Introduction to Machine Learning.
CSCI-UA 9473, Lecture 5.

Augustin Cosse

Ecole Normale Supérieure, DMA & NYU
Fondation Sciences Mathématiques de Paris.

2018

What have we seen so far?

- Bayesian framework and estimators, prior, posterior, MLE, MAP
- Supervised Learning
  - Linear regression
    - Bias variance trade-off (Linear and non linear data)
    - Regularization (Ridge, Lasso, Subset Selection)
  - Linear classification
    - Discriminative vs Generative classifiers
    - Least squares
    - Logistic regression

Today

- Geometry of separating hyperplanes (recap) and Rosenblatt perceptron
- The curse of dimensionality
- A word on non parametric classifiers
- Kernel methods
- Support vector machines (SVM)

Parametric vs Non Parametric (I)

- Remember the difference between parametric and non parametric methods ?
- Linear regression = linear combination of a fixed number of (possibly non linear) basis functions

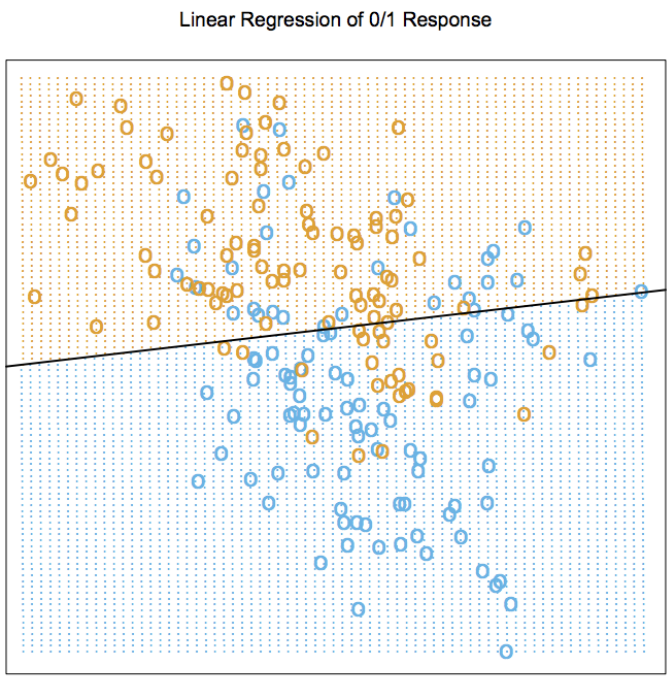
$$Y = \beta_0 + \sum_{k=1}^{d} \beta_k X_k$$

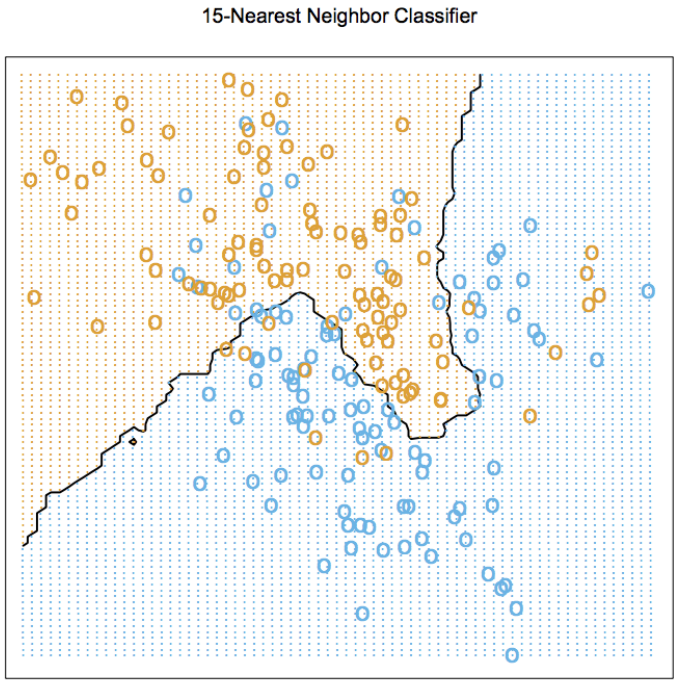
$$Y = \beta_0 + \sum_{k=1}^{d} \beta_k \phi_k(X)$$

- Linearity in the parameters leads to interesting properties such as closed form solution, computational tractability,...

Parametric vs Non Parametric (II)

- Today we will discuss non parametric models
- Parametric = fixed number of parameters, Non parametric = number of parameters/model complexity grows with the amount of training samples





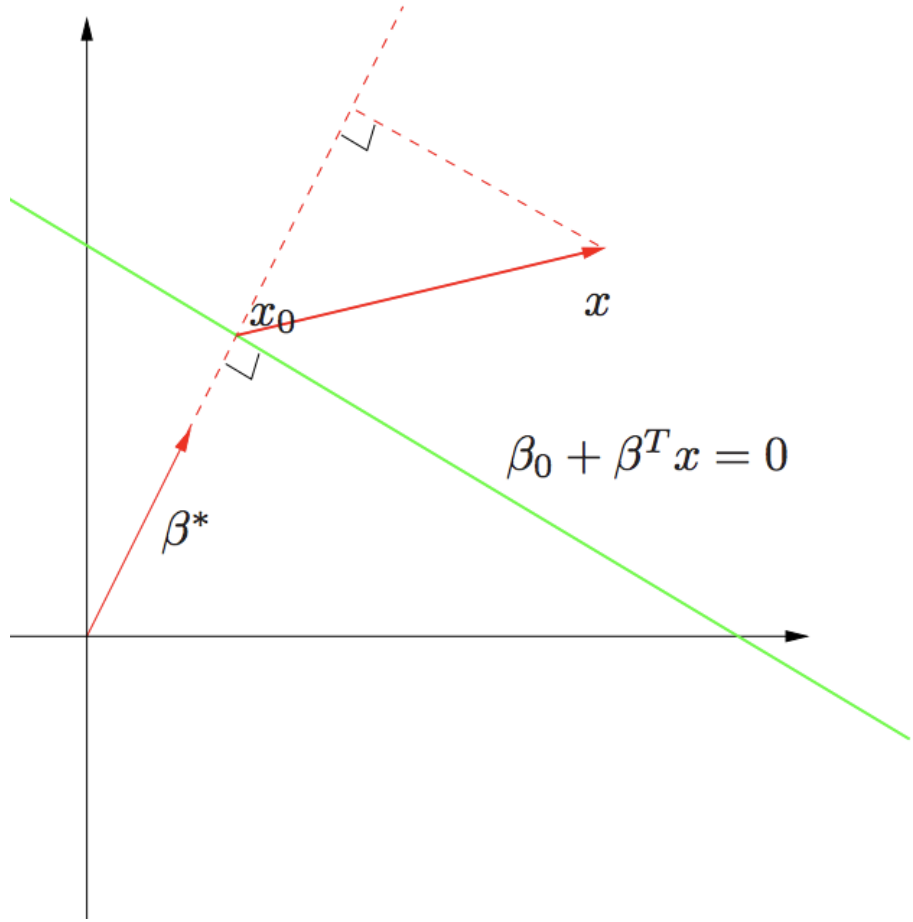
(from H.,T.,F., The Elem. of Stat. Learn.)

