Kernels

Agenda

- Introduction
- Kernel functions
- Using kernels inside GLM
- Kernel trick
  - Examples of kernelized algorithms
- Support vector machines

Introduction

- So far, each object that we want to process (e.g. classify or cluster) can be represented as a **fixed-size feature vector**, $x_i \in \mathbb{R}^D$
- In certain situations it is not clear **how to best represent them** as fixed-size features, e.g. text document, protein sequence, molecular structure (has complex 3d structure)…
- Assume that we have some way of measuring the **similarity between objects**, that doesn't require preprocessing them into feature vector format (e.g. edit distance between two strings)
- Let $\kappa(\boldsymbol{x}, \boldsymbol{x}') \geq 0$ be some measure of similarity between objects $\boldsymbol{x}, \boldsymbol{x}' \in \chi$, where χ is some abstract space; we will call κ a **kernel function**

Kernel functions

- We define a **kernel function** to be a real-valued function of two arguments, $\kappa(\boldsymbol{x}, \boldsymbol{x}') \in \mathbb{R}$, for $\boldsymbol{x}, \boldsymbol{x}' \in \chi$. Typically the function is symmetric (i.e., $\kappa(\boldsymbol{x}, \boldsymbol{x}') = \kappa(\boldsymbol{x}', \boldsymbol{x})$), and non-negative (i.e., $\kappa(\boldsymbol{x}, \boldsymbol{x}') \geq 0$), so it can be interpreted as a measure of similarity, but this is not required.
- Example of kernel functions:
  - RBF kernels
  - Kernels for comparing documents
  - Mercer kernels
  - Linear kernels
  - Matern kernels
  - String kernels
  - Kernels derived from probabilistic generative models

Radial basis function (RBF) kernels

- The **squared exponential kernel** (SE kernel) or **Gaussian kernel** is defined by

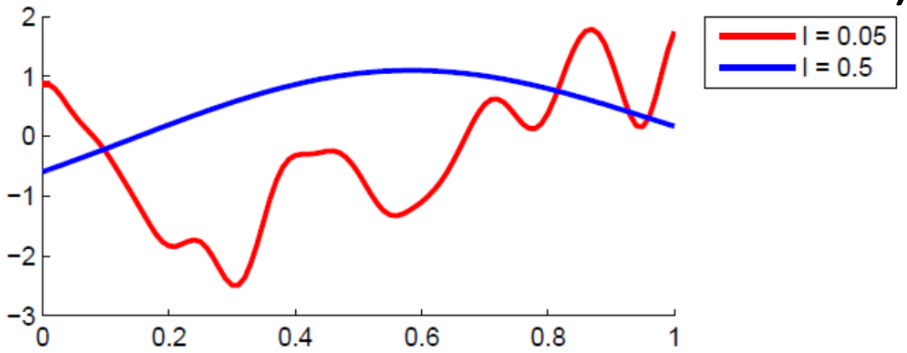
$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \mathbf{x}')\right)$$

- If **Σ** is diagonal, this can be written as $\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}\sum_{j=1}^{D}\frac{1}{\sigma_j^2}(x_j - x_j')^2\right)$

  σ_j defines the **characteristic length scale** of dimension *j*.

- If **Σ** is spherical, we get the isotropic kernel, where σ^2 is known as the **bandwidth** – shared length scale (metric of the period over which local fluctuations are allowed)

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



Kernels for comparing documents

- Bag of words representation, where x_{ij} is the number of times word *j* occurs in document *i*, we can use the **cosine similarity**, which is defined by

$$\kappa(\mathbf{x}_i, \mathbf{x}_{i'}) = \frac{\mathbf{x}_i^T \mathbf{x}_{i'}}{||\mathbf{x}_i||_2 ||\mathbf{x}_{i'}||_2}$$

- This quantity measures the cosine of the angle between $\mathbf{x}_i$ and $\mathbf{x}_{i'}$ when interpreted as vectors.
- Since $\mathbf{x}_i$ is a count vector (and hence non-negative), the cosine similarity is between 0 and 1, where 0 means the vectors are orthogonal and therefore have no words in common.
- Problems, popular words (stop words) like “the” and “and” – make documents similar, burstiness of word usage - artificial similarity value

Kernels for comparing documents

- Simple preprocessing- replace the word count vector with a new feature vector called the **TF-IDF** representation, which stands for "term frequency inverse document frequency".
- The term frequency is defined as a log-transform of the count:

$$\text{tf}(x_{ij}) \triangleq \log(1 + x_{ij})$$

this reduces the impact of words that occur many times within one document

- The inverse document frequency is defined as $\text{idf}(j) \triangleq \log \frac{N}{1 + \sum_{i=1}^{N} \mathbb{I}(x_{ij} > 0)}$
- Where N is the total number of documents and denominator counts how many documents contain the word j
- Finally, we define: $\text{tf-idf}(\mathbf{x}_i) \triangleq [\text{tf}(x_{ij}) \times \text{idf}(j)]_{j=1}^{V}$

Kernels for comparing documents

- We then use this inside the cosine similarity measure. That is, our new kernel has the form

$$\kappa(\mathbf{x}_i, \mathbf{x}_{i'}) = \frac{\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_{i'})}{||\phi(\mathbf{x}_i)||_2 ||\phi(\mathbf{x}_{i'})||_2}$$

where $\phi(\mathbf{x}) = \text{tf-idf}(\mathbf{x})$

Mercer (positive definite) kernels

- Some methods define that the kernel function satisfy the requirement that the **Gram matrix** defined by:

$$\mathbf{K} = \begin{pmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_1, \mathbf{x}_N) \\ & \vdots & \\ \kappa(\mathbf{x}_N, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix}$$

be positive definite (symmetric with positive eigenvalues) for any set of inputs $\{\mathbf{x}_i\}_{i=1}^N$

- This kind of kernel is called **Mercer kernel** (e.g. Gaussian and cosine kernels are Mercer)

Mercer's theorem

- Importance of Mercer kernel is the following
- If the Gram matrix is positive definite, the eigenvalue decomposition is the following: $\mathbf{K} = \mathbf{U}^T \mathbf{\Lambda} \mathbf{U}$, where $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues, $\lambda_i > 0$
- Consider an element of K: $k_{ij} = (\mathbf{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:,i})^T (\mathbf{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:j})$
- Let us define $\phi(\mathbf{x}_i) = \mathbf{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:i}$
- Then we can write $k_{ij} = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$
- We see that the **entries in the kernel matrix** can be **computed** by performing **an inner product of some feature vectors** that are implicitly defined by the eigenvectors U

