Politecnico di Milano

Machine Learning

AA 2019/2020

Summary - Machine Learning

Author:
Valerio Colombo

October 28, 2020

Contents

1 Linear models for regression

The goal of regression is to predict the value of one or more continuous target variables t given the value of a D-dimensional vector x of input variables. The simplest form of linear regression models are also linear functions of the input variables. However, we can obtain a much more useful class of functions by taking linear combinations of a fixed set of nonlinear functions of the input variables, known as basis functions. Such models are linear functions of the parameters, which gives them simple analytical properties, and yet can be nonlinear with respect to the input variables. So the "linear" part in the title refers to the linearity respect to the parameters. So even though the model we are building is linear in the parameter space, it can estimate nonlinear models in input/output space.

1.1 Linear Basis Function Models

1.1.1 Linear regression

The simplest linear model for regression is one that involves a linear combination of the input variables. Given a set comprising M-1 observations x_m, where $m = 1, ..., M-1$ we have,

$$\begin{aligned} &y(x, w) = w_0 + w_1 x_1 + ... + w_{M-1} x_{M-1} = w_0 + \sum_{n=1}^{M-1} (w_n x_n) = w^T x, \\ &where \quad w^T = \begin{bmatrix} w_0 & w_1 & \dots & w_{M-1} \end{bmatrix}, \quad x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_{M-1} \end{bmatrix}^{1} \end{aligned} \tag{1}$$

This is often simply known as linear regression. The key property of this model is that it is a linear function of the parameters $w_0, ..., w_{M-1}$. It is also, however, a linear function of the input variables x_i, and this imposes significant limitations on the model.

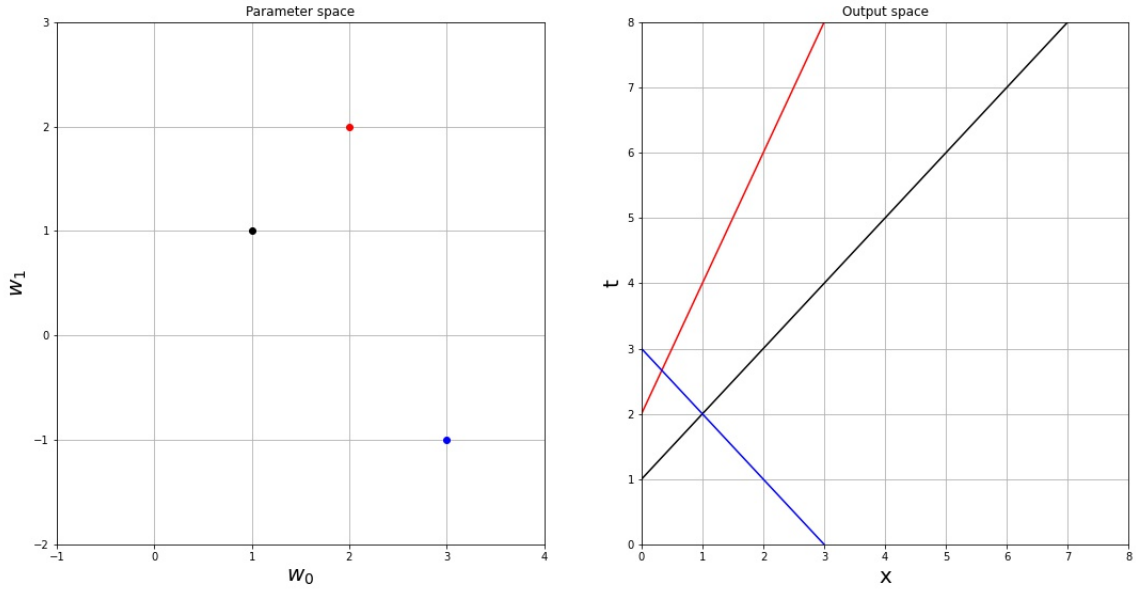
Example We want to estimate the weight of a person based on his age, height and gender So our inputs variables would be

$$\begin{aligned} &age, && x_a \in \mathbb{N} \\ &height, && x_h \in \mathbb{N} \\ &gender, && x_g \in \mathbb{N} \\ &bias, && 1 \end{aligned}$$

$$y(x, w) = \begin{bmatrix} w_0 & w_a & w_h & w_g \end{bmatrix} \begin{bmatrix} 1 \\ x_a \\ x_h \\ x_g \end{bmatrix}$$

[1] Note that we added a dummy sample $x_0 = 1$ to include w_0 in $w^T x$.

Note In linear regression we can distinguish two spaces. Hypothesis space(Parameters space) and Output space(Input space)



1.1.2 Linear regression with non-linear basis functions

We can extend the class of models by considering linear combinations of fixed nonlinear functions of the input variables, of the form

$$y(x, w) = w_0 + \sum_{j=1}^{M-1} (w_j \Phi_j(x)) = w^T \Phi(x),$$
$$where \quad \Phi(x) = \begin{bmatrix} 1 \\ \Phi_1(x) \\ \vdots \\ \Phi_{M-1}(x) \end{bmatrix} \tag{2}$$

By using nonlinear basis functions[2], we allow the function $y(x, w)$ to be a nonlinear function of the input vector x. Functions of the form (2) are called linear models because they are linear in w. It is this linearity in the parameters that will greatly simplify the analysis of this class of models. From a geometric point of view the task of basis functions is to linearize input space "bending" the point into a straight line.

[2]Non-linear functions like: $e^x, sin(x), x^2$,...

1.1.3 Loss functions

To evaluate which model, and so parameters, is the best, we need to quantify what it means to do well or poorly on a task. To do so we use loss functions.

$$L(t, y(x)) \qquad \text{Loss function}$$

$$E[L] = \int\int L(t, y(x))p(x, t)dxdt \qquad \text{Average loss function}$$

Our goal is to find the model $y(x)$ that minimize the loss function $L(t, y(x))$. If we take the Minkowsky loss

$$E[L] = \int\int (t - y(x))^q p(t, x)dtdx \tag{3}$$

Based on q the model that minimize $E[L]$ is

- $q = 2: \quad y(x) = E[t|x]$ Conditional mean. Note $E[t|x] = \int tp(t|x)dt$
- $q = 1: \quad y(x) = median(t|x)$ Conditional median
- $q \to 0: \quad y(x) = mode(t|x)$ Conditional mode

Note For $q = 2$ we have the squared loss function. This is the most used one.

1.2 Least square minimization

The method of least squares is a standard approach in regression analysis to approximate a model by minimizing the sum of the squares of the residuals. Given N samples, we consider the following loss(error) function

$$L(w) = \frac{1}{2}\sum_{n=1}^{N}(y(\underset{n^{th}input}{x_n}, w) - \underset{n^{th}target}{t_n})^2 \tag{4}$$

This loss function is called SSE(Squared Sum of Errors) or half RSS(Residual Sum of Squares). It can be also written as

$$RSS(w) = \|\epsilon\|_2^2 \quad \overset{square}{\underset{l_2norm}{}}, \quad where \quad \epsilon = \begin{bmatrix} y(x_1, w) - t_1 \\ \vdots \\ y(x_N, w) - t_N \end{bmatrix} \tag{5}$$

Note Given $x = \begin{bmatrix} x_1 & x_2 & \dots & x_N \end{bmatrix}$, the l_2norm of x corresponds to

$$l_2norm(x) = \|x\|_2 = \sqrt{\sum_{n=1}^{N} x_i^2} \tag{6}$$

1.2.1 Ordinary Least Squares (Closed Form)

Given N samples and M parameters, we construct $\Phi = \begin{bmatrix}\Phi(x_1) & \dots & \Phi(x_N)\end{bmatrix}^T$ and $t = \begin{bmatrix}t_1 & \dots & t_N\end{bmatrix}^T$.
We can rewrite the SSE in matrix notation

$$L(w) = \frac{1}{2}RSS(w) = \frac{1}{2}(t - \Phi w)^T(t - \Phi w) \quad [1x1]^3 \tag{7}$$

Note $\Phi[NxM]$, $w[Mx1]$, $t[Nx1]$
To minimize L(w) we have to calculate the first and the second derivative

$$\frac{\partial L(w)}{\partial w} = \frac{\partial(\frac{1}{2}(t - \Phi w)^T(t - \Phi w))}{\partial w} = -\Phi^T(t - \Phi w) \quad [Mx1] \tag{8}$$

Note $\frac{\partial L(w)}{\partial w}$ are the directions of every w_i to minimize $L(w)$

$$\frac{\partial L(w)}{\partial w \partial w^T} = \Phi^T\Phi \quad [MxM] \tag{9}$$

Note $\frac{\partial L(w)}{\partial w \partial w^T}$ is a semi-definite positive matrix[4], so all eigen values≥ 0 and it's invertible. It also means that for $\frac{\partial L(w)}{\partial w} = 0$ we find the minimum of $L(w)$

Now we have to find w imposing the first derivative to zero

$$\begin{aligned}
\frac{\partial L(w)}{\partial w} &= 0 \\
-\Phi^T(t - \Phi w) &= 0 \\
-\Phi^T t + \Phi^T\Phi w &= 0 \\
\Phi^T\Phi w &= \Phi^T t \\
\hat{w} &= (\Phi^T\Phi)^{-1}\Phi^T t \quad [Mx1]
\end{aligned} \tag{10}$$

Second derivative can give us some infos about features(basis functions) importance. If we have some eigen values close to zero we could have the following situations

- $N < M$, less samples than dimensions(parameters)
- The feature with null eigen value is linearly dependents from other features

From a computational point of view the matrix invertion is a very complex operation. In fact, the temporal complexity of OLS is

$$O(NM^2 + M^3)$$

[3] The [ ] next to the equations indicates result dimensions
[4] $x^T\Phi^T\Phi x \geq 0, \quad \forall x \in \mathbb{R} \setminus \emptyset$

Geometric interpretation We can give a geometric interpretation of the problem. Given $\hat{t} = \begin{bmatrix} y(x_1, w) & \dots & y(x_N, w) \end{bmatrix}^T$ from the previous calculation we can say that $\hat{t}$ is a linear combination of the column of Φ. So $\hat{t}$ lies in a M-subspace S and since $\hat{t}$ minimize $L(w)$ with respect to t, it represents the projection of t in S

$$\hat{t} = \underbrace{\Phi(\Phi^T\Phi)^{-1}\Phi^T}_{\text{Hat matrix H}} t = Ht, \quad \text{H projects t on S}$$

1.2.2 Maximum Likelihood ML (Closed Form)

We assume that the target variable t is given by a deterministic function $f(x)$ with additive Gaussian noise ϵ so that

$$t = f(x) + \epsilon$$

To estimate t we can make the following assumptions

- Approximate $f(x)$ with $y(x, w)$
- Assume $\epsilon \sim \mathcal{N}(0, \sigma^2)$

Given N i.i.d.[5]samples, with input $X = \{x_1, ..., x_N\}$ and outputs $t = \begin{bmatrix} t_1 & \dots & t_N \end{bmatrix}^T$ we can say that

$$t_i \sim \mathcal{N}(t_i | w^T\Phi(x_i), \sigma^2) \tag{11}$$

For the properties of the Gaussian distribution we can write the following

$$p(t|X, w, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}(t_n | w^T\Phi(x_n), \sigma^2)$$

$$p(t|X, w, \sigma^2) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2}}$$

$$\Downarrow \quad \text{Transition to ln to simplify calculus.}$$
$$\text{Min \& Max remain the same}$$

$$ln(p(t|X, w, \sigma^2)) = ln \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2}}$$

$$= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2}}\right)$$

$$= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right) + ln\left(e^{-\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2}}\right)$$

[5]Independent and Identically Distributed random variables

$$
\begin{aligned}
&= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right) + \sum_{n=1}^{N} ln\left(e^{-\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2}}\right) \\
&= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right) + \sum_{n=1}^{N} -\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2} \\
&= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(t_n - w^T\Phi(x_n))^2 \\
&= \sum_{n=1}^{N} ln\left((2\pi\sigma^2)^{-\frac{1}{2}}\right) - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(t_n - w^T\Phi(x_n))^2 \\
&= -\frac{N}{2}ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\underbrace{\sum_{n=1}^{N}(t_n - w^T\Phi(x_n))^2}_{RSS} \qquad (12)
\end{aligned}
$$

To find the maximum likelihood, we equal the gradient of $ln(p(t|X, w, \sigma^2)) = 0$

$$
\begin{aligned}
\overset{w}{\nabla} ln(p(t|X, w, \sigma^2)) &= \overset{w}{\nabla}\left(-\frac{1}{2\sigma^2}\sum_{n=1}^{N}(t_n - w^T\Phi(x_n))^2\right) \\
&= -\frac{1}{2\sigma^2}\sum_{n=1}^{N} 2(t_n - w^T\Phi(x_n))(-\Phi^T(x_n)) \\
&= -\frac{1}{\sigma^2}\sum_{n=1}^{N} t_n\Phi^T(x_n) - w^T\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n) \\
0 &= \sum_{n=1}^{N} t_n\Phi^T(x_n) - w^T\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n) \\
w^T &= \sum_{n=1}^{N} t_n\Phi^T(x_n)\left(\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n)\right)^{-1} \\
w &= \left(\sum_{n=1}^{N} t_n\Phi^T(x_n)\left(\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n)\right)^{-1}\right)^T \\
&= \left(\left(\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n)\right)^{-1}\right)^T\left(\sum_{n=1}^{N} t_n\Phi^T(x_n)\right)^T \\
&= \left(\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n)\right)^{-1}\sum_{n=1}^{N}\Phi(x_n)t_n \qquad having \quad \Phi = \begin{bmatrix} [\Phi^T(x_1)] \\ \vdots \\ [\Phi^T(x_N)] \end{bmatrix} \\
&= (\Phi^T\Phi)^{-1}\Phi^T t \quad [Mx1] \qquad (13)
\end{aligned}
$$

As we can see both OLS and ML give the same result (13).

W variance estimation As we said before for ML calculation we made some assumptions. The most important for calculating the variance are

- the observations t_i are uncorrelated and have constant variance σ^2
- x_i are fixed(non-random)

Given that the variance-covariance matrix of least-squares estimates is

$$Var(\hat{w}_{ML}) = (\Phi^T\Phi)^{-1}\sigma^2 \tag{14}$$

As we have seen before $\Phi^T\Phi$ matrix can give us some insights on features importance. If a features is relevant its eigen value is high. An effect of this is the reduction of parameters variance. In fact the $\Phi^T\Phi$ matrix is inverted(same as divided) and multiplied to error variance σ, so high eigen values reduce variance. We can achieve variance reduction by gathering more samples for our estimation.

Multiple outputs To solve a regression problem with multiple outputs we could use different sets of basis function for each output, having independents problems. Usually, a single set of basis function is used

$$\hat{W}_{ML} = \underbrace{(\Phi^T\Phi)^{-1}\Phi^T}_{\Phi^\dagger \text{Pseudo-inverse}} T \quad [MxK], \quad where \quad T \quad [NxK] \tag{15}$$

1.2.3 Gradient optimization (Open form)

Closed form solutions are not practical with large datasets because they are computationally too complex. To overcome this problem we can use iterative approaches which make sequential(online) updates of the parameters instead of calculating them directly. an example is stochastic[6]gradient descent. If the loss function can be represented as a sum over samples ($L(x) = \sum_n L(x_n)$) this iterative approach is applicable. This is the case for least squares.

$$w^{\overset{\text{iteration}}{(k+1)}} = w^{(k)} - \underset{\text{learning rate}}{\alpha^{(k)}} \nabla L(x_n) \tag{16}$$

The learning rate α is an important hyperparameter[7]of the model. It defines the length of each iteration step. A big step makes the iterative process converge faster, but is less precise because it can "jump" the minimum that we are looking for. Similarly if we take small steps

[6]Stochastic means that we won't use all data at once to update the parameters, but we can update them from smaller subsets of the dataset. This is done to reduce the complexity.

[7]In machine learning, a hyperparameter is a parameter whose value is used to control the learning process. By contrast, the values of other parameters are derived via training.

we are more precise but we could get stuck in local minima and the convergence is slow. To be sure of algorithm convergence the learning rate must have the following properties

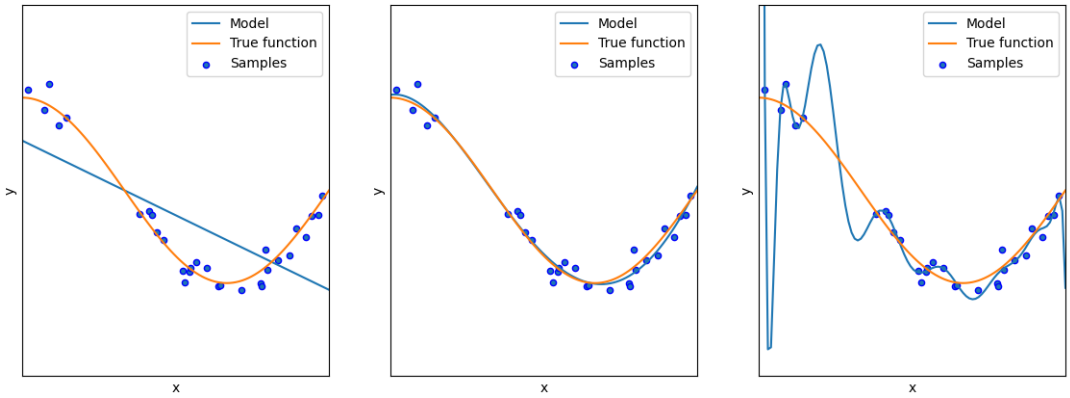
$$\sum_{k=0}^{\infty} \frac{1}{\alpha^{(k)}} = \infty \quad \wedge \quad \sum_{k=0}^{\infty} \frac{1}{{\alpha^{(k)}}^2} = M, \quad M \in \mathbb{R} \tag{17}$$

1.2.4 Underfitting - Overfitting

Model complexity play a very important role for the success of an algorithm. A simple definition of 'model complexity' can be define as the number of parameters and features of a model. The complexity influence model capability to generalize. An optimal model is one that generalize well the problem and generalization is the model's ability to give sensible outputs to sets of input that it has never seen before. We can have either two cases for bad generalization.

- **Underfitting** The model is too simple and it generalize too much. In practice we don't have enough parameter to estimate the true model.
- **Overfitting** The model is too complex and it relies too much on the given dataset. The model will behave very well on our dataset, but will perform poorly on other ones. In practice the model tries to interpolate the dataset learning the true model and the noise of our starting dataset because it has too many parameters.

A naive way to measure the performance of our model is to look at the loss function value. A lower value corresponds to a better performance. This is true, but loss function value lacks of valuable information regarding model complexity. In fact, as we had said before, an overfitting model tries to interpolate the dataset, this will produce a loss function value that tends to zero but the model true performance is far from optimal.



