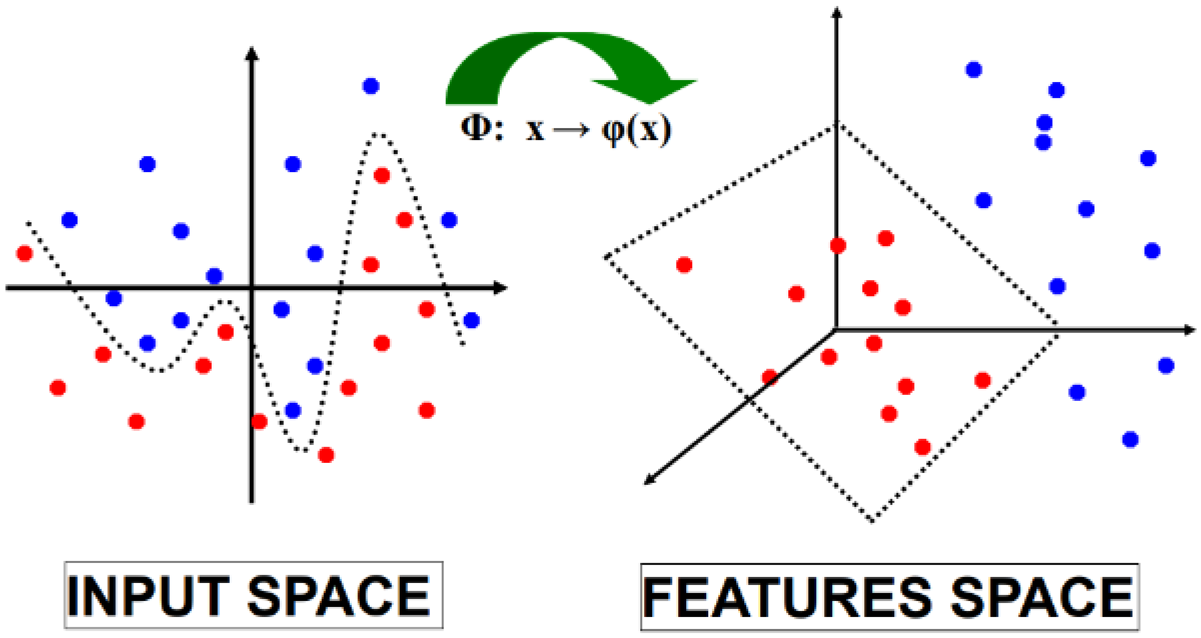
Kernel Methods

Introduction to Kernel Methods

Often we want to capture nonlinear patterns in the data

- Nonlinear Regression: input-output relationship may not be linear
- Nonlinear Classification: Classes may not be separable by a linear boundary
  Linear models are not just rich enough, it is not feasible for the complexity of the model.
  Kernel methods allow to make linear models work in nonlinear settings by mapping data to higher dimensions where it exhibits linear patterns.
  We want to map a non linear input space in a linearly separable feature space



And then we want to find a linear separator. This come with a price that is the increasing the number of features effectively needed to solve our problem.
This take in some problem:

- Curse of dimensionality! The number of features grows significantly with the number of input variable and the mapping become quickly computationally unfeasible
  Kernels methods deal with this issue: they don't require to explicitly compute the feature mapping, they are expensive but computationally feasible. The idea of kernel methods is finding a way to solve the problem in a way with I'm not affected too much from the dimensionality of the features space.
  The first ingredient is the kernel function: $k(x, x') = \phi(\mathbf{x})^T\phi(\mathbf{x}')$. The idea is to use this function as I know it is able to catch similarities between the samples. This function can help me building a model not affected by the dimensionality of the features space as there are many spaces where I can find a more cheaper representation than computing the dot

product between features vector in the features space. The kernel methods tries to revisit all the techniques seen so far and finding a way to only use the kernel function.