Mercer's theorem

- In general, if the kernel is Mercer, then there exists a function ϕ mapping $\boldsymbol{x}$ such that

$$\kappa(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}')$$

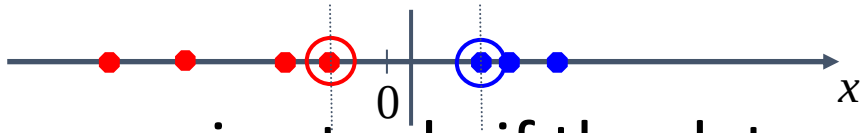
- Where ϕ depends on the eigen functions of k
- For example, **polynomial kernel** $\kappa(\mathbf{x}, \mathbf{x}') = (\gamma \mathbf{x}^T \mathbf{x}' + r)^M$ where $r > 0$.
- If, for example, $M = 2, \gamma = r = 1$ and $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^2$, we have

$$\begin{aligned} (1 + \mathbf{x}^T \mathbf{x}')^2 &= (1 + x_1 x_1' + x_2 x_2')^2 \\ &= 1 + 2x_1 x_1' + 2x_2 x_2' + (x_1 x_1)^2 + (x_2 x_2')^2 + 2x_1 x_1' x_2 x_2' \end{aligned}$$

- This can be written as $\phi(\mathbf{x})^T \phi(\mathbf{x}')$ where $\phi(\mathbf{x}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2]^T$
- So using this kernel is equivalent to working in a 6-dimensional feature space.

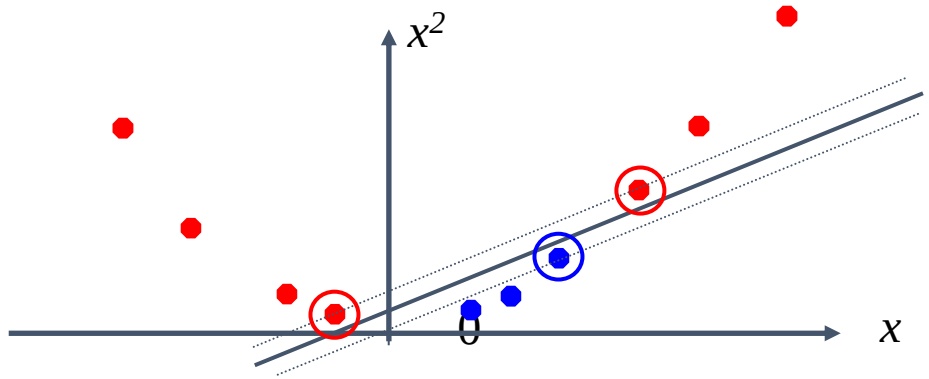
Mercer's theorem

- Datasets that are linearly separable with some noise:



- But what are we going to do if the dataset is just too hard?

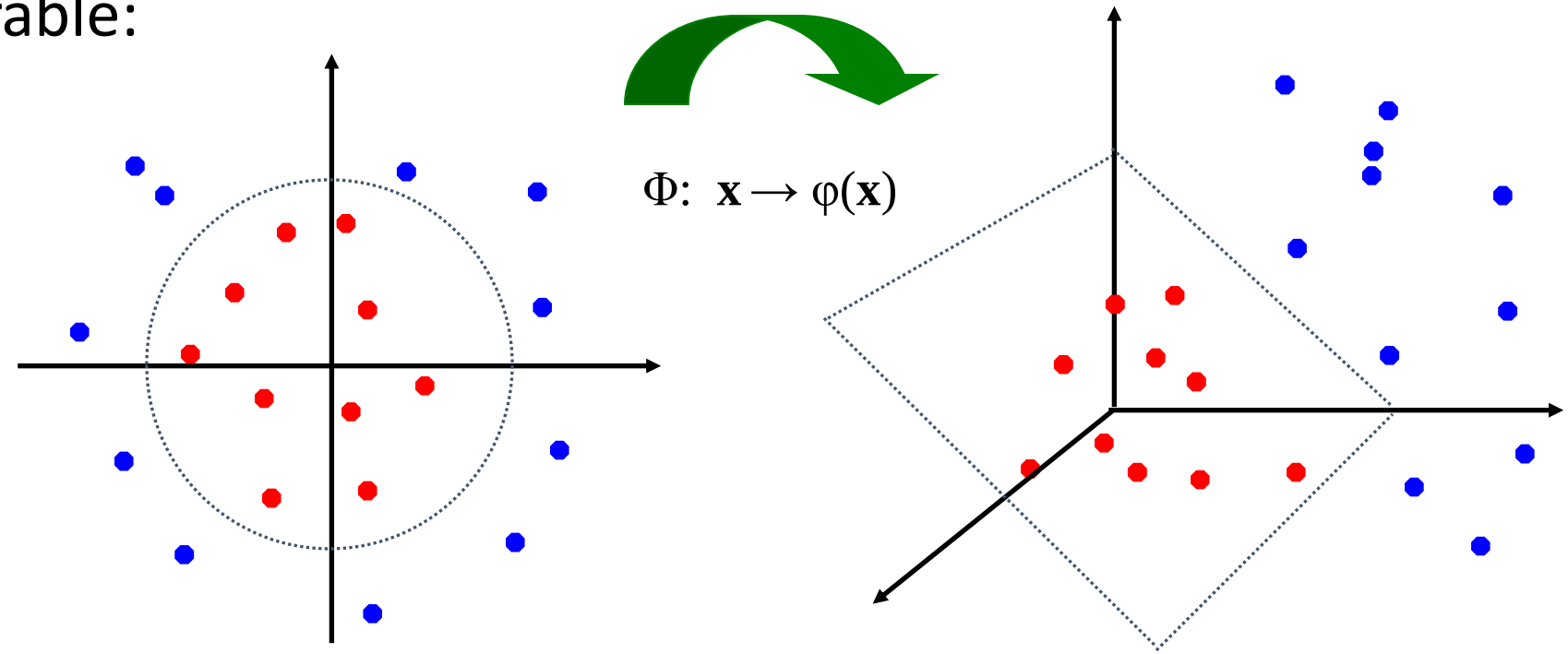
- How about… mapping data to a higher-dimensional space:



- SVMWithPolynomialKernelVisualization.mp4

Mercer's theorem

- General idea: the original input space can always be mapped to some higher-dimensional feature space where the training set is separable:



Linear kernels

- Deriving the feature vector implied by a kernel is in general quite difficult, and only possible if the kernel is Mercer.
- However, deriving a kernel from a feature vector is easy: we just use

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^T \boldsymbol{\phi}(\mathbf{x}') = \langle \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}') \rangle$$

- If $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$ we get the **linear kernel**, defined by

$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$$

- This is **useful if the original data is already high dimensional**, and if the original features **are individually informative**, e.g., a bag of words representation where the vocabulary size is large, or the expression level of many genes.
- In such a case, the decision boundary is likely to be representable as a linear combination of the original features, so it is not necessary to work in some other feature space.

String kernels

- The real power of kernels arises when the inputs are structured objects.
- As an example, we now describe one way of comparing **two variable length strings** using a string kernel
- Consider two strings **x** and **x'** of lengths *D*, *D'*, each defined over the 20 letter alphabet *A*.

