NEURAL NETWORKS AND SUPPORT VECTOR MACHINES

Gedi Lukšys
Applied Data Science 2
April 22, 2024

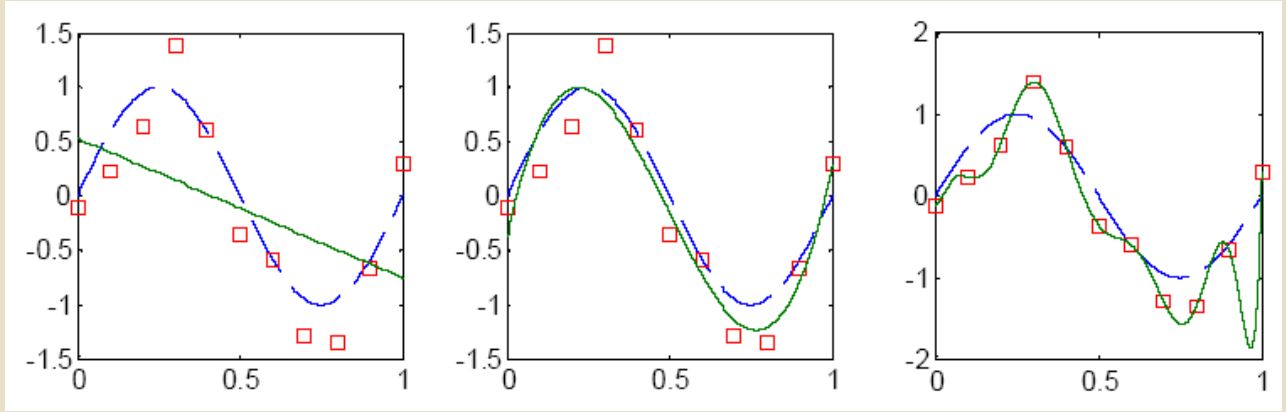
Outline

- Data & model preparation and evaluation
- Perceptron
- Multilayer perceptron (MLP = ANN)
- Support vector machines (SVM)

Learning objective: understand the main principles of different supervised learning methods and differences between them

Bias-variance dilemma and features

- Imagine fitting sine data with a polynomial model: M = 3 works best. Why?
- It has sufficient flexibility but does not overfit the training data
- Best model size can be determined with cross-validation

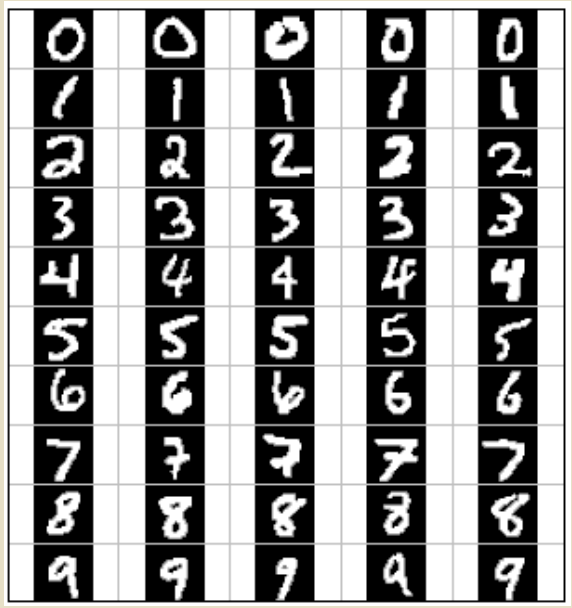


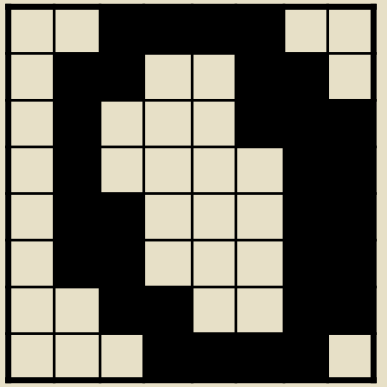
H.Jaeger, *'Machine Learning'*, Bremen, 2003

Generally, more parameters need more data points to avoid overfitting. However, higher dimensional data lead to more parameters

What if the number of samples is limited?

