Lecture Notes on Machine Learning

Sébastien DA VEIGA
ENSAI - CREST

2022–2023

Abstract

These lecture notes have been prepared for third-year students at ENSAI following the Machine Learning course. They consist of the last three 3-hours lectures, dedicated to the following topics:

- Support Vector Machines
- Kernel methods
- Aggregation of ML models

Contents

Lecture 1: Support Vector Machines

October 28, 2022

- Reminder on classification
  - Risk & Bayes classifier
  - Linear classifiers & Logistic Regression
- Support Vector Machines
  - Linearly separable case – Hard margin classifier
  - Non-linearly separable case – Soft margin classifier
  - Non-linear extension with kernels
  - The kernel jungle / The kernel zoo
  - Concluding notes
    - Hyperparameters
    - Extension to multi-class classification
    - Numerical examples with code

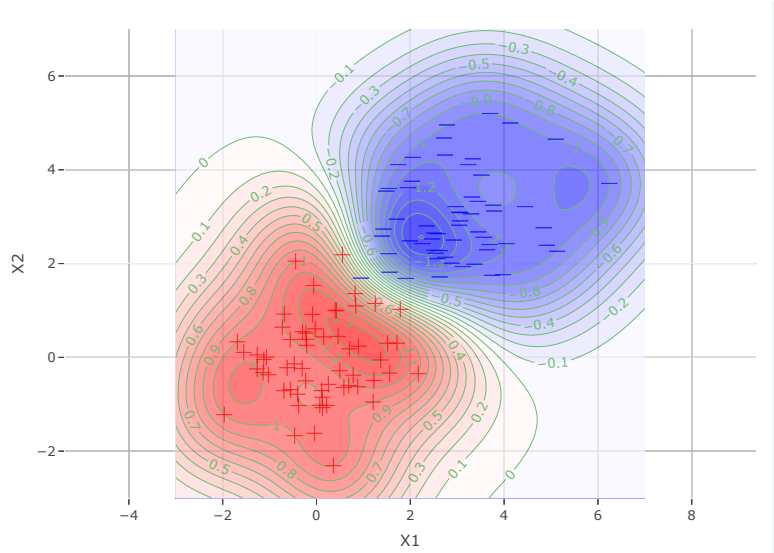
Chapter 1

Reminder on classification

1.1 Problem setting and notations

During this lecture, we will focus on a classification problem, where we are given a training sample $(\boldsymbol{x}_i, y_i)_{i=1,\dots,n}$ of size n from an unknown joint distribution $\mathbb{P}_{\boldsymbol{X}Y}$. Here $\boldsymbol{X} = \left(X^{(1)}, \dots, X^{(d)}\right) \in \mathcal{X} \subset \mathbb{R}^d$ is the vector of the d-dimensional features and $Y \in \{C_1, \dots, C_M\}$ is the target variable with values taken from the M classes $\{C_1, \dots, C_M\}$.
Our goal is to build a *classifier* or *classification rule* $h : \mathbb{R}^d \to \{C_1, \dots, C_K\}$ from the training sample, which predicts the class $h(\boldsymbol{x})$ of any other individual with features $\boldsymbol{x}$. In the following, we will focus on **binary** classification, where the target variable can only take two values. Be aware that in the next sections these two labels will be encoded differently depending on the mathematical context (mainly $\{0, 1\}$ and $\{-1, +1\}$), but we will always make clear which encoding is being used.
Any classifier h in binary classification can be written through a so-called **decision function** $f(\boldsymbol{x})$, which serves as an intermediary to predict the class. When $f(\boldsymbol{x}) > 0$, the classifier takes the value $h(\boldsymbol{x}) = +1$ and when $f(\boldsymbol{x}) < 0$ we have $h(\boldsymbol{x}) = -1$ when using the $\{-1, +1\}$ label encoding. This means that in this case we can write the classifier as $h(\boldsymbol{x}) = \text{sign}(f(x))$ where $\text{sign}(a) = a/|a|$ is the **sign** function. The set of feature values $\left\{\boldsymbol{x} \in \mathbb{R}^d : f(\boldsymbol{x}) = 0\right\}$ corresponding to decision function values equal to 0 is known as the **decision boundary**.

Example (Illustration of the decision function in two dimensions). We give below an example of a decision function with a training sample of size 120 with two features $X^{(1)}$ and $X^{(2)}$ and a target variable Y with values in $\{-1, +1\}$. Samples from each class are represented with a '+' sign in red or a '−' sign in blue. We draw the level set values of the decision function. Observe in particular in the middle the decision boundary corresponding to a 0 value.



Recall that the **risk** $R(h)$ of a classifier h is defined through a **loss function** $l(y, y')$:

$$R(h) = \mathbb{E}_{\mathbb{P}_{\boldsymbol{X}Y}} \left[l(Y, h(\boldsymbol{X})) \right].$$

When the loss function is equal to the 0-1 loss $l(y, y') = \mathbb{1}_{y \neq y'}$, we can show that the binary classifier h^* that minimizes the risk relies on a decision function related to the conditional probability of the target variables given the features.

Definition 1 (Bayes classifier). For binary classification, the classifier h^* that minimizes the risk $R(h)$ with a loss function $l(y, y') = \mathbb{1}_{y \neq y'}$ is the **Bayes classifier** defined as

$$h^*(\boldsymbol{x}) = \begin{cases} 1 & \eta(\boldsymbol{x}) > \frac{1}{2} \\ 0 & \text{otherwise} \end{cases}$$

where $\eta(\boldsymbol{x}) = \mathbb{P}(Y = 1 | \boldsymbol{X} = \boldsymbol{x})$.
Here the two classes are encoded as $\{0, 1\}$.

The Bayes classifier is thus the best possible classifier with respect to the risk. Unfortunately its expression depends on the unknown joint distribution $\mathbb{P}_{\boldsymbol{X}Y}$, meaning that it is out of reach in practice since we only observe a n-sample $(\boldsymbol{x}_i, y_i)_{i=1,\dots,n}$ from $\mathbb{P}_{\boldsymbol{X}Y}$. Most machine learning algorithms for classification aim at building an approximation of the Bayes classifier h^* from this training sample. Since $\mathbb{P}_{\boldsymbol{X}Y}$ is unknown and that the risk cannot be computed, they typically rely on the **empirical risk** $\hat{R}(h)$ defined as the empirical mean of the loss function on the training sample instead of the true expectation with respect to $\mathbb{P}_{\boldsymbol{X}Y}$:

$$\hat{R}(h) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, h(\boldsymbol{x}_i)).$$

The difference between all of them lies in the assumptions they make, either on the joint distribution $\mathbb{P}_{\boldsymbol{X}Y}$, on the form of the classifier h they use to approxi-

mate h^* and on the loss function appearing in the risk. In the next Section, we recall some basic algorithms which belong to the class of linear classifiers.