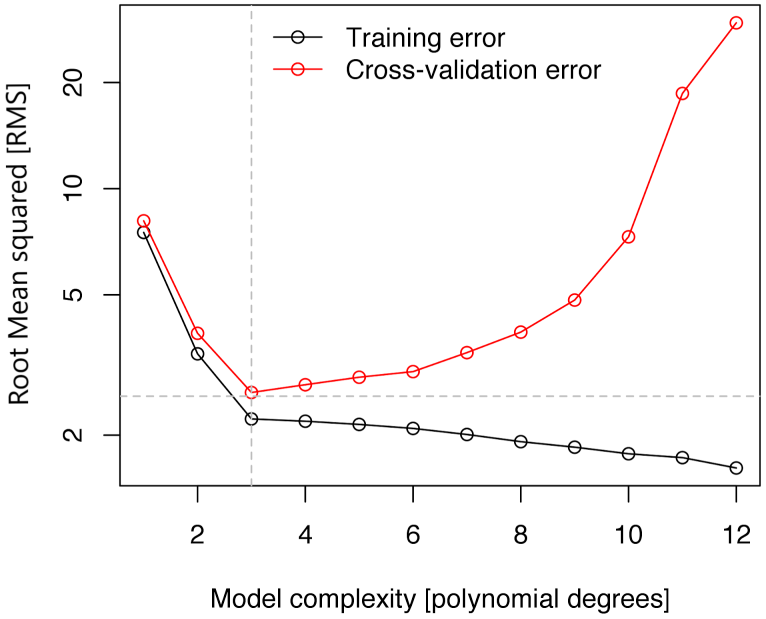
To overcome this problem we can split the dataset in two subset,

- **Training set** This set is used to learn the model's parameters.
- **Test set** This set is used to test model performance on unseen data. This is very helpful for complexity selection.

This technique is called **cross-validation**. A good error function for testing the complexity is

$$E_{RMS} = \sqrt{\frac{2RSS(\hat{w})}{N}} \tag{18}$$

Differently from the loss function used at training time E_{RMS} is not monotonically decreasing with model complexity. It has a U shape and the minimum corresponds to the optimal model complexity.



Another indicator for poor generalization is parameters absolute value. If we have very large parameters it means that the model oscillate very rapidly. This is an evidence of overfitting.

1.3 Regularization

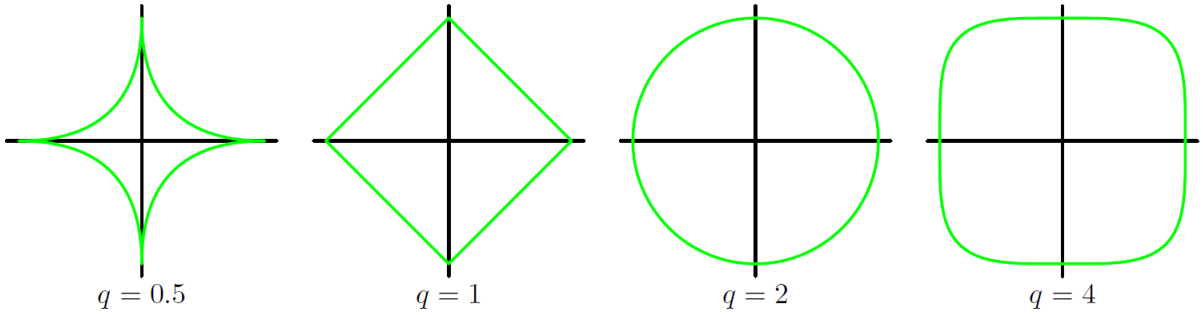
One of the major aspects of training your machine learning model is avoiding overfitting. The model will have a low accuracy if it is overfitting. This happens because your model is trying too hard to capture the noise in your training dataset. One way to avoid overfitting is using cross validation, that helps in estimating the error over test set, and in deciding what parameters work best for your model [1.2.4]. Another way to reduce overfitting is constrain/ regularize or shrink the parameters estimates towards zero. To do so, we can change the loss function

$$L(w) = \underbrace{L_D(w)}_{\text{error on data}} + \underbrace{\lambda L_W(w)}_{\text{error on complexity}} \tag{19}$$

For tractability we take

$$L_D(w) = \frac{1}{2}RSS \qquad\qquad L_W(w) = \frac{1}{2}\sum_{j=1}^{M}|w_j|^q, \quad q \in \mathbb{N}^+$$

As we can see L_W depends on parameter absolute value. This loss function component discourages high parameters values. So a parameters have to justify its high value by giving a very good contribution to L_D, otherwise the model is penalized. The penalization of L_W can be controlled with λ(regularization coefficient) and q. Furthermore we can interpret L_W as a constraint. If we consider the parameters space, L_w is bounding the parameters at a certain distance from the origin. The distance is imposed by λ and q. In particular, q modifies the shape of the boundary(constraint) in the parameters space. The most used value for q are 1 and 2.



Regularization allows complex models to be trained on data sets of limited size without severe over-fitting, essentially by limiting the effective model complexity. However, the problem of determining the optimal model complexity is then shifted from one of finding the appropriate number of basis functions to one of determining a suitable value of the regularization coefficient λ.

1.3.1 Ridge regression

For $q = 2$ we have the so called Ridge regression.

$$\begin{aligned} L_W(w) &= \frac{1}{2}w^T w = \frac{1}{2}\|w\|_2^2 \\ L(w) &= \frac{1}{2}\sum_{j=1}^{N}(t_i - w^T\Phi(x_i))^2 + \frac{\lambda}{2}w^T w \\ &= \frac{1}{2}(t - \Phi w)^T(t - \Phi w) + \frac{\lambda}{2}w^T w \end{aligned} \tag{20}$$

The main advantage of ridge regression is that the loss function is still quadratic in w, so we can still derive a closed form solution.

$$\hat{w}_{ridge} = (\lambda I + \Phi^T\Phi)^{-1}\Phi^T t \tag{21}$$

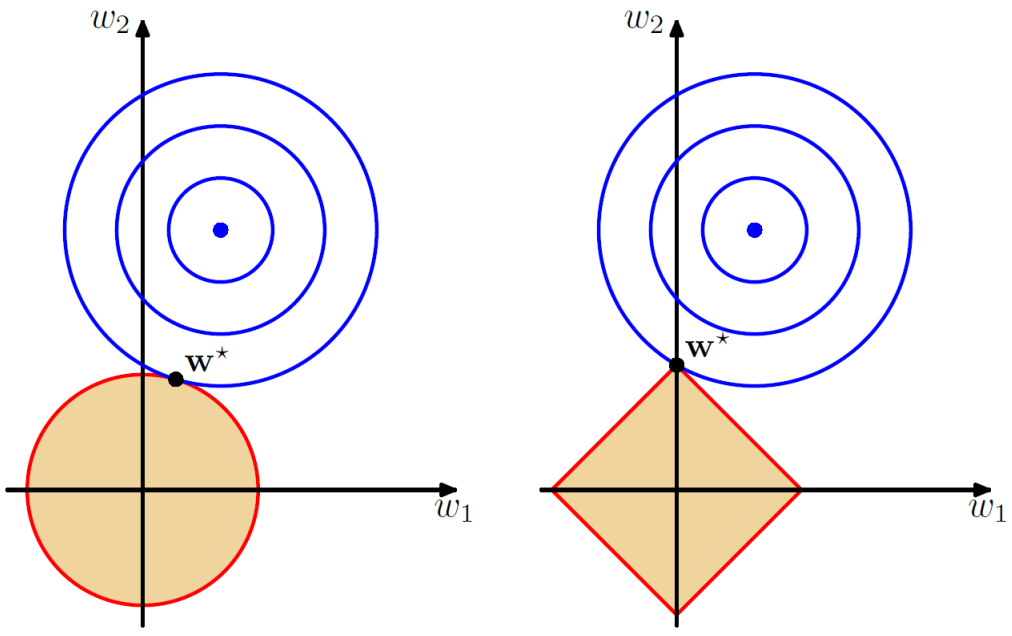
Note The eigen values of $(\lambda I + \Phi^T\Phi)$ are still greater or equal to zero because $\Phi^T\Phi$ is semi-definite positive and λI simply imposes a positive lower bound to the eigen values. $(\lambda I + \Phi^T\Phi)$ is still semi-definite positive and so invertible.

1.3.2 Lasso

For $q = 1$ we have the so called Lasso.

$$
\begin{aligned}
L_W(w) &= \frac{1}{2}\sum_{j=1}^{M}|w_j| = \frac{1}{2}\|w\|_1 \\
L(w) &= \frac{1}{2}\sum_{j=1}^{N}(t_i - w^T\Phi(x_i))^2 + \frac{\lambda}{2}\sum_{j=1}^{M}|w_j| \\
&= \frac{1}{2}\sum_{j=1}^{N}(t_i - w^T\Phi(x_i))^2 + \frac{\lambda}{2}\|w\|_1
\end{aligned} \tag{22}
$$

Differently from ridge regression, lasso is not linear. No closed form solution exists because we have the absolute value operator in L_W. In contrast, a very good advantage is the capability to make some weights equal to zero for values of λ large enough. This means that lasso leads to sparser models[8].



The red lines in the figure represents $\sum_{j=1}^{M}|w_j|^q$ respectively for $q = 2$ and $q = 1$. The blue lines are the unregularized error functions.

[8] A model is sparse if some parameters tend to zero, eliminating some features from the model

1.4 Bayesian Linear regression

In our discussion of maximum likelihood for setting the parameters of a linear regression model, we have seen that the effective model complexity, governed by the number of basis functions, needs to be controlled according to the size of the data set. Adding a regularization term to the log likelihood function means the effective model complexity can then be controlled by the value of the regularization coefficient, although the choice of the number and form of the basis functions is of course still important in determining the overall behaviour of the model. This leaves the issue of deciding the appropriate model complexity for the particular problem, which cannot be decided simply by maximizing the likelihood function, because this always leads to excessively complex models and over-fitting. We therefore turn to a Bayesian treatment of linear regression, which will avoid the over-fitting problem of maximum likelihood, and which will also lead to automatic methods of determining model complexity using the training data alone.

Note Bayes theorem is obviously the hearth of this type of regression. As a reminder, Bayes theorem states

$$\underbrace{P(A|B)}_{Posterior} = \frac{\overbrace{P(B|A)}^{Likelihood}\overbrace{P(A)}^{Prior}}{\underbrace{P(B)}_{Marginalization}} \tag{23}$$

Example We can estimate the probability of getting head or tail with a coin flip. We don't know if the coin is tricked or not. We know that a coin flip follow a Bernoulli distribution

$$P(r) = \begin{cases} p, & \text{if } r = \text{Head} \\ q = 1 - p, & \text{if } r = \text{Tail} \end{cases}$$

- **Prior** $P(r)$ we assume a regular coin so $P(r) = p = \frac{1}{2}$ ($P(Head) = \frac{1}{2}$ and $P(Tail) = \frac{1}{2}$)
- **Posterior** $P(r|D)$ Probability of the coin having $p = \frac{1}{2}$ given the Data.
- **Likelihood** $P(D|r)$ Probability of observing the Data given that the coin have $P = \frac{1}{2}$
- **Marginalization** $P(D)$ Probability of observing the Data

$$P(r|D) = \frac{P(D|r)P(r)}{P(D)}$$

So the Bayesian approach formulate our knowledge as follow,

1. We formulate our knowledge about the world in a probabilistic way
   (a) We define the model that expresses our knowledge qualitatively
   (b) We capture our assumptions about unknown parameters by specifying the prior distribution over those parameters before seeing the data

2. We observe the data

3. We compute the posterior probability distribution for the parameters, given observed data

4. We use the posterior distribution to make predictions or take decisions

As the example we have made before we have

$$P(w|D) = \frac{P(D|w)P(w)}{P(D)}$$

- **Prior** $P(w)$ probability distribution of the parameters
- **Posterior** $P(w|D)$ Probability of w given training data D
- **Likelihood** $P(D|w)$ Probability of observing the Data given parameters w
- **Marginalization** $P(D)$ Probability of observing the Data $P(D) = \int P(D|w)P(w)dw$

Our objective is to find the most probable value of w given the data maximum a posteriori (MAP), which is the mode of the posterior $P(w|D)$.

Note An advantage of the Bayesian approach is the introduction of the prior distribution. This greatly reduce our hypothesis space(parameter space) reducing overfitting. Assuming a Gaussian likelihood model we can take a Gaussian prior. The Gaussian family is conjugate to itself (or self-conjugate) with respect to a Gaussian likelihood function: if the likelihood function is Gaussian, choosing a Gaussian prior over the mean will ensure that the posterior distribution is also Gaussian.

$$P(w) \sim \mathcal{N}(w|w_0, S_0) \tag{24}$$

- w_0 [MX1] Mean of the distribution. We guess that w is equal to w_0 in the parameter space.
- S_0 [MxM] Covariance matrix of the distribution. The matrix is diagonal because we assume i.i.d. parameters.

So P(w) is a multivariate[9]Gaussian. As we have said before, the posterior will be Gaussian

$$P(w|t, \Phi, \sigma^2) \propto \mathcal{N}(w|w_0, S_0)\mathcal{N}(t|\Phi w, \sigma^2 I_N) = \mathcal{N}(w_N, S_N)$$
$$w_N = S_N\left(S_0^{-1}w_0 + \frac{\Phi^T t}{\sigma^2}\right)$$
$$S_N^{-1} = S_0^{-1} + \frac{\Phi^T\Phi}{\sigma^2}$$

[9]In probability theory and statistics, the multivariate normal distribution or multivariate Gaussian distribution is a generalization of the one-dimensional (univariate) normal distribution to higher dimensions.

For two particular prior distribution cases, Bayesian regression estimate reduces to already known regressions,

- $S_0 = \infty I$ In this case we have no prior knowledge of the parameters distribution. If we substitute S_0 in w_N definition we find,

$$\begin{aligned} S_N^{-1} &= S_0^{-1} + \frac{\Phi^T\Phi}{\sigma^2} = \frac{\Phi^T\Phi}{\sigma^2} \\ S_N &= \sigma^2(\Phi^T\Phi)^{-1} \\ w_N &= S_N\left(S_0^{-1}w_0 + \frac{\Phi^T t}{\sigma^2}\right) \\ &= \sigma^2(\Phi^T\Phi)^{-1}\frac{\Phi^T t}{\sigma^2} \\ &= (\Phi^T\Phi)^{-1}\Phi^T t \end{aligned} \tag{25}$$

  As we can see (25) is equal to the ML estimator. So Bayesian regression reduces to the maximum likelihood case.

- $w_0 = 0$, $S_0 = \tau^2 I, \quad \tau \in \mathbb{R}$

$$\begin{aligned} S_N^{-1} &= S_0^{-1} + \frac{\Phi^T\Phi}{\sigma^2} \\ &= \frac{1}{\tau^2}I + \frac{\Phi^T\Phi}{\sigma^2} \\ w_N &= S_N\left(S_0^{-1}w_0 + \frac{\Phi^T t}{\sigma^2}\right) \\ &= S_N\frac{\Phi^T t}{\sigma^2} \\ &= \left(\frac{1}{\tau^2}I + \frac{\Phi^T\Phi}{\sigma^2}\right)^{-1}\frac{\Phi^T t}{\sigma^2} \\ &= \left(\frac{\sigma^2}{\tau^2}I + \Phi^T\Phi\right)^{-1}\Phi^T t \end{aligned} \tag{26}$$

  We can notice that (26) is equal to Ridge regression with $\lambda = \frac{\sigma^2}{\tau^2}$. Small values of S_0 corresponds to high values of λ and viceversa.

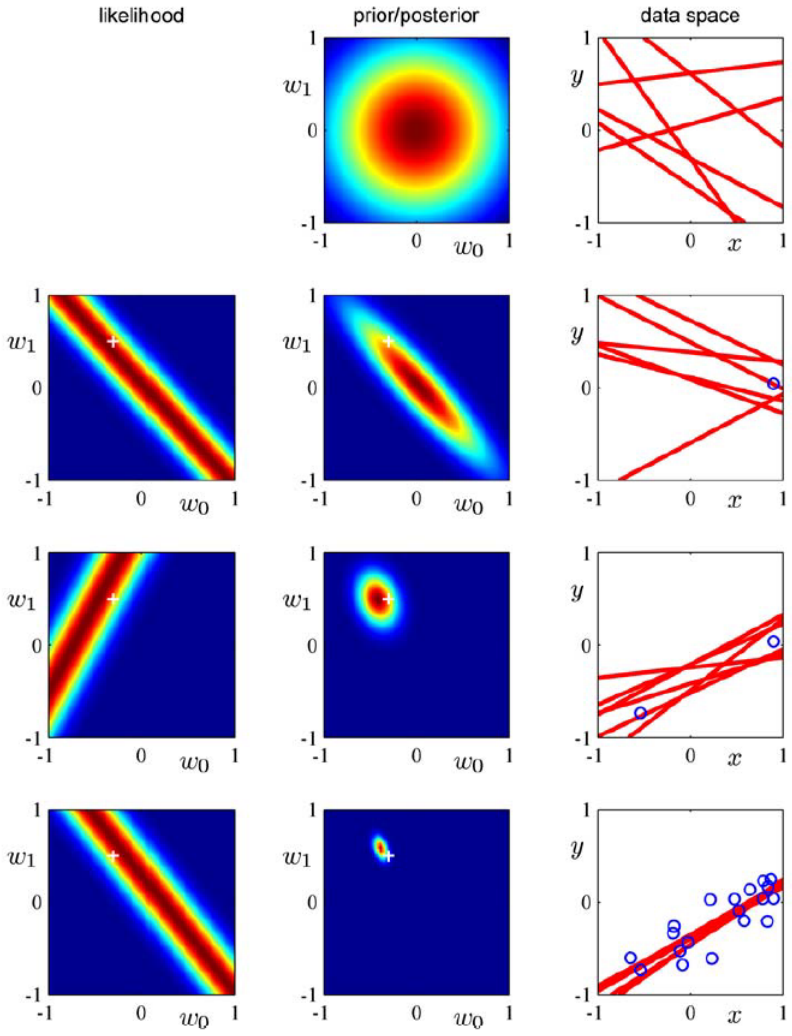
Example We generate some data from

$$t(x) = -0.3 + 0.5x + \epsilon, \quad where \quad \epsilon \sim \mathcal{N}(0, 0.04)$$

As a model we take

$$y(x, w) = w_0 + w_1 x, \quad \sigma^2 = 0.04, \quad \tau^2 = 0.5$$

To find the posterior distribution we use an iterative approach as follow. We start from a multivariate Gaussian prior $P(w) \sim \mathcal{N}(0, \tau^2 I)$ (prior). Then we take one data point and we find the parameters that make the model pass through that point, also considering data noise σ^2 (likelihood). The next step is to multiply the prior with the likelihood to find a posterior distribution in parameters space. The new posterior can be used as a prior for the next iteration. We take a new point, we find the parameters that make the model pass through that point and again we multiply together prior and likelihood. Note that at each iteration, the likelihood distribution considers only one point at the time.