Kernel trick

It is possible to rework the representation of linear models to replace all the terms that involve $\phi(\mathbf{x})$ with other terms that involve only $k(\mathbf{x}, .)$.
In other words, the output of linear model can be computed only on the basis of the similarities between data samples (computed with the kernel function)

Kernel Ridge Regression

- Let's go back to the loss function used for the **ridge regression:**

$$L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\left(\mathbf{w}^T\phi(\mathbf{x}_n) - t_n\right)^2 + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w} = \frac{1}{2}(\mathbf{t} - \mathbf{\Phi}\mathbf{w})^T(\mathbf{t} - \mathbf{\Phi}\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

- To solve it, we set to zero the gradient of L with respect to $\mathbf{w}$:

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \lambda\mathbf{w} - \mathbf{\Phi}^T(\mathbf{t} - \mathbf{\Phi}\mathbf{w}) = 0$$

- Now, instead of solving it for $\mathbf{w}$, let do a variable change:

$$\boxed{\mathbf{a} = \lambda^{-1}(\mathbf{t} - \mathbf{\Phi}\mathbf{w})}$$

$$\mathbf{w} = \mathbf{\Phi}^T\lambda^{-1}(\mathbf{t} - \mathbf{\Phi}\mathbf{w}) = \mathbf{\Phi}^T\mathbf{a}$$

With this we can easily find the relation $\overline{w} = \phi^T\mathbf{a}$. Now we see how I can reapply the new variable in the solution.

- Now we can replace $\mathbf{w}$ in the gradient:

$$\boxed{\mathbf{w} = \mathbf{\Phi}^T\mathbf{a}}$$

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \lambda\boxed{\mathbf{w}} - \mathbf{\Phi}^T(\mathbf{t} - \mathbf{\Phi}\boxed{\mathbf{w}}) = 0$$

$$\Rightarrow \quad \mathbf{\Phi}^T\left(\lambda\mathbf{a} - \left(\mathbf{t} - \mathbf{\Phi}\mathbf{\Phi}^T\mathbf{a}\right)\right) = 0$$

$$\Rightarrow \quad \mathbf{\Phi}\mathbf{\Phi}^T\mathbf{a} + \lambda\mathbf{a} = \mathbf{t}$$

$$\Rightarrow \quad \mathbf{a} = (\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{t}$$

$$\boxed{\mathbf{K} = \mathbf{\Phi}\mathbf{\Phi}^T}$$

Gram Matrix

The matrix Φ is composed of all the vector samples and the matrix dimensionality is NxM. Φ^T each column is a feature vector for a sample so the dimensionality is MxN. So the result will be a NxN matrix called K. K will contains all the kernel function of the various sample vector.

The **Gram matrix** is a NxN matrix, where each element is the inner product between the feature vectors:

$$K = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \dots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \dots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$$

$$\boxed{k(\mathbf{x}_n, \mathbf{x}_m) = \phi(\mathbf{x}_n)^T \phi(\mathbf{x}_m)}$$

- K matrix represents the **similarities** between each pair of samples in the training data

How can we compute the prediction using the dual represantation?

$$\boxed{\mathbf{w} = \mathbf{\Phi}^T \mathbf{a}}$$

$$y(\mathbf{x}) = \boxed{\mathbf{w}}^T \phi(\mathbf{x}) = \boxed{\mathbf{a}}^T \mathbf{\Phi} \phi(\mathbf{x}) = \mathbf{k}(\mathbf{x})^T (K + \lambda \mathbf{I}_N)^{-1} \mathbf{t}$$

$$\boxed{\mathbf{a} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{t}}$$

We replace w with its new value and then resolve. We obtain a prediction computed as the linear combination of the target values of the samples in the training set. I don't need in any point to compute the feature mapping. I am multiplying a vector by something that would be a Nx1 and this is the target vector. Linear combination of the target value in my dataset.