IPTSALVKETLALLSTHRTLLIANETLRIPVPVHKNHQLCTEEIFQGIGTLESQTVQGGTV
ERLFKNLSLIKKYIDGQKKKCGEERRRVNQFLDYLQEFLGVMNTEWI

PHRRDLCSRSIWLARKIRSDLTALTESYVKHQGLWSELTEAERLQENLQAYRTFHVLLA
RLLEDQQVHFTPTEGDFHQAIHTLLLQVAAFAYQIEELMILLEYKIPRNEADGMLFEKK
LWGLKVLQELSQWTVRSIHDLRFISSHQTGIP

- These strings have the substring LQE in common. We can define the similarity of two strings to be the number of substrings they have in common.

String kernels

- Let $\phi_s(x)$ denote the number of times that substring *s* appears in string *x*. We define the kernel between two strings *x* and *x'* as

$$\kappa(x, x') = \sum_{s \in \mathcal{A}^*} w_s \phi_s(x) \phi_s(x')$$

- where $w_s \geq 0$ is the weight of string *s* and $A*$ is the set of all strings (of any length) from the alphabet *A* (this is known as the Kleene star operator)
- This is a Mercer kernel
- Variations, bag-of-characters, bag-of-words, strings of fixed length

Using kernels inside Generalized linear models - Kernel machines

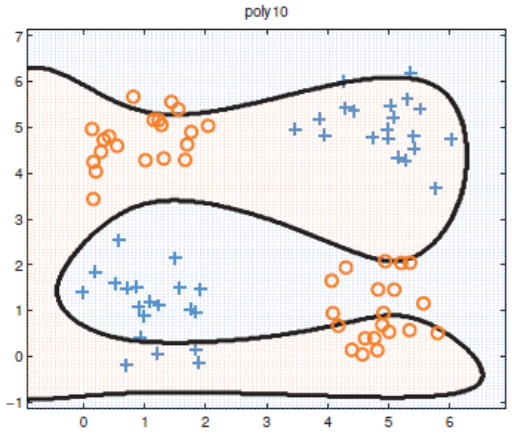
- We can define **kernel machine** with the input feature vector

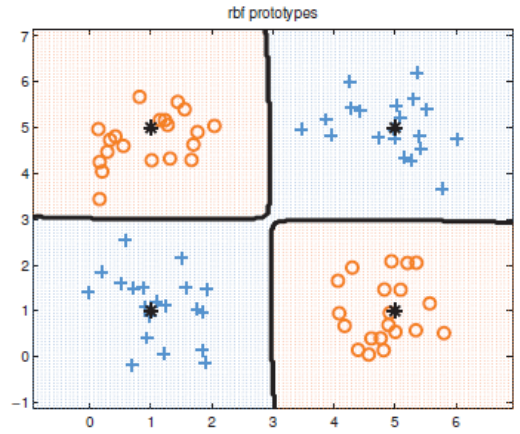
$$\phi(\mathbf{x}) = [\kappa(\mathbf{x}, \boldsymbol{\mu}_1), \ldots, \kappa(\mathbf{x}, \boldsymbol{\mu}_K)]$$

where $\mu_k \in \chi$ are a set of K **centroids**, and $\phi(x)$ is **kernelized feature vector.** If k is RBF kernel, this is called RBF network.

- Kernelized feature vector for **logistic regression** $p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathrm{Ber}(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}))$
- Simple way to define non-linear decision boundary

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0





a) xor truth table. (b) Fitting a logistic regression classifier using degree 10 polynomial expansion. (c) Same model, but using an RBF kernel with centroids specified by the 4 black crosses.

The kernel trick

- Some models relies on dot product between vectors $K(x_i, x_j) = x_i^T x_j$
- If every data point is mapped into high-dimensional space via some transformation Φ: x → ϕ(x), the dot product becomes:

$$\boldsymbol{k}(\mathbf{x_i},\mathbf{x_j}) = \boldsymbol{\phi}(\mathbf{x_i})^{\mathrm{T}}\,\boldsymbol{\phi}(\mathbf{x_j})$$

- A kernel function is in fact some function that **corresponds to an inner product in some expanded feature space.**
- Rather than defining our feature vector in terms of $\boldsymbol{\phi}(\mathbf{x})$ we can instead work with the original feature vectors **x**, but modify the algorithm so that it replaces all inner products of the form **x**, **x'** with a call to the kernel function, κ(**x**, **x'**). This is called the **kernel trick**.
- It turns out that many algorithms can be kernelized in this way.
- Note that we require that the kernel be a Mercer kernel for this trick to work.

Examples of kernelized algorithms

- **Kernelized nearest neighbor classification** – compute the Euclidean distance of a test vector to all the training points, find the closest one and look up its label. This can be kernelized by observing that

$$||\mathbf{x}_i - \mathbf{x}_{i'}||_2^2 = \langle \mathbf{x}_i, \mathbf{x}_i \rangle + \langle \mathbf{x}_{i'}, \mathbf{x}_{i'} \rangle - 2\langle \mathbf{x}_i, \mathbf{x}_{i'} \rangle$$

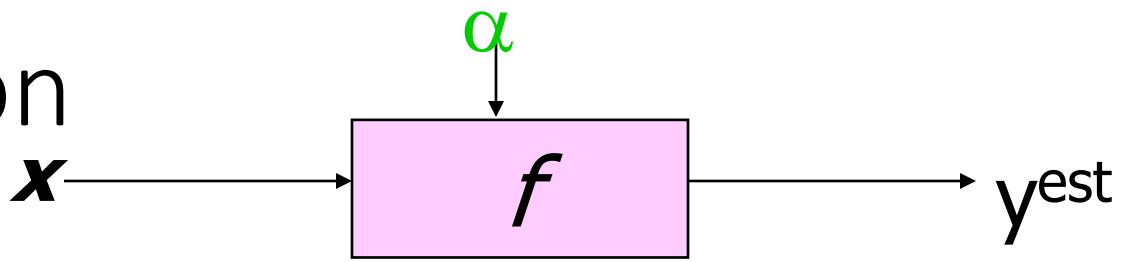
  - features produced by these feature maps can bring the points from the same class closer to each other and push points from different classes away if correct kernel is selected for the data
  - Enables NN classifier to be applied to structured data
- **Kernelized K-medoids clustering** – k-medoids clustering also uses Euclidean distance to measure dissimilarity, which can also be kernelized by using the same equation as in kernelized NN classification