Separating Hyperplanes (quick recap)

- Consider the separating hyperplane $\beta_0 + \boldsymbol{\beta}^T \boldsymbol{x}$
- $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ belong to the plane if they satisfy

$$\beta_0 + \boldsymbol{\beta}^T \boldsymbol{x}_1 = \beta_0 + \boldsymbol{\beta}^T \boldsymbol{x}_2$$

- We thus have $\boldsymbol{\beta}^T(\boldsymbol{x}_1 - \boldsymbol{x}_2) = 0$ for all $\boldsymbol{x}_1, \boldsymbol{x}_2$ in the plane
- $\boldsymbol{\beta}$ ($\boldsymbol{\beta}^* = \boldsymbol{\beta}/\|\boldsymbol{\beta}\|$) is the vector normal to the hyperplane



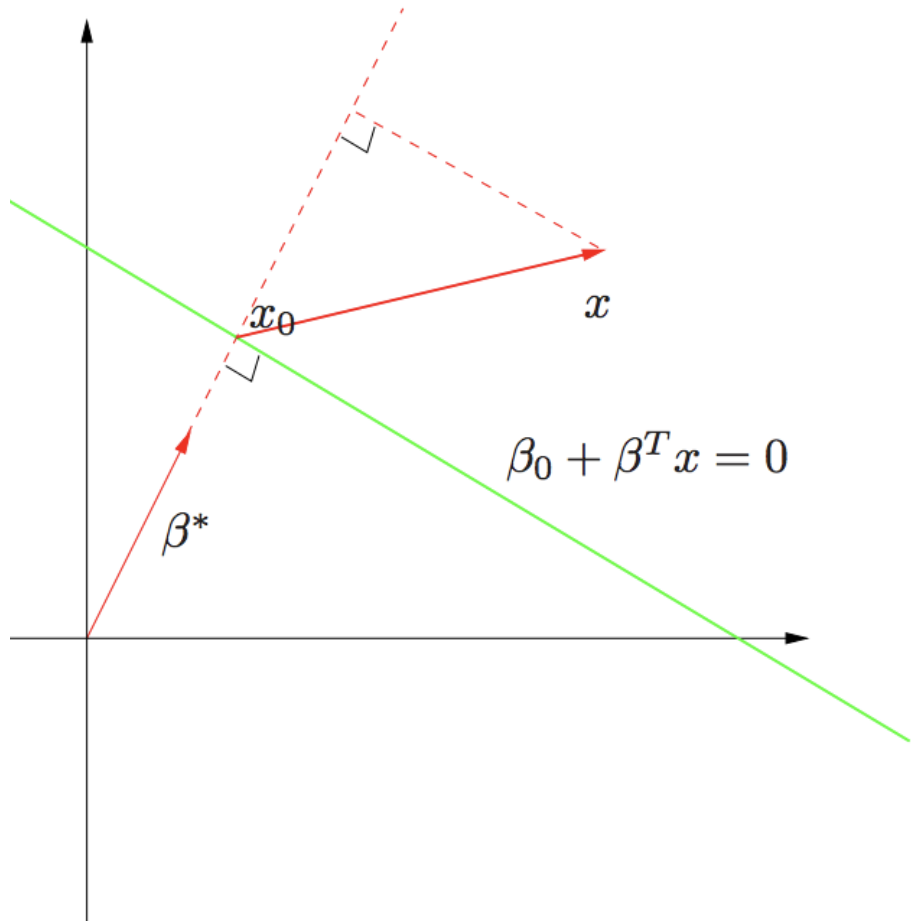
H,T,F, Elem. of Stat. Learn.

- The signed distance of a point $\boldsymbol{x}$ to the hyperplane is defined as

$$(\boldsymbol{\beta}^*)^T(\boldsymbol{x} - \boldsymbol{x}_0) = \frac{1}{\|\boldsymbol{\beta}\|}(\boldsymbol{\beta}^T\boldsymbol{x} + \beta_0)$$

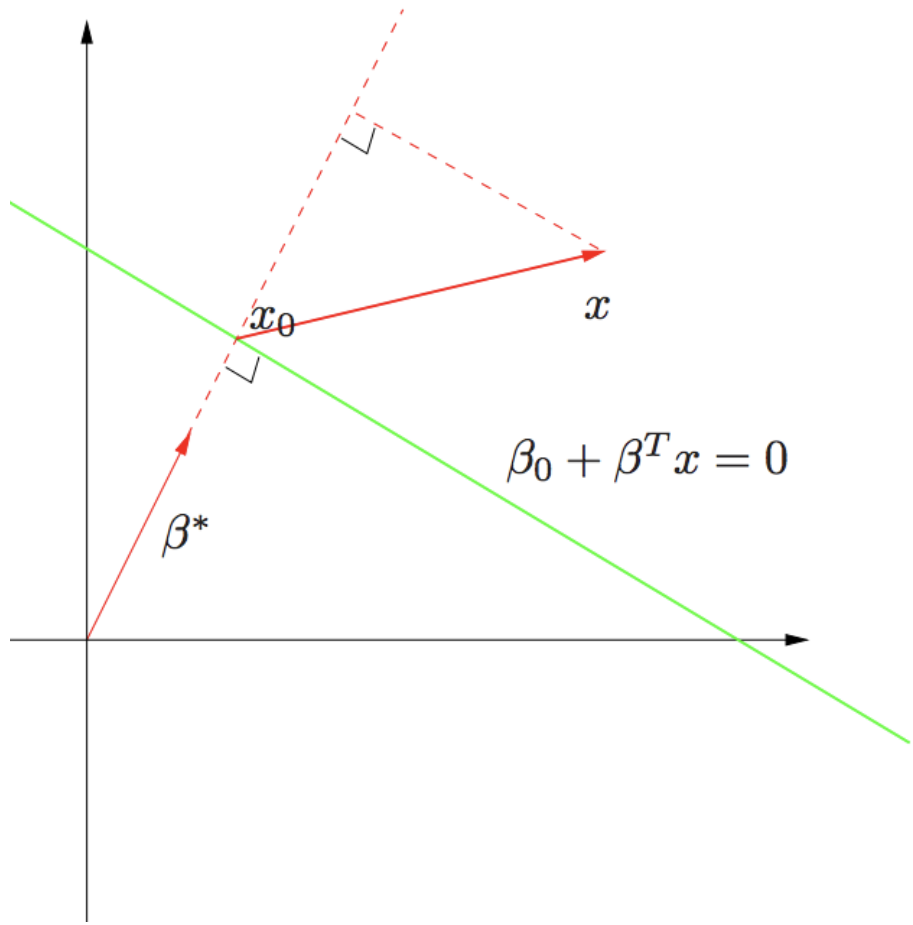
- Points that are located above thus lead to positive values $\beta^T\boldsymbol{x} + \beta_0 > 0$

- Points that are located below lead to negative values $\beta^T\boldsymbol{x} + \beta_0 < 0$



H,T,F, Elem. of Stat. Learn.

- A separating plane thus gives a natural way to associate positive or negative labels to points
- For a two class classification problem, we can look for the plane that gives positive labels to one class and negative labels to the other
- This idea leads to the perceptron algorithm of Rosenblatt



H,T,F, Elem. of Stat. Learn.

