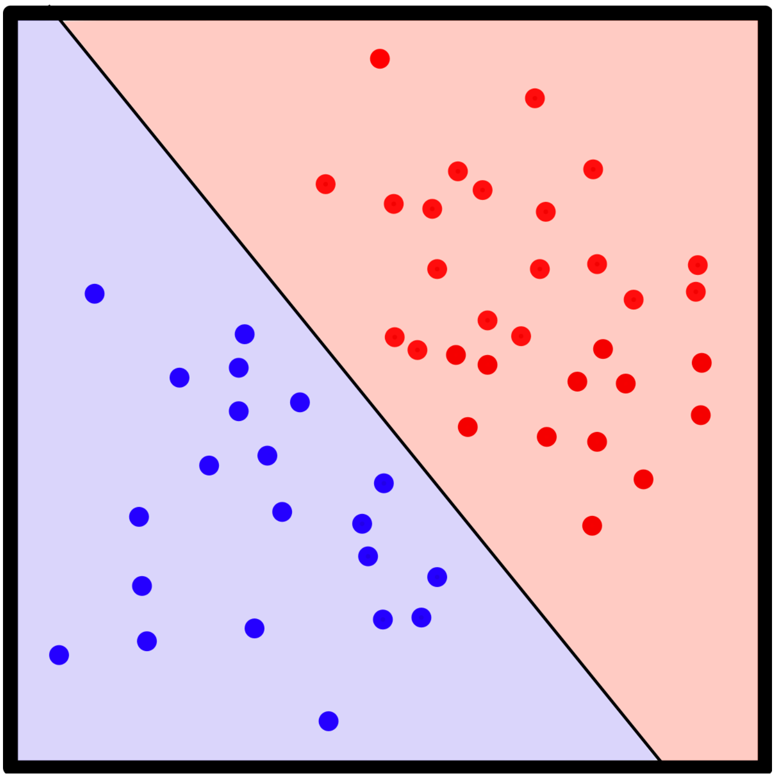
Linear models for classification

Introduction

The main asssumption when working with linear models is that the samples in the dataset have the property of being *linearly separable:*

We say that a dataset is linerarly separable if exists a linear function that separates the space in two or more region, where each region belongs to a certain class.



An example of a linearly separable dataset

The linear separator function can be a line in a 2-D case, while in generale it is a d-dimensional hyperplane when dealing with d-dimensional spaces.

An ideal resolution for this kind of problem is defining a linear model and train its parameters in order to predict new sample based on the region of space they belong.

The k-classes case

In the k-classes case we will define k linear combinations (one for each class) such that:

$$y_1(x) = w_1^T x + w_{10}$$
$$\dots$$
$$y_k(x) = w_k^T x + w_{k0}$$

In order to simplify the notation we define $\tilde{w} = \begin{bmatrix} w_0 \\ w \end{bmatrix}$ and $\tilde{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}$.

We can now express the model we want to learn as a column vector made by the k models:

$$y(x) = \begin{bmatrix} y_1(x) \\ ... \\ y_k(x) \end{bmatrix} = \begin{bmatrix} w_1^T x + w_{10} \\ ... \\ w_k^T x + w_{k0} \end{bmatrix} = \begin{bmatrix} \tilde{w}_1^T \\ ... \\ \tilde{w}_k^T \end{bmatrix} \tilde{x} = \tilde{W}^T \tilde{x}$$

Where we $\tilde{W}^T$ is the matrix that contains all the parameters for our model:

The goal of machine learning is to find the optimal parameters for $\tilde{W}$.

It is very important to keep in mind that a linear model is valid as long as the parameters are linear, meaning that for example $y(x) = w_1 x_1 + w_2 x_2^2 + w_3 x_3^3 + w_0$ is still a linear model.

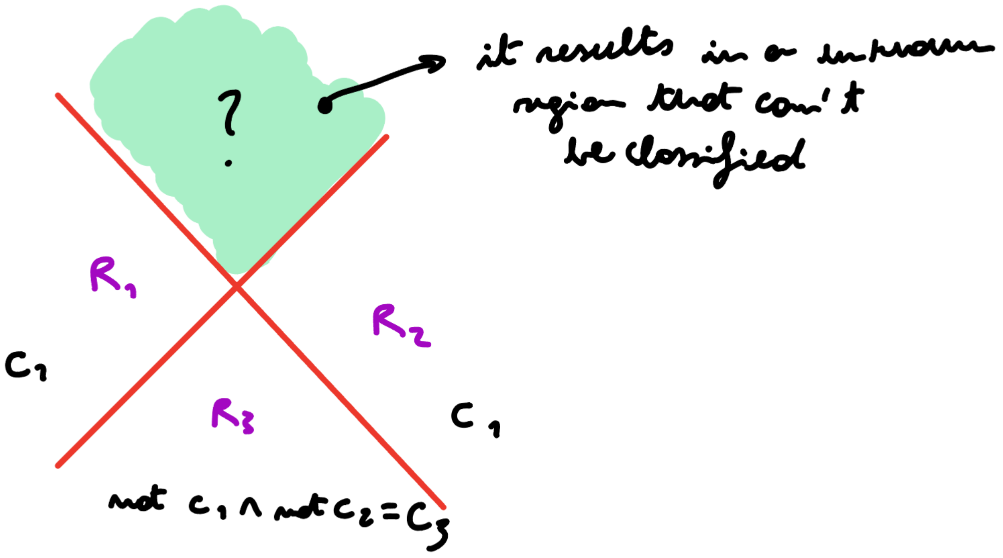
Some bad classifiers

When dealing with this problem we might think about a solution that involves binary classifiers, namely classifiers that tell apart one class from the other. The following examples show why it is a bad idea.