Support vector machines

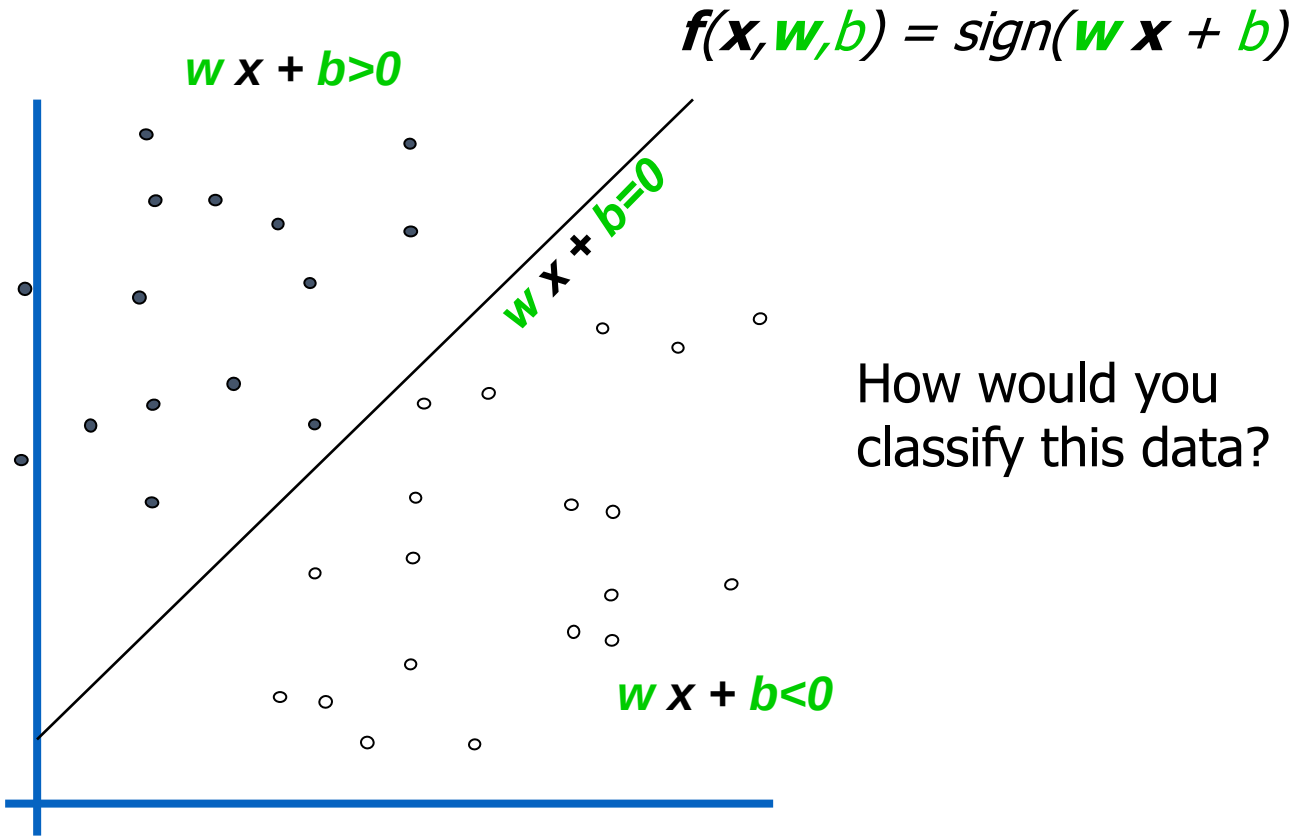
- Unnatural from a probabilistic point of view
- Originally designed for binary classification, but can be extended to regression and multi-class classification
- Very popular and widely used
- The predictions only depend on a subset of training data, known as **support vectors**
- This, together with the kernel trick is known as **support vector machine** or **SVM**

SVM for classification

Linear Classifiers

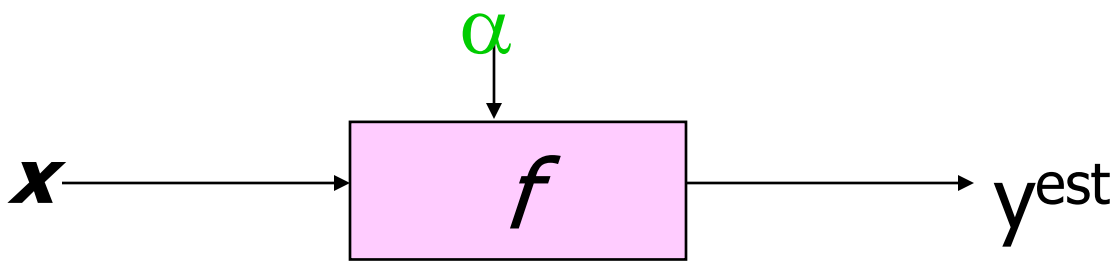


- denotes +1
- denotes -1



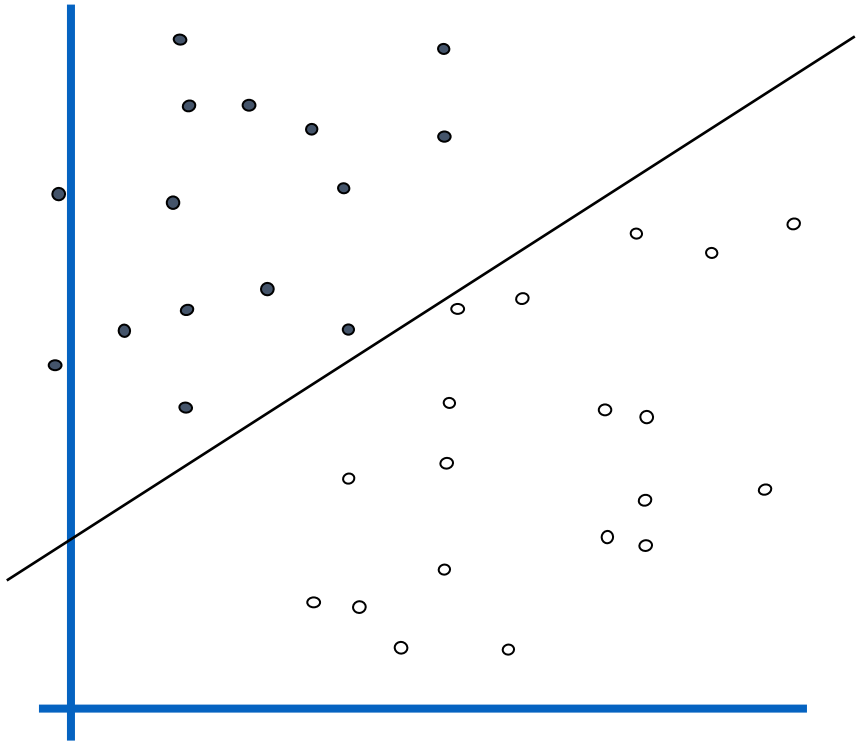
How would you classify this data?