- Use not the raw data, but features!





http://w3.impa.br/~lhf/sib2003/p023/img2.gif

Model setup & training for classification

- For classification tasks the output is binary (0/1) instead of continuous, therefore it becomes inefficient to let the outputs take much smaller or larger values than that...

Solution: use an **activation function** g! Usually it's a sigmoid/logistic function

- Now y = g(f(x))

Why?

- Takes care of bounds
- Creates a nonlinearity

Data preparation

- **Random** division into training and validation sets
  - If there is any structure in data, it should be equally reflected in training and validation sets – e.g. equal/proportional number of samples from all classes
- If there are enough samples, one division is sufficient (often 70-30 ratio)
  - If not, leave-one-out (or leave-some-out) cross validation needs to be performed.
- For classification, (R)MSE and classification errors are different quantities! (although closely related)
- For **multiple classes**, the approach depends on method: for neural network-based models, can have multiple output "neurons" and pick the most active; for SVMs can train separating one class vs. the rest or in pairs.

Chance levels / baseline performance

- Simplest case in classification: **1/(Number of classes)** !
- If classes have different sizes in a sense that the likelihood to encounter different classes is substantially different, then the chance level (for a 2-class classification) is p(larger class).
- Only classification accuracy *minus* chance level counts!
- E.g. if we want to predict individual's recognition memory (recognized an item or not), with average performance 81%, our classifier performing at 82% would be poor and 75% would be worse than trivial.
- Sometimes (e.g. for regression) calculation of chance levels can be tricky: in that case use **random shuffling**

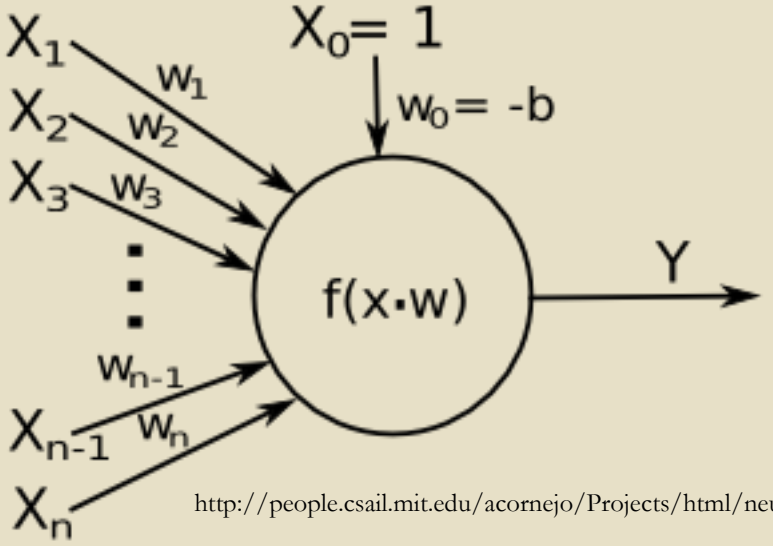
The (single layer) perceptron

- The simplest neural network – a linear classifier

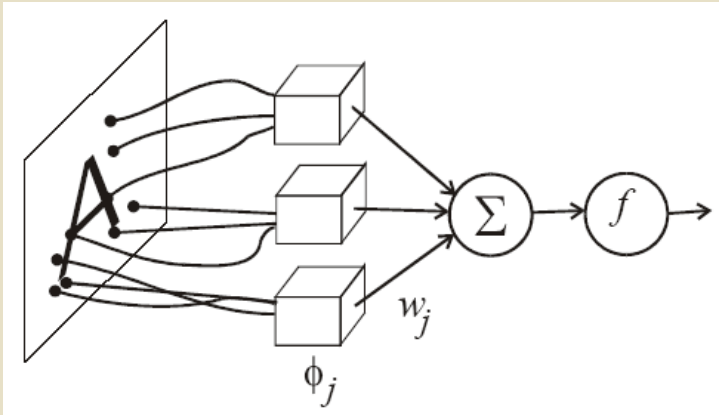
$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