1.2 Linear classifiers

We first start by recalling what linear classifiers are.

Definition 2 (Linear classifier)**.** A linear classifier h is a classifier with a decision function f such that the decision boundary $\left\{\boldsymbol{x} \in \mathbb{R}^d : f(\boldsymbol{x}) = 0\right\}$ is linear.

The following remark is trivial but essential.

Remark. A linear classifier does not necessarily require the decision function f itself to be linear.
In particular, any monotonic transformation of a linear function produces a decision function which is no longer linear, but with a linear decision boundary.

The first linear classifier you should have seen during the Supervised Learning course in 2nd year is the Bayesian Linear Discriminant Analysis (LDA) one. In this model, the main assumption is that for each class label, the features are distributed according to a multivariate normal distribution, i.e.

$$\mathbb{P}(\boldsymbol{x}|Y = C_m) \sim \mathcal{N}_{\boldsymbol{x}}\left(\mu_m, \Sigma\right).$$

The LDA classifier is obtained by replacing inside the Bayes classifier the value of $\eta(\boldsymbol{x})$ which derives from this assumption:

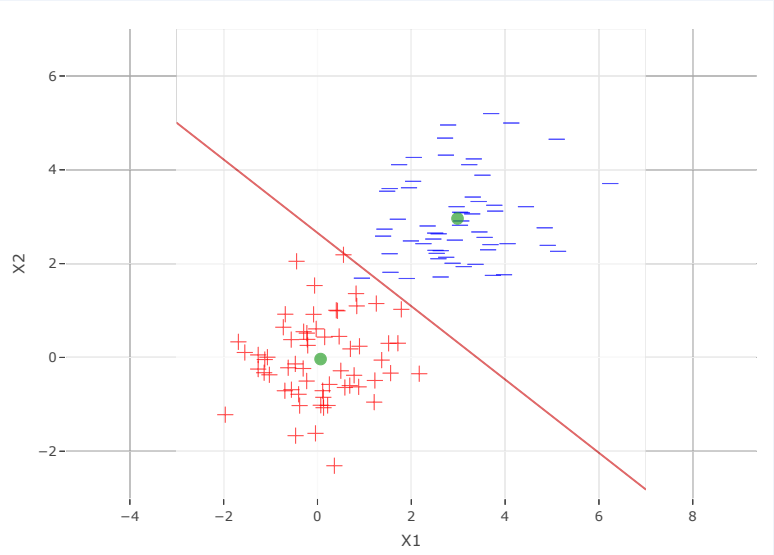
$$\begin{aligned}\eta_{\mathrm{LDA}}(\boldsymbol{x}) &= \mathbb{P}_{\mathrm{LDA}}(Y = 1|\boldsymbol{X} = \boldsymbol{x}) = \frac{\mathbb{P}_{\mathrm{LDA}}(X = \boldsymbol{x}\,|Y = 1)\pi_1}{\sum_{m=0}^{1}\mathbb{P}_{\mathrm{LDA}}(X = \boldsymbol{x}\,|Y = m)\pi_m} \\ &= \frac{\mathcal{N}_{\boldsymbol{x}}\left(\mu_1, \Sigma\right)\pi_1}{\sum_{m=0}^{1}\mathcal{N}_{\boldsymbol{x}}\left(\mu_m, \Sigma\right)\pi_m}\end{aligned}$$

by Bayes theorem and the law of total probability, where $\pi_m := \mathbb{P}(Y = m)$. The decision boundary then corresponds to $\eta_{\mathrm{LDA}}(\boldsymbol{x}) = 1/2$, or equivalently

$$\begin{aligned}&\mathcal{N}_{\boldsymbol{x}}\left(\mu_1, \Sigma\right)\pi_1 = \mathcal{N}_{\boldsymbol{x}}\left(\mu_0, \Sigma\right)\pi_0 \\ &-\frac{1}{2}(\boldsymbol{x} - \mu_1)^T\Sigma^{-1}(\boldsymbol{x} - \mu_1) + \log\pi_1 = -\frac{1}{2}(\boldsymbol{x} - \mu_0)^T\Sigma^{-1}(\boldsymbol{x} - \mu_0) + \log\pi_0 \\ &\boldsymbol{x}^T\Sigma^{-1}(\mu_1 - \mu_0) - \frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_0^T\Sigma^{-1}\mu_0 + \log\frac{\pi_1}{\pi_0} = 0\end{aligned}$$

which is the equation of an hyperplane as expected. Note that in practice μ_k, π_k and Σ are estimated with the training sample. We give below an example of Bayesian LDA on the dataset already used above.

Example (Illustration of Bayesian LDA). The linear decision boundary from Bayesian LDA corresponds to the line in the middle, and the estimated mean for each class is represented with a green point.



Remark. If for each class label we assume different covariance matrices Σ_k in the formula above, this leads to Bayesian Quadratic Discriminant Analysis which involves a quadratic decision boundary. The equation of the decision boundary is left as an exercise.

As opposed to Bayesian LDA, another approach is to not make such strong assumptions on $\mathbb{P}_{\boldsymbol{X}Y}$, but instead build a so called *plug-in classification rule* $\hat{h}$ defined as

$$\hat{h}(\boldsymbol{x}) = \begin{cases} 1 & \hat{\eta}(\boldsymbol{x}) > \frac{1}{2} \\ -1 & \text{otherwise} \end{cases}$$

where $\hat{\eta}(\boldsymbol{x})$ is an estimate of $\eta(\boldsymbol{x})$ built such that the empirical risk is minimized. If we focus on linear classifiers, the first intuitive idea is to propose a **parametric form** of $\hat{\eta}(\boldsymbol{x})$ such that the decision boundary is linear. Following what you saw for regression problems, you may be tempted to consider for example

$$\hat{\eta}(\boldsymbol{x}) = \beta_0 + \boldsymbol{x}^T\beta.$$

This will obviously produce a linear decision boundary, but since $\eta(\boldsymbol{x})$ is a (conditional) probability it may not be suited to use such an unconstrained parametric form (that does not guarantee values between 0 and 1). Instead, it is wiser to build upon a monotonic transformation following our previous remark. For example, we may write

$$\text{logit}\left(\hat{\eta}(\boldsymbol{x})\right) = \beta_0 + \boldsymbol{x}^T\beta$$