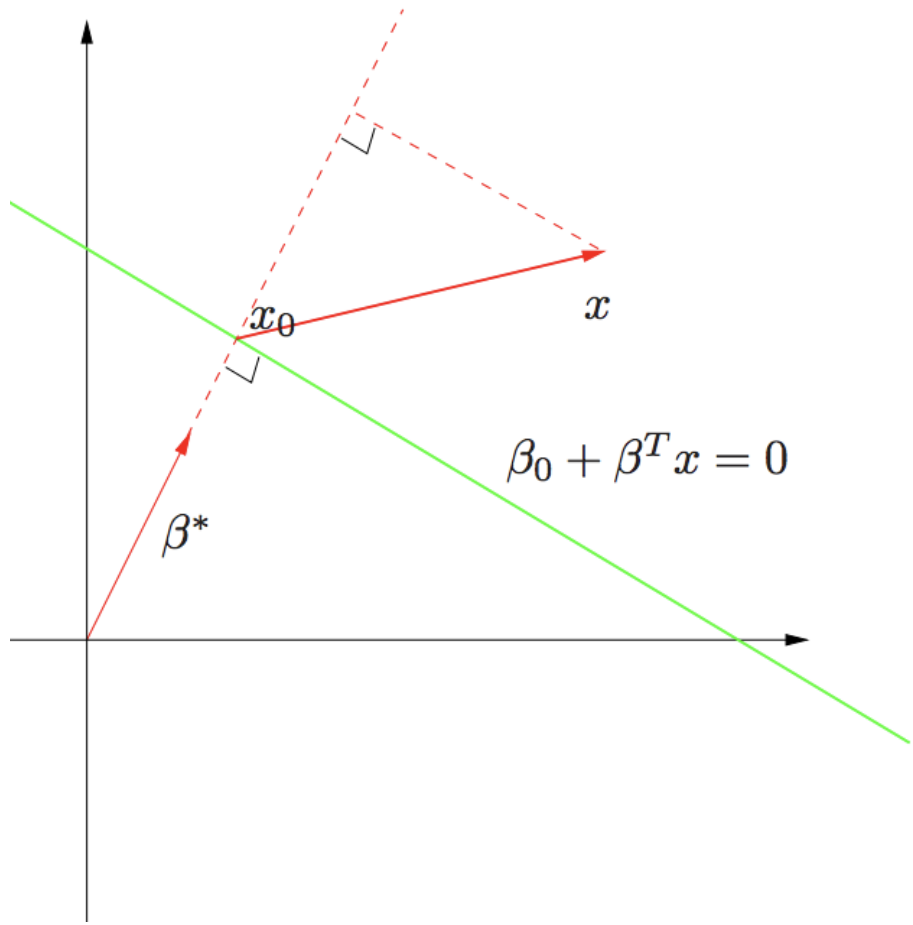
- The perceptron thus simply reads as

$$y(\boldsymbol{x}) = f(\beta_0 + \boldsymbol{\beta}^T \boldsymbol{x})$$

Where

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

- During training, we associate $+1$ labels to points in cluster $\mathcal{C}_1$ and -1 labels to points in cluster $\mathcal{C}_2$



H,T,F, Elem. of Stat. Learn.

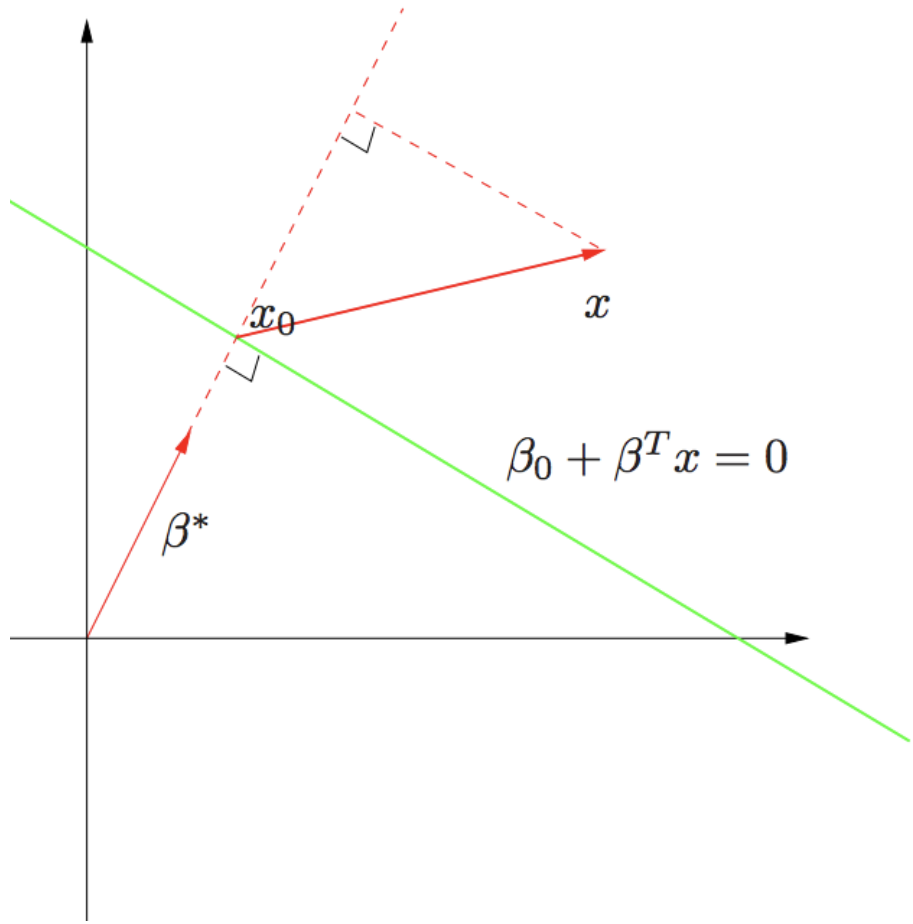
- In the perceptron, a point in $\mathcal{C}_1$ ($y_i = +1$) is thus misclassified if $\beta^T x_i + \beta_0 < 0$
- Generally we would like all points to satisfy

$$y_i(\beta^T x_i + \beta_0) > 0$$

- so we minimize

$$-\sum_{i \in \mathcal{M}} y_i(\beta^T x_i + \beta_0)$$

(contributions should be ≥ 0)



H,T,F, Elem. of Stat. Learn.