  - x – input vector
  - w – weight vector
  - b – bias term

- Name comes from early attempts to model perceptual abilities *(e.g. character recognition)*



http://people.csail.mit.edu/acornejo/Projects/html/neuralnet.htm



Training the perceptron

- Compute output (for each sample j): $y^{(j)} = f\left(\vec{w} \cdot \vec{x}^{(j)}\right) = f\left(w_0 + w_1 x_1^{(j)} + w_2 x_2^{(j)} + \ldots + w_m x_m^{(j)}\right)$

- Compute error and its derivatives w.r.t. weights

$$MSE = \frac{1}{2n}\sum_{j=1}^{n}\left(d^{(j)} - y^{(j)}\right)^2 = \frac{1}{2n}\sum_{j=1}^{n}\left(d^{(j)} - f\left(\vec{w} \cdot \vec{x}^{(j)}\right)\right)^2$$

($d^{(j)}$ – correct output of sample j)

$$\partial MSE / \partial w_i = \frac{1}{2n}\sum_{j=1}^{n} 2\left(d^{(j)} - y^{(j)}\right) \cdot (-1) \cdot f' \cdot \left(x_i^{(j)}\right) = -\frac{1}{n}\sum_{j=1}^{n}\left(d^{(j)} - y^{(j)}\right) \cdot f' \cdot x_i^{(j)}$$

- Adapt each weight w_i as follows:

$$\Delta w_i = -\alpha \, \partial MSE / \partial w_i = \frac{\alpha}{n}\sum_{j=1}^{n}\left(d^{(j)} - y^{(j)}\right) \cdot f' \cdot x_i^{(j)}$$

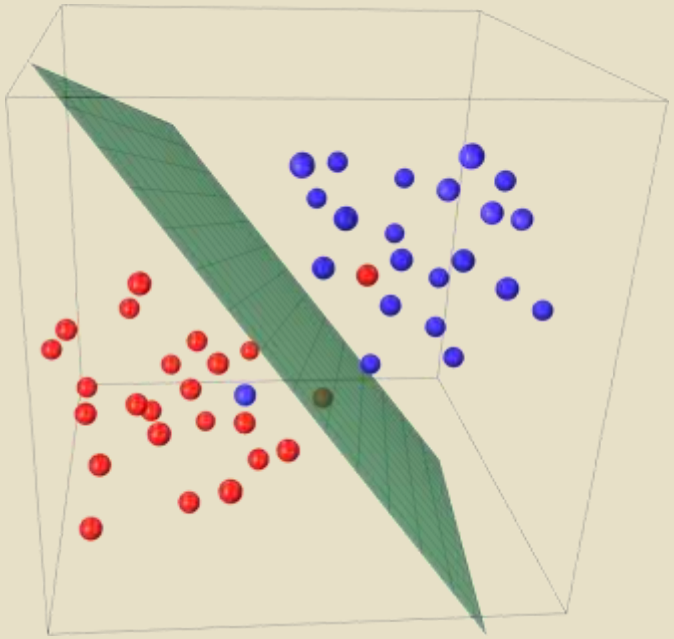
(α – learning rate)

- Stop when the error doesn't go down anymore

Batch vs. **online** training

When can perceptron work well?

It converges to 0 error when the dataset is linearly separable (i.e. there exists a *separating hyperplane*)

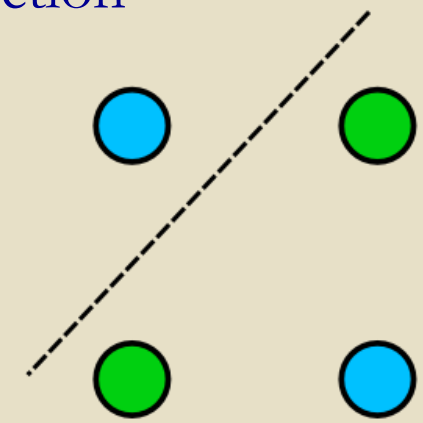


http://www.bwaldvogel.de/liblinear-java/svm3d_big.png

It cannot converge to 0 error (leading to inaccurate predictions or misclassifications) for datasets that are not linearly separable (as are most of real-life data...)