Original representation
- Requires to compute the inverse of $(\boldsymbol{\Phi}^T \boldsymbol{\Phi} + \lambda \boldsymbol{I}_M)$ which is a MxM matrix
- Is computationally convenient when M is rather small

Dual representation
- Requires to compute the inverse of $(\mathbf{K} + \lambda \boldsymbol{I}_N)$ which is a NxN matrix
- Is computationally convenient when M **is very large or even infinite**
- Does not require to explicitly compute $\boldsymbol{\Phi}$, making it possible to apply this approach also to complex type of data (e.g., graphs, sets, strings, text, etc.)
- The **similarity between data samples** (i.e., the **kernel function**) is generally both less expensive to compute and easy to design than computing $\boldsymbol{\Phi}$

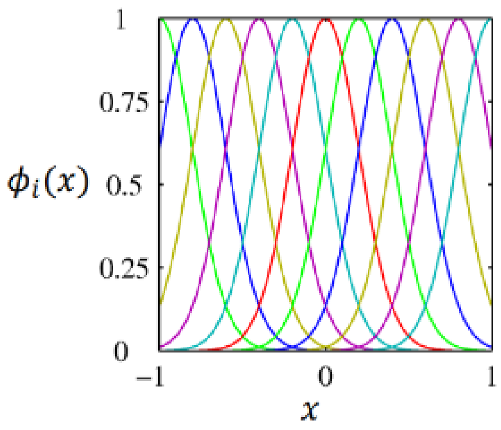
In the original representation we don't need to know how to compute the real space into a feature space but we only need the kernel function. This will open the possibilities to do something more interesting.

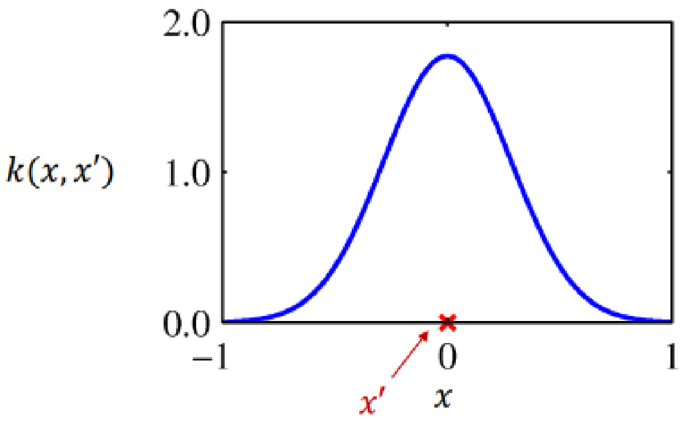
Kernel Function Design

We defined the kernel function as the dot product in the feature space:

$$k(x,x') = \phi(\mathbf{x})^T\phi(\mathbf{x}') = \sum_{i=1}^{M}\phi_i(\mathbf{x})\phi(\mathbf{x}')$$

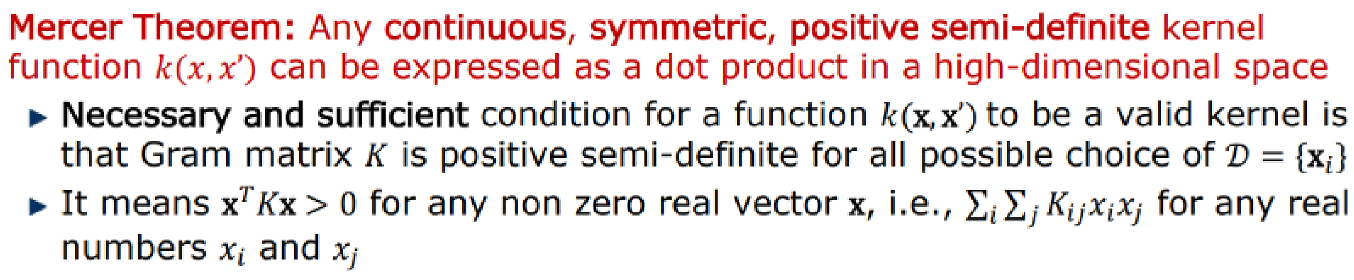
Examples (1D input space)