1.4.1 Predictive distribution

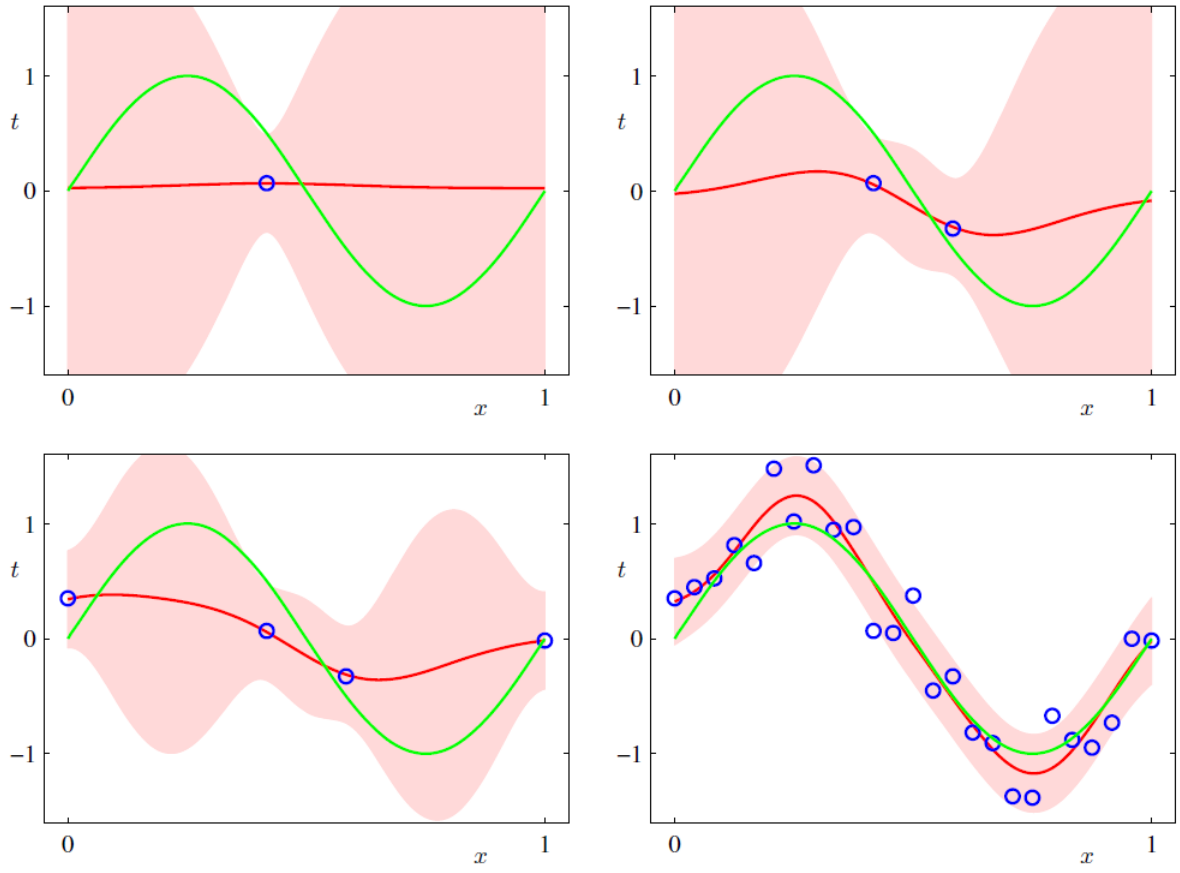
In practice, we are not usually interested in the value of w itself but rather in making predictions of t for new values of x. This requires that we evaluate the posterior predictive distribution defined by,

$$p(t|x, D, \sigma^2) = \int \mathcal{N}(t|w^T\Phi(x), \sigma^2)\mathcal{N}(w|w_N, S_N)dw = \mathcal{N}(t|w_N^T\Phi(x), \sigma_N^2(x)), \tag{27}$$

$$\sigma_N^2(x) = \underbrace{\sigma^2}_{\substack{\text{noise in the} \\ \text{target values}}} + \underbrace{\Phi(x)^T S_N \Phi(x)}_{\substack{\text{Uncertainty associated} \\ \text{with parameters values}}} \tag{28}$$

σ^2 is also called irreducible noise, in fact for $N \to \infty$, the second term of $\sigma_N^2(x)$ goes to zero, but σ^2 remain constant. In the figure below we can observe,

- **Green line** True model
- **Blue dots** Samples
- **Red line** Mean of the Gaussian predictive distribution
- **Red area** Predictive distribution spanning one standard deviation either side of the mean.



2 Linear models for classification

The goal in classification is to take an input vector x and to assign it to one of K discrete classes C_k where $k = 1, ..., K$. In the most common scenario, the classes are taken to be disjoint, so that each input is assigned to one and only one class. The input space is thereby divided into decision regions whose boundaries are called decision boundaries or decision surfaces. In this chapter, we consider linear models for classification, by which we mean that the decision surfaces are defined by (D − 1)-dimensional hyperplanes within the D-dimensional input space. Data sets whose classes can be separated exactly by linear decision surfaces are said to be linearly separable.

2.1 Linear classification

We will consider linear models for classification. In the linear regression case, the model is linear in parameters,

$$y(x, w) = \sum_{j=0}^{D-1} w_j x_j = x^T w$$

To have a simpler notation in future steps we explicit w_0 from w

$$= w_0 + \sum_{j=1}^{D-1} w_j x_j = w_0 + x^T w \tag{29}$$

For classification, we need to predict discrete class labels, or posterior probabilities that lie in the range of $(0, 1)$, so we use a nonlinear function(discriminant function) to remap the input space to the output space.

$$y(x, w) = \underbrace{f(w_0 + x^T w)}_{\text{activation function}}$$

A naive way to perform classification with two output classes is to choose an arbitrary activation function and for output less than 0.5 we have class 0 and viceversa class 1. We could be tricked into thinking that this classifier can predict nonlinear boundaries, but it's not the case. In fact, taken the boundary

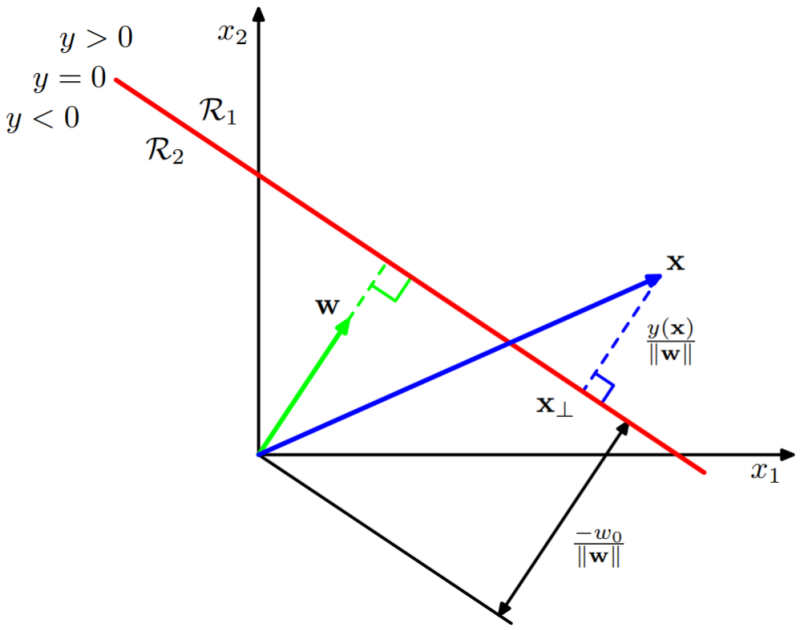
$$\begin{aligned} f(w_0 + x^T w) &= 0.5 \\ f^{-1}(f(w_0 + x^T w)) &= f^{-1}(0.5) \\ w_0 + x^T w &= f^{-1}(0.5) \\ w_0^* + x^T w &= 0, \\ with \quad w_0^* &= w_0 - f^{-1}(0.5) \end{aligned} \tag{30}$$

As we can see (30) represents an hyperplane, hence the decision surfaces are linear functions of x, even if the activation function is nonlinear. As in regression we can use basis function to make the input space linearly separable. The bad thing about this approach is that the model is no longer linear in the parameters, so no closed form solution exists.

2.1.1 Geometric interpretation

To have a better understanding of the discriminant function we can analyze it from a geometric point of view. The simplest representation of a linear discriminant function is obtained by taking a linear function of the input vector so that $y(x, w) = w_0 + x^T w$. w_0 is called bias and its negative is called threshold. An input vector x is assigned to class C_1 if $y(x) \geq 0$ and to class C_2 otherwise. The corresponding decision boundary is therefore defined by the relation $y(x) = 0$, which corresponds to a (D − 1)-dimensional hyperplane within the D-dimensional input space. Consider two points x_A and x_B both of which lie on the decision surface. Because $y(x_A) = y(x_B) = 0$, we have $w^T(x_A - x_B) = 0$ and hence the vector w is orthogonal to every vector lying within the decision surface[10], and so w determines the orientation of the decision surface. We can also say that w_0 regulates the normal distance(d) of the boundary from the origin. To find r we can project[11]a point x on the boundary on w

$$\begin{aligned} w^T x + w_0 &= 0 \\ w^T x &= -w_o \\ d = \frac{w^T x}{\|w\|_2} &= -\frac{w_0}{\|w\|_2} \end{aligned} \tag{31}$$



Furthermore, we can obtain the signed distance(r) of a point x from the boundary. Let's consider the projection $x_\perp$ of x on the boundary. Then

$$x = x_\perp + r\frac{w}{\|w\|_2}$$

[10]Given two vector their dot product is 0 when they are perpendicular to each other. $a \cdot b = |a||b|cos\theta$

[11]We know that the projection of a vector a on another vector b is $proj_b(a) = \frac{ab}{\|b\|}$.

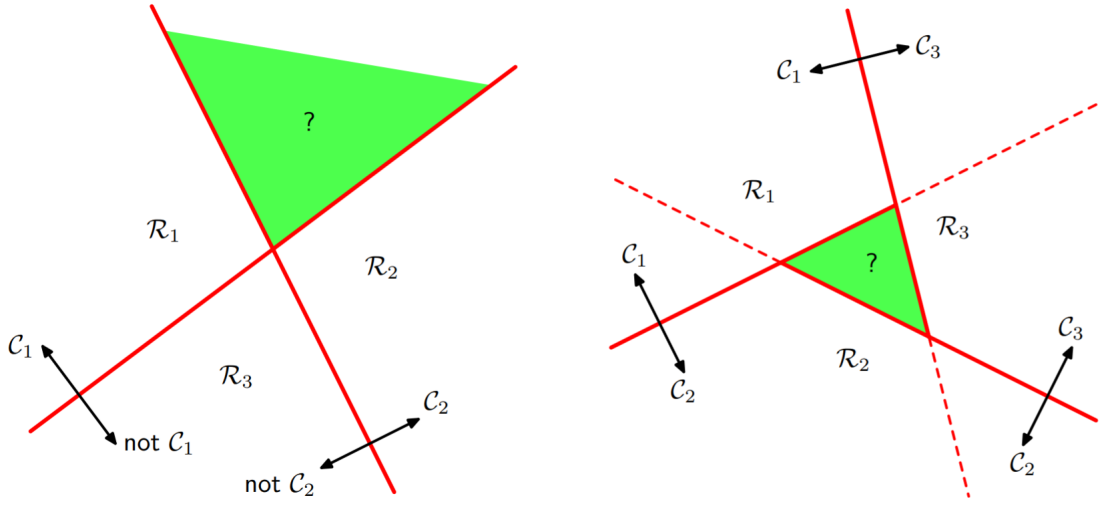
$$
\begin{aligned}
w^T x &= w^T x_\perp + w^T r \frac{w}{\|w\|_2} \\
w^T x + w_0 &= \underbrace{w^T x_\perp + w_0}_{=0} + w^T r \frac{w}{\|w\|_2} \\
y(x) &= w^T r \frac{w}{\|w\|_2} \\
y(x) &= r \frac{\|w\|_2^2}{\|w\|_2} \\
r &= \frac{y(x)}{\|w\|_2}
\end{aligned} \tag{32}
$$

2.1.2 Multiple outputs

Now consider the extension of linear discriminants to $K > 2$ classes. We might be tempted be to build a K-class discriminant by combining a number of two-class discriminant functions. However, this leads to some serious difficulties.

One-versus-the-rest Consider the use of K−1 classifiers each of which solves a two-class problem of separating points in a particular class C_k from points not in that class. There are regions in input space that are ambiguously classified, as we can see in the left-hand diagram.

One-versus-one An alternative is to introduce K(K − 1)/2 binary discriminant functions, one for every possible pair of classes. Each point is then classified according to a majority vote amongst the discriminant functions. However, this too runs into the problem of ambiguous regions, as illustrated in the right-hand diagram.



Linear discriminant functions We can avoid these difficulties by considering a single K-class discriminant comprising K linear functions of the form

$$y_k(x) = w_k^T x + w_{ko}, \quad where \quad k = 1, \ldots, K \tag{33}$$

and then assigning a point x to class C_k if $y_k(x) > y_j(x), \quad \forall j = k$. The decision boundary between class C_k and class C_j is therefore given by $y_k(x) = y_j(x)$ and hence corresponds to a (D − 1)-dimensional hyperplane. The resulting decision boundaries are singly connected[12]and convex. . Convexity imposes that taken two points x_A and x_B that belong to the same region R_k, every point $\hat{x}$ in between is still inside region R_k

$$\hat{x} = \lambda x_A + (1 - \lambda) x_B, \quad \hat{x} \in R_k. \quad \lambda \in [0, 1]$$

Convexity and linearity of the discriminant functions imply that

$$f_k(\lambda x_A + (1 - \lambda) x_B) > f_j(\lambda x_A + (1 - \lambda) x_B), \quad \forall j \in [1, K] \setminus k$$

2.2 Least square for classification

Consider a general classification problem with K classes, with a one-hot encoding for the target vector t. One justification for using least squares in such a context is that it approximates the conditional expectation $E[t|x]$ of the target values given the input vector. Each class is described by its own linear model

$$y_k(x) = w_k^T x + w_{ko}, \quad where \quad k = 1, \ldots, K \tag{34}$$

Using vector notation,

$$y(x) = \tilde{W}^T \tilde{x}, \quad where$$

$$\tilde{W} = \left[\underbrace{\begin{bmatrix} w_{10} \\ \vdots \\ w_{1D} \end{bmatrix}}_{w_1^T} \quad \cdots \quad \underbrace{\begin{bmatrix} w_{K0} \\ \vdots \\ w_{KD} \end{bmatrix}}_{w_K^T} \right], \quad [D + 1 x K]$$

$$\tilde{x} = (1, x^T)^T, \quad [D + 1 x 1]$$

We classify the input x into class C_k if $y_k(x) > y_j(x), \quad \forall j \in [1, K] \setminus k$. $y_k(x)$ corresponds to the k^{th} element of $y(x)$. To estimate the parameter we can follow what we did for the regression problem. To minimize least square,

$$\tilde{W} = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T T, \quad [(D + 1) x K] \tag{35}$$

$$\tilde{X} \quad [N x (D + 1)]$$

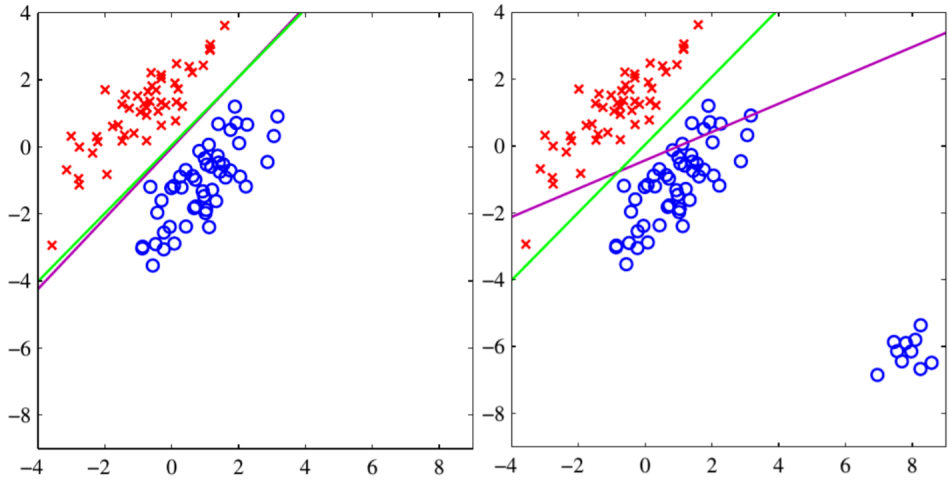
$$\tilde{T} \quad [N x K]$$

[12]It means that all the linear functions are ray originating from a common point

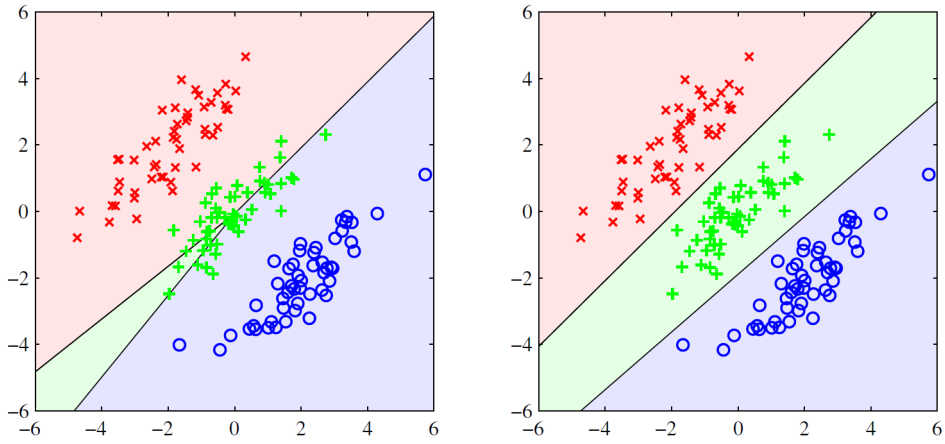
2.2.1 Least squares problems

The least square approach is problematic in some cases.