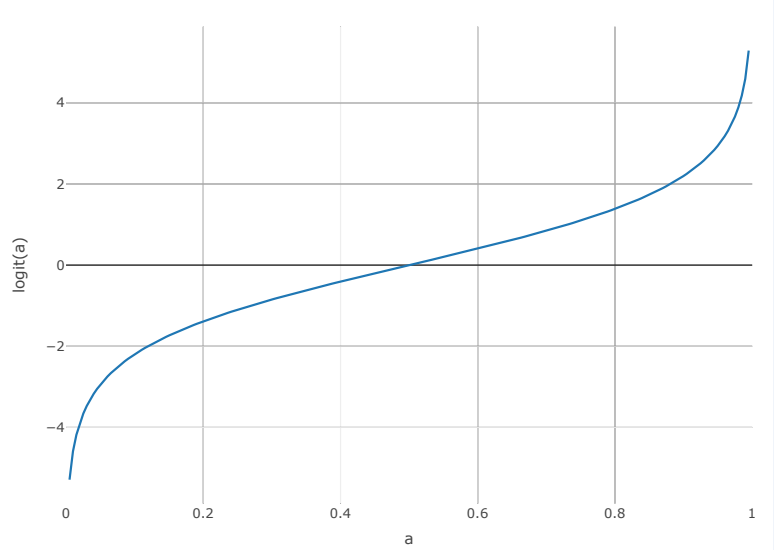
where $\text{logit}(a) = \log(a/(1-a))$, $a \in [0,1]$ is the so-called *logit* function, see the

example below for a visualization of this function. Equivalently, we have

$$\hat{\eta}(\boldsymbol{x}) = \frac{\exp(\beta_0 + \boldsymbol{x}^T\beta)}{1 + \exp(\beta_0 + \boldsymbol{x}^T\beta)}.$$

The logit function is monotonic, and so is its inverse, meaning that the corresponding plug-in classification rule will have a linear decision boundary.

Example (Logit function). Here is the shape of the logit function.



Now the last step is to find the best values for the parameters β_0 and β, and it is natural to try to minimize the empirical risk. If we consider the 0-1 loss, the corresponding empirical risk minimization problem writes

$$\min_{\beta_0,\beta} \hat{R}(\hat{h}) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{\text{sign}(\beta_0+\boldsymbol{x}^T\beta)\neq y_i}.$$

But here the indicator function is neither continuous nor differentiable, making the optimization problem hard to solve. A workaround is to consider a so-called *surrogate loss*, i.e. a loss function which penalizes classification errors similarly to the 0-1 loss while being smoother. A popular surrogate loss is the **cross-entropy loss** l_{CE}, given by

$$l_{\text{CE}}(y, \eta(\boldsymbol{x})) = -\mathbb{1}_{y=0}\log(1-\eta(\boldsymbol{x})) - \mathbb{1}_{y=1}\log(\eta(\boldsymbol{x})).$$

When applied to our parametric model, this leads to the new empirical risk minimization problem with $\{0, 1\}$ labels:

$$\min_{\beta_0,\beta} \hat{R}(\hat{h}) = \frac{1}{n}\sum_{i=1}^{n} \left[\log(1+\exp(\beta_0 + \boldsymbol{x}_i^T\beta))) - y_i(\beta_0 + \boldsymbol{x}_i^T\beta))\right].$$

The proof is left as an exercise. You may recognize this minimization problem from your Generalized Linear Regression course in 2nd year, it is called Logistic Regression (because it uses the logit function).

Remark. On a side note, you certainly derived it from another point of view. Indeed the same estimator can also be recovered with a (conditional) maximum likelihood approach by considering that Y is a Bernoulli random variable with probability $\hat{\eta}(\boldsymbol{x})$.

A few practical words about Logistic Regression:

- There is no closed-form solution to the minimization problem as in Linear Regression;
- But the loss function is convex, so you can use gradient descent, stochastic gradient descent or even more advanced optimizers, following what you have learned in the first part of this course;
- For high-dimensional problems, you can also use regularization with L_1 or L_2 penalties (you will have the opportunity to try that during the last computer class assignment).

Chapter 2

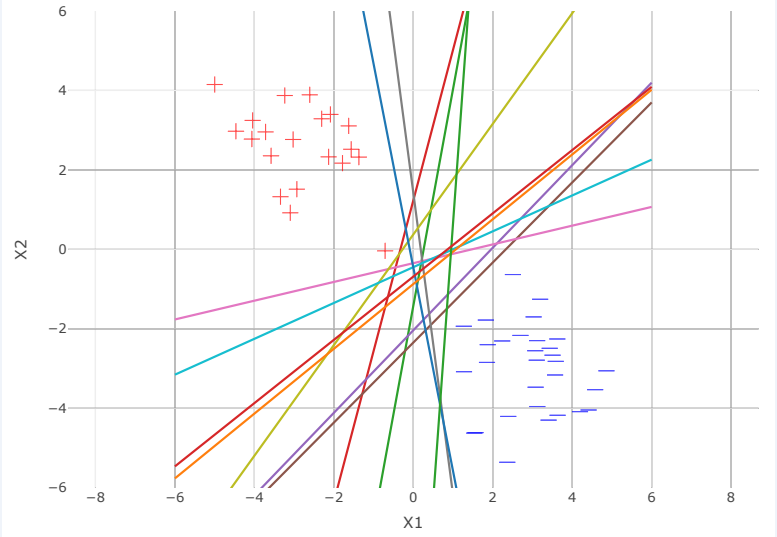
Support Vector Machines

Now that we have seen a brief summary of linear classifiers, we can take a step back and ask ourselves the kind of decision boundaries they can produce.
We will first assume that the training sample is *linearly separable*, meaning that there exists a hyperplane which can perfectly separate the two classes. This simplified situation will guide our intuition towards a linear classifier which should be robust, and help us design the *hard margin classifier*. When no perfect linear separation is possible, we will then extend the underlying ideas to the *soft margin classifier*. Finally, to account for even more non-linearity, we will introduce the concept of feature functions and kernels. The theory of kernels will be briefly tackled during Lecture 2.

2.1 Linearly separable case

To begin with, we will assume that the training sample is *linearly separable*, meaning that there exists a hyperplane which can perfectly separate the two classes. From a classification perspective, this means that we could train many linear classifiers with an empirical risk equal to 0 (i.e. they do not make classification errors). For instance, we may encounter the following situation depicted in the example below.

Example. We depict a linearly separable case where several linear classifiers have been trained on the training samples which are identified with a '+' sign in red or a '−' sign in blue according to their class. The different classifiers are represented with different colors.



By looking at their empirical risk, it is impossible to choose the 'best' one since they all have zero classification error on the training set. However, intuitively we could argue that from a generalization perspective (i.e. the classification errors made on an independent test set), it may be wiser to select a linear classifier with a decision boundary lying as far as possible to the training samples. This intuition is formalized mathematically through the concept of *margin*.

Definition 3 (Margin). The **margin** γ of a separating hyperplane is the distance between this hyperplane and the closest training sample.