But the typical way to design kernel. 3 way:

- You know the feature mapping but is cheaper to compute the kernel function
- Design the kernel directly without any knowledge of the corresponding feature space. You need to guarantee that the resulting function is valid(It need to exist a feature space with this kernel function)
- Set of rules with which we can compute a kernel function starting from another one
  In general, we must make sure that the designed kernel functions is valid, that is it correspond to a scalar product into any feature space. The Gram matrix has to be positive semi-definite(For every possible non real zero vector the property holds).



Mercer Theorem: Any **continuous, symmetric, positive semi-definite** kernel function $k(x,x')$ can be expressed as a dot product in a high-dimensional space

- **Necessary and sufficient** condition for a function $k(\mathbf{x},\mathbf{x}')$ to be a valid kernel is that Gram matrix K is positive semi-definite for all possible choice of $\mathcal{D} = \{\mathbf{x}_i\}$
- It means $\mathbf{x}^T K\mathbf{x} > 0$ for any non zero real vector $\mathbf{x}$, i.e., $\sum_i\sum_j K_{ij}x_ix_j$ for any real numbers x_i and x_j

Rules to design valid kernels

Given valid kernels $k_1(x,x')$ and $k_2(x,x')$ the following rules can be applied to design a new valid kernel:

1. k(x, x')=ck_1(x.x') where c > 0 is a constant
2. k(x, x')=f(x)k_1(x, x')f(x') where f(.) is any function
3. k(x, x')=q(k_1(x,x')) where q(.) is a polynomial with non-negative coefficients

4. k(x, x')=exp(k_1(x, x'))
5. k(x, x')=k_1(x, x')+ k_2(x, x')
6. k(x, x')=k_1(x, x')k_2(x, x')
7. k(x, x')=$k_3(\phi(\mathbf{x}), \phi(\mathbf{x}'))$ where

Gaussian Kernel

- This is a commonly used kernel:

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

  - We can check it is a valid kernel, expanding the square:

$$\left\|\mathbf{x} - \mathbf{x}'\right\|^2 = \mathbf{x}^T\mathbf{x} + \mathbf{x}'^T\mathbf{x}' - 2\mathbf{x}^T\mathbf{x}'$$

$$\Rightarrow k(\mathbf{x}, \mathbf{x}') = \exp(-\mathbf{x}^T\mathbf{x}/2\sigma^2)\exp(\mathbf{x}^T\mathbf{x}'/\sigma^2)\exp(-\mathbf{x}'^T\mathbf{x}'/2\sigma^2)$$ Rule 2 and 4

- The feature space corresponding to Gaussian kernel has infinite dimension
- We can extend Gaussian Kernel by replacing $\mathbf{x}^T\mathbf{x}'$ with a nonlinear kernel $\kappa(\mathbf{x}, \mathbf{x}')$:

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2\sigma^2}(\kappa(\mathbf{x}, \mathbf{x}) + \kappa(\mathbf{x}', \mathbf{x}') - 2\kappa(\mathbf{x}, \mathbf{x}'))\right)$$

Its feature space has infinite dimension. It is the most used kernel used among kernel methods techniques. It permit to learn an arbitrary complex problem.

Kernels for Symbolic Data

- Kernel methods can be extended also to inputs different from real vectors, such graphs, sets, strings, texts, etc.
- In fact, the kernel function represents a measure of the similarity between two samples
- A common kernel used for set is:

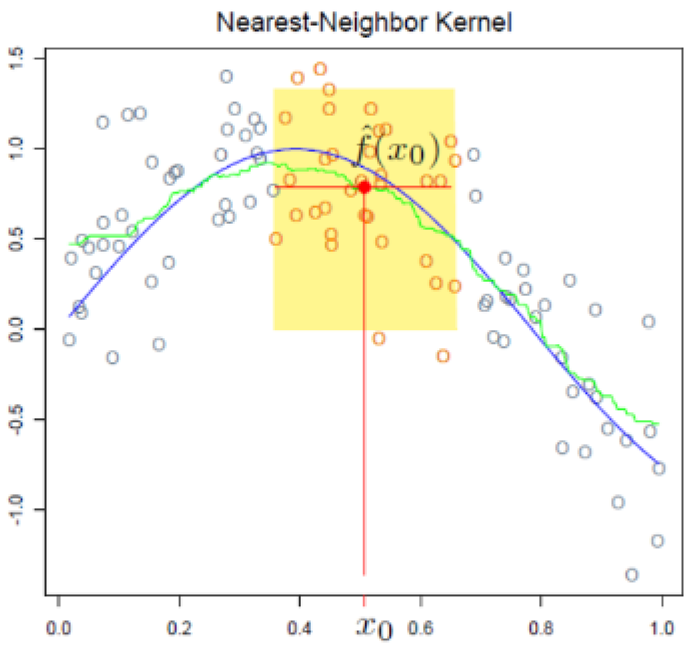
$$k(A_1, A_2) = 2^{|A_1 \cap A_2|}$$

Kernel Regression

k-NN Regression

The k Nearest Neighbour (k-NN) can be applied to solve a regression problem by averaging the K nearest samples in the training data:

$$\hat{f}(\mathbf{x}) = \frac{1}{k}\sum_{\mathbf{x}_i \in N_k(\mathbf{x})} t_i$$



Nadaraya-Watson Model

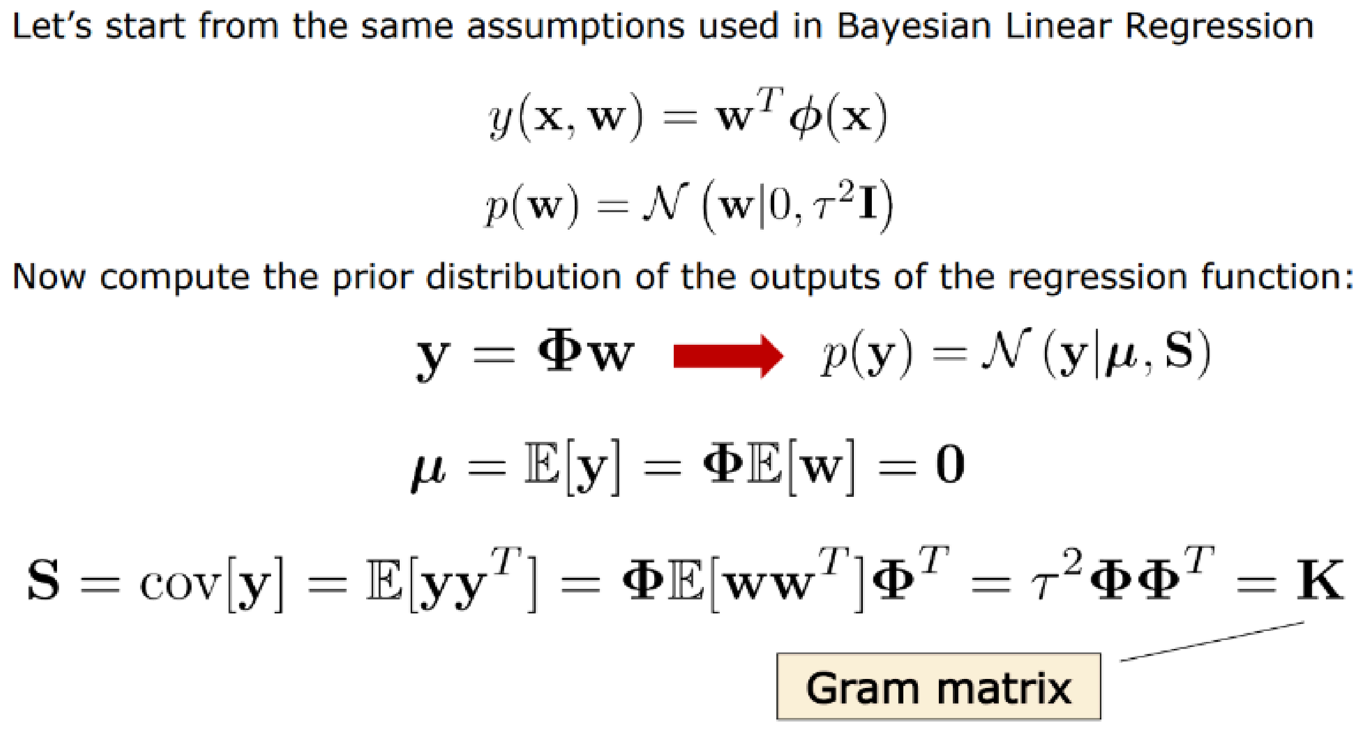
In k-NN regression the model output is generally very noisy due to the discontinuity of neighborhood averages. Nadaraya-Watson model (or kernel regression) deal with this issue by using kernel function to compute a weighted average of samples:

$$\hat{f}(\mathbf{x}) = \frac{\sum_{i=1}^{N} k(\mathbf{x}, \mathbf{x}_i) t_i}{\sum_{i=1}^{N} k(\mathbf{x}, \mathbf{x}_i)}$$

Typical choices for kernels are: Epanechnikov Kernel (bounded support) and Gaussian Kernel (infinite support)

Gaussian Processes

Find a way to generalise Bayesian linear regression to use kernel methods.



Let’s start from the same assumptions used in Bayesian Linear Regression

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \phi(\mathbf{x})$$

$$p(\mathbf{w}) = \mathcal{N}\left(\mathbf{w} | 0, \tau^2 \mathbf{I}\right)$$

Now compute the prior distribution of the outputs of the regression function:

$$\mathbf{y} = \mathbf{\Phi w} \longrightarrow p(\mathbf{y}) = \mathcal{N}\left(\mathbf{y} | \boldsymbol{\mu}, \mathbf{S}\right)$$

$$\boldsymbol{\mu} = \mathbb{E}[\mathbf{y}] = \mathbf{\Phi}\mathbb{E}[\mathbf{w}] = \mathbf{0}$$

$$\mathbf{S} = \operatorname{cov}[\mathbf{y}] = \mathbb{E}[\mathbf{y}\mathbf{y}^T] = \mathbf{\Phi}\mathbb{E}[\mathbf{w}\mathbf{w}^T]\mathbf{\Phi}^T = \tau^2 \mathbf{\Phi}\mathbf{\Phi}^T = \underbrace{\mathbf{K}}_{\text{Gram matrix}}$$

In general, a Gaussian Process is defined as a distribution probability over a function y(x) such that the set of values y(x_i) − for an arbitrary $\{x_i\}$ − jointly have a Gaussian Distribution. In our specific case $p(y) = \mathcal{N}(y|0, \mathbf{K})$ where $K_{nm} = k(\mathbf{x}_n, \mathbf{x}_m) = \tau^2 \phi(\mathbf{x}_n)^T \phi(\mathbf{x}_m)$. This gives a probabilistic interpretation of the kernel function as: $k(\mathbf{x}_n, \mathbf{x}_m) = \mathbb{E}[y(\mathbf{x}_n), y(\mathbf{x}_n)]$. The τ^2 constant is not giving a problem as in the calculus it is irrelevant. This gives a probabilistic interpretation to the model. The Gram matrix is the covariance of the y variable. This implies that the kernel function can give me the correlation between two elements of the space. Different kernel will give different representation of the correlation between two points of the space.
The typical choice of kernel are the Gaussian Kernel or the Exponential Kernel. Gaussian Kernel usually used to resolve the problem of regress function and if you need to find a smooth function if you need also noise variation in the space you should use Exponential Kernel.