Outliers Least square is very sensitive to outliers. Least square tries to find a line which is the most close to all points. It does so evaluating the square distance between the samples and the line. It means that an outlier will have a greater impact on the line position because it will be more distant with respect to the probable samples. In classification this could degrade a lot the performance, because the boundary could deviate so much that some samples, previously well classified, now lie on the other side of the boundary.



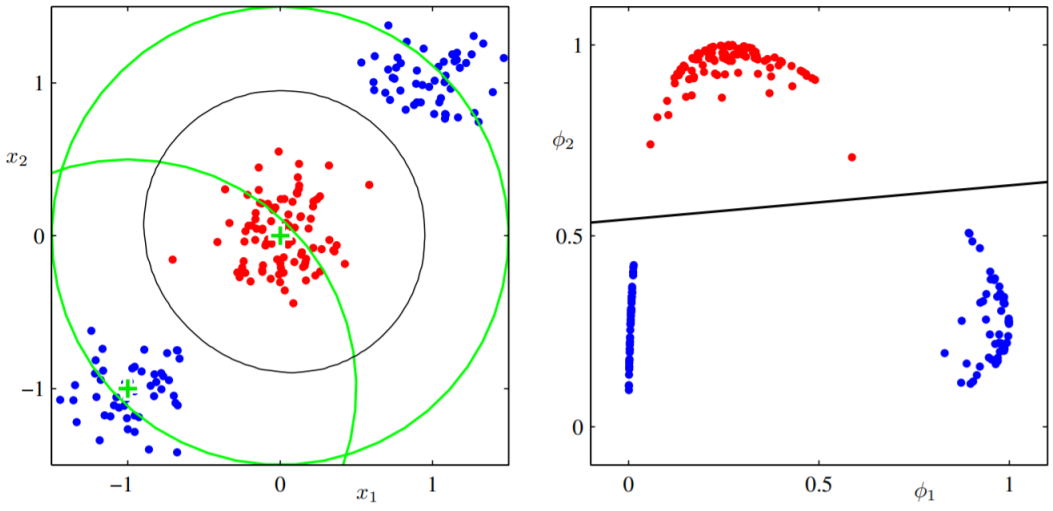
Non-Gaussian distributions We recall that least squares corresponds to the maximum likelihood under the assumption of a Gaussian conditional distribution, whereas binary target vectors clearly have a distribution that is far from Gaussian. As we can see in the figure below, even though the three classes are linearly separable, least square is not able to find good boundaries.



2.2.2 Basis functions

So far, we have considered classification models that work directly in the input space. All considered algorithms are equally applicable if we first make a fixed nonlinear transformation of the input space using vector of basis functions $\Phi(x)$. Decision boundaries will be linear in the feature space, but would correspond to nonlinear boundaries in the original input space.

Example Illustration of the role of nonlinear basis functions in linear classification models. The left plot shows the original input space (x_1, x_2) together with data points from two classes labelled red and blue. Two 'Gaussian' basis functions $\Phi_1(x)$ and $\Phi_2(x)$ are defined in this space with centres shown by the green crosses and with contours shown by the green circles. The right-hand plot shows the corresponding feature space $(\Phi_1(x), \Phi_2(x))$ together with the linear decision boundary. This corresponds to a nonlinear decision boundary in the original input space, shown by the black curve in the left-hand plot.



2.3 Perceptron

The perceptron is an algorithm for online[13]supervised learning of binary classifiers. . The algorithm tries to find a threshold function: a function that maps its input x (a real-valued vector) to an output value

$$y(x) = f(w^T \Phi(x)), \quad where$$
$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

Target values are +1 for C_1 and -1 for C_2. The algorithm finds the separating hyperplane by minimizing the distance of misclassified points to the decision boundary. Our objective is

[13]Online means that it is an iterative approach which calculate the solutions with multiple steps

to find a parameter vector w such that $w^T\Phi(x_n) \geq 0$ when $x_n \in C_1$ and $w^T\Phi(x_n) < 0$ when $x_n \in C_2$. Now we define a an error function as follow,

$$\epsilon_p(w, x_n) = \begin{cases} 0, & \text{If x is classified correctly} \\ w^T\Phi(x_n)t_n, & \text{If x is not classified correctly (proportional to boundary distance)} \end{cases}$$

Now we define an error function for the parameter optimization,

$$L_P(w) = -\sum_{n \in M} w^T\Phi(x_n)t_n \tag{36}$$

To perform minimization we use stochastic gradient descent(online)

$$w^{(k+1)} = w^{(k)} - \alpha\nabla L_P(w) = w^{(k)} + \alpha\Phi(x_n)t_n \tag{37}$$

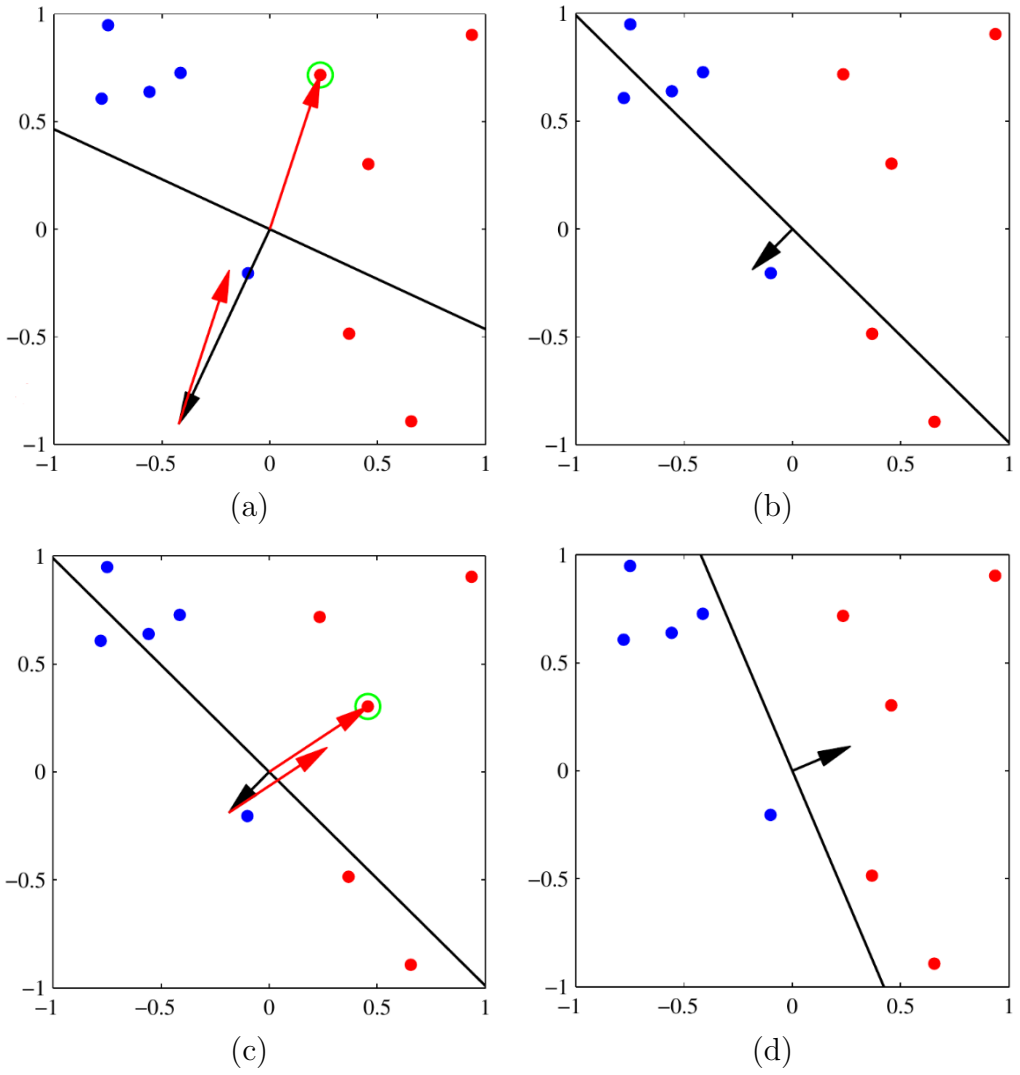
Note We have a minus sign in the loss function because $w^T\Phi(x_n)t_n$ will always be negative. This is due to the fact that if $w^T\Phi(x_n)$ is misclassified, then it will have an opposite sign compared to t_n.

Note The learning rate α can be set to 1 because it doesn't change the direction of w. We assume that w starts from the origin, so a scaling of the vector doesn't affect the boundary definition.

2.3.1 Perceptron algorithm

Algorithm 1: Perceptron algorithm

Output : A parameter vector $w^{(k)}$ that correctly classifies the two classes
Input : Data set $x_n \in \mathbb{R}^D$
$t_n \in \{-1, +1\}$, $\forall n \in [1, N]$
Initialize: w_0
$k \leftarrow 0$;
while *!converged* **do**
  $k \leftarrow k + 1$;
  $n \leftarrow k\%N$;
  if $\hat{t_n} \neq t_n$ **then**
    $w^{(k+1)} \leftarrow w^{(k)} + \Phi(x_n)t_n$;
  end
end



Theorem 2.1 (Perceptron convergence theorem)**.** *If the training data set is linearly separable in the feature space* Φ*, then the perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps*

The number of steps before convergence may be substantial. We are not able to distinguish between non-separable problems and slowly converging ones. If multiple solutions exist, the one found depends by the initialization of the parameters and the order of presentation of the data points. Another fact about algorithm steps is that the effect of a single update is to reduce the error due to the misclassified pattern, but this does not imply that the loss is reduced at each stage. This means that we reduce the error of the misclassified point we are considering, but we have no guarantee that the error of the other points gets better.

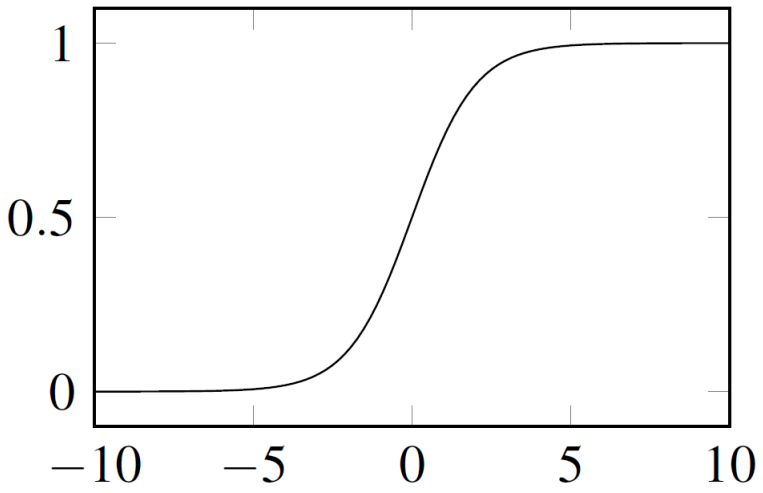
2.4 Logistic regression

Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary variable. So it is capable of resolve two-class classification. Logistic regression is a discriminative model so we model directly the posterior probability $P(C_k|\Phi)$. In detail we use a logistic sigmoid function[14].

$$P(C_1|\Phi) = \sigma(w^T\Phi) = \frac{1}{1+e^{-w^T\Phi}} \tag{38}$$

$$P(C_2|\Phi) = 1 - P(C_1|\Phi) \tag{39}$$

Note Logistic regression is a classification method not a regression one.



2.4.1 Maximum Likelihood for logistic regression

We now use maximum likelihood to determine the parameters of the logistic regression model. Now we have to define a suitable loss function to use. To do so, we first analyze our inputs and outputs. We have a dataset $D = \{x_n, t_n\}$, $n = 1, \ldots, N$

$$y_n(\Phi(x_n)|w) = P(t_n|\Phi(x_n)), \quad t_n \in [0,1] \Rightarrow t_n \sim Be(y_n(\Phi(x_n)|w))$$

Note For $t_n = 1$ we have C_1 and for $t_n = 0$ we have C_2.

Knowing that t_n follows a Bernoulli distribution we have,

$$P(t_n) = y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n} \tag{40}$$

[14]Sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$

The equation (40) describes the probability of the result being t_n given the input $\Phi(x_n)$ and the parameters w. So we can take as loss function the product of all $P(t_n)$.

$$
\begin{aligned}
l(w) &= \prod_{n=1}^{N} P(t_n) \\
&= \prod_{n=1}^{N} y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n} \\
\Downarrow &\quad \text{Transition to ln to simplify calculus.} \\
&\quad \text{Min \& Max remain the same} \\
l(w) &= ln(\prod_{n=1}^{N} y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n}) \\
&= \sum_{n=1}^{N} ln(y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n}) \\
&= \sum_{n=1}^{N} ln(y_n(\Phi(x_n)|w)^{t_n} + \sum_{n=1}^{N} ln(1 - y_n(\Phi(x_n)|w))^{1-t_n}) \\
&= \sum_{n=1}^{N} t_n ln(y_n(\Phi(x_n)|w) + \sum_{n=1}^{N} (1 - t_n) ln(1 - y_n(\Phi(x_n)|w))) \\
&= \sum_{n=1}^{N} t_n ln(y_n(\Phi(x_n)|w) + (1 - t_n) ln(1 - y_n(\Phi(x_n)|w))) \\
\Downarrow &\quad \text{For simplicity we remove } y_n \text{ parameters} \\
&= \sum_{n=1}^{N} t_n ln(y_n) + (1 - t_n) ln(1 - y_n)
\end{aligned}
$$

To find the optimal parameters we would like to maximize $l(w)$. Usually loss function are minimized. To be coherent with the literature, we put a minus sign in from of our loss function $l(w)$.

$$
L(w) = -\sum_{n=1}^{N} t_n ln(y_n) + (1 - t_n) ln(1 - y_n) \quad \text{(Binary cross-entropy)} \tag{41}
$$

Now we have to minimize the loss function. Unfortunately L(w) is no longer linear, so we have to use iterative methods. Nonetheless, we need to find the gradient of $L(w)$.

$$
\begin{aligned}
\overset{w}{\nabla} L(w) &= \sum_{n=1}^{N} \frac{\partial L(w)}{\partial y_n} \frac{\partial y_n}{\partial w} \quad \text{Chain rule} \\
&= \sum_{n=1}^{N} \frac{y_n - t_n}{y_n(1 - y_n)} y_n(1 - y_n)\Phi(x_n)
\end{aligned}
$$

$$= \sum_{n=1}^{N} (y_n - t_n)\Phi(x_n)$$

There is no closed form solution, due to non-linearity of the logistic sigmoid function, but the error function is convex[15]and can be optimized by standard gradient-based optimization techniques.

Note If we replace the sigmoid with a step function we obtain the perceptron algorithm. Both algorithm use the same updating rule $w \leftarrow w + \alpha(y(x_n, w) - t_n)\Phi(x_n)$

2.4.2 Multiclass logistic regression

Logistic regression can be expanded to multiclass classification. Before starting, it can be useful to talk about the role of the sigmoid function in standard logistic regression. σ is used to remap the infinite space $w^T\Phi$ in a finite output space. Furthermore, the output space have to be a probability distribution, so every output must be between 0 and 1, and the sum of the two classes must be 1. In logistic regression the first property is ensured by the sigmoid function and the second by the fact that $P(C_2|\Phi) = 1 - P(C_1|\Phi)$. In the multiclass case, we have to find a new way to ensure that the second property is still valid. To comply with both properties we can use the **softmax** operator. If we have K classes we construct K classifier as follow,

$$P(C_k|\Phi) = y_k(\Phi|w) = \frac{exp(w_k^T\Phi)}{\sum_{j=1}^{K} exp(w_j^T\Phi)} \tag{42}$$

Given T, a [NxK] matrix containing all output vector t_n [Kx1]

$$P(T|\Phi) = \prod_{n=1}^{N} \underbrace{\left(\prod_{k=1}^{K} P(C_k|\Phi(x_n))^{t_{nk}} \right)}_{\substack{\text{Only one term corresponding} \\ \text{to correct class}}}$$
$$= \prod_{n=1}^{N} \left(\prod_{k=1}^{K} y_{nk}^{t_{nk}} \right)$$

So the loss function can be expressed as

$$L(w_1, \ldots, w_K) = -ln(P(T|\Phi)) = -\sum_{n=1}^{N} \left(\sum_{k=1}^{K} t_{nk} ln(y_{nk}) \right) \tag{43}$$

Last but not least the gradient will be

$$\nabla L_{w_k} = \sum_{n=1}^{N} (y_{nk} - t_{nk})\Phi(x_n) \tag{44}$$

[15]Convex in this case implies that $L(w)$ has only one minimum

3 Bias-Variance and Model Selection

3.1 "No Free Lunch" Theorems

In this section we will talk about a generic learner performance on different problems. We want to know if a given algorithm is better than the others in every case. The short answer is no, any two optimization algorithms are equivalent when their performance is averaged across all possible problems. In a more formal manner we can define $ACC_G(L)$ as the accuracy of L on unseen data and $\mathcal{F}$ as the set of all possible concept $y = f(x)$ (all possible problems).

Theorem 3.1 (No Free Lunch theorem)**.** $\forall L$, $\frac{1}{|\mathcal{F}|}\sum_{\mathcal{F}} ACC_G(L) = \frac{1}{2}$*, given any distribution* $\mathcal{P}$ *over x and training set size N.*

Corollary 3.1.1. *For any two learner* L_1 *and* L_2*, if* $\exists$ *a learning problem s.t.* $ACC_G(L_1) > ACC_G(L_2)$ *then* $\exists$ *a learning problem s.t.* $ACC_G(L_1) < ACC_G(L_2)$

In practice we are saying that it doesn't exist a perfect learning algorithm that performs well in every scenario. So every algorithm is "specialized" on a given learning task. No algorithm is universally better than another one.

3.2 Bias-Variance trade-off

To efficiently select the model complexity we want to analyze its error on unseen data.