The main idea is thus to find the hyperplane with the largest margin, while still separating perfectly the training sample. In the following, when we will mention a maximal or optimal margin, it will always imply that it is optimal among all hyperplanes which make zero classification errors. Before diving into the mathematical details, let us comment a few points (we will only consider the $\{-1, +1\}$ encoding below):

- Previously we have denoted a linear decision function as $f(\boldsymbol{x}) = \beta_0 + \boldsymbol{x}^T\beta$, but now following standard textbooks on support vector machines we will use the equivalent notation $f(\boldsymbol{x}) = b + \langle \boldsymbol{x}, \boldsymbol{w} \rangle$ where $\langle \cdot, \cdot \rangle$ is the Euclidean dot product in $\mathbb{R}^d$;
- When the margin is maximal, there is at least one $+1$ sample and one -1 sample with a distance equal to the margin γ (this is easy to prove);
- We will denote H the hyperplane with maximum margin, H^+ the hyperplane parallel to H at a distance γ on the $+1$ side and similarly H^- the hyperplane parallel to H at a distance γ on the -1 side;
- Training samples with a distance to H equal to the maximum margin γ are called **support vectors**, the terminology coming from the fact that they *support* H^+ and H^-;

- Multiplying b and $\boldsymbol{w}$ by a constant does not change the hyperplane equation. So for identification purposes we need to impose a scaling, and choose the following standard one:

$$b + \langle \boldsymbol{x}^+, \boldsymbol{w} \rangle = +1, \quad \forall \boldsymbol{x}^+ \in H^+$$
$$b + \langle \boldsymbol{x}^-, \boldsymbol{w} \rangle = -1, \quad \forall \boldsymbol{x}^- \in H^-$$

  by symmetry of H^+ and H^-.

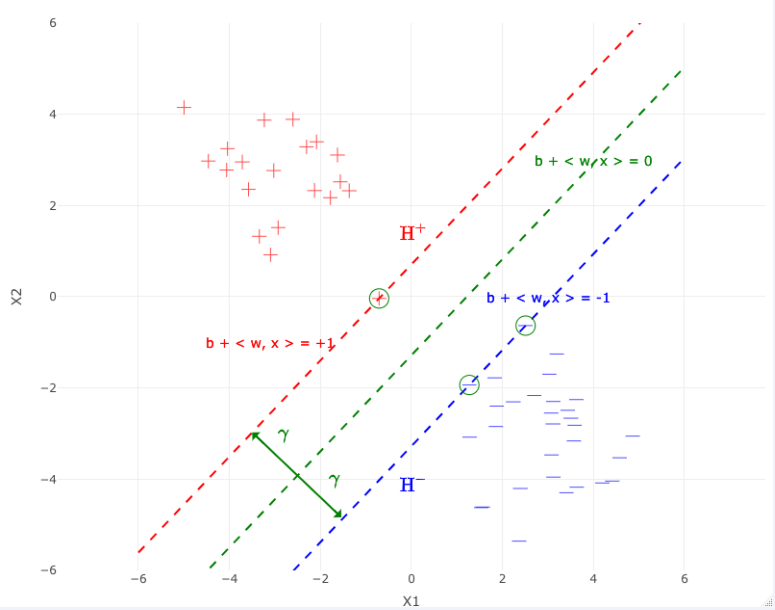
- With this scaling, the optimal margin is equal to

$$\gamma = \frac{1}{\|w\|_2}.$$

  The proof is left as an exercise (hint: consider two points on H^+ and H^- which are symmetric with respect to H and compute their distance).

We represent below all the introduced notations on a simple two-dimensional example.

Example. In the same example as above, the decision boundary with maximum margin is depicted in dashed green, while the H^+ and H^- hyperplanes are represented in dashed red and blue, respectively. Support vectors are identified with green circles.



Wrapping things up, we can now write the final optimization problem we have to solve:

1. We want to maximize $\gamma = 1/\|w\|_2$, **this is equivalent to minimizing** $\frac{1}{2}\|w\|_2^2$...;

2. ... while respecting that the hyperplane separates the classes, i.e.

$$\begin{aligned} \forall i = 1, \dots, n \qquad & \operatorname{sign}(f(\boldsymbol{x}_i)) && = y_i \\ \iff & y_i f(\boldsymbol{x}_i) && \geq 1 \\ \iff & y_i \left(b + \langle \boldsymbol{x}_i, \boldsymbol{w} \rangle\right) && \geq 1 \end{aligned}$$

Optimization Problem (Linearly separable maximum margin problem – **Primal form**). When the training sample is linearly separable, the maximum margin hyperplane $f(\boldsymbol{x}) = b + \langle \boldsymbol{x}, \boldsymbol{w} \rangle$ is obtained by solving

$$\begin{aligned} & \min_{\boldsymbol{w}, \beta} \ \tfrac{1}{2} \|w\|_2^2 \\ \text{subject to} \quad & y_i \left(b + \langle \boldsymbol{x}_i, \boldsymbol{w} \rangle\right) \geq 1 \, \forall i = 1, \dots, n. \end{aligned}$$

This is a quadratic optimization problem in dimension $d + 1$ with n linear inequality constraints. Usually, we consider instead the *dual problem* (i.e. with Lagrange multipliers) which leads to the following equivalent optimization problem.

Optimization Problem (Linearly separable maximum margin problem – **Dual form**). When the training sample is linearly separable, the maximum margin hyperplane $f(\boldsymbol{x}) = b + \langle \boldsymbol{x}, \boldsymbol{w} \rangle$ is obtained by solving the dual problem

$$\begin{aligned} & \min_{\alpha_1, \dots, \alpha_n} \ \tfrac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle \\ \text{subject to} \quad & 0 \leq \alpha_i \, \forall i = 1, \dots, n \\ \text{and} \quad & \sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned}$$

The optimal hyperplane is retrieved with

- $\boldsymbol{w}^* = \sum_{i=1}^{n} \alpha_i^* y_i \boldsymbol{x}_i$;
- b^* is obtained with the scaling constraints for support vectors;
- The training samples with $\alpha_i^* > 0$ correspond to **support vectors**;
- By plugging $\boldsymbol{w}^*$ inside f, the final decision boundary writes

$$f^*(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i^* y_i \langle \boldsymbol{x}_i, \boldsymbol{x} \rangle + b^*.$$

In particular, **only support vectors contribute to the decision boundary** (since all other training samples have $\alpha_i^* = 0$).

The proof is left as an exercise. This time, this is a quadratic optimization problem in dimension n with one linear equality constraint.

Since this classifier makes zero classification errors and leaves no sample at a distance to the hyperplane less than the margin, it is called a **hard margin classifier**.