One-versus-the-rest classifier

Here we use k-1 classifiers in order that tell apart the i-th class from all the others. This basically means that for example the classifier for class C1 just knows whether a sample belongs to class C1 or not.

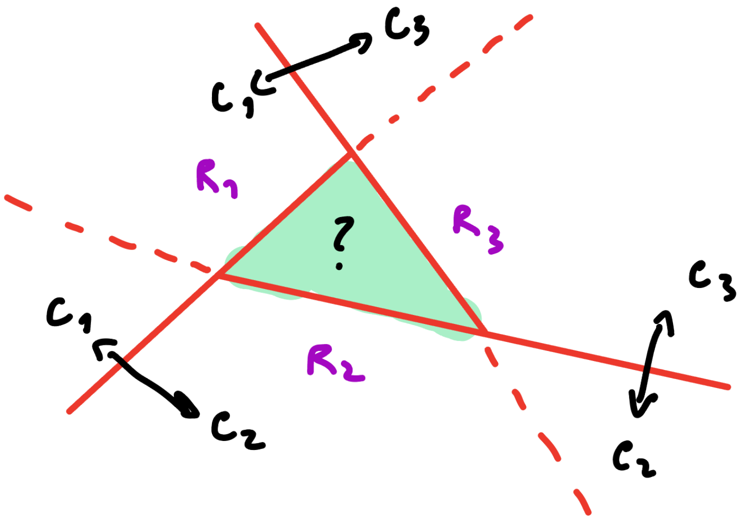
There is a major logic fault with this approach: there is always an unknown region that does not belong to any class, and it will cause conflict during the decision. The following is the example with three classes:



What class is the region that doesn't belong to any class?

One-versus-one classifier

This approach defines a classifier for each possible pair of classes to tell apart each class from the other. We use $\frac{k(k-1)}{2}$ classifiers. This approach still produces an uknown region at the inner intersection of the regions:



The right approach

The right approach to this problem is to define our classifier as a compound of k linear models:

$$y(x) = \begin{bmatrix} y_1(x) \\ ... \\ y_k(x) \end{bmatrix} = \begin{bmatrix} \tilde{w}_1^T \\ ... \\ \tilde{w}_k^T \end{bmatrix} \tilde{x} = \tilde{W}^T \tilde{x}$$

Each new sample will be classified according to the class that will produce the highest value for predictions, namely x is classfied as C_k if $y_k(x) > y_i(x)\ \forall i \neq k$.

The result is that a k-class discriminant will divide the space in a correct way. The following models show different solutions starting from this approach.

Least squares

Our input dataset is defined as $D = \{(x_n, t_n)_{n=1}^N\}$ implementing a 1-out-of-k encoding.

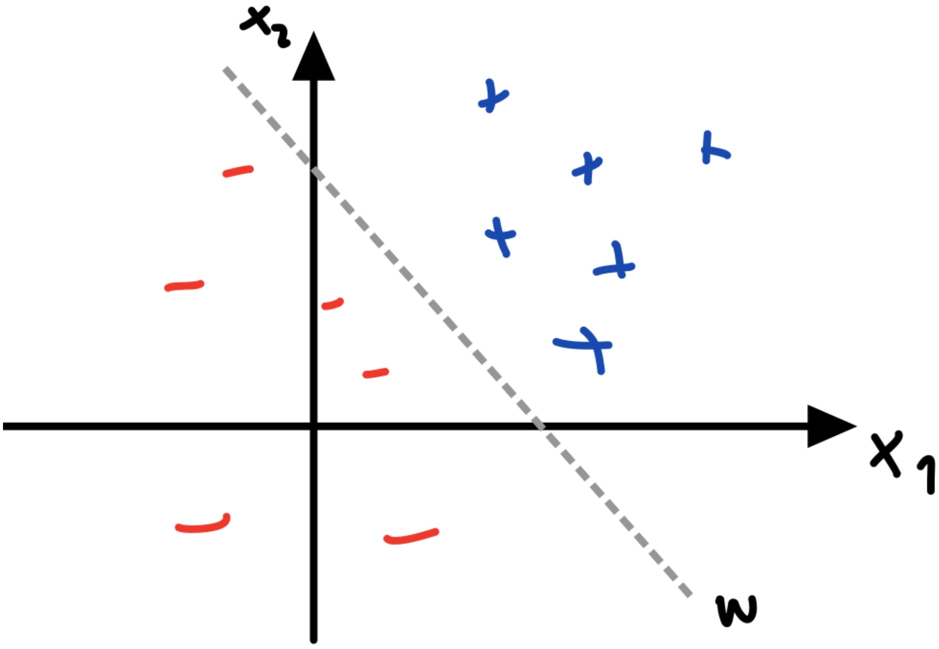
The error function for this model is defined as the sum of the squared errors, where the error is represented as the distance between the prediction $\tilde{X}\tilde{W}$ and the truth T :

$$E(\tilde{W}) = \frac{1}{2}\, Tr\{(\tilde{X}\tilde{W} - T)^T(\tilde{X}\tilde{W} - T)\}$$

The operator Tr stands for the trace of the matrix, namely the sum of the elements on its main diagonal (thus the sum of the squared errors). The constant 1/2 is there just for the purpose of being simplified when computing the derivative of the squared error.

Our goal is to have a zero distance between the prediction matrix and the true one, meaning that we want them to have the same values.