Perceptron

(Perceptron) $$D(\boldsymbol{\beta}, \beta_0) = -\sum_{i \in \mathcal{M}} y_i(\beta_0 + \boldsymbol{\beta}^T \boldsymbol{x}_i)$$

- How do we train the perceptron?
- One way is to use stochastic gradient descent (we will come back to that idea later)
- General idea (perceptron learning algorithm)
  - Choose initial vector of prefactors $\boldsymbol{\beta}$
  - Then for each misclassified points $\boldsymbol{x}_n$, do

$$\begin{bmatrix} \boldsymbol{\beta}^{k+1} \\ \beta_0^{k+1} \end{bmatrix} \leftarrow \begin{bmatrix} \boldsymbol{\beta}^{k} \\ \beta_0^{k} \end{bmatrix} - \eta \nabla D^n(\boldsymbol{\beta}, \beta_0), \quad D^n = y_n(\beta_0 + \boldsymbol{\beta}^T \boldsymbol{x}_n)$$

Perceptron: intuition

Perceptron w/
gen. features $\phi(X)$

$$D(\beta, \beta_0) = -\sum_{i \in \mathcal{M}} y_i(\beta_0 + \beta^T \phi_i)$$

$$\begin{bmatrix} \beta^{k+1} \\ \beta_0^{k+1} \end{bmatrix} \leftarrow \begin{bmatrix} \beta^k \\ \beta_0^k \end{bmatrix} - \eta \nabla D^n(\beta, \beta_0), \quad D^n = -y_n(\beta_0 + \beta^T \phi(\boldsymbol{x}_n))$$

$$\text{if no intercept} \quad \beta^{k+1} \leftarrow \beta^k + \eta \phi_n y_n$$

- Perceptron convergence Theorem: If there exists an exact solution (data is linearly separable), then the perceptron algorithm is guaranteed to find an exact solution in a finite number of steps.

Perceptron: more intuition

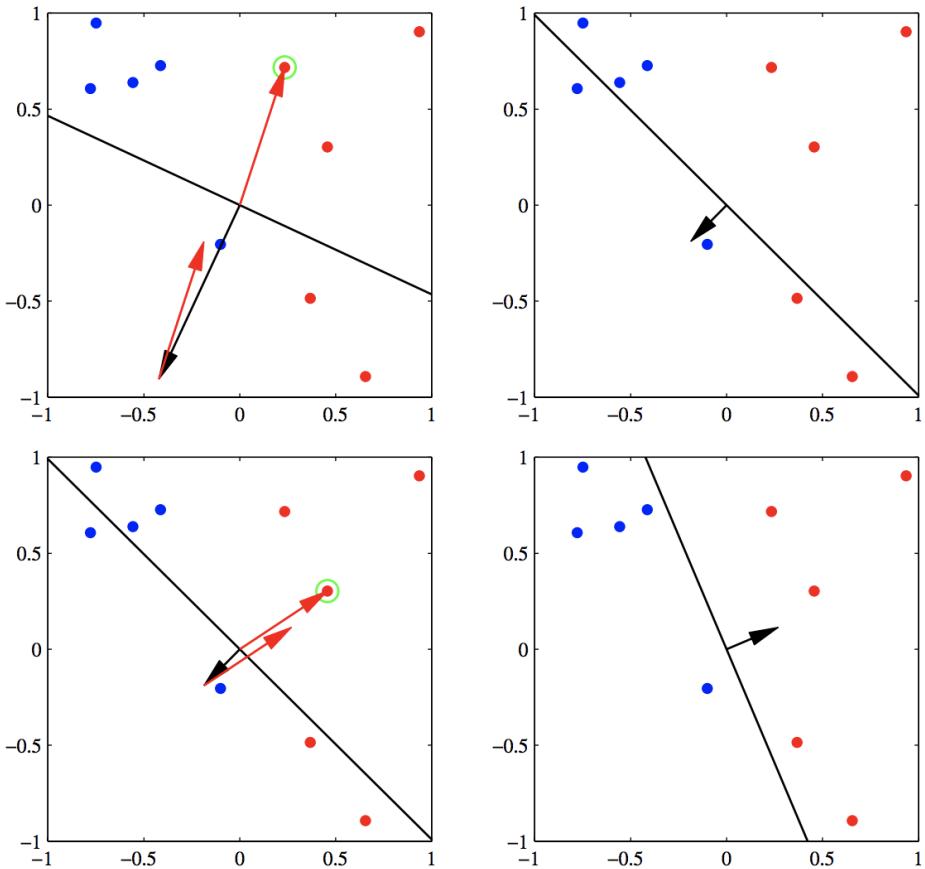


Figure 4.7 Illustration of the convergence of the perceptron learning algorithm, showing data points from two classes (red and blue) in a two-dimensional feature space (ϕ_1, ϕ_2). The top left plot shows the initial parameter vector $\mathbf{w}$ shown as a black arrow together with the corresponding decision boundary (black line), in which the arrow points towards the decision region which classified as belonging to the red class. The data point circled in green is misclassified and so its feature vector is added to the current weight vector, giving the new decision boundary shown in the top right plot. The bottom left plot shows the next misclassified point to be considered, indicated by the green circle, and its feature vector is again added to the weight vector giving the decision boundary shown in the bottom right plot for which all data points are correctly classified.

Bishop, Pattern Recogn. and ML.

- Let us assume no intercept (data has been centered)
- For each misclassified points $\boldsymbol{x}_n$ (resp. $\phi(X_n)$), the algorithm adds the pattern of the misclassified point to the weight vector $\boldsymbol{\beta}$

$$
\begin{aligned}
& -(\beta^{k+1})^T \phi_n y_n \\
= & -(\beta^k)^T \phi_n y_n \\
& -(\phi_n y_n)^T \phi_n y_n \\
< & -(\beta^k)^T \phi_n y_n
\end{aligned}
$$

Curse of dimensionality

- Two difficulties in (generalized) linear models are related to the dimension

1. Fitting a simple linear model is tricky in high dimension because when $d \gg n$, the solution is usually not unique. (one possibility is to use regularization)

2. As soon as we consider advanced features, the number of coefficients needed grows non linearly with the dimension. Ex: linear regression model made from degree-3 polynomial features

$$y(\boldsymbol{x}, \boldsymbol{w}) = w_0 + \sum_{i=1}^{D} w_i x_i + \sum_{k=1}^{D} \sum_{j=1}^{D} w_{ij} x_i x_j + \sum_{i,j,k}^{D} w_{i,j,k} x_i x_j x_k$$

Curse of dimensionality

- Those difficulties are part of a general phenomenon known as curse of dimensionality
- The expression curse of dimensionality which which was apparently coined by Bellman (1961) was roriginally eferring to the fact that many algorithms that work fine in low dimension become intractable in high dimension.
- In machine Learning, it refers to much more and in particular, it includes the idea that generalization becomes harder in large dimension because a fixed size training set covers a dwindling fraction of the space as d increases. (see Dave Donoho, *High-Dimensional Data Analysis: The Curses and Blessings of Dimensionality*, Pedro Domingos, *A few useful things to know in machine Learning*)

Curse of dimensionality

- As an example, consider a regression model that would divide the space into subcells.
- As the dimension grows, we would need an exponentially large amount of training data to ensure that the cells are not empty