2.2 Non-linearly separable case

What we just saw before is only suited if classes are linearly separable, since the previous optimization problems do not have a solution if this is not the case. The main question now is: How can we generalize the previous ideas to the non-separable case?

Ingredient 1. We still keep the idea of maximizing $1/\|w\|_2$

Ingredient 2. But we will authorize classification errors, meaning that some examples will be on the wrong side of the hyperplane, and as well tolerate samples which are classified correctly but which are at a distance smaller than $1/\|w\|_2$ (inside the margin). But of course we want these errors to be as small as possible.

As was done for logistic regression, we will minimize the errors by minimizing the empirical risk with a loss function. Recall that a linear classifier writes $h(\boldsymbol{x}) = \text{sign}(f(\boldsymbol{x})) = \text{sign}(b + \langle \boldsymbol{x}, \boldsymbol{w} \rangle)$ for the $\{-1, +1\}$ encoding. Previously, we talked about:

- The 0-1 loss $\mathbb{1}_{y \neq h(\boldsymbol{x})}$, which can be rewritten $\mathbb{1}_{yf(\boldsymbol{x})<0}$;
- The cross-entropy loss, which can be rewritten $\log(1 + \exp(yf(\boldsymbol{x})))$.

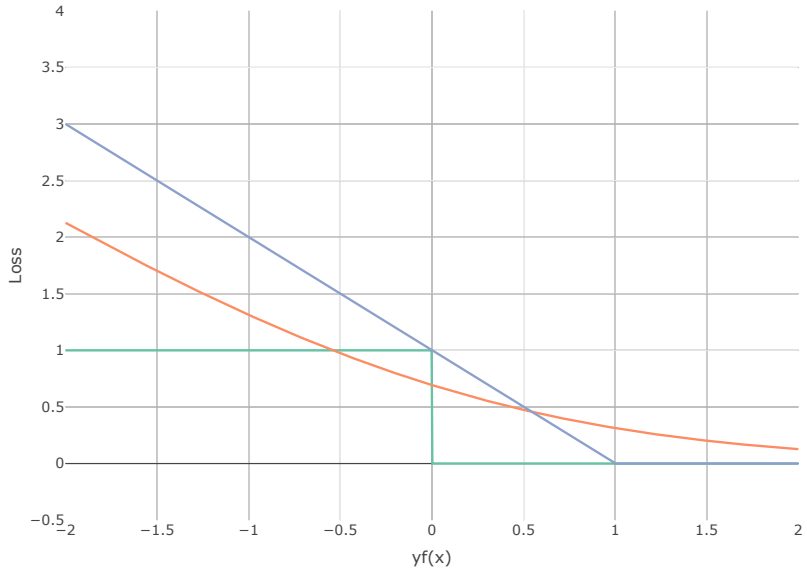
For support vector machines, another loss function was introduced, the *hinge loss*.

Definition 4 (Hinge loss). With the $\{-1, +1\}$ encoding, the **hinge loss** of a binary classifier $h(\boldsymbol{x}) = \text{sign}(f(\boldsymbol{x}))$ is given by

$$\max(0, 1 - yf(\boldsymbol{x})) = [1 - yf(\boldsymbol{x})]_+$$

where $[\cdot]_+$ is the positive part function.

We give below a graphical comparison of these three loss functions (0-1 in green, cross-entropy in orange and hinge in blue).



Remark. A justification for using the hinge loss is sometimes found in textbooks: this is the smallest convex function which lies above the 0-1 loss.

We can now come back to our original problem: we want to maximize the margin and at the same time minimize the empirical risk measured with the hinge loss. This is achieved by defining an objective function as a weighted combination of these two objectives:

$$\min_{\boldsymbol{w},\beta} \frac{1}{2}\|w\|_2^2 + \frac{C}{n}\sum_{i=1}^{n}[1 - y_i f(\boldsymbol{x}_i)]_+.$$

Here, the constant $C > 0$ determines a **trade-off** between increasing the margin and ensuring that the training samples lie on the correct side of the hyperplane. To make a connection with what you saw in the previous class, we can rewrite the problem as

$$\min_{\boldsymbol{w},\beta} \frac{1}{n}\sum_{i=1}^{n}[1 - y_i f(\boldsymbol{x}_i)]_+ + \lambda\|w\|_2^2$$

where $\lambda = 1/2C$ is a regularization parameter for an empirical risk minimization with L_2 regularization. This looks a lot like Ridge Regression, but with another loss function specific to classification (the hinge loss).

This optimization problem involves a continuous but non-differentiable loss function (due to the hinge loss). Even if now more modern approaches have been proposed to directly handle it (sub-gradient or stochastic sub-gradient descent), below we will detail the original proposal from the literature on support vector machines, which is the one implemented in most legacy packages. It is based on the classical trick to add *slack variables* to the optimization problem.

Optimization Problem (Non-linearly separable maximum margin problem – **Primal form**). When allowing classification errors, the maximum margin hyperplane $f(\boldsymbol{x}) = b + \langle \boldsymbol{x}, \boldsymbol{w} \rangle$ is obtained by solving

$$\begin{aligned} & \min_{\boldsymbol{w}, \beta, \boldsymbol{\xi_1}, \ldots, \boldsymbol{\xi_n}} \frac{1}{2}\|w\|_2^2 + \frac{C}{n}\sum_{i=1}^{n} \boldsymbol{\xi_i}^2 \\ \text{subject to} \quad & \boldsymbol{\xi_i} \geq 0 \, \forall i = 1, \ldots, n. \\ \text{and} \quad & y_i \left(b + \langle \boldsymbol{x}_i, \boldsymbol{w} \rangle\right) \geq 1 - \boldsymbol{\xi_i} \, \forall i = 1, \ldots, n. \end{aligned}$$

where the $\boldsymbol{\xi_i}$'s are **slack variables**.

This is a quadratic optimization problem in dimension $d + n + 1$ with n linear inequality constraints. As before, we will prefer working with its dual formulation.

Optimization Problem (Non-linearly separable maximum margin problem – **Dual form**). When allowing classification errors, the maximum margin hyperplane $f(\boldsymbol{x}) = b + \langle \boldsymbol{x}, \boldsymbol{w} \rangle$ is obtained by solving the dual problem

$$\begin{aligned} & \min_{\alpha_1, \ldots, \alpha_n} \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle \\ \text{subject to} \quad & 0 \leq \alpha_i \leq \frac{\boldsymbol{C}}{\boldsymbol{n}} \, \forall i = 1, \ldots, n \\ \text{and} \quad & \sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned}$$

An **important remark** is that this dual problem is exactly the same as in the linear separable case **with just the additional bound constraints** $\boldsymbol{\alpha_i} \leq \frac{\boldsymbol{C}}{\boldsymbol{n}}$.
The optimal hyperplane is retrieved with

- $\boldsymbol{w}^* = \sum_{i=1}^{n} \alpha_i^* y_i \boldsymbol{x}_i$;
- b^* is obtained with the scaling constraints for support vectors;
- The training samples with $\alpha_i^* > 0$ correspond to **support vectors**;
- By plugging w^* inside f, the final decision boundary writes

$$f^*(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i^* y_i \langle \boldsymbol{x}_i, \boldsymbol{x} \rangle + b^*.$$

In particular, **only support vectors contribute to the decision boundary** (since all other training samples have $\alpha_i^* = 0$).