Linear Classifiers



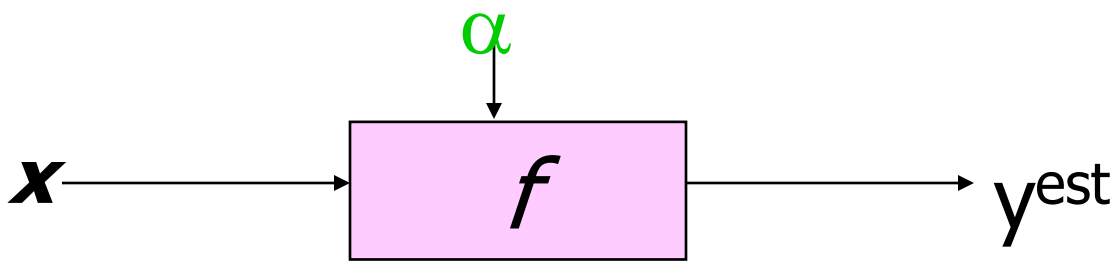
$f(\mathbf{x},\mathbf{w},b) = sign(\mathbf{w}\,\mathbf{x} + b)$

- denotes +1
- denotes -1



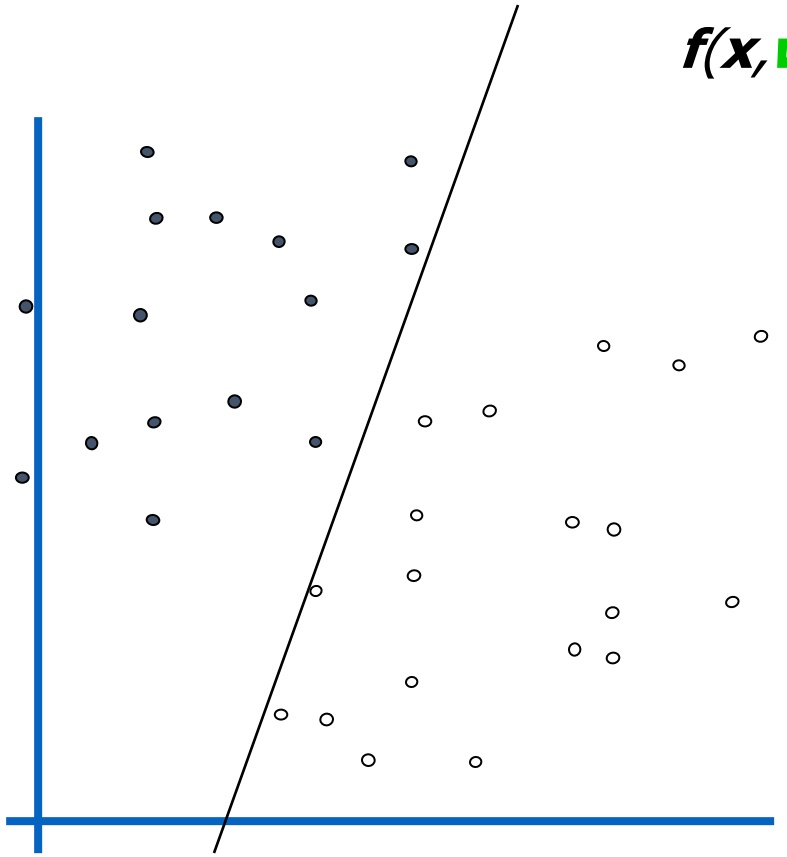
How would you classify this data?

Linear Classifiers



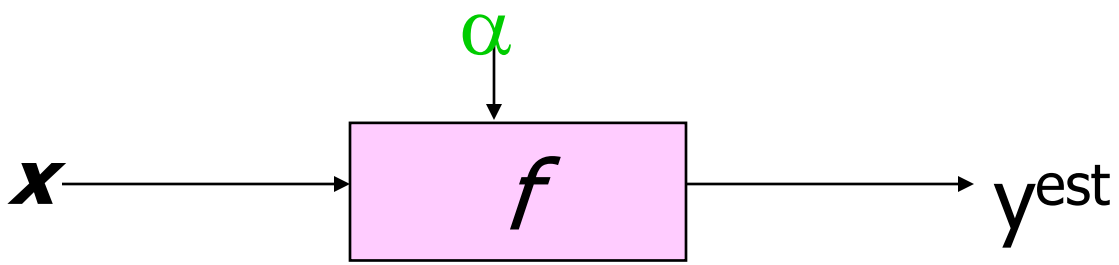
- denotes +1
- denotes -1

$f(\mathbf{x}, \mathbf{w}, b) = sign(\mathbf{w}\,\mathbf{x} + b)$



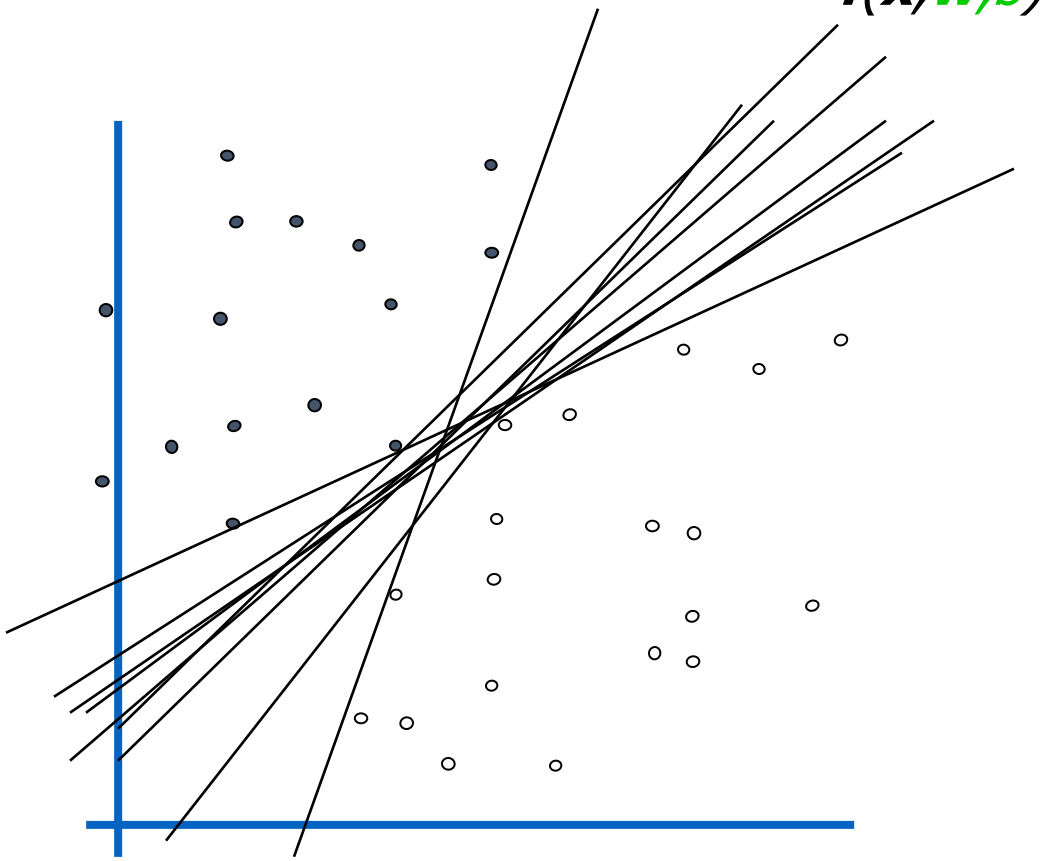
How would you classify this data?