3.2.1 Bias-Variance decomposition

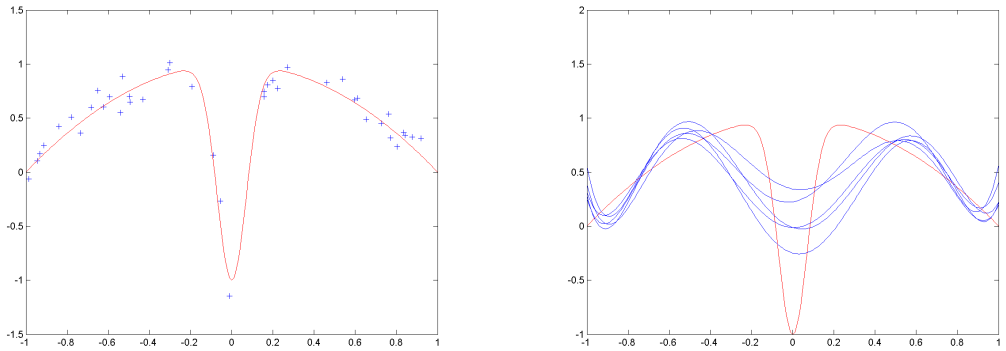
The bias–variance decomposition is a way of analyzing a learning algorithm's expected generalization error with respect to a particular problem as a sum of three terms, the bias, variance, and a quantity called the irreducible error, resulting from noise in the problem itself. Assume to have a dataset $\mathcal{D}$ with N samples taken from $t_i = f(x_i) + \epsilon$, where $E[\epsilon] = 0$ and $Var[\epsilon] = \sigma^2$ Our objective is to find a model $y(x)$ that approximate f as well as possible on unseen data. Let's consider the expected square error on an unseen sample x

$$
\begin{aligned}
E[(t - y(x))^2] &= E[t^2 + y(x)^2 - 2ty(x)] \\
&= E[t^2] + E[y(x)^2] - E[2ty(x)] \\
&\text{We can substitute t with its true function } f(x) \\
&= E[t^2] \pm E[t]^2 + E[y(x)^2] \pm E[y(x)]^2 - f(x)E[2y(x)] \\
&= Var[t] + E[t]^2 + Var[y(x)] + E[y(x)]^2 - 2f(x)E[y(x)] \\
&= Var[t] + Var[y(x)] + E[t]^2 + E[y(x)]^2 - 2f(x)E[y(x)] \\
&= Var[t] + Var[y(x)] + (f(x) - E[y(x)])^2 \\
&= \underbrace{Var[t]}_{\sigma^2} + \underbrace{Var[y(x)]}_{\text{Variance}} + \underbrace{E[f(x) - y(x)]^2}_{\text{Bias}^2}
\end{aligned} \tag{45}
$$

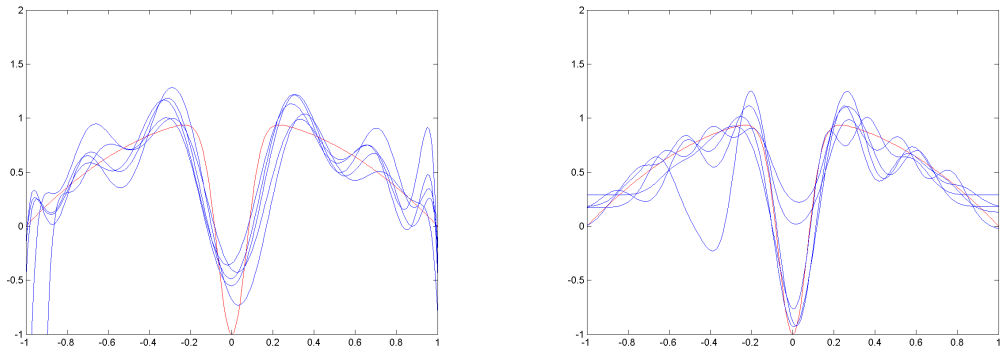
As we can see we have three sources of error

- σ^2: This is the irreducible error. It generates directly from the problem.
- Variance: This is an error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting).
- Bias: this is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).

It's worth mentioning that the expectation is performed over different realization of the training set D. Let's see an example



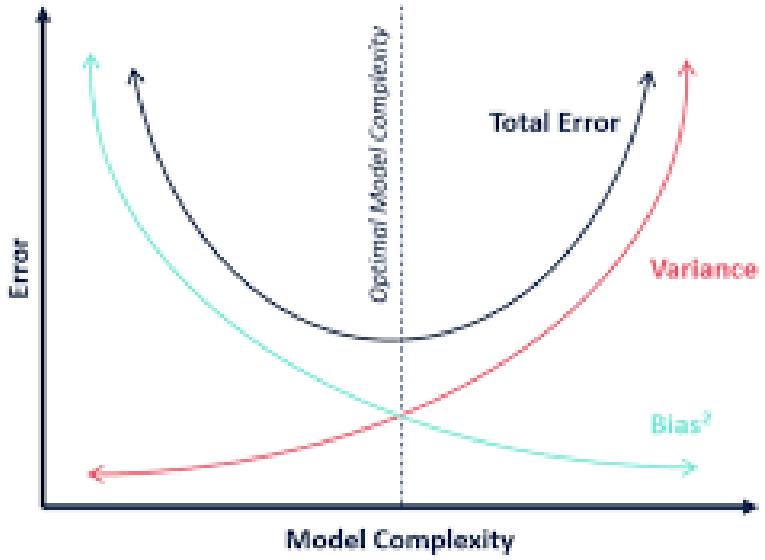
(a) (b)



(c) (d)

So let's take the dataset in figure(a) where the true function is the red line. For each case we take five different realization of the dataset and we estimate a model(blue line). In figure(b) we have an underfitting situation because our model is too simple. We can observe that we have a big bias because the models don't fit on data. But the variance between the models is very low. In figure(c) we have the right trade-off between bias and variance. In figure(d) we have a very low bias because all the models estimate the dataset well, but the variance between the trials is very big. In this case we are overfitting the data.

From what we have said, we can see how model complexity can affect the estimation error. Generally speaking the bias decreases with model complexity and variance increase with model complexity.



Example (K-NN) In the case of K-Nearest Neighbor we can derive an explicit analytical expression of the expected squared prediction error

$$E[(t - y(x))^2] = \sigma^2 + \frac{\sigma^2}{K} + \left(f(x) - \frac{1}{K} \sum_{i=1}^{K} f(x_i) \right)^2 \tag{46}$$

- σ^2 is the irreducible error
- $\frac{\sigma^2}{K}$ is the variance term. It depends on the irreducible error and decreases as K increases
- $\left(\frac{1}{K} \sum_{i=1}^{K} f(x_i) \right)^2$ is the bias term. It depends on how rough the model space is. The rougher the space, the faster the bias term will increase as further away neighbors are brought into the estimates

3.2.2 Training-test error

Given a data set $\mathcal{D}$ with N samples, we can split it in a training set and a test set(cross-validation). To calculate the training error we have to choose a loss function(e-g RSS). We can distinguish between

- Regression: $L_{train} = \frac{1}{N} \sum_{n=1}^{N} (t_n - y(x_n))^2$

- Classification: $L_train = \frac{1}{N}\sum_{n=1}^{N}(I(t_n \neq y(x_n)))$

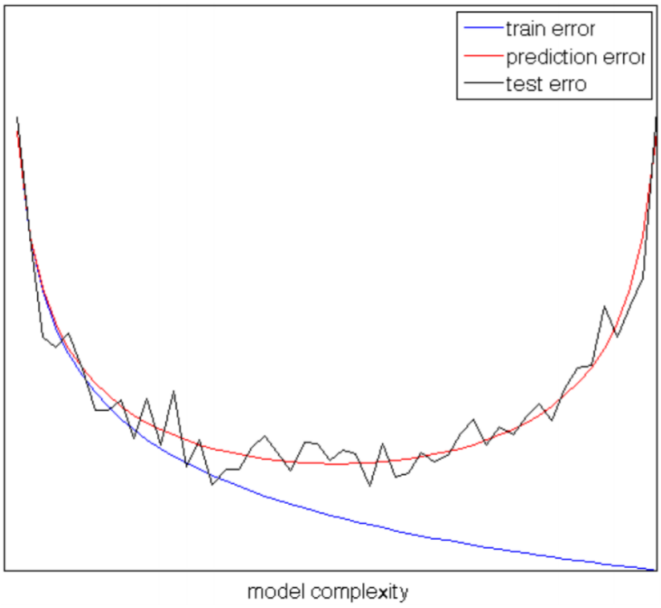
The training error measures how close our model is to training data. As we can imagine, increasing model complexity decreases the training error. But we also know that passed a certain complexity the generalization capability of our model will be decrease. So training error is not a good estimator of our model performance because is monotonically decreasing with model complexity, so it is an optimistically biased estimate of prediction error. Our objective is to estimate the true prediction error,

- Regression: $L_{true} = \int (f(x) - y(x))^2 p(x) dx$
- Classification: $L_{true} = \int I(f(x) \neq y(x)) p(x) dx$

This is impossible to estimate directly because we would need the true model f(x). A good way to estimate the prediction error is through the test error.

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (t_n - y(x_n))^2 \tag{47}$$

The test error is calculated with the test set. It is very important to not mix the training set and the test set, in order to get an unbiased estimation of the prediction error. We use the training set to estimate the model's parameters and the test set to estimate the prediction error.



3.3 Model selection

Model selection cover a key role in the performance of our model and can be performed in several ways. Increasing the complexity of a model means to add dimensions to the input space. This could have really bad consequences.

3.3.1 Curse of dimensionality

In ML we have the so called curse of dimensionality. It is related to the exponential increase in volume associated with adding extra dimensions to the input space. Working with high-dimensional data is difficult because the variance of the model becomes larger, we need more samples and computational power to counteract this phenomenon. So a common pitfall when we can't solve a learning problem is to add features to the input space. However the number of samples remain the same and so the importance of the old features decreases leading to worse performance. To have a better grasp of what can lead to an increasing search volume we can think of the following. Imagine the parameter space of a learning problem. Adding a feature will add a dimension in the parameter space. Let's imagine a uni-dimensional parameters space. Suppose we have two points on a line, 0 and 1. These two points are unit distance away from one another. Suppose we introduce a second axis, again distributed a unit distance away. Now we have two points, (0,0) and (1,1). But the distance between the points has grown to $\sqrt{2}$. If we iterate this process adding new dimension the two point will go further away. More formally, consider a p-dimensional hypercube with unit volume. Suppose that we have n points uniformally distributed inside the hypercube. Let r be the ratio of points inside the cube which are within some neighborhood. To capture an r-full of points in the data, we need to grow a cube which takes up r of the unit cube's volume on every axis. Since the length of an edge on the cube is simply 1, we have that $r = e^p$. So expressed in terms of e_p, the edge length necessary to fill a p-hypercube, we have that $e_p = r^{\frac{1}{p}}$. For example, to take 10% of the point in a 2-dimensional space we have $e_p(0.1) = 0.1^{\frac{1}{2}} = 0.31$. Similarly, for a 10-dimension space we would have $e_p(0.1) = 0.1^{\frac{1}{10}} = 0.8$. To include 10% of the data in a 10-dimensional space we need to take up to 80% of the volume, in contrast in a 2-dimensional space we only need 31%. The searching space grows exponentially.

3.3.2 Feature selection

Identifies a subset of input features that are most related to the output. We can use the following approach to select the best features.

1. Let $\mathcal{M}_0$ be the null model which contains no input feature
2. For $k = 1, \ldots, M$ (M different features)
   (a) Fit all $\binom{M}{k}$ models containing k features
   (b) Pick the best model and call it $\mathcal{M}_k$. To define the *best* model we need to define a metric.

3. Select a single best model among $\mathcal{M}_0, \ldots, \mathcal{M}_M$ using some criterion

We can start from understanding how we can evaluate and select a single best model. Obviously we can't use training error, because the most complex model will perform "better", even though is overfitting. We would like to use the test error to evaluate which is the best model among a collection of models with different numbers of features. There are two approaches to estimate the test error

- Direct estimation with a validation approach
- Making an adjustment to the training error to account for model complexity

Direct estimation So far we have divided the dataset into two subsets: training and test set. We do so to decouple the test data from the training phase in order to have an unbiased estimation of model performance. This decoupling must be preserved so **we can't use both training and test data to evaluate the various models and their features**. To have a fair evaluation we can introduce a new subset of data: **Validation set**.

- **Training dataset**: The sample of data used to fit the model
- **Validation dataset**: The sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters[16].
- **Test dataset**: The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.

In practice training set is used to learn the parameters, the validation set to tune the hyperparameter and the test set to evaluate the performance of our fit. Validation set can't be use directly to estimate the error, because is used indirectly in the training phase. The solution is to use cross validation. In practice we want to train the model on training set and evaluate it on the validation set. The novelty in the cross validation approach is that training and validation set are not fixed[17]. We can randomly divide the training data into k equal subsets: $\mathcal{D}_1, \ldots, \mathcal{D}_k$. We learn k times the parameters producing k different models, each time excluding a different $\mathcal{D}_i$. Then we evaluate the performance of every model on the relative excluded set. To estimate the validation error we can calculate the average of every subset $\mathcal{D}_i$ error. Based on the parameter k we can have different type of cross validation error.

[16]The hyperparameters are those parameters describing a model representation that cannot be learned by common optimization methods but nonetheless affect the loss function. An example is the regularization parameter λ in lasso

[17]Every sample is used as training data and validation data in different phases of the learning process

K-fold cross validation k-fold cross validation is the most used, because is a good trade-off between performance and accuracy. Usually we have $k \sim 10$

Algorithm 2: k-fold cross validation

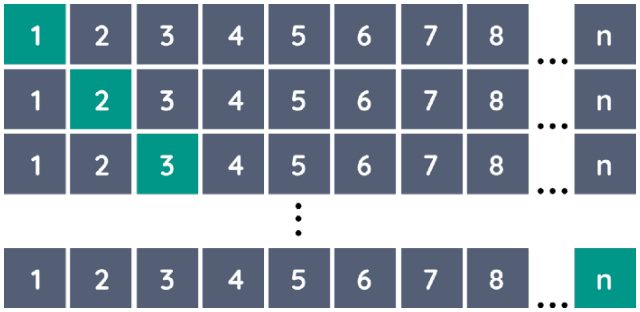
Output: Validation error L_{k-fold}
Input : Data set $\mathcal{D}$ splitted in $\mathcal{D}_1, \ldots, \mathcal{D}_k$
for $i \leftarrow 0$ **to** k **do**
  Train model $y_{\mathcal{D}\setminus\mathcal{D}_i}$ on $\mathcal{D} \setminus \mathcal{D}_i$
  $L_{\mathcal{D}_i} \leftarrow \frac{k}{N} \sum_{(x_n,t_n)\in\mathcal{D}_i} (t_n - y_{\mathcal{D}\setminus\mathcal{D}_i}(x_n))^2$
end
$L_{k-fold} \leftarrow \frac{1}{k} \sum_{i=1}^{k} L_{\mathcal{D}_i}$

It's worth mentioning that sometimes the validation partition is called test. Be aware that is not the same test set used ultimately to evaluate model performance.

LOO (Leave One Out) In this case we consider at each iteration a validation set with only one sample.

Algorithm 3: LOO cross validation

Output: Validation error L_{LOO}
Input : Data set $\mathcal{D}$ with N samples
for $i \leftarrow 0$ **to** N **do**
  Train model $y_{\mathcal{D}\setminus\{n\}}$ on $\mathcal{D} \setminus \{n\}$
  $L_{\{n\}} \leftarrow (t_n - y_{\mathcal{D}\setminus\{n\}}(x_n))^2$
end
$L_{LOO} \leftarrow \frac{1}{N} \sum_{n=1}^{N} L_{\{n\}}$



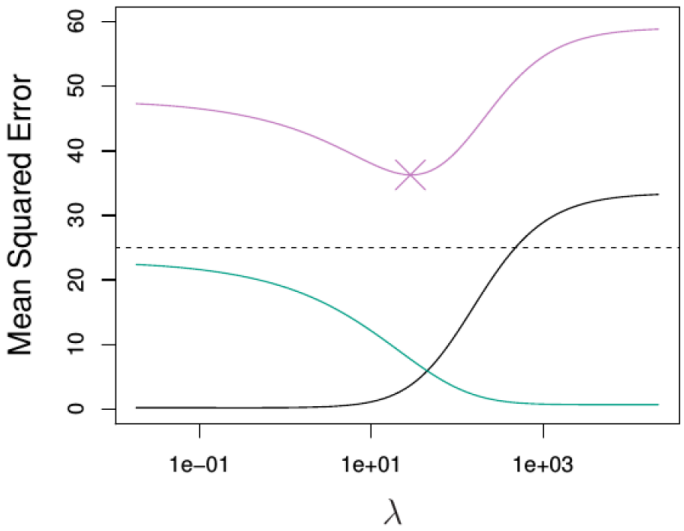
To evaluate different input features set, we apply the cross validation for every model having those input features. LOO is almost unbiased(pessimistically) as opposed to k-fold, but from a computational point of view k-fold is far better. For example, if we have 100.000 samples and each iteration of the algorithm takes 1 seconds, computing L_{LOO} will take more than a full day. If you have to do it for every permutation of the input features it will take a very long time.

Adjustment techniques This approach tries to account for model complexity when evaluating the training error. There are several way to do that

- $C_p = \frac{1}{N}(RSS + 2d\tilde{\sigma}^2)$
  Where d is the number of parameters, $\tilde{\sigma}^2$ is an estimate of the variance of the noise ϵ
- $BIC = \frac{1}{N}(RSS + log(N)d\tilde{\sigma}^2)$
  We replaces $2d\tilde{\sigma}^2$ of C_p with $log(N)d\tilde{\sigma}^2$. Since $log(N) > 2$ when $N > 7$, BIC selects smaller models.
- $AIC = -2log(L) + 2d$
  Where L is the maximized value of the likelihood function for the estimated model
- $AdjustedR^2 = 1 - \frac{RSS/(N-d-1)}{TSS/(N-1)}$
  where TSS is the total sum of squares. Differently from the other criteria, here a large value indicates a model with a small test error.

3.3.3 Regularization

We have already seen regularization approaches applied to linear models like ridge regression and lasso. Such methods shrink the parameters towards zero. It may not be immediately obvious why such a constraint should improve the fit, but it turns out that shrinking coefficient estimates can significantly reduce the variance. A possible motivation could be that smaller parameters are less prone to produce fast changing output function, so they are less prone to overfitting because they can't interpolate directly all the samples. Regularization methods are very useful when we have limited dataset and a large number of features compared to the dataset size. To visualize the importance of choosing the correct we can consider the following example. We pick $N = 50$ samples from a given function and we try to fit it with a model with 45 features. As a regression method we use ridge regression. From the figure below we can see in action the bias-variance trade-off. When λ gets bigger we obtain simpler models, so the variance is reduced and the bias gets bigger. We need to find the optimal λ which minimizes the MSE. Cross-validation provides a simple way to tackle this problem. We choose a grid of λ values, and compute the cross-validation error rate for each value of λ. We then select the tuning parameter value for which the cross-validation error is smallest. Finally, the model is re-fit using all of the available observations and the selected value of the tuning parameter.