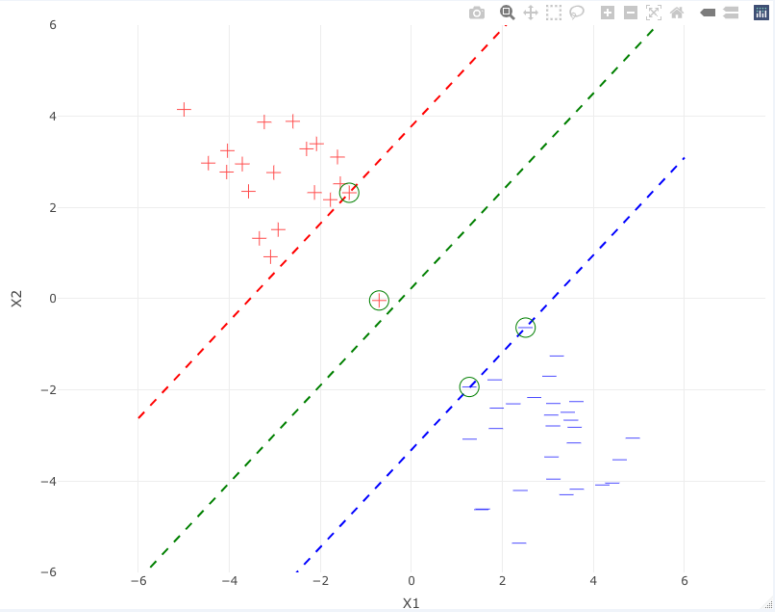
Contrary to the previous hard-margin classifier, since now the classifier is allowed to make misclassifications and tolerates samples at a distance to the hyperplane less than the margin, it is called a **soft margin classifier**.

As mentioned before, solving this dual problem is what is done in legacy libraries, with specific ultra-optimized solvers. A famous library is LIBSVM, which is in particular used in scikit-learn.

Concerning the choice of C, it is typically found with cross-validation following standard practice for regularized problems. But note that there is a support vector machines variant called ν-SVM where C is replaced with a more interpretable hyperparameter ν (number of margin errors and support vectors).

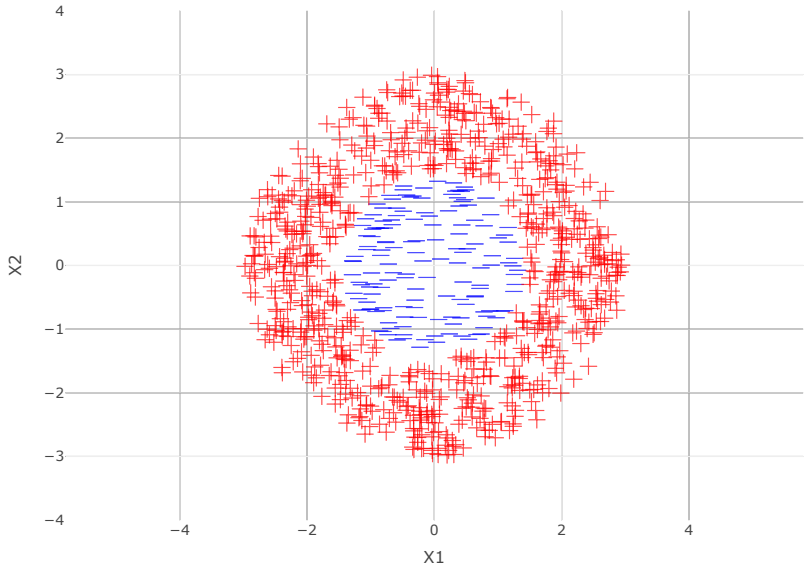
Finally, even if data are linearly separable, using a soft margin classifier instead of the hard one may be beneficial! We discuss the example presented at the beginning of the Section below.

Example. In the same example as above, the '+' support vector in the middle implies that the hard margin hyperplane is much closer to the '-' class than to the '+' class in average. The '+' support vector is not really an outlier, but a value more 'extreme' than the other '+' samples. In such a case to balance the classes, a soft margin classifier, allowing the '+' support vector in the middle to be at a distance smaller than γ, may produce a more robust hyperplane. We show below how it looks like.



2.3 Non-linear extension with kernels

Now you know how to deal with the non-separable case, by balancing the margin and the classification errors. That is great, but do you think what you have learned would perform well if the training samples look like this?



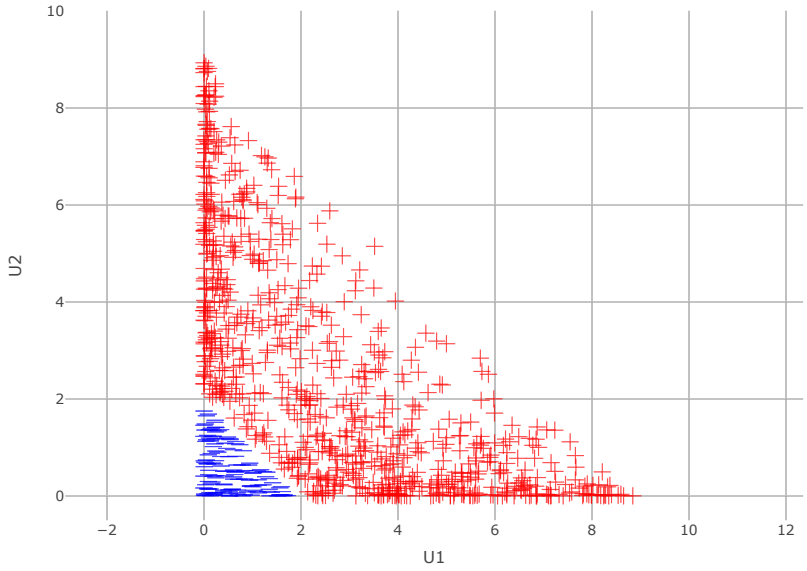
Every linear classifier will make a lot of classification errors in this case! But here, in this specific example, a boundary decision function like

$$f(\boldsymbol{x}) = (x^{(1)})^2 + (x^{(2)})^2 - r^2,$$

would lead to a much better classifier. Of course, f is not linear in $x^{(1)}$ and $x^{(2)}$. But it is linear in $(x^{(1)})^2$ and $(x^{(2)})^2$! Now if we define the function

$$\begin{array}{rcl} \phi: \mathbb{R}^2 & \to & \mathbb{R}^2 \\ \left(x^{(1)}, x^{(2)}\right) & \mapsto & \left(u^{(1)} = (x^{(1)})^2, u^{(2)} = (x^{(2)})^2\right), \end{array}$$

then f is linear in $\phi(\boldsymbol{x})^{(1)}$ and $\phi(\boldsymbol{x})^{(2)}$.
This remark thus gives us a straightforward method to recycle what we developed for linear classifiers: we first transform the training sample through function ϕ, meaning that we replace the original features $X^{(1)}$ and $X^{(2)}$ with $U^{(1)} = \phi(\boldsymbol{X})^{(1)}$ and $U^{(2)} = \phi(\boldsymbol{X})^{(2)}$, see how the training samples look like after such transformation below.