Linear Classifiers



$f(\mathbf{x}, \mathbf{w}, b) = sign(\mathbf{w}\,\mathbf{x} + b)$

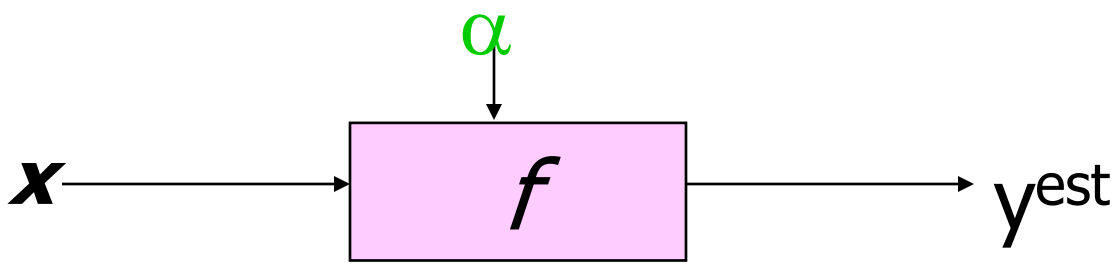
- denotes +1
- denotes -1



Any of these would be fine..

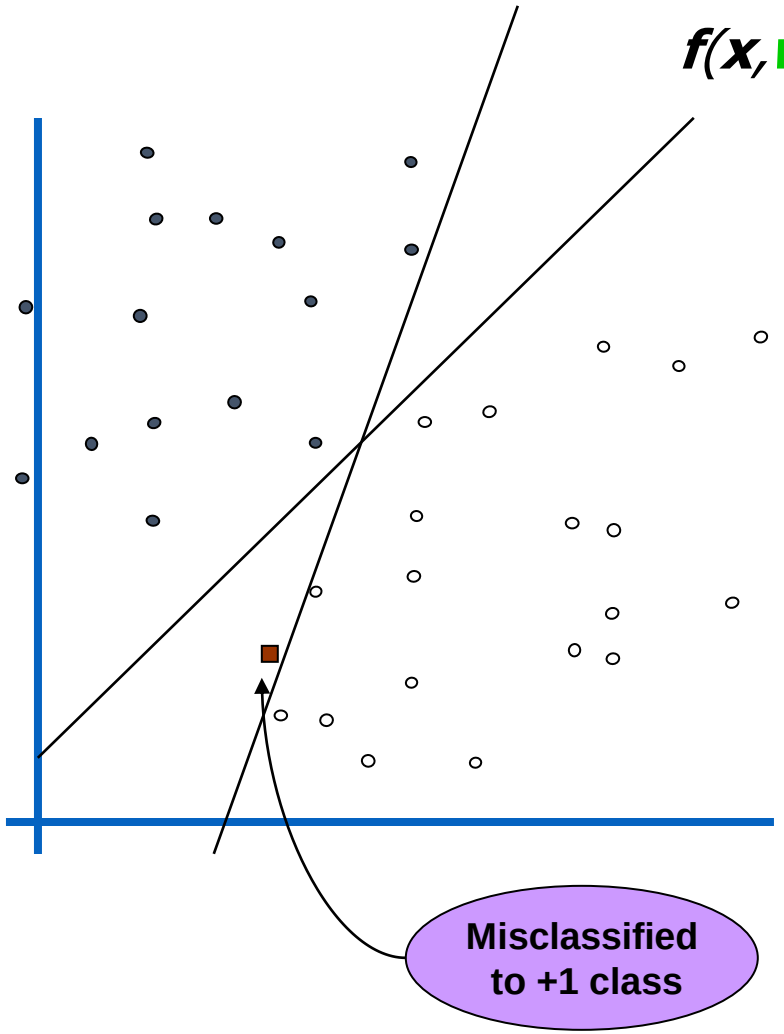
..but which is best?

Linear Classifiers



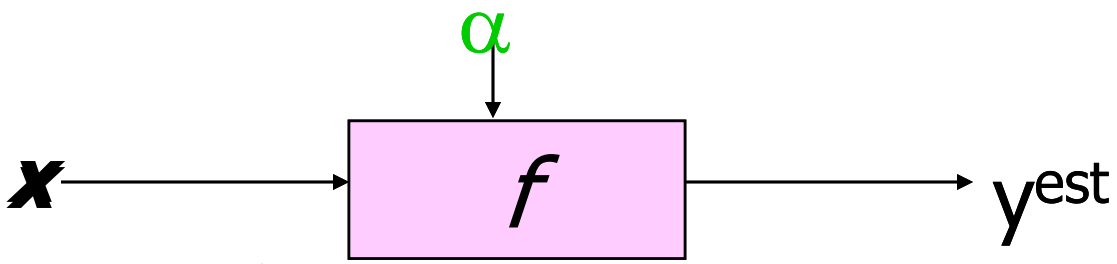
- denotes +1
- denotes -1

$f(\mathbf{x}, \mathbf{w}, b) = sign(\mathbf{w}\,\mathbf{x} + b)$



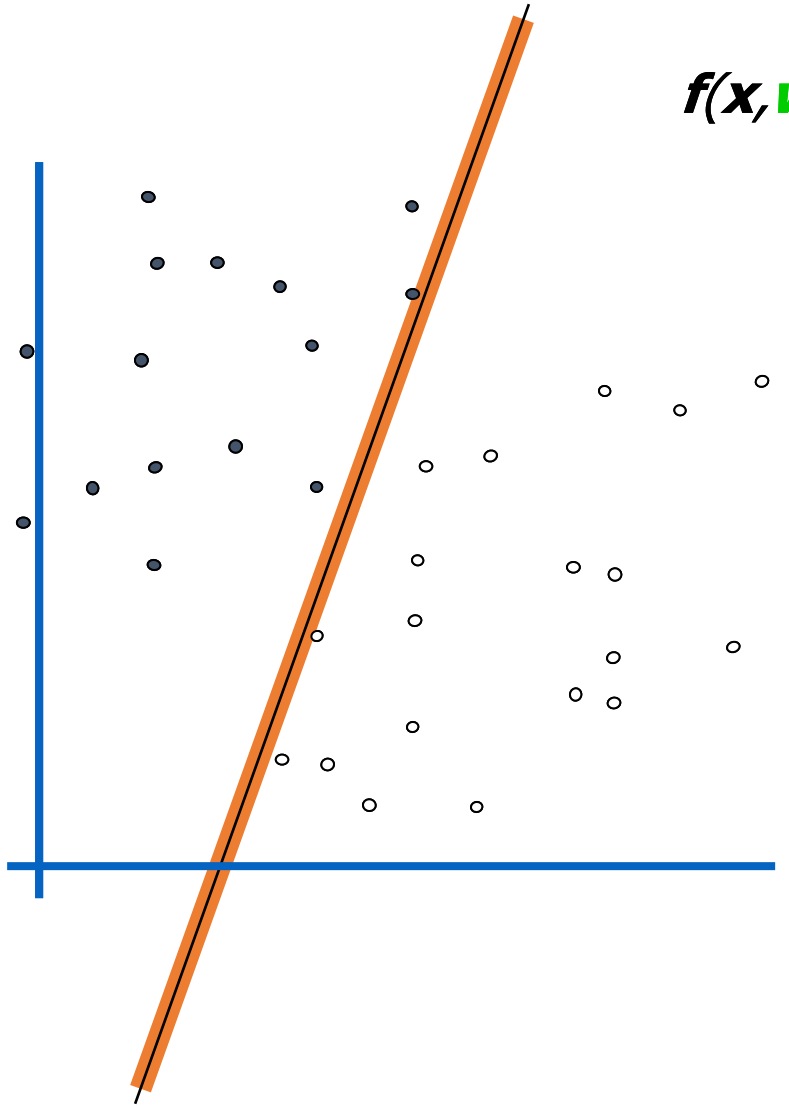
How would you classify this data?