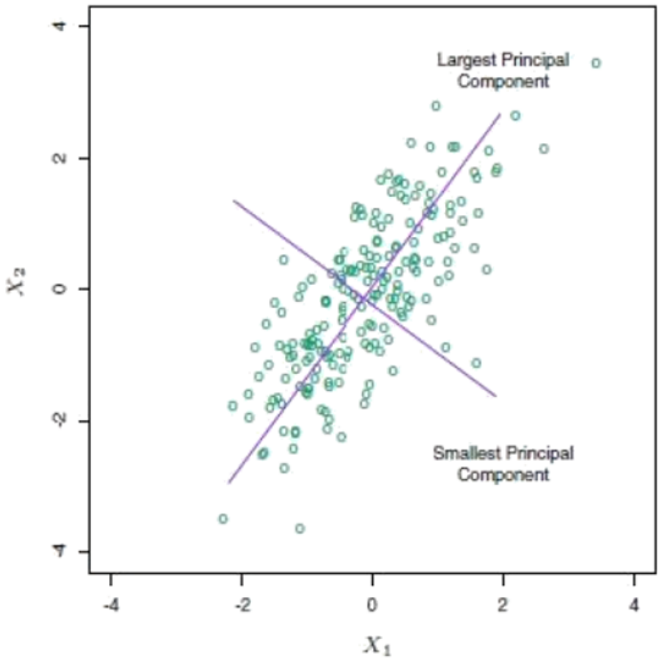
Black: squared bias, Green: variance, Purple: MSE, Dashed: minimum possible MSE, Purple cross: ridge regression model with minimum MSE

3.3.4 Dimension reduction

The approach we are going to present differs from the previous one because it doesn't operate directly on the original features. Dimension reduction methods transform the original features and then the model is learned on the transformed variables. It does so with an unsupervised learning approach. There are many techniques to perform dimensionality reduction,

- PCA (Principal Component Analysis)
- ICA (Independent Component Analysis)
- Self-organizing Maps
- Autoencoders
- ...

PCA The most used methodology is PCA. The idea is to find a new orthogonal base in the input space, which accounts for most on input variance. In other word we want to find a line such that when the data is projected onto that line, it has the maximum variance. Then we find a new line, orthogonal to the first one, that has maximum projected variance.We repeat this process until we have covered a certain percentage of input variance or we have reached a given number of dimensions(lines).

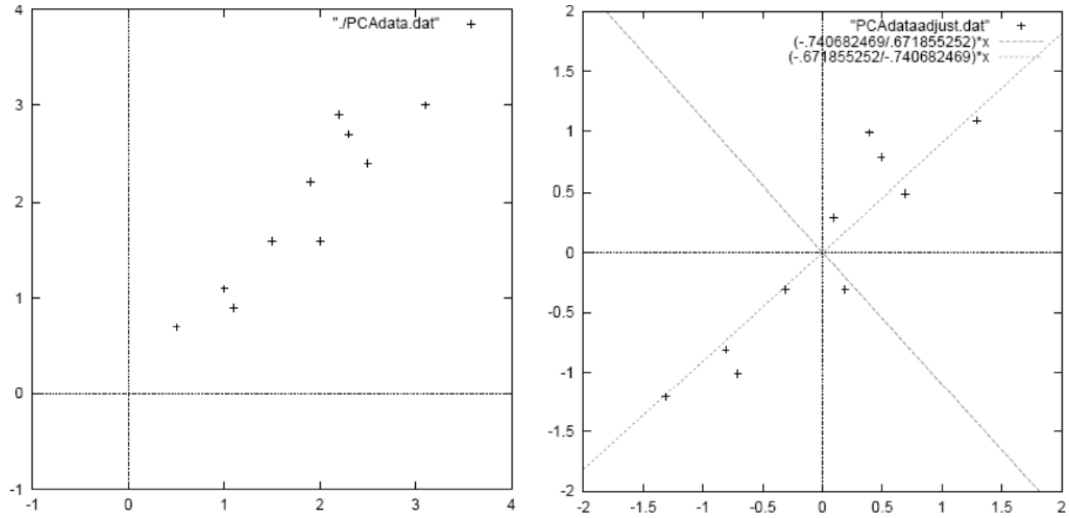


In a more formal way, the complete process consists of,

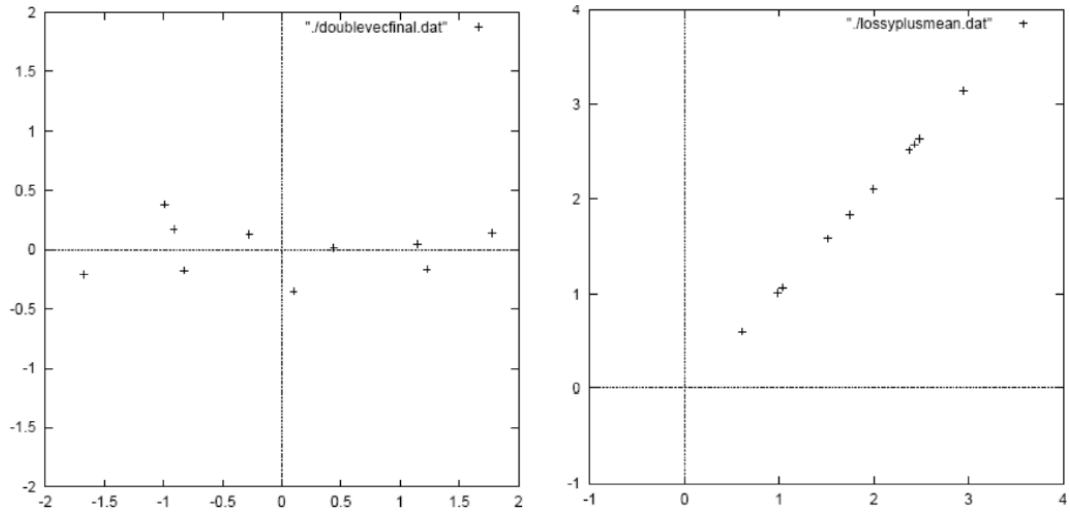
1. Compute the data mean $\overline{x} = \frac{1}{N}\sum_{n=1}^{N} x_n$ to later center the data on the origin
2. Compute the covariance matrix $S = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \overline{x})(x_n - \overline{x})^T$
3. Get the eigen values λ_i and eigen vector e_i of S
4. Select the first k largest eigen values. The relative eigen vector are the PCA components. $\frac{\lambda_i}{\sum_i \lambda_i}$ is the percentage of variance captured by the i^{th} principal component(PC)

Once we have found the new PCs, we can project the data onto the new orthogonal base $E_k = \begin{bmatrix} e_1 & \dots & e_k \end{bmatrix}$ [Mxk].

$$X' = XE_k, \quad [Nxk] \tag{48}$$



Left: Original data, Right: Mean centered data with PCs overlayed



Left: Original data projected into full PC space, Right: Original data reconstructed with first PC

PCA is very useful to reduce computational complexity. In supervised learning reducing the number of dimension leads to smaller hypothesis space resulting in models that are less prone to overfitting. PCA can be seen as a noise reduction reduction. This technique have some defects,

- Fails when data consists of multiple cluster
- Directions of greatest variance may not be most informative

- Computational problems with many dimensions
- PCA computes linear combination of features, but data often lies on a nonlinear manifold

3.4 Model Ensembles

The methods seen so far can reduce bias by increasing variance or vice versa. But there's a class of technique that can reduce variance or bias only. To decrease the variance without increasing the bias er can use **bagging**. To decrease the bias without increasing the variance we can use **boosting**. Bagging and Boosting are meta-algorithms. Their basic idea is to learn several models and combine them, instead of learning only one model. Typically they greatly improve accuracy.

3.4.1 Bagging

As we have said before bagging reduces the variance. We do so by averaging multiple models together. We know that

$$Var[\bar{X}] = \frac{Var[X]}{N} \tag{49}$$

In order to be able to apply this method we need to find a way to generate multiple models from one dataset. To do so we use bootstrap. This statistical technique consists in generating samples of size B (called bootstrap samples) from an initial dataset of size N by randomly drawing with replacement B observations.

After boostraping we train a model for each bootstrap sample. In the prediction phase, in case of classification the result will be the majority vote on the classification results of every model, and for regression will be the average of the predicted values estimated by every model. Bagging improves performance for unstable learners which vary significantly with small changes in the data set. In practice we want to average multiple overfitting model. From what we have studied before, an overfitting model has low bias and high variance. By combining different model we reduce the variance maintaining the low bias. In practice bagging almost always helps.

3.4.2 Boosting

The aim of boosting is to reduce the bias. It achieve so by sequentially train weak learners[18]. Now you are wondering, what the fuck means sequentially. Sequentially means that we train a model based on the prediction of the previous. The steps to perform boosting are the following,

1. Give an equal weight to all training samples
2. Train a weak model on the training set
3. Compute the error of the model on the training set
4. For each training samples increase its weight if the model predicted wrong that sample. Doing so we obtain a new training set.
5. Iterate the training on the new training set, error computation and samples re-weighting until we are satisfied by the result

The final prediction is the weighted prediction of every weak learner. In practice we are combining a set of sequential underfitting model. Doing so, we have low variance and the bias is improved by combining the weak learner to form a strong learner. On average, boosting helps more than bagging, but it is also more common for boosting to hurt performance.

[18] A weak learner is a learner that has a slightly better performance than chance prediction on any training set

4 PAC-Learning and VC-Dimension

In this section we will have a look to some statistical learning method. Previously we focused our attention on estimating the generalization error of a given model in order to measure the true performance. The training error usually is not a good indicator of how a model is behaving, but is way easier to estimate than other error terms. So we would like to extract as many information as possible from training error. Also there are cases where the training error is somewhat a good estimation of the performance. For example when we have a good number of samples relative to our hypothesis space we are less prone to overfitting. A question arises naturally. Can we estimate how many samples are necessary given an hypothesis space? We can answer this question in a theoretical way. Please remind that from a practical point of view is rarely used.

4.1 PAC-Learning

To introduce the ingredients of the theoretical setting we use a character recognition task given an array of n bits encoding a binary-valued image.

- X **Instances set**. In the character recognition problem, the instance space is $X = \{0,1\}^n$. The set of all possible input binary images.
- H **Hypothesis space**. The space where lies all possible combination of parameters.
- C **Set of target concept**. A concept is a subset $c \subset X$. One concept is the set of all patterns of bits in $X = \{0,1\}^n$ that encode a picture of the letter "P".
- $\mathcal{P}$ **Probability distribution over X**. Training instances are generated by a fixed, unknown probability distribution over X

The learner observes a sequence $\mathcal{D}$ of training example $< x, c(x) >$ for some target concept $c \in C$ and x is drawn from $\mathcal{P}$. The learner must output a hypothesis h estimating c. h is evaluated by its performance on subsequent instances drawn according to $\mathcal{P}$

$$L_{true} = P_{x \in \mathcal{P}}(c(x) \neq h(x))$$

Our objective is to bound L_{true} given L_{train}. Now we introduce the so called version space $VS_{H,\mathcal{D}}$. It is a subset of H where the training error L_{train} is zero. So the hypothesis h are always correct on training instances. For now, we assume that VS is non-empty. Making some consideration we can bound L_{train} inside VS.

Theorem 4.1 (L_{train} bound in VS)**.** *If the hypothesis space H is finite and $\mathcal{D}$ is a sequence of $N \geq 1$ independent random examples of some target concept c, then for any $0 \leq \epsilon \leq 1$, the probability that $VS_{H,\mathcal{D}}$ contains a hypothesis error greater then ϵ is less than $|H|e^{-\epsilon N}$:*

$$P(\exists h \in H : L_{train}(h) = 0 \wedge L_{true}(h) \geq \epsilon) \leq |H|e^{-\epsilon N}$$

Proof.

$$P((L_{train}(h_1) = 0 \wedge L_{train}(h_1) \geq \epsilon) \vee \cdots \vee (L_{train}(h_{|H|}) = 0 \wedge L_{train}(h_{|H|}) \geq \epsilon))$$

$$\begin{aligned}
&\leq \sum_{h \in H} P(L_{train}(h) = 0 \wedge L_{train}(h) \geq \epsilon) && \text{Union bound} \\
&\leq \sum_{h \in H} P(L_{train}(h) = 0 | L_{train}(h) \geq \epsilon) && \text{Bound using Bayes' rule} \\
&\leq \sum_{h \in H_{bad}} (1-\epsilon)^N && \text{Bound on individual } h_i s \\
&\leq |H|(1-\epsilon)^N && |H|_{bad} \leq |H| \\
&\leq |H| e^{-\epsilon N} && (1 - \epsilon \leq e^{-\epsilon}, for \quad 0 \leq \epsilon \leq 1)
\end{aligned}$$

□

We can notice that the dimension of the hypothesis space influences negatively the bound, in fact a larger searching space will give us less guarantees on the value of L_{true}. On the other hand, having more samples is always better, in fact $e^{-\epsilon N}$ is monotonically decreasing with N. Larger ϵ will lead to smaller bound because we are less demanding on the similarity between L_{true} and L_{train}. ϵ is the probability of making a wrong guess. Now we can bound the probability of $P(\exists h \in H : L_{train}(h) = 0 \wedge L_{true}(h) \geq \epsilon) \leq |H| e^{-\epsilon N}$. We can set a parameter δ

$$|H| e^{-\epsilon N} \leq \delta$$

After choosing δ we can calculate N or ϵ.
Given ϵ and δ

$$N \geq \frac{1}{\epsilon}\left(ln(|H|) + ln\left(\frac{1}{\delta}\right)\right) \tag{50}$$

Given N and δ

$$\epsilon \geq \frac{1}{N}\left(ln(|H|) + ln\left(\frac{1}{\delta}\right)\right) \tag{51}$$

Note $|H|$ can be very large. If we take as an example a binary decision problem with M binary inputs(features), the size of H will be 2^{2^M}. So N will have a exponential relationship with M. This is related to the curse of dimensionality.

Example Suppose H contains conjunctions of constraints on up to M boolean attributes (i.e., M literals). In this case $|H| = 3^M$. How many examples are sufficient to ensure with probability at least $(1 - \delta)$ that every h in $VS_{H,\mathcal{D}}$ satisfies $L_{true}(h) \leq \epsilon$?

$$N \geq \frac{1}{\epsilon}\left(Mln(3) + ln\left(\frac{1}{\delta}\right)\right)$$

Now we are ready to define formally what PAC[19]is. Consider a class C of possible target concepts defined over a set of instances X of length n, and a learner L using hypothesis space H.

Definition 4.1. *C is **PAC-learnable** if there exists an algorithm L such that for every $f \in C$, for any distribution $\mathcal{P}$, for any ϵ such that $0 \leq \epsilon < \frac{1}{2}$, and δ such that $0 \leq \delta < \frac{1}{2}$, then algorithm L, with probability at least $(1-\delta)$, outputs a concept h such that $L_{true}(h \leq \epsilon)$ using a number of samples that is polynomial of $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$*

Definition 4.2. *C is **efficiently PAC-learnable** by L using H $\iff$ $\forall c \in C$, distributions $\mathcal{P}$ over X, ϵ such that $0 \leq \epsilon < \frac{1}{2}$, and δ such that $0 \leq \delta < \frac{1}{2}$, algorithm L, with probability at least $(1-\delta)$, outputs a concept h such that $L_{true}(h \leq \epsilon)$, in time that is polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, M and size(c)*[20]

Now we need to generalize to problems where the VS is empty, because usually the train error is not equal to zero(agnostic learning). We can simply bound the difference between L_{train} and L_{true} in H.

$$L_{true}(h) \leq L_{train} + \epsilon$$

From now on we will consider only binary classification problems for simplicity. As we did before, we need to find an upper bound for the probability of having a "bad event", which in this case consists in having a gap between L_{train} and L_{true} bigger than ϵ. To achieve this we use the **Hoeffding bound**, which states

Definition 4.3. *For N i.i.d. coin flips $X_1, \ldots, X_N$, where $X_i \in \{0,1\}$ and $0 < \epsilon < 1$, we define the empirical mean $\bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$, obtaining the following bound*

$$P(E[\bar{X}] - \bar{X} > \epsilon) \leq e^{-2N\epsilon^2}$$

Theorem 4.2. *Given an hypothesis space H, a dataset $\mathcal{D}$ with N i.i.d. samples, $0 < \epsilon < 1$: for any learned hypothesis h:*

$$P(\exists h \in H | L_{true}(h) - L_{train}(h) > \epsilon) \leq |H| e^{-2N\epsilon^2}$$

This is very similar to what we have found in the non-empty case. Like we did before, we can calculate the number of example needed given ϵ and δ

$$N \geq \frac{1}{2\epsilon^2}\left(ln(|H|) + ln\left(\frac{1}{\delta}\right)\right)$$

Or ϵ given N and δ.

$$\epsilon \geq \sqrt{\frac{ln(|H|) + ln(\frac{1}{\delta})}{2N}}$$

[19]Probably Approximately Correct learning. Probably refers to δ and it is the confidence. Approximately refers to ϵ and it is the accuracy

[20]size(c): Number of bit necessary to express c. It comes from information theory

Now we can rewrite the gap between L_{train} and L_{true} as

$$L_{true}(h) \leq \underbrace{L_{train}(h)}_{Bias} + \underbrace{\sqrt{\frac{ln(|H|) + ln(\frac{1}{\delta})}{2N}}}_{Variance} \tag{52}$$

Once more, we can see how $|H|$ influences the loss function. For large $|H|$ we assume a low bias because it's more probable to find a good h and a high variance. For small $|H|$ we have high bias because we have a low probability of including a good h and low variance. In practice what we are saying is that we have to justify a large H with a lot of data. If we do so the training error will be a good estimation of the overall performance(test/true error).