In this transformed space, the samples can easily be classified with a linear boundary decision function. From a practical point of view, in our previous optimization problems we just need to do one thing: everytime an $\boldsymbol{x}_i$ appears, we simply replace it with the transformed sample $\phi(\boldsymbol{x}_i)$, the image of the original sample $\boldsymbol{x}_i$ through the function ϕ. For example, the soft margin dual problem is now

$$\begin{aligned} & \min_{\alpha_1,\dots,\alpha_n} \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j)\rangle_{\mathcal{H}} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq \frac{C}{n}\ \forall i = 1,\dots,n \\ \text{and} \quad & \sum_{i=1}^{n}\alpha_i y_i = 0 \end{aligned}$$

and the final decision function writes

$$f^*(\boldsymbol{x}) = \sum_{i=1}^{n}\alpha_i^* y_i \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x})\rangle_{\mathcal{H}} + b^*.$$

Note that we added a subscript $\mathcal{H}$ for the dot product, we will comment this point soon below.
We now give a few words about the function ϕ:

(1) In our example, we have defined ϕ **manually**, being guided by the training samples.
This was easy because we were just in dimension two (easy visualization) and the pattern was simple (a circle). But in general, this would be much more complicated! So this manual trick is not really generalizable.

(2) Related to the above, we just used $\phi : \mathcal{X} \to \mathbb{R}^2$ with $\mathbb{R}^2$ only because it was easy to identify.
In general, we may want to build a much larger number of new features,

with the hope that with some of them the data will become linearly separable. In other words, in a high-dimensional new space, we expect to increase the chances of a linear separation. We even can (and will do!) use a function $\phi : \mathcal{X} \to \mathcal{H}$ going to an infinite-dimensional space $\mathcal{H}$.

(3) But to have the right to do so with our previous results, $\mathcal{H}$ cannot be any space. Since ϕ is only involved in scalar products, we only need $\mathcal{H}$ to be an Hilbert space.

(4) However specifying a (potentially) high-dimensional ϕ is cumbersome. But we have observed that ϕ only appears through dot products in both the optimization problem and the final decision function, so that we only have terms like

$$\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle_{\mathcal{H}}.$$

Let us write such dot products as

$$k(\boldsymbol{x}, \boldsymbol{x}') := \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle_{\mathcal{H}}.$$

Here k is a function $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which is symmetric and positive definite (due to the properties of dot products), which is called a **kernel**.

(4.1) From a computational and practical perspective, this means that we only need to specify the kernel k instead of ϕ (as long as it is symmetric and positive definite). This is easier, because:

- It is defined on $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$ versus $\phi : \mathcal{X} \to \mathcal{H}$ where $\mathcal{H}$ is high-dimensional;
- It can be intuitively interpreted as a dot product in an Hilbert space, that is to say a **function which measures the similarity between objects in** $\mathcal{X}$ (the feature space).
  This means for example that in classification we can easily handle features that are more 'exotic' than simple real variables, such as images, text sequences, time series and so on, as long as we are able to provide a similarity measure between them in the form of the kernel.

(4.2) From a theoretical perspective, specifying just the kernel k is also justified by the so-called **Moore-Aronszajn theorem**:

> **Theorem 1** (Moore-Aronszajn)**.** For any symmetric positive definite function $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, there exists a unique Hilbert space $\mathcal{F}$ and a function ψ such that for all $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X} \times \mathcal{X}$,
>
> $$\kappa(\boldsymbol{x}, \boldsymbol{x}') := \langle \psi(\boldsymbol{x}), \psi(\boldsymbol{x}') \rangle_{\mathcal{F}}.$$
>
> Note that $\mathcal{F}$ is unique, but ψ is not.

(5) Such a kernel k is at the heart of the famous class of **kernel** methods in machine learning and the famous **kernel trick**.
This kernel trick is simple to state: everytime you have an algorithm which involves dot products $\langle \boldsymbol{x}, \boldsymbol{x}' \rangle$ for features $\boldsymbol{x}$ (like SVM, but also PCA, and also Ridge Regression, more on this next class), meaning that

this is a linear algorithm, you can build a nonlinear counterpart by using $k(\boldsymbol{x}, \boldsymbol{x}')$ instead of $\langle \boldsymbol{x}, \boldsymbol{x}' \rangle$. Internally this will correspond to a nontrivial transformation ϕ of the features that you do not need to specify.

(6) Now coming back to SVM, the dual problem with the kernel k writes in the following form.

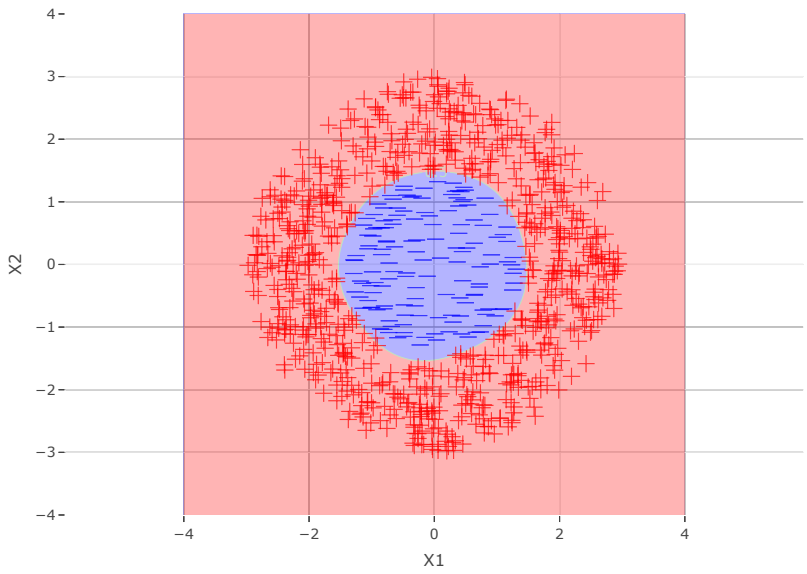
Optimization Problem (Non-linearly separable maximum margin problem **with a kernel k – Dual form**). When allowing classification errors, the maximum margin hyperplane with a kernel k is obtained by solving the dual problem

$$\begin{aligned} & \min_{\alpha_1,\dots,\alpha_n} \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq \frac{C}{n} \, \forall i = 1, \dots, n \\ \text{and} \quad & \sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned}$$

The final decision boundary writes

$$f^*(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i^* y_i k(\boldsymbol{x}_i, \boldsymbol{x}) + b^*.$$

If you now use a kernel on our motivating example (instead of specifying a function ϕ), we obtain the following decision boundary. We choose here the Gaussian kernel (see the next Section for the definition).



