cdot x_1, & x_2 \cdot x_2, & \dots, & x_2 \cdot x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_n \cdot x_1, & x_n \cdot x_2, & \dots, & x_n \cdot x_n \end{bmatrix}$$

Nonlinear dual perceptron

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

$$x \rightarrow \varphi(x)$$

- Then our perceptron will be:

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

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

The kernel trick

Let $x = (x_1, x_2)$ and $x' = (x'_1, x'_2)$ 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) $\mathcal{X} \rightarrow \mathcal{Z}$ (new feature space)

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

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

The dot product of these two feature vectors is

$$z \cdot z' = x_1^2 x_1'^2 + x_2^2 x_2'^2 + 2x_1 x_1' x_2 x_2' = (x_1 x_1' + x_2 x_2')^2 = (x_1 \cdot x_2)^2$$

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 $K(x_1, x_2) = (x_1 \cdot x_2)^2$.
- 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 $x = (x_1, x_2)$ and $x' = (x'_1, x'_2)$, is the following function a valid kernel? If so, what is the feature map function?

$$K(x, x') = (1 + x \cdot x')^2$$

$$\begin{aligned} x \cdot x' &= \\ (1 + x \cdot x')^2 &= 1 + x_1^2 x'^2_1 + x_2^2 x'^2_2 + 2x_1 x'_1 + 2x_2 x'_2 + 2x_1 x'_1 x_2 x'_2 \end{aligned}$$

This is the dot product of:

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

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 an inner 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:

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

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:

$$K(x, x') = (x \cdot x' + c)^q$$

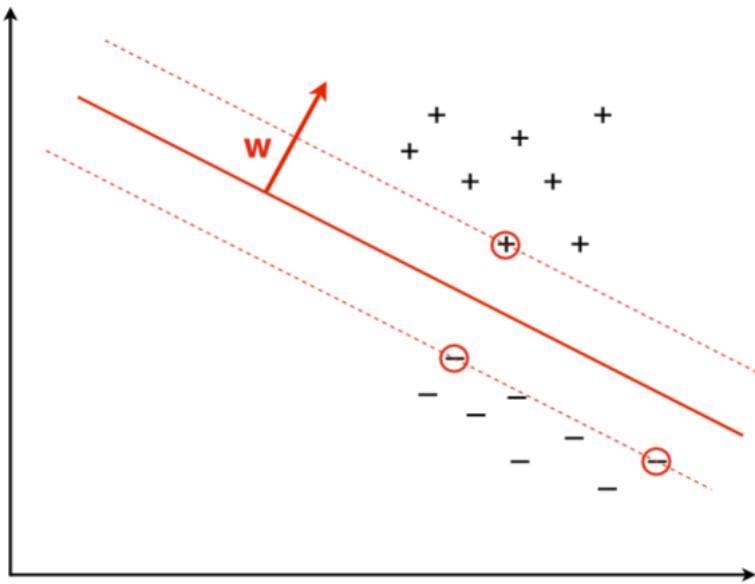
- RBF kernel is defined as:

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

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

Support Vector Machine

Support vector machine

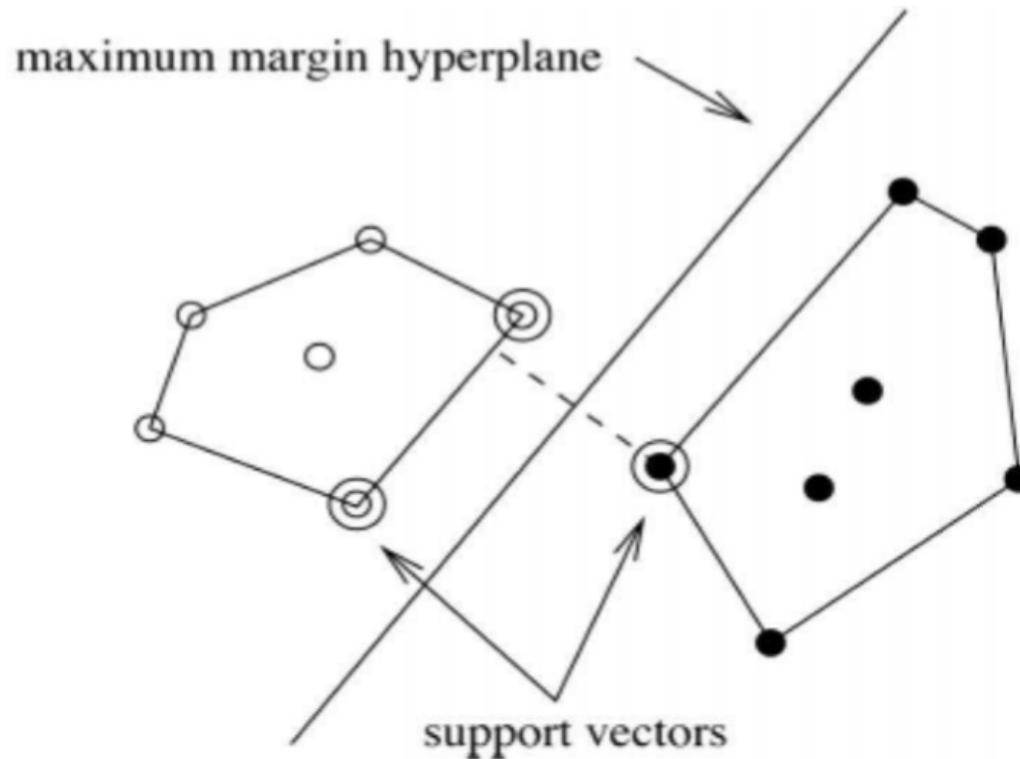


The decision boundary learned by a support vector machine maximizes the margin, which is indicated by the dotted lines. The circled data points are the support vectors.

Support vector machine

- Support vector machines (machine \equiv algorithm) learn linear classifiers
- Can avoid overfitting – learn a form of decision boundary called the maximum margin hyperplane
- Fast for mappings to nonlinear spaces
 - employ a mathematical trick to avoid the actual creation of new “pseudo-attributes” in transformed instance space
 - i.e. the nonlinear space is created implicitly

Training a support vector machine (SVM)



“Convex hull” of instances in each class is tightest enclosing convex polygon

Training a support vector machine

- For a linearly separable two-class data set convex hulls do not overlap
- Learning problem: fit maximum margin hyperplane, i.e. a kind of linear model
- For a linearly separable two-class data set the maximum margin hyperplane is the classification surface which
 - correctly classifies all examples in the data set
 - has the greatest separation between classes

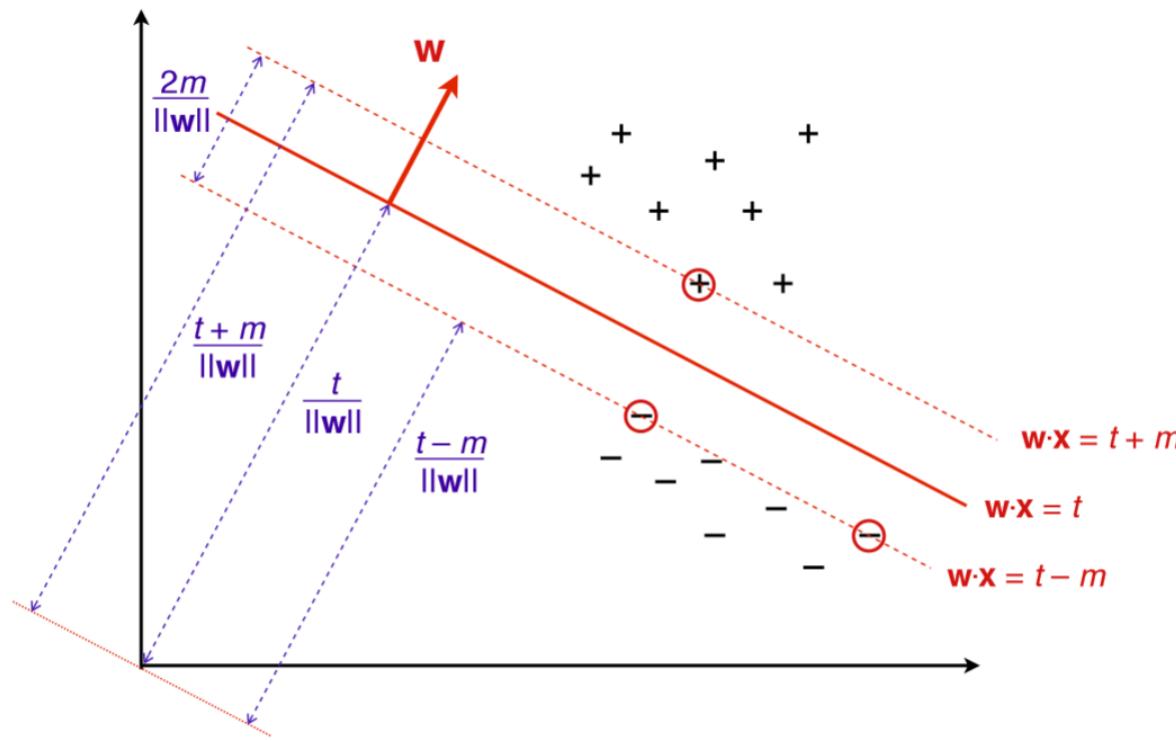
Training a support vector machine

- maximum margin hyperplane is orthogonal to shortest line connecting convex hulls, intersects with it halfway
- The more “separated” the classes, the larger the margin, the better the generalization
- Instances closest to maximum margin hyperplane are support vectors
- Important observation: support vectors define maximum margin hyperplane!
 - All other instances can be deleted without changing position and orientation of the hyperplane!