The result will be a linear separator parametrized by W:



The solution for finding the optimal parameters is:

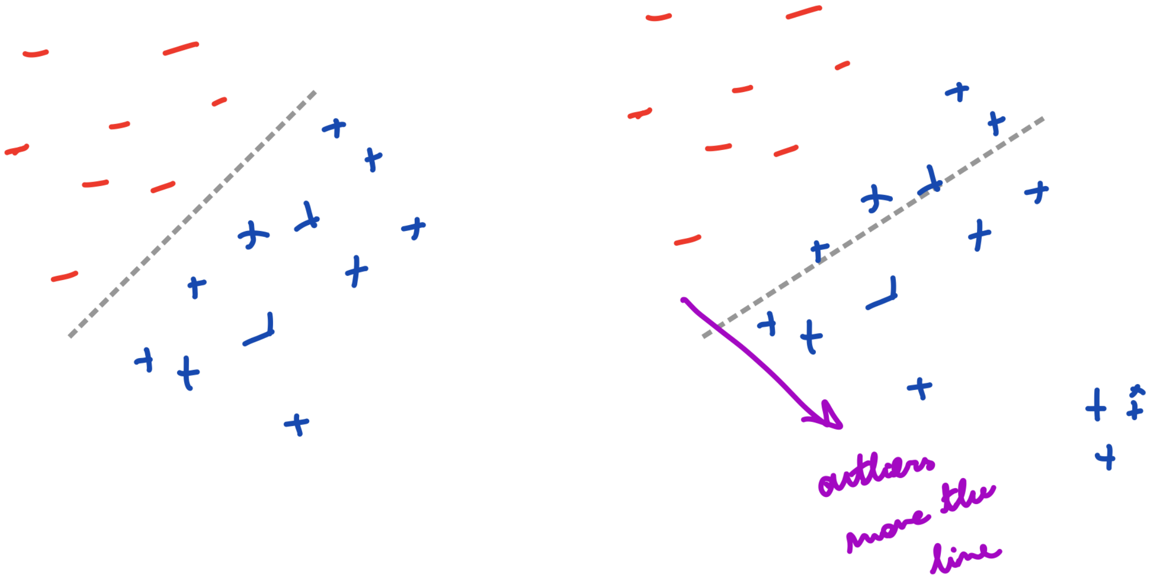
$$\tilde{W} = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T T$$

Where the term $(\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T$ is the pseudo inverse of $\tilde{X}$. Once the solution is computed, we will use the obtained weights inside our model to make predictions:

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

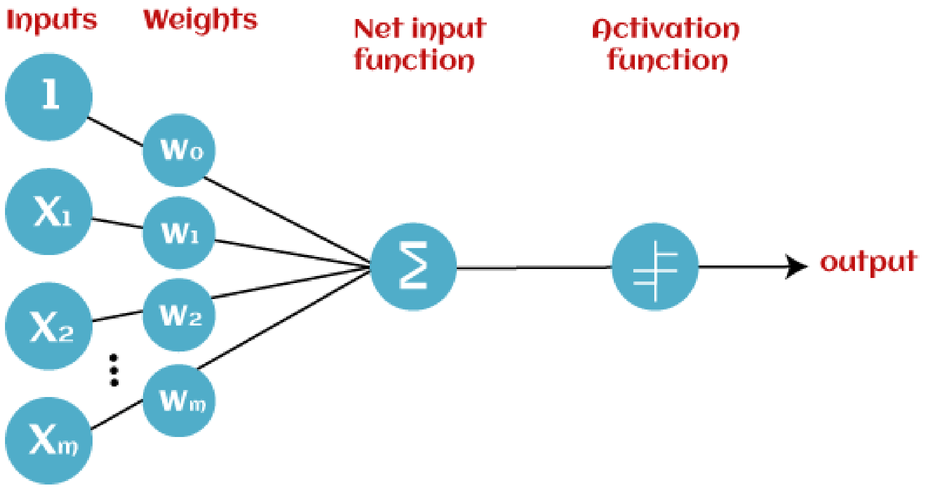
And the predicted class will be $C_k = argmax_{i=1,\ldots,k}(y_i(x))$.

The fundamental problem with the least squares method is that the linear separator depends only on the distance error, meaning that outliers will tend to move the line towards them, making the entire model less performant:



Perceptron

A perceptron unit is made by pipeline of two operations. The first one is computing weighted linear combination of the inputs, which is then fed to a sign function:



The first operator is $\Sigma = \sum_i w_i x_i$, while in this case the activation function is the sign function, which takes the weighted sum as an input, and maps its values to -1 if the sum is negative, 1 otherwise.

The whole model can be summarized as $y(x) = sign(W^T X)$.

Let's now take a step back and consider the unthresholded linear unit:

$$o = w_0 + w_1 x_1 + ... + w_n x_n$$

Given a training dataset $D = \{(x_n, t_n)_{n=1}^{N}\}$, we define the error function as the sum of the squared errors:

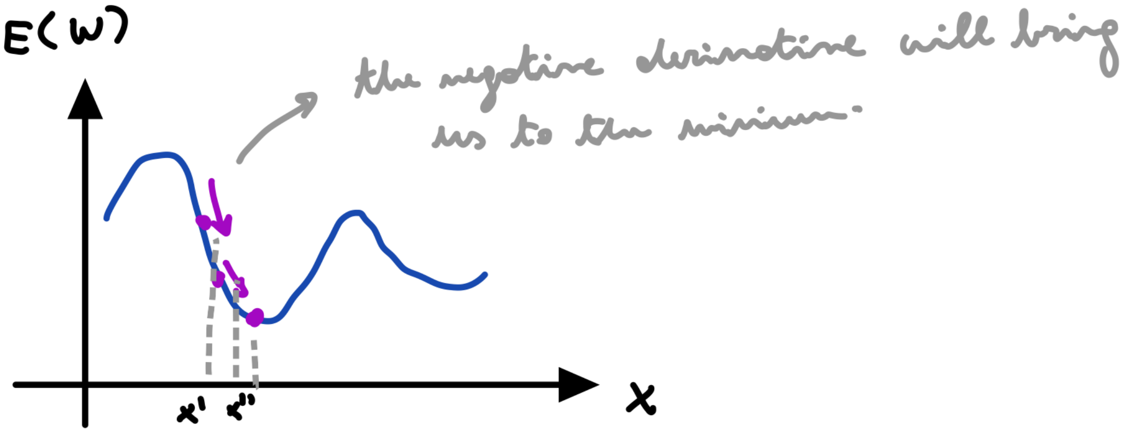
$$E(W) = \frac{1}{2}\sum_{n=1}^{N}(t_n - o_n)^2 = \frac{1}{2}\sum_{n=1}^{N}(t_n - w^T x_n)^2$$

In order to minimize the loss function we want to satisfy $E(w) \iff T = W^T X$.

In order to find the minimum of the error function we start by computing its gradient:

$$\frac{\delta E}{\delta w_i} = ... = \sum_{n=1}^{N}(t_n - w^t x_n)(-x_{i,n})$$

The negative gradient will show us the direction of maximum decrease, that we will follow using an iterative approach in order to reach the minimum.



For each step we update the weights by summing a certain delta, then we compute the new derivative:

$$w_i \leftarrow w_i + \Delta w_i$$
$$w_i = w_i - \eta\frac{\delta E}{\delta w_i}$$

Notice that we introduce the term η, that is a constant called the **Learning Rate**. The choice of the value for the learning rate is fundamental, since it represents by how much we move towards the negative direction. We will see better how this parameter will affect the computation of our model.

Now we have to pay attention to the composition of the delta term:

$$\Delta w_i = -\eta \frac{\delta E}{\delta w_i} = -\eta \sum_{n=1}^{N} (t_n - w^T x_n) x_{i,n}$$

The two terms of the subtraction $(t_n - w^T x_n)$ are incompatible, since $t_n \in \{0, 1\}$ while $w^t x_n \in \mathbb{R}$. In order to fix this, we can just use the sign function in order to map the real values of the second term to either 1 or -1. The resulting formula will be:

$$w_i \leftarrow w_i + \Delta w_i$$
$$\Delta w_i = w_i - \eta \frac{\delta E}{\delta w_i} = \eta \sum_{n=1}^{N} (t_n - \text{sign}(w^T x_n)) x_{i,n}$$

This formulation makes the computation of the model really robust to outliers, since even extreme values will be mapped to 1 or -1. The error will be 0 when the line will perfectly separate the data.

Implementation

When feeding the data to the training phase, we have three possible approaches:

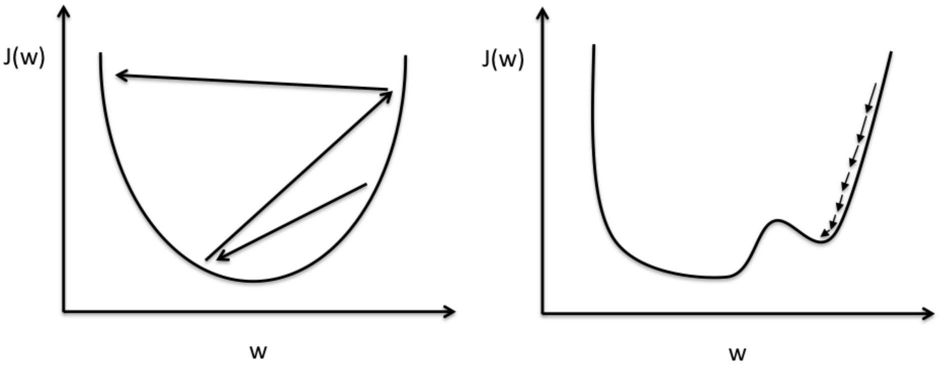
- **Batch mode:** we feed the whole datase $\Delta w_i = \eta \sum_{(x,t)\in D}^{N} (t_n - \text{sign}(w^T x_n)) x_{i,n}$
- **Mini-batch mode:** for each iteration we take a small subset $S \subset D$ $\Delta w_i = \eta \sum_{(x,t)\in S}^{N} (t_n - \text{sign}(w^T x_n)) x_{i,n}$
- **Incremental mode:** we feed one sample per step

The best approach happend to be the mini-batch mode, since batch mode can be too computationally expensive, while incremental mode becomes slower as the dataset gets larger.

Moreover, we need to consider these two conditions needed for the convergence of the algorithm:

- Data is linearly separable.
- The learning rate is sufficiently small.

The choice of learning rate is in fact crucial for the convergence of our algorithm. In fact if η is too large, we risk make large steps and missing the minimum, often resulting in an oscillation that will lead to divergence:



Large learning rate: Overshooting.

Small learning rate: Many iterations until convergence and trapping in local minima.