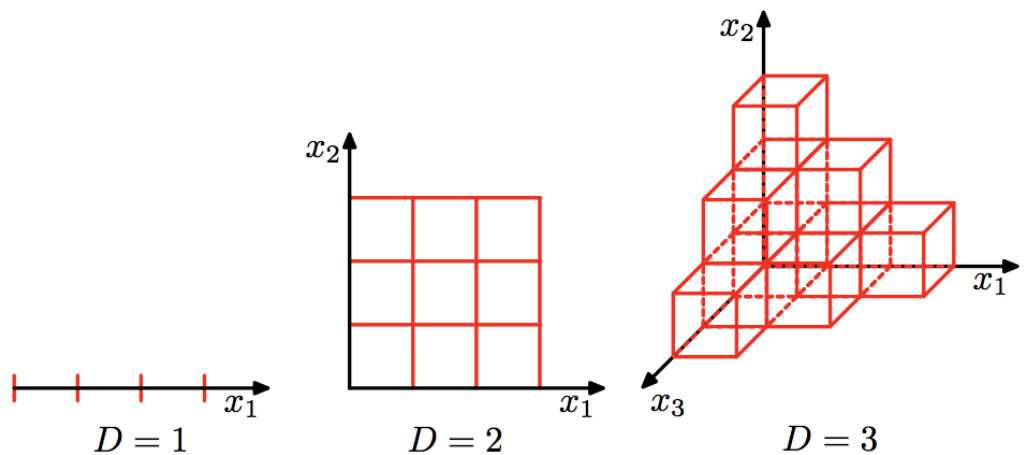


Figure 1.21 Illustration of the curse of dimensionality, showing how the number of regions of a regular grid grows exponentially with the dimensionality D of the space. For clarity, only a subset of the cubical regions are shown for $D = 3$.

Bishop, Pattern Recognition and ML

Kernels

- There are two main uses of kernels in machine learning
- Either as a way to encode similarity between inputs
  - So far we have assumed that the data could be represented by means of some feature vector $X = (X_1, X_2, \ldots, X_n)$
  - Computing explicit features is often difficult and we only have access to some form of similarity/dissimilarity between the samples (think of comparing texts for example)
  - Kernel trick = turning classification/regression models based on features to models based on similarity/kernel
- Kernels can also be used for localization, to build smooth (generative) models for regression and classification (Kernel smoothing)

Kernels

- We define a kernel to be a function of two arguments $\kappa(\boldsymbol{x}, \boldsymbol{x}')$ that is symmetric, $\kappa(\boldsymbol{x}, \boldsymbol{x}') = \kappa(\boldsymbol{x}', \boldsymbol{x})$ and non negative $\kappa(\boldsymbol{x}, \boldsymbol{x}') \geq 0$ (I.e the idea is to use it as a measure of similarity)
- Two important examples are the Gaussian kernel and the class of RBF kernels

$$\kappa(\boldsymbol{x}', \boldsymbol{x}) = \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}')\boldsymbol{\Sigma}(\boldsymbol{x} - \boldsymbol{x}')\right)$$

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

(σ here is called the bandwidth)

Kernels

- When used to encode similarity between points, Kernel are often represented through their corresponding Gram matrix

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

- If the only thing we can compute with the data is the similarity matrix $\boldsymbol{K}$, it is usually hard to extract the feature map from $\boldsymbol{K}$.

- However, there is an important class of kernels for which the existence of such feature map is guaranteed

- For these kernels, the feature vector can be computed from the Gram matrix

Mercer Kernels

- A Mercer Kernel is a Kernel for which the Gram matrix $\boldsymbol{K}$ is positive definite for any set of inputs $\{\boldsymbol{x}_i\}$
- When the kernel is Mercer, there always exists a feature mapping $\phi(\boldsymbol{x})$ such that

$$\boldsymbol{K}_{ij} = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$$

- In other words, the kernel corresponds to an inner product in some finite feature space.
- Mercer Kernel are especially useful when used in Support Vector Machines (SVM) because they guarantee that there exists a unique optimal separating hyperplane in the feature space.

Kernel machines

- There are two main approaches to define models based on kernels
  1. Use any generalized linear model where you replace the features by kernels centered at centroids $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K$ (Kernel machine)

     $$\hat{f}(\boldsymbol{x}) = \sum_{n=1}^{K} w_n \kappa(\boldsymbol{x}, \boldsymbol{\mu}_n)$$

     When using Radial basis functions, this model is known as RBF network
  2. Instead of building a new model from a feature vector defined in terms of kernels, one can instead start from existing models and replace all inner products $\langle \boldsymbol{x}, \boldsymbol{x}' \rangle$ by a call to the Kernel function $\boldsymbol{\kappa}(\boldsymbol{x}, \boldsymbol{x}')$. This idea is known as the Kernel trick

Kernel machines

- The main issue with kernel machines is how to choose the centroids
- When in low dimension, one can choose the centroids to uniformly tile the space (but this will fail in high dimension because of the curse of dimensionality)
- Another approach could be to optimize over the centroids (but the problem is highly multimodal $\Rightarrow$ finding the optimum is hard)
- One could also start by finding clusters in the data and then assign a centroid to each cluster (That would require choosing a number of clusters)

Kernel machines

- Finally a last approach, which is the simplest is to take each sample $\boldsymbol{x}_i$ as a prototype. The feature vector is then given by

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

- The advantage is that the model is now fully non parametric
- But it requires many kernels
- One solution to select a subset of these kernels is to use any of the regularization penalties that we have studied in inear regression
- This gives models known as L1 or L2 regularized Vector Machines

The Kernel trick (I)

- The Kernel trick starts with existing models and tries to replace all inner products in those models with a call to the Kernel to define to define a corresponding version of these models that would rely only on similarity
- For this trick to work, the kernel should be a Mercer kernel
- As an example, consider Ridge regression. In this model the objective is given by

$$J(\boldsymbol{w}) = (\boldsymbol{y} - \boldsymbol{X}\beta)^T(\boldsymbol{y} - \boldsymbol{X}\beta) + \lambda\|\beta\|$$

And the optimal solution can be computed exactly (cfr assignment) as

$$\beta = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}_D)^{-1}\boldsymbol{X}^T\boldsymbol{y} = \left(\sum_{i=1}^{N}\boldsymbol{x}_i\boldsymbol{x}_i^T + \lambda\boldsymbol{I}_D\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

Here $\boldsymbol{X}$ is the $N \times D$ design matrix encoding the points

The Kernel trick (II)

$$\boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}_D)^{-1}\boldsymbol{X}^T\boldsymbol{y} = \left(\sum_{i=1}^{N} \boldsymbol{x}_i\boldsymbol{x}_i^T + \lambda \boldsymbol{I}_D\right)^{-1} \boldsymbol{X}^T\boldsymbol{y}$$

- By using an inversion trick, one can write $\boldsymbol{\beta}$ equivalently as

$$\boldsymbol{\beta} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda \boldsymbol{I}_N)^{-1}\boldsymbol{y}$$

- But now $\boldsymbol{X}\boldsymbol{X}^T$ is exactly the Gram matrix $\boldsymbol{K}$, i.e we can write

$$\boldsymbol{\beta} = \boldsymbol{X}^T(\boldsymbol{K} + \lambda \boldsymbol{I}_N)^{-1}\boldsymbol{y}$$

The Kernel trick (III)

$$\boldsymbol{\beta} = \boldsymbol{X}^T(\boldsymbol{K} + \lambda \boldsymbol{I}_N)^{-1}\boldsymbol{y}$$