- The most famous toy example:

  XOR function

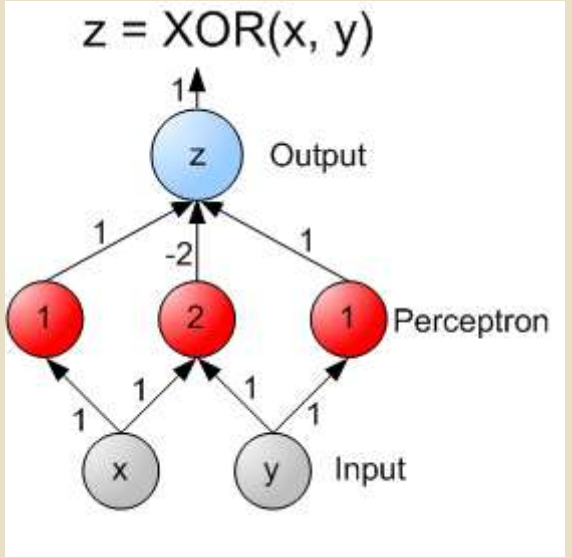


http://people.csail.mit.edu/acornejo/Projects/html/neuralnet.htm

How to separate the non-linearly separable datasets?

Introduce at least 1 hidden layer with non-linear activation functions! (for example, a simple network with step activation function in the hidden layer can solve the XOR problem)