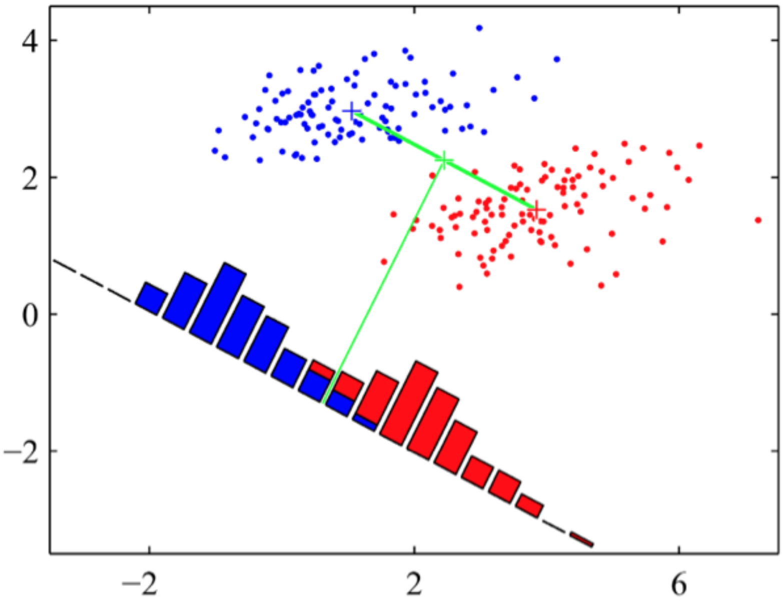
The learning rate will also affect the ability of our model of splitting the regions of our space. In fact for small values of η the linear separator will accumulate next to one of the classes, resulting in overfitting.

The termination conditions for our algorithm will be two:

- When the error is zero: this is the optimal solutions, but not always reachable.
- We define a thresholds for the changes of the loss function. When the following step will produce a change smaller than the threshold, we will stop.

Fisher's linear discriminant

The core idea about Fisher's linear discriminant is to place the linear separator in the middle of the line that connects the centers of mass (means) of the distribution:

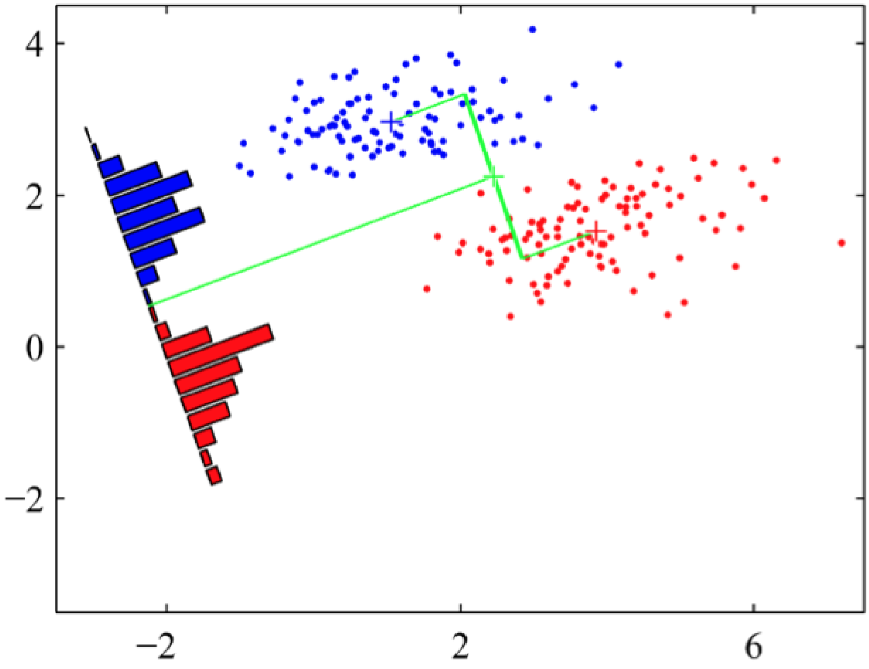


Let's consider a dataset with N_1 elements belonging to class C_1 and N_2 elements of class C_2. Their means will be:

$$m_1 = \frac{1}{N_1} \sum_{n \in C_1} x_n \quad m_2 = \frac{1}{N_2} \sum_{n \in C_2} x_n$$

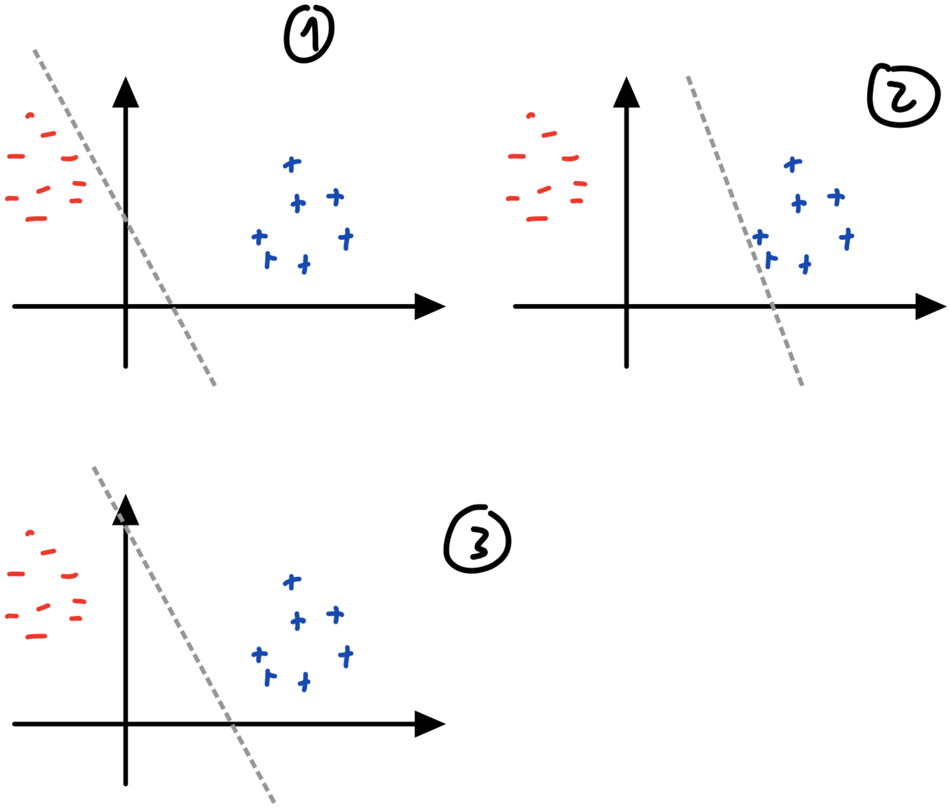
The separation of the two classes will be expressed as $J(W) = W^T(m_2 - m_1)$ and our goal will be to find the parameters for W that maximize separation.