Classifier Margin



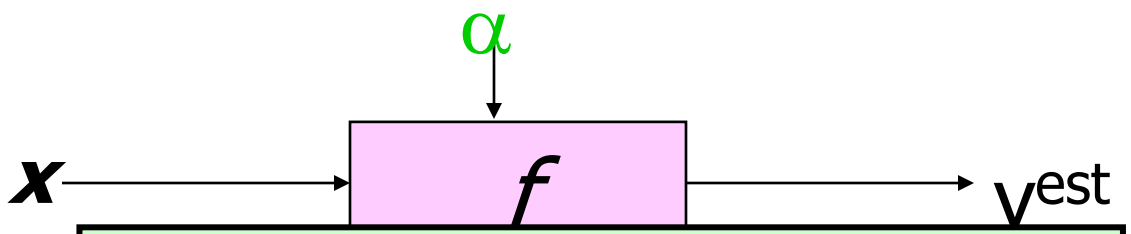
- denotes +1
- denotes -1

$f(\mathbf{x},\mathbf{w},b) = sign(\mathbf{w}\,\mathbf{x} + b)$



Define the margin of a linear classifier as the width that the boundary could be increased by before hitting a datapoint.

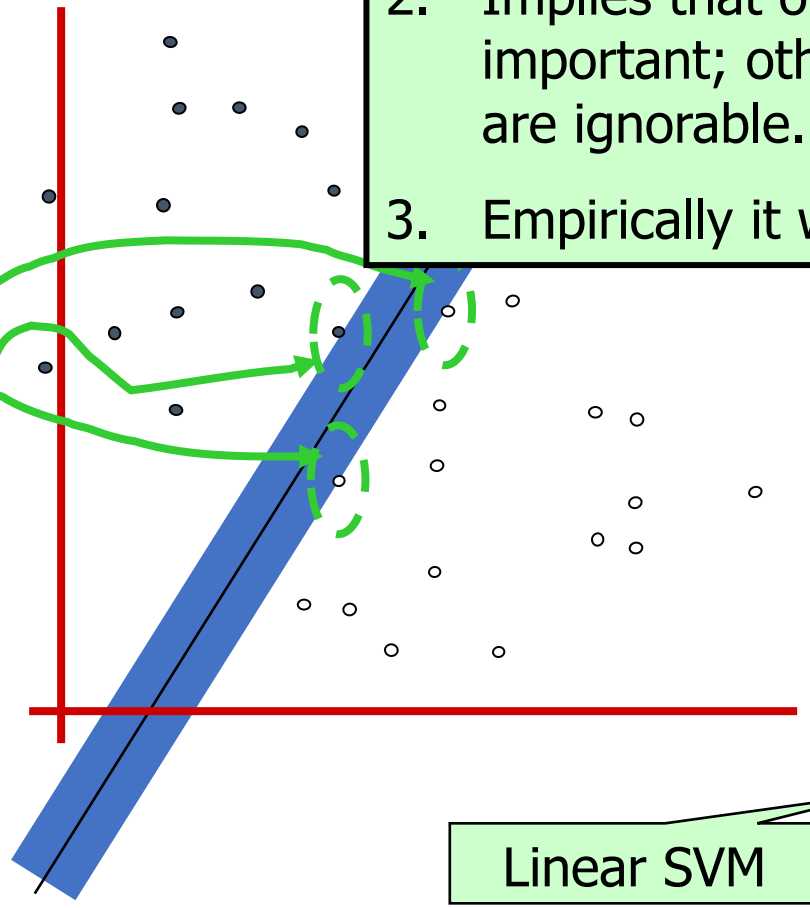
Maximum Margin



1. Maximizing the margin is good according to intuition
2. Implies that only support vectors are important; other training examples are ignorable.
3. Empirically it works very very well.

- denotes +1
- denotes -1

Support Vectors are those datapoints that the margin pushes up against



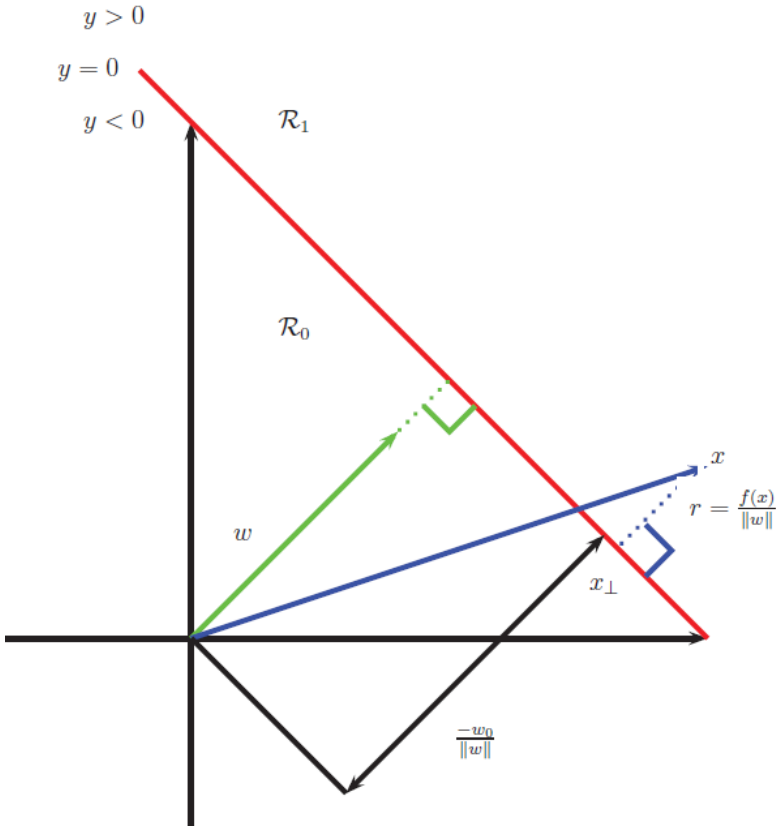
linear classifier with the, um, maximum margin.

This is the simplest kind of SVM (Called an LSVM)

Linear SVM

The large margin problem

- The goal is to derive a discriminant function $f(x)$ which will be linear in the feature space implied by the choice of kernel.