Finally, now that we have simplified the problem by replacing ϕ with k, how do you specify it in practice? A practical answer is given in the next Section.

2.4 The kernel jungle / The kernel zoo

Fortunately, at your level and in the scope of this class, things are quite easy for you because many smart people have worked on this subject and there is a list of **commonly used kernel expressions**. This list of kernels is usually referred to as the **kernel jungle** or **kernel zoo**. Here we will limit ourselves to $\mathcal{X} = \mathbb{R}^d$, and a typical list of kernels in this case is the following:

(a) The **vanilla dot product** or **linear** kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{x}, \boldsymbol{x}' \rangle.$$

Ok with this one, you just do standard linear SVM.

(b) The **polynomial** kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = (a + \langle \boldsymbol{x}, \boldsymbol{x}' \rangle)^p .$$

Not very powerful in practice, because it corresponds to a finite-dimensional Hilbert space.

(c) The **sigmoid** or **hyperbolic tangent** or **neural network** kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \tanh\left(a + \langle \boldsymbol{x}, \boldsymbol{x}' \rangle\right)$$

or more generally

$$k(\boldsymbol{x}, \boldsymbol{x}') = \tanh\left(a + \kappa(\boldsymbol{x}, \boldsymbol{x}')\right).$$

It had quite a success in the 'old days', despite the fact that it is not a kernel because it is not positive definite!

(d) The **Gaussian** or **RBF** (Radial Basis Function) or **Squared-Exponential** kernel

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{x}') &= \exp\left(-\frac{1}{2\theta^2}\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2\right) \\ &= \exp\left(-\gamma\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2\right). \end{aligned}$$

(Be careful that here γ is not the margin, just the notation of a constant which is used in scikit-learn).
This is a very popular choice, which is still used a lot in practice. But in my experience it is always behind other kernels in terms of predictive accuracy.

(e) The **Laplacian** or **Exponential** kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\gamma\|\boldsymbol{x} - \boldsymbol{x}'\|_2\right).$$

Not that is it not available in scikit-learn but in kernlab.

(f) The **spline** kernel (we will see in next class why such a name)

$$k(\boldsymbol{x}, \boldsymbol{x}') = \prod_{l=1}^{d}\left[1 + x^{(l)}x'^{(l)} + \frac{1}{2}x^{(l)}x'^{(l)}\min\left(x^{(l)}, x'^{(l)}\right) - \frac{1}{6}\min\left(x^{(l)}, x'^{(l)}\right)^3\right]$$

if $\boldsymbol{x}\ \boldsymbol{x}' \in [0,1]^d$. It is not available in scikit-learn but in kernlab.

(g) The **Matérn** or **Sobolev** kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu} \frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_2}{\theta} \right)^{\nu} K_\nu \left(\sqrt{2\nu} \frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_2}{\theta} \right)$$

where $\Gamma(\cdot)$ is the Gamma function and K_ν is a modified Bessel function. There are two interesting particular cases:

$$\text{for } \nu = 3/2,\ k(\boldsymbol{x}, \boldsymbol{x}') = \left(1 + \sqrt{3}\tfrac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2}{\theta}\right) \exp\left(-\sqrt{3}\tfrac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2}{\theta}\right),$$

$$\text{for } \nu = 5/2,\ k(\boldsymbol{x}, \boldsymbol{x}') = \left(1 + \sqrt{5}\tfrac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2}{\theta} + \tfrac{5}{3}\tfrac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2^2}{\theta^2}\right) \exp\left(-\sqrt{5}\tfrac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2}{\theta}\right).$$

Such kernels are very efficient in practice, but they are not available for SVMs in scikit-learn nor in kernlab.

2.5 Concluding notes

2.5.1 Choice of hyperparameters

For almost all previous kernels, there are hidden tuning parameters: a, p, γ, θ, ν and so on. These are additional hyperparameters (on top of the regularization constant C) which **must be tuned by cross-validation**.
But even in this case, you will sometimes see that it is not sufficient to reach a high prediction accuracy. This is often explained by the intrinsic limitation called *isotropic* kernels: in the list above you can see that there is only **one hyperparameter** in front of $\|\boldsymbol{x} - \boldsymbol{x}'\|_2$, which means that there is one hyperparameter common to all features. This implies two important things:

1. Before using a SVM algorithm with a isotropic kernel, you should **always normalize the features**.

2. If you have a problem where some features have predictive power larger than the others (this is often the case in practice), it is wiser to consider **one hyperparameter per feature**. Of course this will increase the computational cost of cross-validation, but this is the price to pay for flexibility.
   Unfortunately this is not an option for SVMs in scikit-learn or kernlab. You can either code it yourself, or use other similar kernel methods for which scikit-learn or other R packages offer the possibility to activate anisotropy, namely Gaussian Process Classification or Regression (but these methods are outside the scope of this class).

2.5.2 Extension to multi-class classification

We only tackled binary classification here. How do we generalize SVM to multi-class classification? There are two main approaches:

(1) The **one-versus-rest (OVR)** or **one-versus-all (OVA)** approach.
If there are M classes, we start by building M separate classifiers, each of

them being trained on a binary classification problem (the class of interest versus all the others). For prediction, we finally use

$$h(\boldsymbol{x}) = \text{sign}\left(\underset{m=1,\ldots,M}{\arg\max} f_m(\boldsymbol{x})\right)$$

where f_m is the decision function trained for the binary classification problem of class m versus all the others. This is the default in scikit-learn for Logistic Regression, for example.

(2) The **one-versus-one (OVO)** approach.
As the name suggests, it splits the training dataset into one dataset for each class versus every other class. This implies that we train $M(M-1)/2$ separate classifiers, but with smaller datasets. For prediction, each model predicts one class label and the class which receives the most votes is selected. This is the default in scikit-learn for SVM.

2.5.3 Numerical examples with code

See Computer Class 2!