The current solution has the major flaw of not taking into account the covariance of the two distributions. A straight forward solution is to introduce an additional parameters to the optimization problem in the form of a rotation matrix. The result will be a rotation of the linear separator, that will provide a better split for the two regions:



Support Vector Machines (SVMs)

❗ **Exam question:**
Let’s suppose to have the following three solutions: which one is better?
At first sight we might think that they are all equal, since they perfectly partition the dataset. But actually the better solution is the third one, since it will be more likely for it to classify new instances because it divides the space more evenly.



Support Vector Machines were born with the idea of addressing the problem of finding the best division in space in order to provide better accuracy.

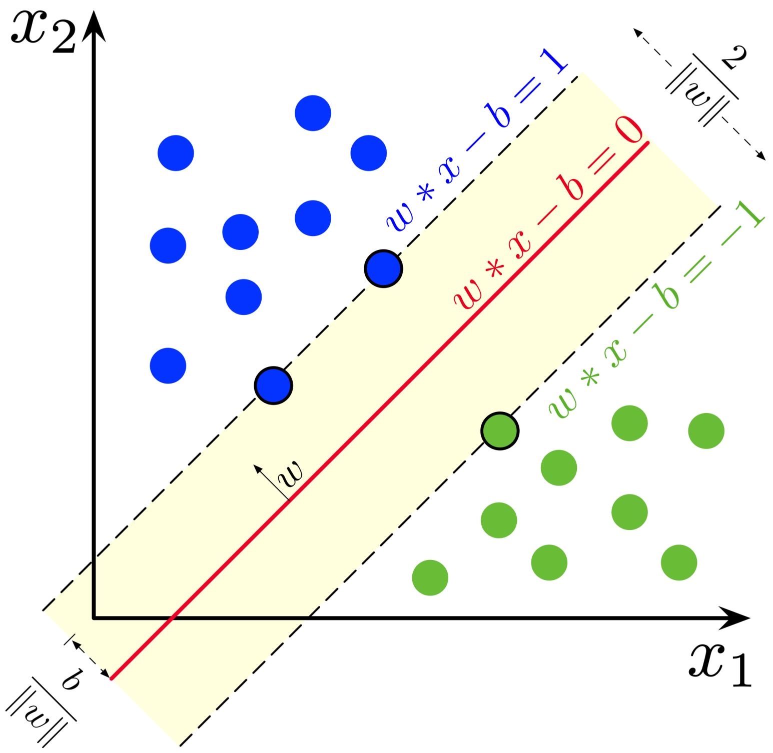
The core concept for this model lies into the definition of margins. A margin is the distance between the closest point of the dataset and the linear separator. Let's now analyze the problem.

We consider a function $f : X \to \{+1, -1\}$ and a dataset $D = \{(x_n, t_n)_{n=1}^{N}\}$ with $t_n \in \{+1, -1\}$.

We want to find a linear model $y(x) = w^T x + w_0$. Assuming thay D is linearly separable, there exists a linear combination of parameters such the resulting line separates the data:

$$\exists\ w, w_0 \text{ such that } \ y(x_n) \begin{cases} > 0 \text{ if } t_n = +1 \\ < 0 \text{ if } t_n = -1 \end{cases}$$

We define the margin of a point as $\frac{y(x)}{||w||}$, namely the distance between a cetain point and the separator.



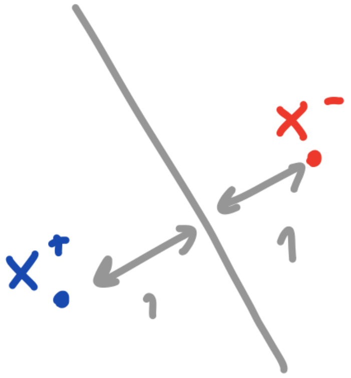
Using the concept of margin, we will define our optimization problem in the terms of maximizing the margin from the closest point to the line, namely:

$$min_{n=1,...,n}\frac{|y(x_n)|}{||W||} = ... = \frac{1}{||W||}min_{n=1,...,N}[t_n(\bar{w}^T x_n + \bar{w}_0)]$$

Minimizing this quantity means finding the hyperplane $h^* = w^{*T} x + w_0$ that will maximize the margins. The parameters that will satisfy this conditions are computes as:

$$w^*, w_0^* = argmax_{w,w_0}\frac{1}{||W||}min_{n=1,..,N}[t_n(w^T x_n + w_0)]$$

A fundamental step before computing the solution is the normalization of the dataset. This means to scale all the points such that the minimum margin for the closest point is greater or equal than 1. Once normalized, we will have that we will have at least two samples (one for C1, the other for C2) having a margin of 1.