- Now let $\boldsymbol{\beta} = \boldsymbol{X}^T\boldsymbol{\alpha}$ with $\boldsymbol{\alpha} = (\boldsymbol{K} + \lambda \boldsymbol{I}_N)^{-1}\boldsymbol{y}$ and substitute this in the regression model, we get

$$\begin{aligned}\hat{f}(\boldsymbol{x}) = \boldsymbol{\beta}^T\boldsymbol{x} = \boldsymbol{\alpha}^T\boldsymbol{X}\boldsymbol{x} &= \sum_{i=1}^{N} \alpha_i \langle \boldsymbol{x}_i, \boldsymbol{x} \rangle \\ &= \sum_{i=1}^{N} \alpha_i \kappa(\boldsymbol{x}, \boldsymbol{x}_i)\end{aligned}$$

The Kernel trick (III)

$$\hat{f}(\boldsymbol{x}) = \sum_{i=1}^{N} \alpha_i \kappa(\boldsymbol{x}, \boldsymbol{x}_i), \quad \boldsymbol{\alpha} = (\boldsymbol{K} + \lambda \boldsymbol{I}_N)^{-1} \boldsymbol{y}$$

- The cost of computing the dual variables $\boldsymbol{\alpha}$ is $\mathcal{O}(N^3)$ whereas the cost of computing the primal variables $\boldsymbol{\beta}$ is $\mathcal{O}(D^3)$. The kernel method is thus essentially useful in high dimension
- However, prediction using the dual variables α takes $\mathcal{O}(ND)$ time whether prediction using the primal variables $\boldsymbol{\beta}$ takes $\mathcal{O}(D)$ time. Making $\boldsymbol{\alpha}$ sparse (few non zero entries) can speed up prediction $\Rightarrow$ that is precisely the point of SVMs !

Usage

- There are two main frameworks in which you might want to use Kernels as a way to encode similarity
  1. You have some data and you've come up with a function which you think might be a good way to encode similarity of your data
  2. You don't want to explicitely deal with feature vectors
  3. $D > N$
- Kernels are especially useful when combined with Support Vector Machines (see later slides)

Smoothing Kernels

- A smoothing kernel is a function $\kappa(\boldsymbol{x})$ of one argument which decreases sufficiently quickly

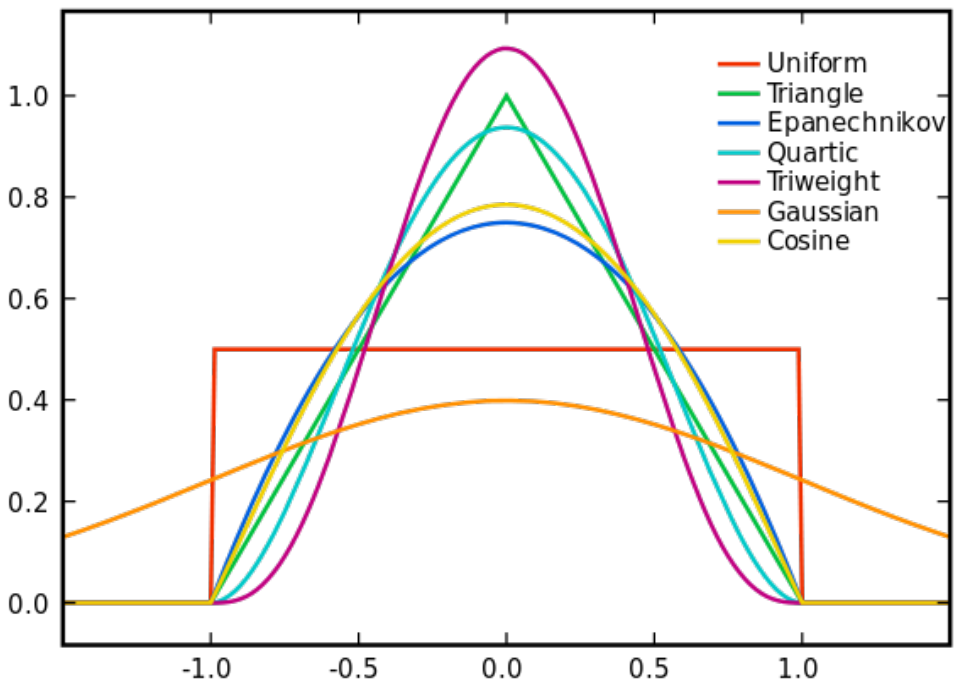
$$\int \kappa(x)\, dx = 1, \quad \int x\kappa(x)\, dx = 0, \quad \text{and} \quad \int x^2\kappa(x)\, dx > 0$$

- A useful example is the Gaussian kernel

$$\kappa(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

- When the inputs are vectors, we can simply take any kernel and use it with the norm $\|\boldsymbol{x}\|$, (e.g. $\|\boldsymbol{x}\|^2 = \sum_{n=1}^{N} X_n^2$)

Smoothing Kernels: examples



Smoothing Kernels

- Smoothing kernels are essentially used in local regression models to get an approximation for the conditional expectation $\mathbb{E}\{y|\boldsymbol{x}\}$
- In this framework, the kernels are used to approximate $p(\boldsymbol{x}, y)$

$$\mathbb{E}\{y|\boldsymbol{x}\} = \int yp(y|\boldsymbol{x})\, dy = \frac{\int p(\boldsymbol{x}, y)y\, dy}{\int p(\boldsymbol{x}, y)\, dy}$$

- Then use smoothing kernels to define an approximation of the

$$p(\boldsymbol{x}, y) \approx \frac{1}{N}\sum_{i=1}^{N} \kappa_x(\boldsymbol{x} - \boldsymbol{x}_i)\kappa_y(y - y_i)$$

- The model is well defined as the Kernels integrate to 1 ($p(\boldsymbol{x}, y)$ corresponds to a probability)

Smoothing Kernels

$$p(\boldsymbol{x}, y) \approx \frac{1}{N} \sum_{i=1}^{N} \kappa_x(\boldsymbol{x} - \boldsymbol{x}_i)\kappa_y(y - y_i)$$

- Using the kernel decomposition for the joint probability distribution, we can write the conditional expectation (which gives one possible regression model) as

$$\begin{aligned}
\mathbb{E}\{y|\boldsymbol{x}\} &= \int y p(y|\boldsymbol{x})\, dy = \frac{\int p(\boldsymbol{x}, y) y\, dy}{\int p(\boldsymbol{x}, y)\, dy} \\
&= \frac{\frac{1}{N}\sum_{i=1}^{N} \kappa_x(\boldsymbol{x} - \boldsymbol{x}_i) \int y \kappa_x(y - y_i)\, dy}{\frac{1}{N}\sum_{i=1}^{N} \kappa_x(\boldsymbol{x} - \boldsymbol{x}_i) \int \kappa_y(y - y_i)\, dy} \\
&= \frac{\sum_{i=1}^{N} \kappa_x(\boldsymbol{x} - \boldsymbol{x}_i) y_i}{\sum_{i=1}^{N} \kappa_x(\boldsymbol{x} - \boldsymbol{x}_i)}
\end{aligned}$$

Smoothing Kernels

$$\mathbb{E}\{y|\boldsymbol{x}\} = \frac{\sum_{i=1}^{N} \kappa_x(\boldsymbol{x} - \boldsymbol{x}_i) y_i}{\sum_{i=1}^{N} \kappa_x(\boldsymbol{x} - \boldsymbol{x}_i)}$$