$\Rightarrow$ Multilayer perceptrons

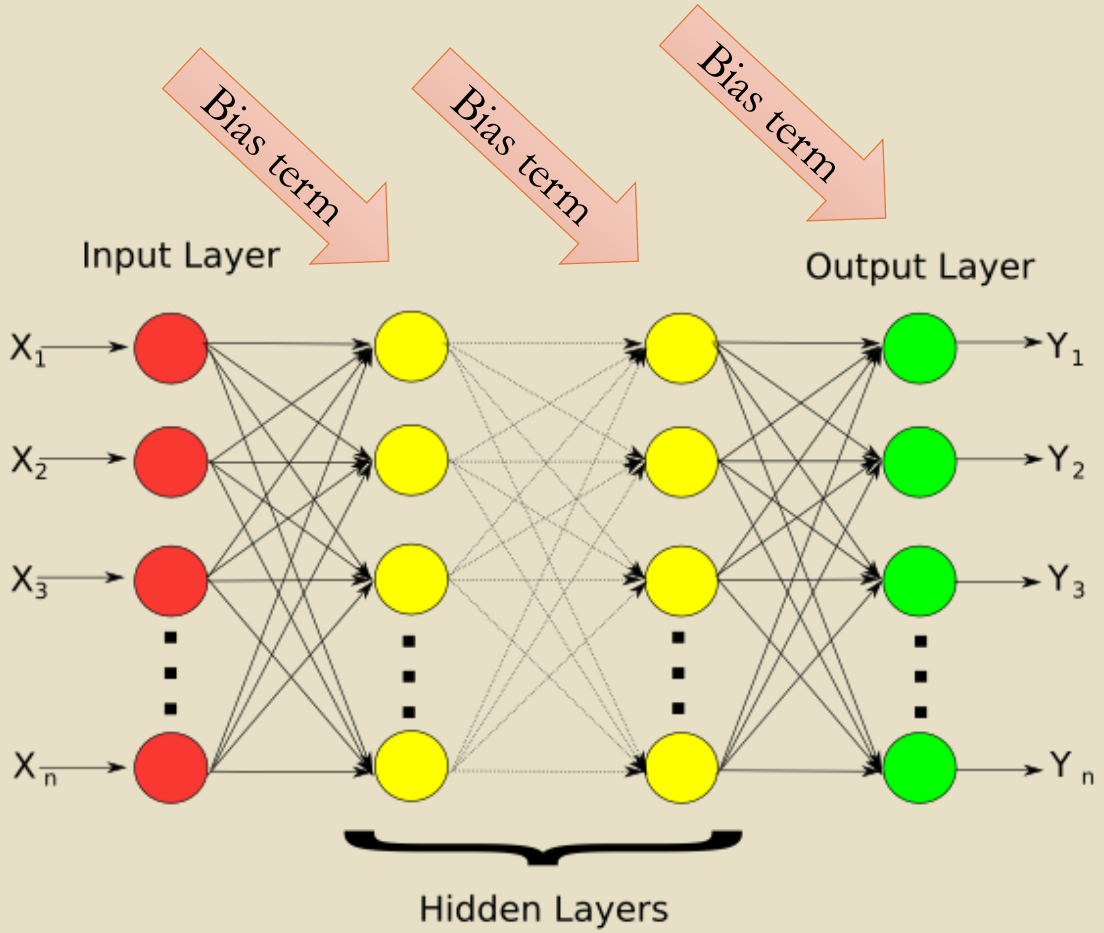


- In fact, it has been shown that a neural network with a single hidden layer and a certain (e.g. sigmoid) activation function can approximate any continuous function! (not necessarily in a practically useful way)

Architecture of multilayer perceptrons (MLPs)

Also known as **A**rtificial **N**eural **N**etworks

- All-to-all connections between the neighboring layers
- Usually 1 hidden layer
  - Many layers with special structure and learning rules: **deep learning!**
- Usually sigmoid (*tanh*) activation functions in the hidden layer(s), more variety for the output layer



http://people.csail.mit.edu/acornejo/Projects/html/neuralnet.htm

How to train this beast? The same approach!

- First, obviously we need to calculate the activations of the network („*forward pass*“) as well as the mean squared error(s) in the output layer
- Use the same derivative based *steepest gradient descent* rule as for the perceptron:
  - $\Delta w = -\alpha\, \delta(MSE)/\delta w$

  where w are the *hidden→output* layer weights w_{kj} & *input→hidden* layer weights w_{ji}.

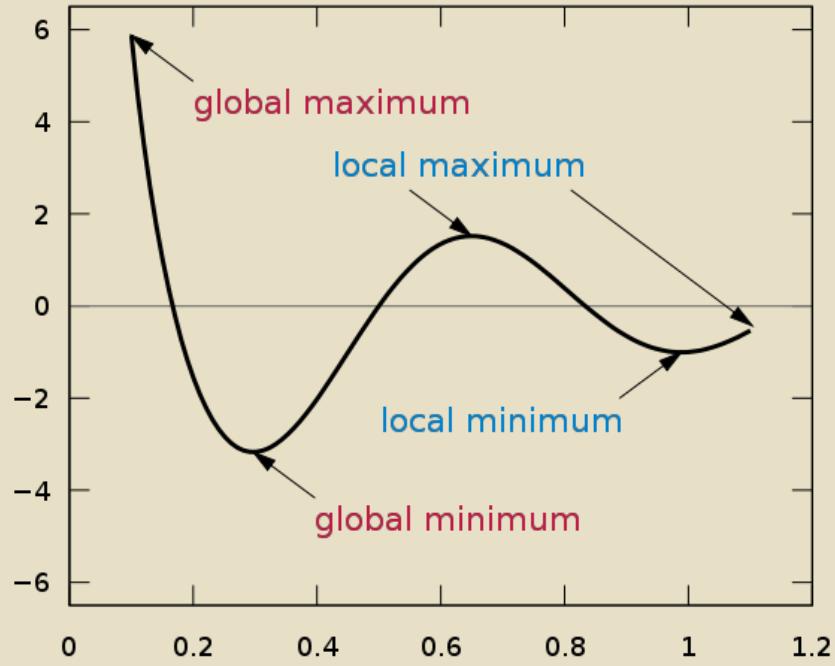
- Problem: how to calculate the derivatives?
  - Use the chain rule: $\delta A/\delta D = \delta A/\delta B \times \delta B/\delta C \times \delta C/\delta D$
  - Use the result of the N^{th} layer to calculate derivatives for the $(N-1)^{th}$ layer: backpropagation!
  - „Backward pass“: update the weights

- 1 forward and 1 backward pass constitute **an epoch**

Practical aspects of MLP training

- MLPs tend to get stuck in local minima that can be far from the global minimum. What to do?
  - Use momentum! $\Delta w(t) = -\alpha\, \delta(MSE)/\delta w + \mu\, \Delta w(t-1)$
  - Experiment with learning rates α to ensure stable yet effective training (e.g. 0.001, 0.003, 0.01, 0.03, 0.1)

- **When to stop training ?**
  - Change in MSE smaller than a certain threshold
  - A predefined number of steps
  - Continue as long as validation error doesn‘t go up



Data preprocessing & initialization

- It‘s necessary to normalize all input (and output) data before training so that
  - Some inputs don‘t initially have disproportionate influence
  - Output ranges become practical for training
- How to deal with the test data?
  - Normalize using means and standard deviations from the training data
- Weight initialization – random but from a narrow interval (like 0..1 or -1..1, to allow stability & flexibility)

Do we need to write code for all of this???