We will define these two points as x_k^+ and x_k^- and they will hold the following property:

$$w^{*T}x_k^+ + w_0 = +1$$
$$w^{*T}x_k^- + w_0 = -1$$

The problem of computing the maximum margin can by simplified by computing the result with respect just to these two points. The optimization problem is formulated as:

$$w^*, w_0 = argmax\frac{1}{||W||} = argmin\frac{1}{2}||W||^2$$

The solution to this problem can be computed using the Lagrangian Multipliers method:

$$w^* = \sum_{n=1}^{N} a_n^* t_n x_n$$

The terms a_n^* are named Lagrange multipliers, and they will be our unknown term. In order to find them we apply this function:

The formulation of this solution is effective, because it exploits a really important property:

The Karush-Kuhn-Tucker condition states that for each $x_n \in D$, the corresponding Lagrange multiplier will be 1 if the point has margin 1 ($t_n y(x_n) = 1$) or 0 otherwise.

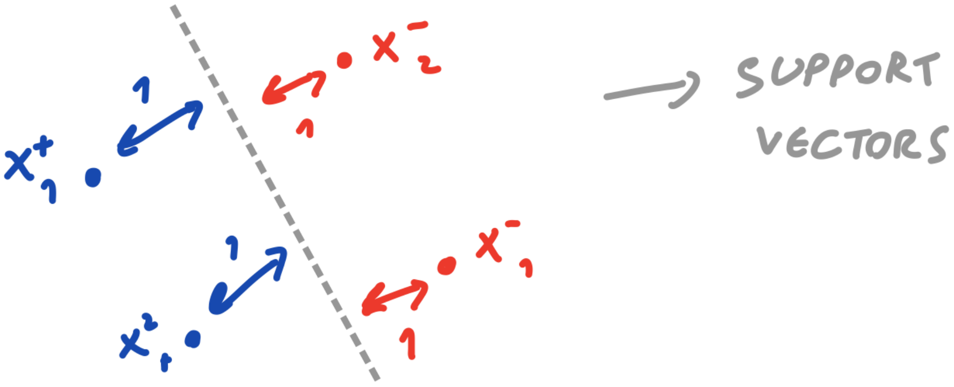
Since the optimization problem is computer as a product of terms, each time the multiplier $a_n^* = 0$, the corresponding term will be 0 and not contribute to the solution. This means that the computation will consider just a small subset of all the training dataset.

The terms that will contribute to the solution are called Support Vectors, and they are formally defined as:

$$SV = \{x_k \in D | t_k y(x_k) = 1\}$$

Therefore, hyperplanes will be describer by support vectors:

$$y(x) = \sum_{x_j \in SV} a_j^* t - j x^T s_k + w_0 = 0$$



In order to compute w_0^* we pick any support vector and solve the equation:

$$w_0^* = t_k - \sum_{x_j \in SV} a_j^* t_k x_k^T x_j$$

Actually, a better solution that is more robust to noisy data is to compute w_0^* with respect to the means of all support vectors:

$$w_0^* = \frac{1}{|SV|} \sum_{x_k \in SV} (t_k - \sum_{x_j \in SV} a_j^* t_k x_k^T x_j)$$

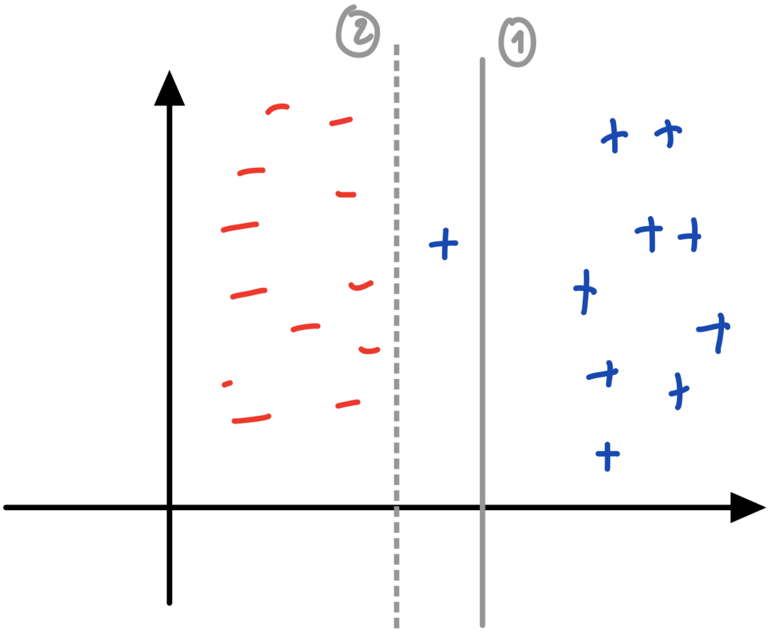
Once we find the maximum margin hyperplante, every new instance will be classified by applying the model

$$y(x') = sign(\sum_{x_k \in SV} a_k^* t_k x'^T x_k + w_0^*)$$

Support Vector Machines with soft margin constraints

We said previously that SVMs are strongly robust to noise produce by outliers, since every point that is outside the margin will give a 0 contribute to the solution of the optimization problem.

Actually, there is another kind of noises that causes problems with SVMs, that is due to points that are too close to the separator with respect to the distribution of their class. Let's see an example:



The result that we would get using the current formulation of the problem will be a separator close to line number 2. But is this the optimal solution? Actually not, because even if line 2 correctly separates the data, a solution like number 1 would be ideal, since it would maximize the probability of correctly classifying new instances by dividing the space more evenly.

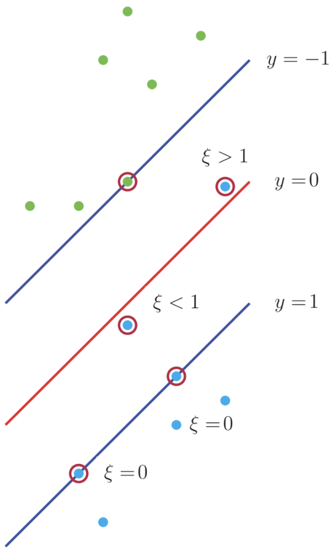
In order to get better performances we can think about relaxing the original constraints of our optimization problem by accepting some exceptions. Remember that our original condition is that every point must have a margin greater or equal that one: $t_n(w^T x + w_0) \geq 1 \; \forall n$.