4.2 VC Dimension

So far we have considered only finite hypothesis space. If we use the bound that we have just found in an infinite[21]H, we would have infinite variance. This is not the case, for infinite H the previous bound is too pessimistic. So we need to find a new one.. In the finite H case we encoded the complexity of H in the number of possible hypothesis. In the infinite case we can't do this, so we need to find a new metric to measure H complexity. We will use the **VC dimension**. To lay the ground for our theoretical discussion, we need to introduce two definitions,

Definition 4.4 (Dichotomy)**.** *A dichotomy of a set S is a partition of S into two disjoint subsets*

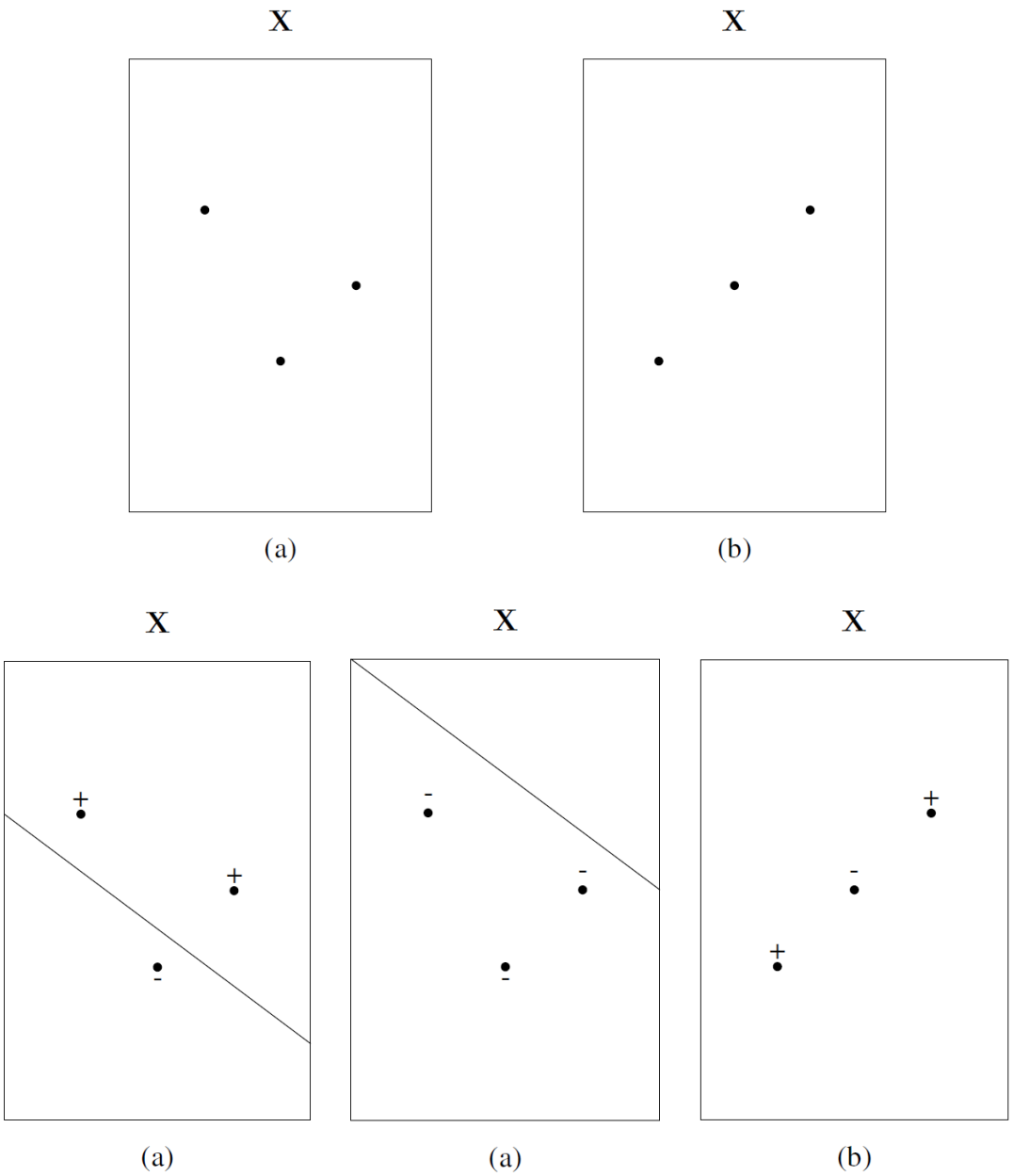
Definition 4.5 (Shattering)**.** *A set of instances S is shattered by hypothesis space H if and only if for every dichotomy of S there exists some hypothesis in H consistent with this dichotomy*

In practice shattering means that if we split our instances set in two not overlapping subset, H shatters S if and only if for every dichotomy we can find an hypothesis which classify correctly every instances. As a reminder we are still considering only binary classification problems.

Example We consider an instances set with three example[22]and an hypothesis space representing a linear classifier. In this case a dichotomy is a specific assignment of class (+) or class (-) to every point. We can have two cases, one where the examples are not aligned and one where they are. H shatters any of the two instances set? Yes, only (a) because we can always find a line that divides every possible dichotomy. In the specific dichotomy shown for (b), we can't find a linear classifier(line), that classifies correctly the three points, so H doesn't shatter (b).

[21]Infinite hypothesis spaces are very common. For example linear regression or classification have infinite hypothesis spaces

[22]This is still infinite because every instance can have a different "position" in the instances set



Note that for four instances, H won't shutter S[23] Now we can define what is the VC dimension

Definition 4.6 (VC dimension)**.** *The Vapnik-Chervonenkis dimension, $VC(H)$, of hypothesis space H defined over instance space X, is the size of the largest finite subset of X shattered by H. If arbitrarily large finite sets of X can be shattered by H, then $VC(H) \equiv \infty$*

In our previous example, the VC dimension of linear classifiers in two dimension is three. In fact, we can have a configuration of three instances whereby every dichotomy is perfectly separable. This doesn't hold for four instances.

[23]This is the XOR problem. It's not linearly separable

Example Few examples of VC dimensions,

- Linear classifier: $VC(H) = M + 1$, for M features plus the constant term.
- Neural networks: $VC(H) = \#parameters$
- 1-Nearest Neighbor: $VC(H) = \infty$
- SVM with Gaussian Kernel: $VC(H) = \infty$. We will see SVM in future chapters.

Now we can find a new bound for the error between L_{train} and L_{true}, and so we can find how many randomly drawn examples suffice to guarantee an error of at most ϵ with probability at least $(1 - \delta)$

$$N \geq \frac{1}{\epsilon}\left(4log_2\left(\frac{2}{\delta}\right) + 8VC(H)log_2\left(\frac{13}{\epsilon}\right)\right) \tag{53}$$

Equally we can express this as an upper bound for L_{true}

$$L_{true}(h) \leq L_{train}(h) + \sqrt{\frac{VC(H)(ln(\frac{2N}{VC(H)}) + 1) + ln(\frac{4}{\delta})}{N}} \tag{54}$$

Properties

Theorem 4.3. *The VC dimension of a hypothesis space $|H| < \infty$ is bounded from above*

$$VC(H) \leq log_2(|H|)$$

Proof. If $VC(H) = d$ then there exists at least 2^d functions(combination) in H, since there are at least 2^d possible labelings

$$\begin{aligned} |H| &\geq 2^d \\ |H| &\geq 2^{VC(H)} \\ VC(H) &\leq log_2(|H|) \end{aligned}$$

□

Theorem 4.4. *Concept class C with $VC(C) = 1$ is not PAC-learnable.*

5 Kernel methods

Kernel methods is a family of non-parametric techniques. To better explain what it means, we start from what we have already seen in the previous chapters. With parametric method a certain hypothesis in the hypothesis space is defined by the combination of values of the learnable parameters. For example, linear regression is a parametric method, where each hypothesis is defined by the value of the parameters associated with each feature plus the constant term. With non-parametric methods we have no explicit parameters. In parametric methods the training set is used in the training phase to learn the parameters. Then for the prediction phase the training set is not used because all the relevant information are encoded in the learned model. In non-parametric methods the training set is used also in the prediction phase because the model is implicitly encoded in the dataset.

Example A very famous example of non-parametric method is the k-nearest neighbour[24]. This method is used for classification. In practice when we have a new sample, we search for the k nearest training data samples in the training data. Then we assign a class to the new sample equal to the most frequent class between the k nearest training samples. Once classified, the new sample becomes part of the training data.

K-nearest neighbour doesn't utilize parameters, but introduce the concept of "distance" for evaluating the new samples. The distance, more formally, is called **metric**. As in parametric method we need to define the features, in non-parametric methods we need to define a metric. We can notice that in the k-nearest neighbour example we have no training time because we haven't a model. But this comes with a price, in fact we have a time penalty during the prediction phase because we need to review each training data to make our assumption, instead of just use our model.

- Parametric: long training time, short prediction time
- Non-parametric: short training time, long prediction time

So far, all the parametric methods were linear. So in the vanilla configuration they can only solve linear problems. We have also seen how we can extend those linear models to non-linear problems, for both regression and classification through the introduction of the basis functions. In the following sections we will see how we can extend the capability of the linear models to non-linear problems with the **kernels**.

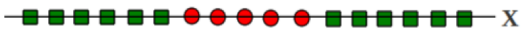
5.1 Kernels

Kernels make linear models work in non-linear settings, by mapping data to higher dimensions, where it may exhibits linear patterns and so linear models are applicable. Another good characteristic of kernel methods is their complexity. In fact, parametric methods complexity is based on the number of features, instead, kernel methods complexity is based on

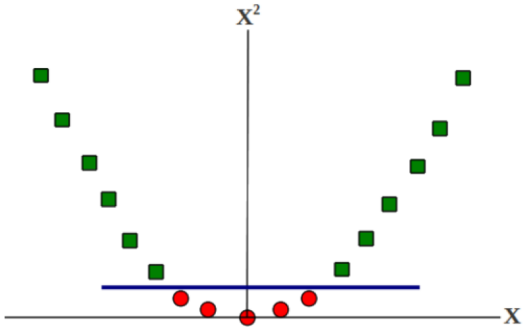
[24]It worth mentioning that k-nearest neighbour is not a kernel method. It's used only as an example for non-parametric methods

the number of samples. This gives us some advantages in some situation. For example, when we have more features than samples, kernel methods are much more efficient, both complexity and performance wise. A mapping to higher dimensions can be very expensive to compute, but kernels can give such mapping almost for free. This process is called kernel trick. In practice we can find a dual representation of a linear model using kernels.

Example - Linearize by dimensions augmentation Consider this binary classification problem



Each example is represented by a single feature x. It clearly doesn't exist a linear separator between the two classes. But if we map $\{x\} \to \{x, x^2\}$ the data are linearly separable.



This was the standard approach for extending linear models to non-linear problems

5.1.1 Kernel functions

Consider the following mapping Φ for an example $x = \{x_1, \ldots, x_M\}$

$$\Phi : x \to \{x_1^2, x_2^2, \ldots, x_M^2, x_1x_2, x_2x_3, \ldots, x_1x_M, \ldots, x_{M-1}x_M\}$$

This particular mapping is called second order monomial. Each new feature uses a pair of the original features. We can observe that the mapping will increase quadratically the number of features. This will have an impact on complexity because computing the mapping itself can be inefficient. Moreover, using the mapped representation could be inefficient too. Thankfully, kernels help avoid both these issues because the mapping doesn't have to be explicitly computed and the computations with the mapped features remain efficient. A kernel is a function which takes as input two data samples, and performs the scalar product between the feature expansions(mapping) of the two samples.

$$k(x, x') = \Phi(x)^T \Phi(x') \tag{55}$$

This kernel function is the metric of our non-parametric method and it measure the similarity between the points x and x'. A good consequence of being a scalar product is symmetry($k(x, x') = k(x', x)$)

Example - Linear kernel Let's consider the simplest kernel possible. The linear kernel correspond to the identity, in fact $\Phi(x) = x$. Given this k(x,x') will be simply the scalar product[25]between the two original samples. The result of the scalar product is maximum when the two vector are pointing in the same direction.
There are different type of kernel

- **Stationary kernel**: Function of difference between arguments. It is called stationary kernel since invariant to translation in space

$$k(x, x') = k(x - x')$$

- **Homogeneous kernel**: Known as radial basis functions, it depends only on the magnitude of the distance between arguments

$$k(x, x') = k(\|x - x'\|)$$

5.1.2 Dual representation

Many linear models for regression and classification can be reformulated in terms of dual representation, in which the kernel function arises naturally. We want this in order to be able to apply the kernel trick. In practice we want to describe our model not using feature but with a kernel. For every linear model exist a dual representation involving kernels. Let's take as an example ridge regression. We recall that loss function for ridge regression is

$$L_w = \frac{1}{2}\sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)^2 + \frac{\lambda}{2}w^T w$$

and its gradient is

$$\begin{aligned}\overset{w}{\nabla} L &= \frac{1}{2}2\sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)\Phi(x_n)^T + \frac{\lambda}{2}2w^T \\ &= \sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)\Phi(x_n)^T + \lambda w^T\end{aligned}$$

Putting $\overset{w}{\nabla} L = 0$ we have

$$\begin{aligned}-\lambda w^T &= \sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)\Phi(x_n)^T \\ w^T &= -\frac{1}{\lambda}\sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)\Phi(x_n)^T\end{aligned}$$

[25] $x \cdot x' = \|x\|\|x'\|\cos\theta$, where θ is the angle between x and x'

$$
\begin{aligned}
&\text{Define } a_n = -\frac{1}{\lambda}(w^T\Phi(x_n) - t_n) \quad [1x1] \\
&= \sum_{n=1}^{N} a_n \Phi(x_n)^T \\
w &= \left(\sum_{n=1}^{N} a_n \Phi(x_n)^T \right)^T \\
&= \sum_{n=1}^{N} (a_n \Phi(x_n)^T)^T \\
&= \sum_{n=1}^{N} (\Phi(x_n) a_n) \\
&\text{Define } \Phi = \begin{bmatrix} [\Phi^T(x_1)] \\ \vdots \\ [\Phi^T(x_N)] \end{bmatrix}, \text{ and } a = \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \\
&= \Phi^T a \quad [Mx1]
\end{aligned}
$$

Now we can substitute w in the loss function. For make the calculus simpler we switch to full matrix notation

$$
\begin{aligned}
L_w &= \frac{1}{2}(\Phi w - t)^T(\Phi w - t) + \frac{\lambda}{2} w^T w \\
&= \frac{1}{2}(\Phi\Phi^T a - t)^T(\Phi\Phi^T a - t) + \frac{\lambda}{2}(\Phi^T a)^T \Phi^T a \\
&= \frac{1}{2}(\Phi\Phi^T a - t)^T(\Phi\Phi^T a - t) + \frac{\lambda}{2} a^T \Phi\Phi^T a \\
&= \frac{1}{2}((\Phi\Phi^T a)^T - t^T)(\Phi\Phi^T a - t) + \frac{\lambda}{2} a^T \Phi\Phi^T a \\
&= \frac{1}{2}(a^T\Phi\Phi^T - t^T)(\Phi\Phi^T a - t) + \frac{\lambda}{2} a^T \Phi\Phi^T a \\
&= \frac{1}{2}(a^T\Phi\Phi^T\Phi\Phi^T a) + \frac{1}{2} t^T t - \frac{1}{2} a^T\Phi\Phi^T t - \frac{1}{2} t^T\Phi\Phi^T a + \frac{\lambda}{2} a^T\Phi\Phi^T a \\
&= \frac{1}{2}(a^T\Phi\Phi^T\Phi\Phi^T a) + \frac{1}{2} t^T t - a^T\Phi\Phi^T t + \frac{\lambda}{2} a^T\Phi\Phi^T a
\end{aligned}
$$

We can observe how Φ is present only when multiplied by its transpose. In this way we can use the kernels we have define before to substitute the features. In order to have a simpler notation, we can observe that the kernel function is a Gram matrix[26] $K = \Phi\Phi^T$ [NxN], where each element is

$$
\begin{aligned}
K_{nm} &= \Phi(x_n)^T\Phi(x_m) = k(x_n, x_m) \\
K &= \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_N) \\ \vdots & \ddots & \vdots \\ k(x_N, x_1) & \dots & k(x_N, x_N) \end{bmatrix}
\end{aligned} \tag{56}
$$

Notes

- Φ [NxM] and K [NxN]
- K is a matrix of similarities of pairs of samples(metric)
- K is symmetric

Now we can substitute $\Phi^T\Phi$ with K. We write L_w as L_a because w is no longer present in the equation so L_a

$$L_a = \frac{1}{2}(a^T K K a) + \frac{1}{2}t^T t - a^T K t + \frac{\lambda}{2}a^T K a$$

Solving for a by combining $w = \Phi^T a$ and $a_n = -\frac{1}{\lambda}(w^T\Phi(x_n) - t_n)$

$$a = (K + \lambda I_N)^{-1}t \quad [Nx1] \tag{57}$$

Solution for a can be expressed as a linear combination of elements of Φ, whose coefficients are entirely in terms of kernel $k(x, x')$, from which we can recover original formulation in terms of parameters w. This means that the loss function is convex thus it has only one global minimum. We can observe how all the element in equation (5.1.2) have dimension dependent only on the number of samples. This is exactly what we were looking for, because we have said that the complexity of kernel methods depends on the number of samples and not features. It practically means that now we have to invert a [NxN] matrix instead of a [MxM].

When we make a prediction for a new x we have our linear regression model. If we substitute $w = \Phi^T a$.

$$\begin{aligned} y(x) &= w^T\Phi(x) \\ &= a^T\Phi\Phi(x) \\ &\text{Define } k(x) = \begin{bmatrix} k(x, x_1) \\ \vdots \\ k(x, x_N) \end{bmatrix} \\ &= k(x)^T(K + \lambda I_N)^{-1}t \end{aligned}$$

As we can see, the prediction is a linear combination of the target values from the training set. We can get a very nice intuition of this. Making a linear combination of the target values is like taking a weighted average over the target samples, based on the similarity between the new samples and the target samples in the training data. In this way, we have completely eliminated the parameters, so we have found a dual representation of the parametric model eliminating the need of parameters and features, for describing the model, by using kernels. So the "model" is now implicit in the data. This approach have several advantages

[26]Given N vectors, the Gram Matrix is the matrix of all inner products. A matrix by its transpose is always a Gram matrix

- Solution for a is entirely described in terms of kernel s functions. Once we get a we can recover w as a linear combination of Φ using $w = \Phi^T a$
- When computing the solution we need to invert a [NxN] matrix and not a [MxM] matrix. This is good when the number of features is very high
- The true advantage is that for some kernel, we don't even need to compute Φ. Doing so we resolve a lot of issues revolving around the high number of features. We will see how we can work even with infinite features.
- Kernel functions can be defined not only over simply vectors of real numbers, but also over objects as diverse as graphs, sets, string, and text documents

5.1.3 Kernel construction

In this section we will see how we can construct a kernel. If you are wondering how the heck we can avoid to compute Φ directly, even if $K = \Phi^T\Phi$, you are in the right section. So the most naive way to construct a kernel is through the scalar product of Φ by its transpose. Nothing special, in fact we don't have any special gain doing this because we still need to use Φ.

$$k(x, x') = \Phi(x)^T\Phi(x') = \sum_{i=1}^{M} \Phi_i(x)\Phi_i(x')$$

Where $\Phi(x)$ are basis functions.

It exist a second and much more interesting method to construct kernels. We choose a function that correspond to a scalar product in some space. We make an example to better understand what it means.