Finding support vectors

- Determining parameters is a constrained quadratic optimization problem
- standard algorithms, or
- special-purpose algorithms are faster, e.g. Platt's sequential minimal optimization (SMO), or LibSVM
- Note: all this assumes separable data!

Support vector machine



The geometry of a support vector classifier. The circled data points are the support vectors, which are the training examples nearest to the decision boundary. The support vector machine finds the decision boundary that maximizes the margin $m/\|w\|$.

Maximizing the margin

Since we are free to rescale t , $\|W\|$ and m , it is customary to choose $m = 1$. Maximizing the margin then corresponds to minimizing $\|w\|$ or, more conveniently, $\frac{1}{2} \|W\|^2$, provided of course that none of the training points fall inside the margin.

This leads to a quadratic, constrained optimization problem:

$$w^*, t^* = \underset{w,t}{\operatorname{argmin}} \frac{1}{2} \|w\|^2, \quad \text{subject to } y_i(w \cdot x_i - t) \geq 1, \quad 1 \leq i \leq n$$

Using the method of Lagrange multipliers, the dual form of this problem can be derived.

Lagrangian multipliers

- Constrained optimization problems are generally expressed as:

$$\min_{x_1, \dots, x_n} J = f(x_1, \dots, x_n)$$

Subject to:

$$g_1(x_1, \dots, x_n) \leq 0, g_2(x_1, \dots, x_n) \leq 0, \dots, g_k(x_1, \dots, x_n) \leq 0$$

- Lagrange multiplier methods involve the modification of the objective function through the addition of terms that describe the constraints. The objective function $J = f(x)$ is augmented by the constraint equations through a set of non-negative multiplicative Lagrange multipliers, $\alpha_j \geq 0$ and it is called the dual Lagrangian:

$$\Lambda(x_1, \dots, x_n, \alpha_1, \dots, \alpha_m) = f(x_1, \dots, x_n) + \sum_{j=1}^k \alpha_j g_j(x_1, \dots, x_n)$$

Deriving the dual problem

Adding the constraints with multipliers α_i for each training example gives the Lagrange function:

$$\begin{aligned}\Lambda(W, t, \alpha_1, \dots, \alpha_n) &= \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i (y_i(w \cdot x_i - t) - 1) \\ &= \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i y_i (w \cdot x_i) + \sum_{i=1}^n \alpha_i y_i t + \sum_{i=1}^n \alpha_i \\ &= \frac{1}{2} w \cdot w - w \cdot \left(\sum_{i=1}^n \alpha_i y_i x_i \right) + t \left(\sum_{i=1}^n \alpha_i y_i \right) + \sum_{i=1}^n \alpha_i\end{aligned}$$

Deriving the dual problem

- By taking the partial derivative of the Lagrange function with respect to t and setting it to 0 we find:

$$\sum_{i=1}^n \alpha_i y_i = 0$$

- Similarly, by taking the partial derivative of the Lagrange function with respect to w and setting to 0 we obtain:

$$W = \sum_{i=1}^n \alpha_i y_i x_i$$

- the same expression as we derived for the perceptron.

Deriving the dual problem

- For the perceptron, the instance weights α_i are non-negative integers denoting the number of times an example has been misclassified in training. For a support vector machine, the α_i are non-negative reals.
- What they have in common is that, if $\alpha_i = 0$ for a particular example x_i , that example could be removed from the training set without affecting the learned decision boundary. In the case of support vector machines this means that $\alpha_i > 0$ only for the support vectors: the training examples nearest to the decision boundary.

Deriving the dual problem

- These expressions allow us to eliminate w and t and lead to the dual Lagrangian

$$\begin{aligned}\Lambda(\alpha_1, \dots, \alpha_n) &= -\frac{1}{2} \left(\sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \right) \cdot \left(\sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \right) + \sum_{i=1}^n \alpha_i \\ &= -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^n \alpha_i\end{aligned}$$

SVM in dual form

- The dual optimization problem for support vector machines is to maximize the dual Lagrangian under positivity constraints and one equality constraint:

$$\alpha_1^*, \dots, \alpha_n^* = \underset{\alpha_1, \dots, \alpha_n}{\operatorname{argmax}} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^n \alpha_i$$

$$\text{subject to } \alpha_i > 0, 1 \leq i \leq n, \sum_{i=1}^n \alpha_i y_i = 0$$