The idea here is to introduce a slack variable $\xi_n \geq 0$ for each data point, thus allowing for points that have margins less than 1:

$$t_n y(x_n) \geq 1 - \xi_n \quad \forall n$$

The values for slack variables will follow this logic:

- $\xi_n = 0$ if the point lies on the margin or in the right side of the boundary.
- $0 < \xi_n \leq 1$ if the point lies between the separator and the margin.
- $\xi > 1$ if the point lies on the wrong side of the boundary.



The balance for this will be introducing some penalties to our optimization solutions for points that violate the margin:

$$w^*, w_0* = argmin \frac{1}{2}||W||^2 + C \sum_{n=1}^{N} \xi_n$$

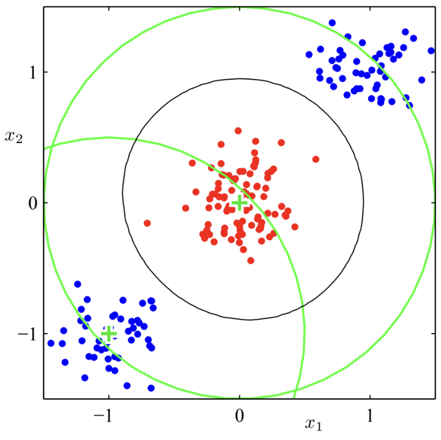
Since we are seeking the argmin, the added term will clearly penalize the terms with higher values for the slack variables. Moreover, we introduced the term C, a coefficient that express how much we want the penalization to impact the result.

Finally, the solution can still be computed using the methods explained before (the lagrangian solution and the KKT condition are still valid).

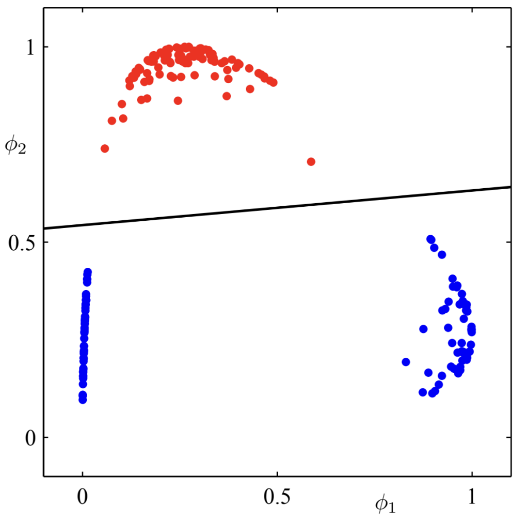
When the dataset is not linearly separable

We already introduced the fact that linear classfication problems works also if the input space is transformed. This means that even if the dataset might not be linearly separable, we can still try to apply some basis transformation in order to make it work.

Let's assume to have the following dataset:

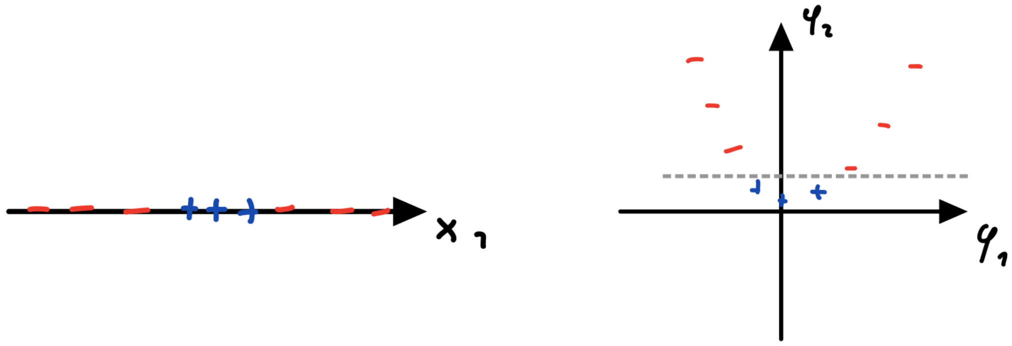


It is clearly non linearly separable. But if we apply a change of basis to its coordinates, expressing it with polar ones, we obtain the following representation:



We obtained a linearly separable representation of the dataset!

Another important property is that our method will still work even if the transformation changes the dimensions of the initial input space, both augmenting or reducing it. For example, we might think to transform data that lies in 1D into a 2D representation of a quadratic curve:



In general, given a dataset $D = \{(x_n, t_n)_{n=1}^N\}$ with $x_n \in \mathbb{R}^d$, we can define a transformation

$$\phi(x) = \begin{bmatrix} \phi_0(x) \\ ... \\ \phi_M(x) \end{bmatrix}$$

with $M \neq d$. This transformation will likely ontroduce nonlinearities. This will result into a transformation of our dataset where $x_n \rightarrow \phi_n = \phi(x_n)$. Therefore we will procude with our newly transformed dataset $D_\phi = \{(\phi_n, t_n)\}$.

For every new instance x', we will proceed transforming it $\phi' = \phi(x')$ and then classifying it with the model.

There are many function for the transformation of basis, the approach is to train them and evaluate which one performs better.

All of those considerations are still valid in the multi-class case, where we just consider the formulation with k linear models, where $y_k(x) = W^T \phi(x)$.