- The result above follows from the properties of smoothing kernels. In particular we use the zero mean property
  To get $\int y \kappa_y(y - y_i) dy = y_i$, let $y' = y - y_i$ and use

$$\int x\kappa(x)\, dx = 0 \;\Rightarrow\; \int (y' + y_i)\kappa_y(y)\, dy'$$
$$= \int y'\kappa_y(y')\, dy' + y_i \int \kappa_y(y') dy'$$
$$= 0 + y_i = y_i$$

Smoothing Kernels

- In other words, smoothing kernels give us a (non parametric) regression model of the form

$$\hat{f}(\boldsymbol{x}) = \sum_{i=1}^{N} w_i(\boldsymbol{x}) y_i$$

where the weight functions are defined from the smoothing kernels as

$$w_i(\boldsymbol{x}) = \frac{\kappa_x(\boldsymbol{x} - \boldsymbol{x}_i)}{\sum_{j=1}^{N} \kappa_x(\boldsymbol{x} - \boldsymbol{x}_i)}$$

- The prediction is now given by a weighted combination of the outputs at the training points
- This method is known as Nadaraya-Watson model (a.k.a Kernel regression)

Kernels (summary)

- Kernels as a way to replace features with similarity
  - Use Kernels machines when you don't know which features to use. We will see an example of this with SVMs.
  - Generally speaking, Kernels are interesting when $d > n$
  - When $d \ll n$, (multi)linear regression is more interesting
  - When $d \gg n$ there is no need to bring the data in higher dimensional space as finding a separating hyperplane should not be hard
- Smoothing Kernels
  - Essentially used in kernel regression or local regression models
- Kernels are also used in unsupervised learning (will discuss that later)

Support vector machines

- Linear models have interesting computational and anlytical properties but their practical applicability is limited by the curse of dimensionality
- Support vector machines are also called sparse vector machines
- SVM start by defining basis functions that are centered on the data and then select a subset of these during training

Support vector machines

- Consider the linear regression model

$$y(\boldsymbol{x}) = \boldsymbol{\beta}^T \phi(\boldsymbol{x}) + \beta_0$$

- Assume we want to do classification so the labels are $t_n = \{\pm 1\}$
- We further assume that the dataset of linearly separable in feature space so that there exist at least one seprating hyperplane with $\boldsymbol{\beta}^T \boldsymbol{\phi}(\boldsymbol{x}_n) + \beta_0 > 0$ for the $\boldsymbol{x}_n$ with $t_n > 0$ and $\boldsymbol{\beta}^T \boldsymbol{\phi}(\boldsymbol{x}_n) + \beta_0 < 0$ otherwise
- When there are multiple choices we should choose the one that gives the smallest generalization error. SVM tries to achieves this through the notion of margin

Support vector machines

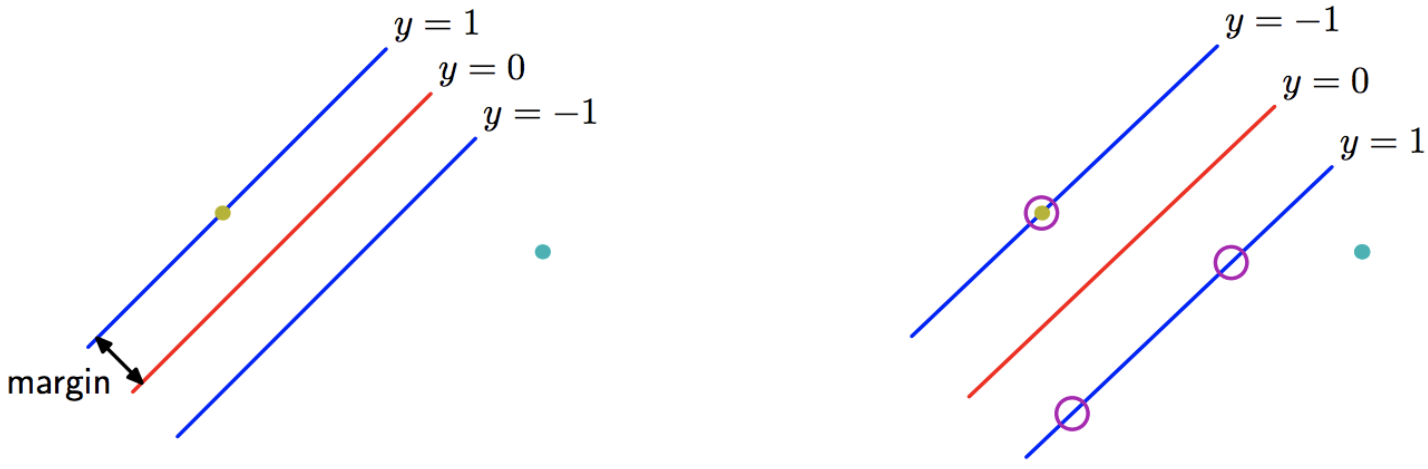


Figure 7.1 The margin is defined as the perpendicular distance between the decision boundary and the closest of the data points, as shown on the left figure. Maximizing the margin leads to a particular choice of decision boundary, as shown on the right. The location of this boundary is determined by a subset of the data points, known as support vectors, which are indicated by the circles.

(Bishop, Pattern recognition and Machine Learning)

SVM as Maximum Margin Classifier

- Recall from the geometry of separating hyperplanes that the distance of a point $\phi(\boldsymbol{x}_n)$ to the hyperplane $\boldsymbol{\beta}^T\boldsymbol{x} + \beta_0$ is defined as $|y(\boldsymbol{x})|/\|\boldsymbol{\beta}\|$
- When all the points are correctly classified, the sign of t_n equals the sign of $\boldsymbol{y}_n = \boldsymbol{\beta}^T\phi(\boldsymbol{x}_n) + \beta_0$ and we can thus write the distance as

$$\frac{t_n y_n}{\|\boldsymbol{\beta}\|} = \frac{t_n(\boldsymbol{\beta}^T\phi(\boldsymbol{x}_n) + \beta_0)}{\|\boldsymbol{\beta}\|}$$

- The margin is the perpendicular distance of the closest point $\phi(\boldsymbol{x}_n)$ to the plane

SVM as Maximum Margin Classifier

$$\frac{t_n y_n}{\|\boldsymbol{\beta}\|} = \frac{t_n(\boldsymbol{\beta}^T \boldsymbol{\phi}(\boldsymbol{x}_n) + \beta_0)}{\|\boldsymbol{\beta}\|}$$

- The maximum margin solution is thus given by

$$\underset{\boldsymbol{\beta}, \beta_0}{\text{argmax}} \left\{ \frac{1}{\|\boldsymbol{\beta}\|} \min_n \left[t_n(\boldsymbol{\beta}^T \phi(\boldsymbol{x}_n) + \beta_0) \right] \right\}$$

- We don't want to solve this problem because deriving a direct solution in this framework would be difficult

SVM as Maximum Margin Classifier

$$\underset{\boldsymbol{\beta},\beta_0}{\text{argmax}} \left\{ \frac{1}{\|\boldsymbol{\beta}\|} \min_n \left[t_n(\boldsymbol{\beta}^T \phi(\boldsymbol{x}_n) + \beta_0) \right] \right\}$$

- First note that for any rescaling $\boldsymbol{\beta} \leftarrow \alpha\boldsymbol{\beta}$, $\beta_0 \leftarrow \beta_0\alpha$, the objective $t_n(\boldsymbol{\beta}^T \boldsymbol{\phi}(\boldsymbol{x}_n) + \beta_0)/\|\boldsymbol{\beta}\|$ is unchanged
- We can thus focus on one of these solution (fix one particular scale for $[\boldsymbol{\beta}, \beta_0]$) as all the others give the same objective
- In particular we can choose to fix the scale by setting

$$t_n(\boldsymbol{\beta}^T \boldsymbol{\phi}(\boldsymbol{x}_n) + \beta_0) = 1$$

For the point that is the closest to the boundary.