- **Mostly not** – machine learning packages exist in R, Matlab, Python, but knowing these details is important for appropriate usage!

Classification with optimal margin

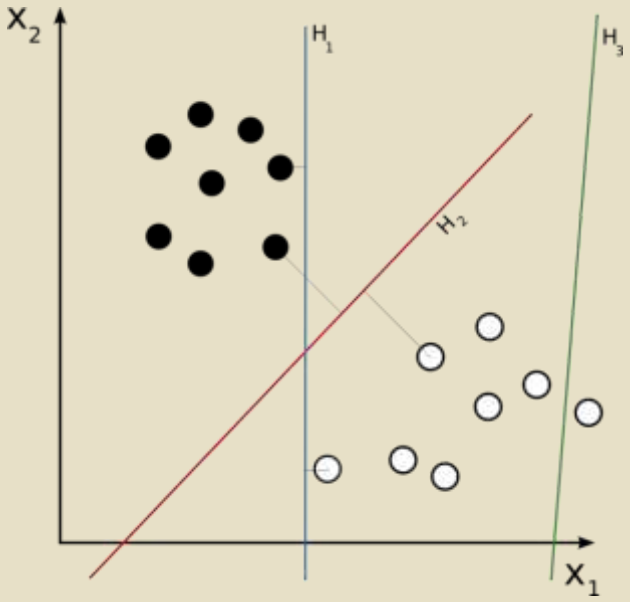
- Perceptron,

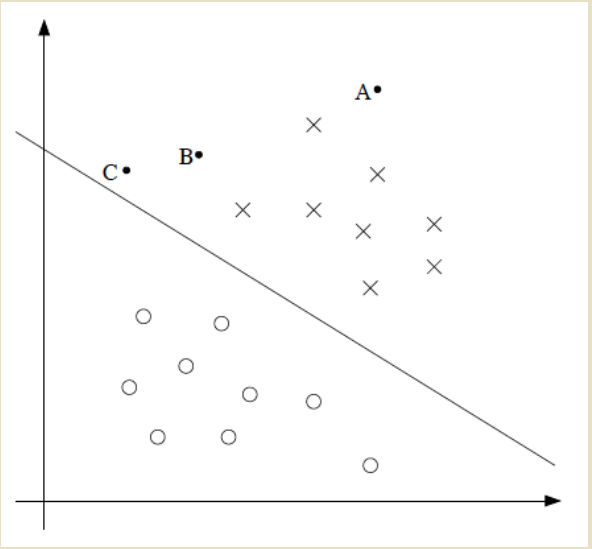
$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

can find a line (or hyperplane) for perfect classification in linearly separable datasets.

- But no guarantees regarding the test data!

- Intuitively a classifier with the largest margin from the nearest data points should work better. **Why?**



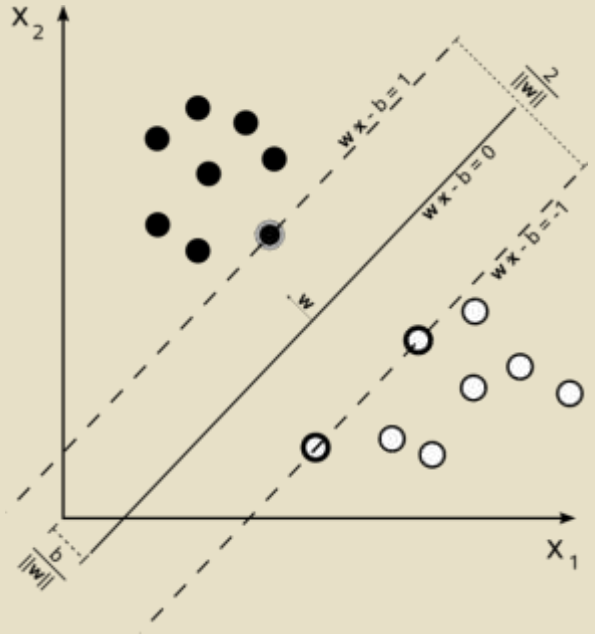


Towards support vector machines (SVMs)…

This is a difficult but solvable quadratic optimization problem

The solution – so-called Lagrangian – has 2 properties:

$$\mathcal{L}(w,b,\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)}.$$



◦ **It's dependent only on data points nearest to the boundary, "support vectors"** (how many?)

$\alpha_i > 0$ for data points at the margin („support vectors") and $\alpha_i = 0$ for all other data points

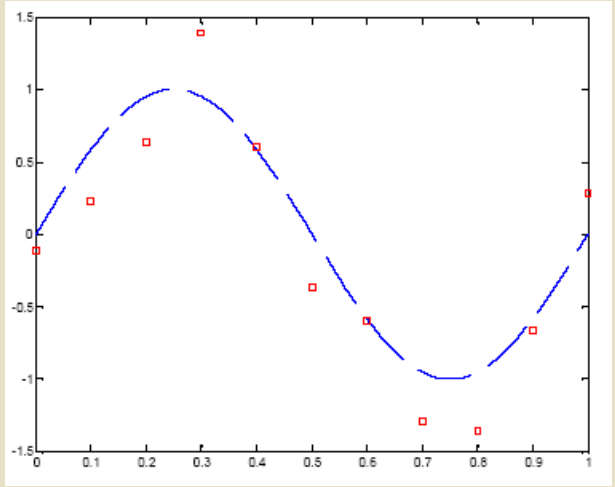
◦ **It operates only on dot products of data points:** $(x^{(i)})^T x^{(j)} = x^{(i)}_1 x^{(j)}_1 + \ldots + x^{(i)}_n x^{(j)}_n$

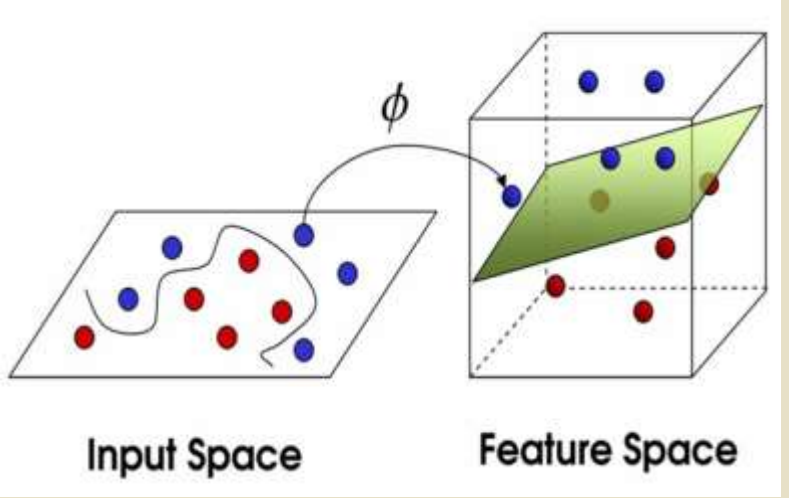
◦ Test data points can be classified as follows: $y = w^T x + b =$ $\sum_{i=1}^{m} \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b.$

Kernels: into multiple dimensions

- You probably remember the polynomial example before...
- Formally speaking what we did was projecting single x into multiple dimensions, i.e. using features like

$$\phi(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}$$





http://www.imtech.res.in/raghava/rbpred/svm.jpg

- Then instead of dot products $\langle x^{(i)}, x \rangle$ we deal with **kernels** $K(x^{(i)}, x) = \langle \phi(x^{(i)}), \phi(x) \rangle$