Example Suppose to have the kernel function $k(x, z) = (x^T z)^2$. To be a valid kernel we need to find a features expansion that is able to provide the same result as the initial kernel definition. Suppose we are in two dimensional space.

$$\begin{aligned} k(x, z) &= (x^T z)^2 \\ &= (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2 \\ &= \begin{bmatrix} x_1^2 & \sqrt{2}x_1 x_2 & x_2^2 \end{bmatrix} \begin{bmatrix} z_1^2 & \sqrt{2}z_1 z_2 & z_2^2 \end{bmatrix}^T \\ &= \Phi(x)^T\Phi(z) \end{aligned}$$

So what's the point? We have checked if $k(x, z)$ is a valid kernel by finding a feature expansion whose inner product is equal to the kernel. Furthermore we can notice that the feature mapping takes the form $\begin{bmatrix} x_1^2 & \sqrt{2}x_1 x_2 & x_2^2 \end{bmatrix}$, comprising all second order terms with a specific weighting. The very nice thing about this is that we can avoid to compute Φ and then perform the scalar product, because we can directly estimate the kernel function(metric). Computing directly the kernel is way cheaper than computing Φ. In this case we have

- **Naive kernel construction**: compute 6 features values and 9 multiplication for inner product
- **Direct estimation of valid kernel**: 2 multiplication and a squaring

This is a very simple example and the gain is very small. If we consider the kernel $k(x, z) = (x^T z + c)^p$, we can demonstrate that the feature expansion that represent this kernel includes all the possible monomial from degree 0 to p.

- **Naive kernel construction**: exponential grow in number of operation
- **Direct estimation of valid kernel**: linear grow in number of operation

We can represent features expansion that include billions of elements with very simple kernel which need few operation to be computed. in this way we have construct a memory based method which doesn't use both features and weights, but it exploit the training data to predict new samples.

Now we can define more formally how we can demonstrate that a given kernel is valid. Necessary and sufficient condition for a function $k(x, x')$ to be a kernel is that the gram matrix K, whose elements are given by $k(x_n, x_m)$, is positive semi-definite[27]for all possible choices of the set $\{x_n\}$.

Theorem 5.1 (Mercer's theorem)**.** *Any continuous, symmetric, positive semi-definite kernel function $k(x, y)$ can be expressed as a dot product in a high-dimensional space*

New kernels can be constructed from simpler kernels as building blocks. Be aware that a really meaningful kernel is the one that define a good metric for representing the similarity between two inputs. So given kernels $k_1(x, x')$ and $k_2(x, x')$, the following new kernels will be valid

1. $k(x, x') = ck_1(x, x')$
2. $k(x, x') = f(x)k_1(x, x')f(x')$, where $f(\cdot)$ is any function
3. $k(x, x') = q(k_1(x, x'))$, where $q(\cdot)$ 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. $k(x, x') = k(\Phi(x), \Phi(x'))$, where $\Phi(x)$ is a function from x to $\mathbb{R}^M$
8. $k(x, x') = x^T Ax'$, where A is a positive semi-definite matrix
9. $k(x, x') = k_a(x_a, x'_a) + k_b(x_b, x'_b)$, where x_a and x_b are variables with $x = (x_a, x_b)$
10. $k(x, x') = k_a(x_a, x'_a)k_b(x_b, x'_b)$

[27]Positive semi-definite means that $x^T Kx \geq 0$, $\forall x : x_i \in \mathbb{R}^+$. A notable consequence is that a positive semi-definite matrix has

Example - Gaussian kernel A commonly used homogeneous kernel is the Gaussian kernel.

$$k(x, x') = exp\bigg(- \frac{\|x - x'\|^2}{2\sigma^2} \bigg) \tag{58}$$

Being an homogeneous kernel, it's based on a distance metric. In particular $\|x - x'\|^2$ represent the euclidean distance between x and x'.

Gaussian kernel validity. We can demonstrate its validity through kernel composition. We can expand the square as

$$\|x - x'\|^2 = x^T x + x'^T x' - 2x^T x'$$

To give

$$\begin{aligned} k(x, x') &= exp(-\frac{1}{2\sigma^2} k_1(x, x')), \quad \text{where} \\ k_1(x, x') &= x^T x + x'^T x' - 2x^T x' \end{aligned}$$

We know that the exponential of a valid kernel is still valid(4) so we need to demonstrate that $-\frac{1}{2\sigma^2} k_1(x, x')$ is valid. For composition (1) we need to demonstrate that $k_1(x, x')$ is valid because $\frac{1}{2\sigma^2}$ is only a coefficient. We also know that for (5) the sum of valid kernel is still valid, so we need to verify the components of $k_1(x, x')$. All three components are just linear kernels with some coefficient(1), so they are valid. □

Here we can appreciate the power of kernel composition. In fact, we don't know which is the feature expansion of the Gaussian kernel, but we are sure that it exists. This is once more a demonstration of how kernel methods don't need to define the features. Still, they correspond to the dual representation of a parametric model, where, in the case of Gaussian kernel, the number of features is infinite.

We can also expand the Gaussian kernel to non-Euclidean distances

$$k(x, x') = exp\bigg(- \frac{1}{2\sigma^2} (k_i(x, x) + k_i(x', x') - 2k_i(x, x')) \bigg) \tag{59}$$

Object kernels We have said how kernels can be defined over real vector numbers but also object such as sets, graph, strings and text.

Example - Sets To define a simple metric over sets we can use

$$k(A_1, A_2) = 2^{|A_1 \cap A_2|} \tag{60}$$

In practice, we find the number elements included in both A_1 and A_2. Then we use this number as an exponent.

Example - Generative models We define the generative model $p(x)$ which is a mapping in a one-dimensional feature space. The kernel is defined as

$$k(x, x') = p(x)p(x') \tag{61}$$

p(x) represent a probability. Performing the multiplication between $p(x)$ and $p(x')$ is like doing a inner product in a one-dimensional space so the kernel is valid. In practice we are multiplying the probability of x and x'. So the kernel defines the probability of having both x and x' and so their "similarity".

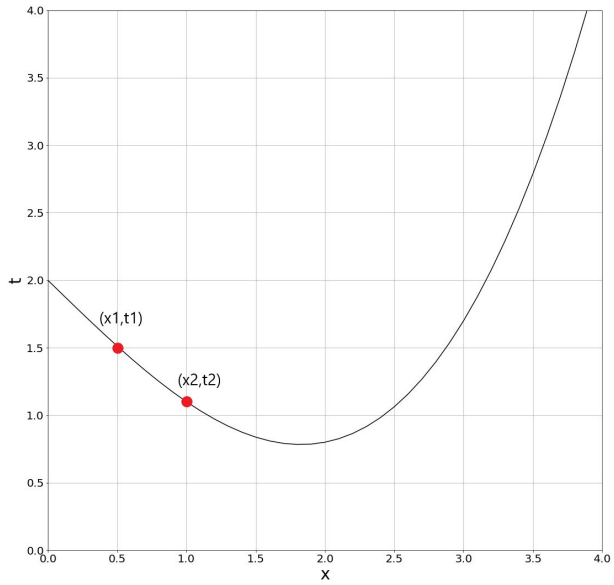
5.2 Gaussian processes

In the previous sections, we have seen how we can find the dual representation of ridge regression based on kernels. Gaussian processes are the kernel version of the Bayesian linear regression when we assume a Gaussian distribution for both prior and likelihood. So far, we have seen how to find the dual model of a non-probabilistic model for regression like ridge regression. We can extend the dual formulation to probabilistic discriminative models.

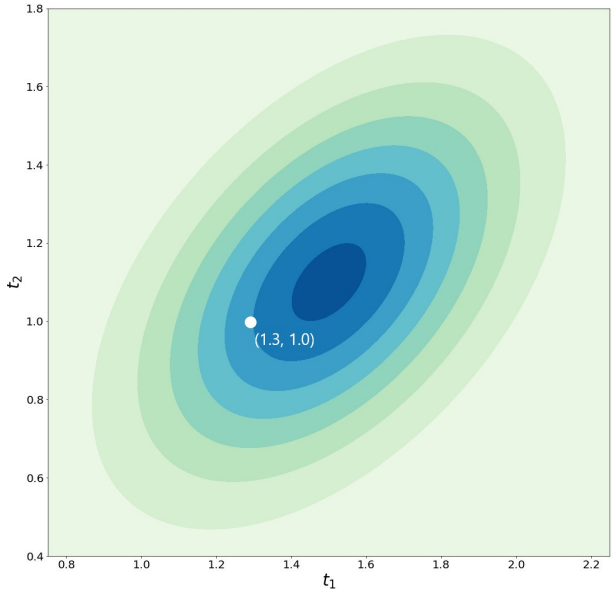
In Bayesian linear regression we have introduced a prior distribution over the parameters. Given a training set, we evaluate a new distribution over the parameters(posterior) by combining the prior and the new data observed(likelihood). From the posterior we can find a predictive distribution $p(t|x)$ for a new input x. With Gaussian processes, we define directly a prior distribution over functions[28]. If before we defined a prior over the parameters associated to the function, now we define a prior directly on the functions, bypassing the parameters. If we recall that Gaussian processes are actually kernel method we can make sense of it. With parametric method the function(model) is defined by its parameters, which decide the "shape" and characteristics of the function. In non-parametric method, the function is defined by the samples. Now we can observe that to define a function with samples, we would need a infinite amount of them. So what Gaussian processes are doing is considering an infinite collection of variables, one for each input point, and considering them as jointly distributed as a infinite-dimensional Gaussian distribution. OK, don't panic now we will see some graph and it will be clearer.

Example - Gaussian processes intuition We plot a function. To define the function we should consider infinite points. For this example we only consider two points x_1 and x_2. For each point we define $p(t|x)$ as a Gaussian distribution. Then, we join the two points distribution in a multivariate Gaussian. This distribution describe the values of our function based on the inputs x. Note that the variables are not independent, because the value of consequent inputs are likely to have similar values.

[28]With function we indicate the model in the parametric world. So in linear regression it would be the hyperplane defined by the parameters

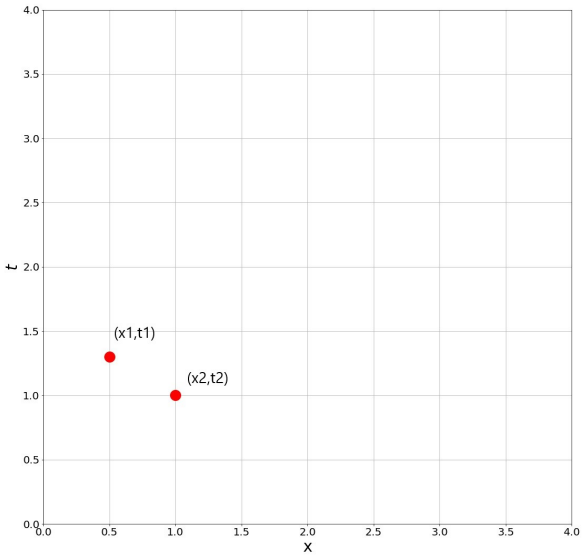


(a) Function we want to estimate



(b) Multivariate Gaussian over t

If we draw a sample from the multivariate Gaussian we would define a new function. In fact, picking a sample is the same as picking an infinite sequence of points in the (x, t) system. So our goal is to define a multivariate Gaussian which describe as well as possible the values of every point of our original function. The white dot in figure (b) is a sample of the multivariate Gaussian representing a guess on the original function.



(c) Function estimated from the multivariate Gaussian. Correspond to the white dot in (b)

In practice we can't work in an infinite space. In fact, we operate over the finite set of the training data. This process will produce a distribution which describe t. This distribution can be used to make prediction for never seen inputs. Now we will explain how we can define a prior distribution over the functions. To do so we recall what we did for linear Bayesian regression. So taken a generic parametric model we have

$$y(x, w) = w^T \Phi(x)$$

In the case of ridge regression we have that the prior over w is

$$w \sim \mathcal{N}(w|0, \tau I)$$

So how is y distributed? We know that the linear combination of Gaussian is still Gaussian. So knowing that y is a linear combination of w, we are sure that it is distributed as a Gaussian. Now we can calculate its mean and variance.

$$\begin{aligned} E[y] &= \Phi E[w] = 0 \\ Cov[y] &= E[yy^T] = \Phi E[ww^T]\Phi^T = \tau \Phi \Phi^T = K, \text{ Gram matrix} \\ K_{nm} &= k(x_n, x_m) = \tau \Phi(x_n)^T \Phi(x_m) \end{aligned}$$

So we have

$$y \sim \mathcal{N}(y|0, K) \tag{62}$$

To justify why K is the covariance matrix of y, we can observe that the Gram matrix components are the kernel function values of input pairs. We have said that the kernel function measure the similarity between two inputs. So K measure the similarity between all inputs pairs and so the correlation of the outputs y. Now we can define a more formal and organized definition of Gaussian processes.

Definition 5.1 (Gaussian process)**.** *A Gaussian process is defined as a probability distribution over function y(x), such that the set of values of y(x), evaluated at an arbitrary set of point* $\{x_1, \ldots, x_N\}$*, jointly have a Gaussian distribution*

Being a Gaussian, the distribution can be completely specified by the mean and covariance.

Note - Fitting As for every kernel method, the choice of the kernel is very important, because it defines how the inputs are correlated. A hyperparameter which control the under-overfitting of the method is the bandwidth of the Gaussian (σ). A narrow bandwidth means that inputs near each other will be highly correlated and the correlation between inputs will decrease very fast as we increase the "distance" between the inputs. On the other hand, for wider bandwidth even slightly far away inputs will be correlated. In case of overfitting we would like to use wider bandwidth because the sample will be less correlated locally, thus decreasing the probability of fitting the noise. From another point of view, if we increase the bandwidth every inputs will "see" more inputs ,and so it will have more samples to estimate the function value. In the same way, if we are underfitting we can have narrower bandwidth.

5.2.1 Prediction

In this section we will see how we can predict the value of our function in input points never seen before. We consider the case in which we use Gaussian processes for regression. As usual for every regression method we define our target value as

$$t_n = y(x_n) + \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2)$$

Under the assumption that the noise is distributed as a Gaussian, we can say that the conditional distribution

$$P(t_n|y_n) = \mathcal{N}(t_n|y_n(x_n), \sigma^2) \tag{63}$$

We can also assume that the noise is independent on each data point. So the joint distribution of t is still Gaussian

$$p(t|y) = \mathcal{N}(t|y, \sigma^2 I) \tag{64}$$

Since $p(y) = \mathcal{N}(0, K)$, we can compute the marginal distribution $p(t)$[29]

$$p(t) = \int p(t|y)p(y)dy = \mathcal{N}(t|0, C), \text{where } C = K + \sigma^2 I_N \tag{65}$$

Now suppose we want to make a prediction t_{N+1} for a new data input x_{N+1}, given the training data. Our goal is to evaluate the predictive distribution $p(t_{N+1}|t^{(N)}, x_1, \ldots, x_N + 1)$[30] We can calculate that

$$\begin{aligned} p(t^{(N+1)}) &= \mathcal{N}(t^{(N+1)}|0, C^{(N+1)}), \text{ where} \\ C^{(N+1)} &= \begin{bmatrix} C^{(N)} & k \\ k^T & c \end{bmatrix}, \quad [(N+1)x(N+1)] \\ k &= \begin{bmatrix} k(x_1, x_{N+1}) & \ldots & k(x_N, x_{N+1}) \end{bmatrix}^T, \quad [Nx1] \\ c &= k(x_{N+1}, x_{N+1}), \quad [1x1] \end{aligned}$$

We can observe that for the Gaussian distribution properties, the predictive distribution is still a Gaussian. From that, we can apply the properties over conditional Gaussian distribution to obtain $p(t_{N+1}|t^{(N)}, x_1, \ldots, x_N + 1)$.

$$p(t_{N+1}|t^{(N)}, x_1, \ldots, x_N + 1) \sim \mathcal{N}(t_{N+1}|k^T {C^{(N)}}^{-1} t, c - k^T {C^{(N)}}^{-1} k) \tag{66}$$

We can observe two things. The mean and the variance depend on x_{N+1}. More interestingly, the mean of the prediction is actually what we obtain for the kernel version of ridge regression. This shouldn't be a surprise, because we have already said that in the parametric word, ridge regression is a particular case of linear Bayesian regression, when the prior is Gaussian and centered around zero. So this particular case holds also in the kernel world. We see that to calculate our prediction we need to invert C. This operation is always possible because by definition K is a Gram matrix and so its semi-definite positive. If we add to K $\sigma^2 I$, we are sure that C is positive definite and so it is for sure invertible.

[29] As $p(w)$ is the prior in liner Bayesian regression, $p(t)$ is the prior with Gaussian processes

[30] $t^{(N)}$ is the t vector when we have N sample. The professor in the lecture uses $\boldsymbol{t}_N$ but I found it a little misleading, because the difference between the bold character and the normal one can be easily missed. Also putting the number of sample as a superscript reminds of the iterations count in other notation

Note - computational cost As usual, inverting a matrix is the most intensive operation of the solution. In our case, the complexity of inverting C will be $\mathcal{O}(N^3)$. Luckily we need to compute this only one once for the given training set. We can also observe that the complexity depends only on the number of samples, as it should be for kernel methods. When we obtain a new sample, we can use the already calculated ${C^{(N)}}^{-1}$ to simplify the complexity of calculating the mean and variance of the predictive distribution. Indeed, we have that the computational cost of the mean is $\mathcal{O}(N)$ and of the variance is $\mathcal{O}(N^2)$. For large dataset is very expensive to calculate exactly the result, so we resort to approximated methods like random sampling and clustering.

Example - Prediction Suppose to have a regression problem we want to solve with Gaussian processes. For simplicity we assume that our function is described by t_1 and t_2, which are the function value for x_1 and x_2. First, we construct our prior p(t) (red ellipses). We assume a zero mean distribution, where the shape of the multivariate Gaussian prior depends on the covariance matrix, and so on the kernel we choose. Now we haven't observed any data, so our best guess for both t_1 and t_2 is zero. Now we observe in x_1 a value for t_1(blue dot). We know that t_1 and t_2 are correlated, so observing t_1 will give us some information about t_2. We know that the predictive distribution $p(t_2|t_1, x_1, x_2)$(green Gaussian) is a Gaussian. In the prior plot below we can visualize this distribution through cutting the prior $p(t)$ parallel to t_2 through t_1(blue line). This slice of $p(t)$ will be $p(t_2|t_1, x_1, x_2)$. Now we can take the mean to estimate t_2(green point).

As we have said before, like we need to define features in the parametric world, we need to define a kernel in the non-parametric case. In some cases, the kernel have some hyperparameters. How can we find the optimal values for this hyperparameters? We have already seen how cross validation can be used to do model selection. This method is very robust, but at the same time it is slow. Another approach uses the maximization of the marginal likelihood using gradient optimization. In practice, you want to find the hyperparameters for which the observed target variables are more likely. This is faster, but it can be stuck in local minima. Usually the gradient approach is the go to method for hyperparameters optimization.