SVM as Maximum Margin Classifier

$$\underset{\boldsymbol{\beta},\beta_0}{\text{argmax}} \left\{ \frac{1}{\|\boldsymbol{\beta}\|} \min_n \left[t_n(\boldsymbol{\beta}^T \phi(\boldsymbol{x}_n) + \beta_0) \right] \right\}$$

- Now all the other points will necessarily satisfy

$$t_n(\boldsymbol{\beta}^T \phi(\boldsymbol{x}_n) + \beta_0) \geq 1$$

- Because we fixed the distance of the closest point to the plane the original optimization problem reduces to

$$\underset{\boldsymbol{\beta}}{\text{argmin}} \frac{1}{2} \|\boldsymbol{\beta}\|^2$$

together with the constraint $t_n(\boldsymbol{\beta}^T \phi(\boldsymbol{x}_n) + \beta_0) \geq 1$

SVM as Maximum Margin Classifier

$$\begin{aligned} \underset{\boldsymbol{\beta}}{\text{argmin}} \quad & \frac{1}{2}\|\boldsymbol{\beta}\|^2 \\ \text{subject to} \quad & t_n(\boldsymbol{\beta}^T\phi(\boldsymbol{x}_n) + \beta_0) \geq 1 \end{aligned}$$

- This constrained optimization problem can be recast as an unconstrained problem by introducing multipliers $\lambda_n \geq 0$

$$L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\lambda}) = \frac{1}{2}\|\boldsymbol{\beta}\|^2 - \sum_{n=1}^{N} \lambda_n \left\{ t_n \left(\boldsymbol{\beta}^T \boldsymbol{\phi}(\boldsymbol{x}_n) + \beta_0 \right) - 1 \right\}$$

(see for example Appendix E in Bishop, Pattern Recognition and Machine Learning)

SVM as Maximum Margin Classifier

$$L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\lambda}) = \frac{1}{2}\|\boldsymbol{\beta}\|^2 - \sum_{n=1}^{N} \lambda_n \left\{ t_n \left(\boldsymbol{\beta}^T \boldsymbol{\phi}(\boldsymbol{x}_n) + \beta_0 \right) - 1 \right\}$$

- To find the minimum of this function, we set the derivatives with respect to $\boldsymbol{\beta}$ and β_0 to zero, getting

$$\boldsymbol{\beta} = \sum_{n=1}^{N} \lambda_n t_n \boldsymbol{\phi}(\boldsymbol{x}_n)$$

$$0 = \sum_{n=1}^{N} t_n \lambda_n$$

- and maximize with respect to λ_n (large λ_n penalize the constraint a lot if it becomes negative)

SVM as Maximum Margin Classifier

- Eliminating $\boldsymbol{\beta}$ and β_0 from $L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\lambda})$, we get

$$L(\boldsymbol{\lambda}) = \sum_{n=1}^{N} \lambda_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \lambda_n \lambda_m t_n t_m \kappa(\boldsymbol{x}_n, \boldsymbol{x}_m)$$

- with the constraints

$$\sum_{n=1}^{N} \lambda_n t_n = 0$$

$$\lambda_n \geq 0$$

- Maximizing $L(\boldsymbol{\lambda})$ under the constraints above is a quadratic programming problem for which efficient techniques exist. Moreover when $\boldsymbol{\kappa}(\boldsymbol{x}_n, \boldsymbol{x}_m)$ is Mercer, there is a single solution

SVM as Sparse Kernel Machines(I)

- Given the function $L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\lambda})$, it is known (Karush-Kuhn-Tucker conditions) that any optimal solution must satisfy the follosing 3 conditions

$$\begin{aligned} \lambda_n &\geq 0 \\ t_n y(\boldsymbol{x}_n) - 1 &\geq 0 \\ \lambda_n \{t_n y(\boldsymbol{x}_n) - 1\} &= 0 \end{aligned}$$

- The last conditions have a very important consequence on SVM
- Either $\lambda_n = 0$ or $t_n y(\boldsymbol{x}_n) = 1$ (support vectors)

SVM as Sparse Kernel Machines (II)

- Either $\lambda_n = 0$ or $t_n y(\boldsymbol{x}_n) = 1$ (support vectors)
- In particular many λ_n will be zero
- Using $\boldsymbol{\beta} = \sum_{n=1}^{N} \lambda_n t_n \phi(\boldsymbol{x}_n)$, and substituting it in $y(\boldsymbol{x}) = \boldsymbol{\beta}^T \phi(\boldsymbol{x}) + \beta_0$, we get the prediction model

$$y(\boldsymbol{x}) = \sum_{n=1}^{N} \lambda_n t_n \kappa(\boldsymbol{x}, \boldsymbol{x}_n) + \beta_0$$

- Which is a combination of the λ_n !

SVM as Sparse Kernel Machines (III)

$$\text{(Karush-Kuhn-Tucker)} \quad \begin{cases} \lambda_n \geq 0 \\ t_n y(\boldsymbol{x}_n) - 1 \geq 0 \\ \lambda_n \{t_n y(\boldsymbol{x}_n) - 1\} = 0 \end{cases}$$

- Using the Karush-Kuhn-Tucker conditions, the SVM prediction model thus reduces to

$$y(\boldsymbol{x}) = \sum_{n \in \mathcal{S}} \lambda_n t_n \kappa(\boldsymbol{x}, \boldsymbol{x}_n) + \beta_0$$

Where $\mathcal{S}$ are the support vectors (all remaining λ_n's are 0)

SVM as Sparse Kernel Machines (IV)

- This sparsity property (i.e the need to only keep a small number of support vectors) is a key property of SVM
- It guarantees efficiency of the prediction step !
- Once you know the support vectors, β and β_0 can be computed using $\boldsymbol{\beta} = \sum_{n=1}^{N} \lambda_n t_n \phi(\boldsymbol{x}_n)$, as well as the fact that at any of the support vectors we must have

$$t_n y_n = t_n \left(\sum_{m \in \mathcal{S}} \lambda_m t_m \kappa(\boldsymbol{x}_n, \boldsymbol{x}_m) + \beta_0 \right) = 1$$

- Sometimes we average the estimate for β_0 over the support vectors (here n) to get more stability

Short summary

- General geometry of separating hyperplane, distance to hyperplane, perceptron
- Curse of dimensionality
- Kernels
  - As a way to encode similarity rather than features
  - As smooth interpolating functions
- SVM
  - Maximum Margin
  - Sparse Kernel machines $\Rightarrow$ efficient prediction