- Consider a point x in the induced space:

$$\mathbf{x} = \mathbf{x}_\perp + r\frac{\mathbf{w}}{||\mathbf{w}||}$$

- Where r is the distance of x from the decision boundary, whose normal vector is w and $x_\perp$ is the orthogonal projection of x onto this boundary. So,

$$f(\mathbf{x}) \quad = \quad \mathbf{w}^T\mathbf{x} + w_0 = (\mathbf{w}^T\mathbf{x}_\perp + w_0) + r\frac{\mathbf{w}^T\mathbf{w}}{||\mathbf{w}||}$$

- Now, $f(\mathbf{x}_\perp) = 0$ so $0 = \mathbf{w}^T\mathbf{x}_\perp + w_0$.
- Hence:

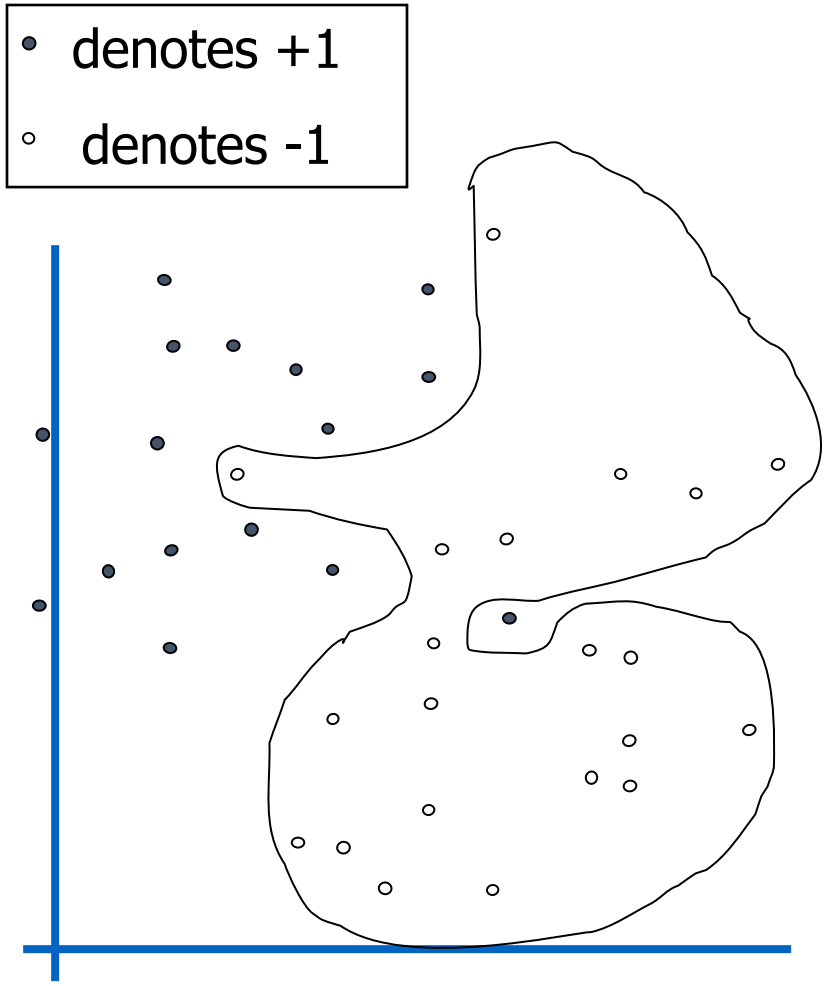
$$f(\mathbf{x}) = r\frac{\mathbf{w}^T\mathbf{w}}{\sqrt{\mathbf{w}^T\mathbf{w}}}, \text{ and } r = \frac{f(\mathbf{x})}{||\mathbf{w}||}$$

The large margin problem

- We would like to make the distance $r = f(\mathbf{x})/||\mathbf{w}||$ as large as possible
- In addition, we want to ensure each point is on the correct side of the boundary. Therefore, we want to optimize:

$$\min_{\mathbf{w},w_0} \frac{1}{2}||\mathbf{w}||^2 \quad \text{s.t.} \quad y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1, i = 1:N$$

Dataset with noise



- **Hard Margin: So far we require all data points be classified correctly**

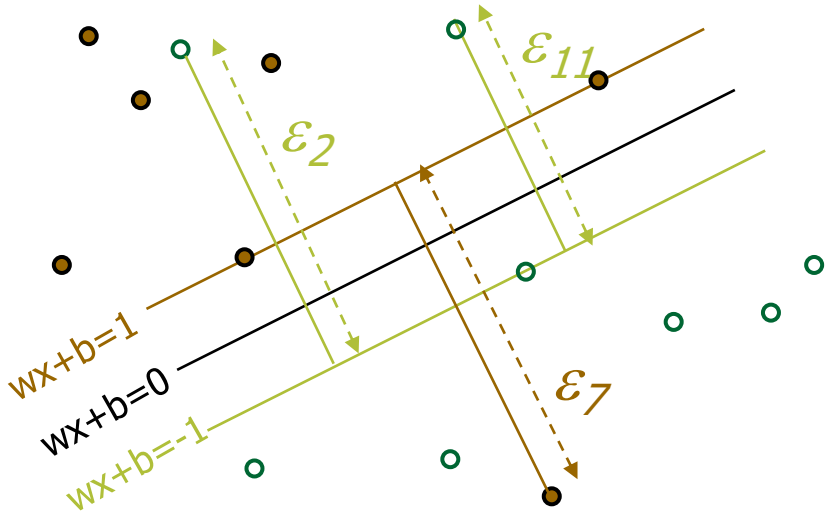
  - No training error

- **What if the training set is noisy?**

  - Solution 1: use very powerful kernels

OVERFITTING!

Soft Margin Classification



***Slack variables εi* can be added to allow misclassification of difficult or noisy examples.**

What should our quadratic optimization criterion be?

$$\min_{\mathbf{w},w_0,\boldsymbol{\xi}} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{N}\xi_i \quad \text{s.t.} \quad \xi_i \geq 0, \;\; y_i(\mathbf{x}_i^T\mathbf{w} + w_0) \geq 1 - \xi_i$$

SVMs for classification

- The optimal solution is: $\hat{\mathbf{w}} = \sum_i \alpha_i \mathbf{x}_i$

- Each non-zero α_i indicates that corresponding $\mathbf{x}_i$ is a support vector.
- At test time, the prediction is done using

$$\hat{y}(\mathbf{x}) = \operatorname{sgn}(f(\mathbf{x})) = \operatorname{sgn}\left(\hat{w}_0 + \hat{\mathbf{w}}^T \mathbf{x}\right)$$

- If we also include the kernel trick, we get:

$$\hat{y}(\mathbf{x}) = \operatorname{sgn}\left(\hat{w}_0 + \sum_{i=1}^{N} \alpha_i \kappa(\mathbf{x}_i, \mathbf{x})\right)$$