SVMs: the kernel trick

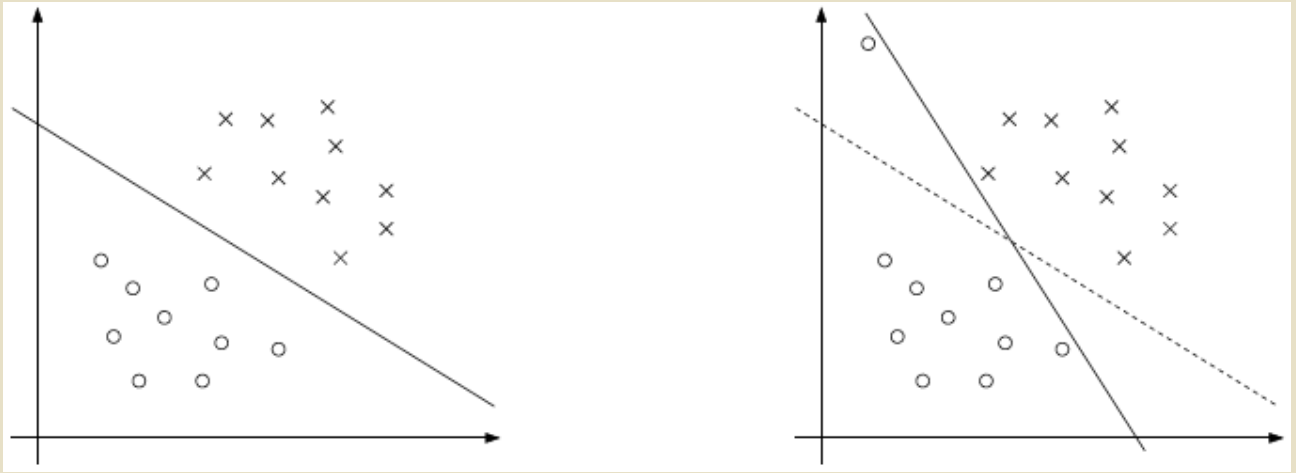
- Kernel functions $\phi(x)$ can be (and usually are) big and ugly, but we don't need to deal with them – we only need to know they exist! Mercer's theorem describes what requirements are necessary and sufficient to qualify as a kernel.

- Without going into that, most often used kernels are the following:
  - **Polynomial** $K(x_i,x_j) = \langle x_i, x_j \rangle^d$
  - Also **polynomial** $K(x_i,x_j) = (\langle x_i, x_j \rangle + 1)^d$
  - **Gaussian radial basis function** (often the most practical):
    $K(x_i,x_j) = \exp(-\beta |x_i - x_j|^2)$

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Why?... Dot products are a measure of similarity (max. if the vectors equal and zero if they are orthogonal) – so are Gaussians!

What to do if perfect classification is not possible?
(i.e. reality)



- Adjust the objective function,

allowing „soft margins“!

- Minimize

$$\frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i$$

$$\text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad i = 1, \ldots, m$$
$$\xi_i \geq 0, \quad i = 1, \ldots, m.$$

- C is a **regularization** parameter – determines the balance between finding a big margin and avoiding misclassifications

Important aspects in SVM training

- Performance depends critically on chosen kernels, kernel parameters and regularization parameter C.

- To find the most appropriate settings it's important to perform search over different parameter values, e.g. in case of Gaussian kernel $K(x_i,x_j) = \exp(-\beta |x_i - x_j|^2)$ it's good to search along log-scale of β and C, like
  - $C = \{2^{-5}, 2^{-3}, ..., 2^{13}, 2^{15}\}$
  - $\beta = \{2^{-15}, 2^{-13}, ..., 2^{1}, 2^{3}\}$

Model size: unlike for MLPs, for SVMs it's more complicated
Effective model size depends on kernel parameters

- For polynomial kernels instead of β it's degree d (1, 2, 3,... – fractions are also used)

Workshop: MNIST digit classification

- In the practical & problem set, we will perform classification of handwritten digits in R, continuing the work from the previous workshop
- **Key steps:**
  - Finding and computing appropriate features (already done!)
  - Splitting data into training and validation sets
  - Training your chosen model and evaluating its performance.
- **The focus will be to go through all the steps as opposed to finding the best features or the best model** (which you can try later for fun ☺)
- We will use the simplest implementation of neural networks in R. More sophisticated options available using classification and regression toolbox *caret*. If interested, you can find a comprehensive guide here: http://topepo.github.io/caret/index.html