Non-Linear Regression

Guillaume Frèche

Version 1.0

Keywords: Newton-Raphson method, gradient descent, single-neuron classifier.

In this document, we extend our study on regression to **non-linear regression**, we talk about the **Newton-Raphson method** and **gradient descent**, and we apply these results to a simplified version of neural networks: the **single-neuron classifier**.

1 Problem presentation

Given a non linear function φ, $n \in \mathbb{N}^*$ vectors $\mathbf{x}_i = (x_{i,1}, \dots, x_{i,m})^\mathsf{T} \in \mathbb{R}^m$ and n scalars $y_1, \dots, y_n$, we are now trying to find weights $w_1, \dots, w_m$ such that for any $i \in [\![1, n]\!]$,

$$y_i = \varphi\left(\sum_{j=1}^{m} w_j x_{i,j}\right) = \varphi\left(\mathbf{x}_i^\mathsf{T}\mathbf{w}_m\right) = f\left(\mathbf{x}_i, \mathbf{w}_m\right)$$

Because of the non-linearity of φ, and thus of f, it is not possible to write this relation in matrix form anymore, and it is not possible to apply classical least squares and recursive least squares. However, we are still looking for an estimator $\widehat{\mathbf{w}}_m$ minimizing the following mean square error function:

$$E(\mathbf{w}_m, \mathbf{x}_1, \dots, \mathbf{x}_n, y_1, \dots, y_n) = \frac{1}{n}\sum_{i=1}^{n} \xi(\mathbf{w}_m, \mathbf{x}_i, y_i)$$

where error function ξ is defined as:

$$\xi(\mathbf{w}_m, \mathbf{x}_i, y_i) = (y_i - f(\mathbf{x}_i, \mathbf{w}_m))^2$$

There are two possible cases:

- we have access to the whole data $\mathbf{x}_1, \dots, \mathbf{x}_n, y_1, \dots, y_n$ and we minimize E directly;
- given the linearity of the sum defining E, we can minimize ξ while we get data $\mathbf{x}_i, y_i$ on the fly. We will focus on this latter case.

2 Newton-Raphson method

2.1 Presentation

We can explicitly compute the partial derivative of error ξ with respect to weight:

$$\frac{\partial}{\partial \mathbf{v}}\xi(\mathbf{v}, \mathbf{x}_i, y_i) = -2\left(y_i - f(\mathbf{x}_i, \mathbf{v})\right)\frac{\partial}{\partial \mathbf{v}}f(\mathbf{x}_i, \mathbf{v}) = 2\left(\varphi\left(\mathbf{x}_i^\mathsf{T}\mathbf{v}\right) - y_i\right)\varphi'\left(\mathbf{x}_i^\mathsf{T}\mathbf{v}\right)\mathbf{x}_i \tag{1}$$

but this equation does not yield a straightforward algebraic solution, contrary to the linear case. We need to introduce a numerical method, the Newton-Raphson method, to provide an approximate solution.
Given $g : [a, b] \to \mathbb{R}$ a $\mathcal{C}^1$ monotonically increasing function such that $g(a) < 0$ and $g(b) > 0$, the intermediate value theorem states that there exists a unique $\theta^* \in]a, b[$ such that $g(\theta^*) = 0$. The Newton-Raphson method is an iterative method constructing a sequence $(\theta_k)_{k\in\mathbb{N}}$ such that $\theta_0 \in [a, b]$ and whose elements are defined by the recursive formula:

$$\theta_{k+1} = \theta_k - \frac{g(\theta_k)}{g'(\theta_k)}$$

This sequence converges quadratically to θ^*, i.e. $\lim_{k\to+\infty} \theta_k = \theta^*$ and there exists a constant $C > 0$ such that

$$\forall k \in \mathbb{N} \qquad (\theta_{k+1} - \theta^*) < C(\theta_k - \theta^*)^2$$

If we are looking for a local optimum of a $\mathcal{C}^2$ function g, we want to solve the equation $g'(\theta^*) = 0$, and we apply Newton-Raphson method to g', yielding to:

$$\theta_{k+1} = \theta_k - \frac{g'(\theta_k)}{g''(\theta_k)}$$

This method can be generalized to a multivariate function $g : \mathbb{R}^m \to \mathbb{R}$. The recursion formula becomes

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - (\mathrm{H}g(\boldsymbol{\theta}_k))^{-1}\,\nabla g(\boldsymbol{\theta}_k)$$

where

$$\nabla g(\mathbf{v}) = \frac{\partial g}{\partial \mathbf{v}}(\mathbf{v}) = \left(\frac{\partial g}{\partial v_1} g(\mathbf{v}), \dots, \frac{\partial g}{\partial v_m} g(\mathbf{v})\right)^{\mathsf{T}} \in \mathbb{R}^m$$

is the gradient of g in $\mathbf{v} \in \mathbb{R}^m$, and

$$\mathrm{H}g(\mathbf{v}) = \begin{pmatrix} \frac{\partial^2 g}{\partial v_1^2}(\mathbf{v}) & \cdots & \frac{\partial^2 g}{\partial v_1 v_m}(\mathbf{v}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 g}{\partial v_m v_1}(\mathbf{v}) & \cdots & \frac{\partial^2 g}{\partial v_m^2}(\mathbf{v}) \end{pmatrix} \in \mathbb{R}^{m\times m}$$

is the Hessian matrix of g in $\mathbf{v} \in \mathbb{R}^m$. This generalized method still converges quadratically to $\boldsymbol{\theta}^*$, i.e. there exists a constant $C > 0$ such that

$$\forall k \in \mathbb{N} \qquad \|\theta_{k+1} - \theta^*\| < C\|\theta_k - \theta^*\|^2$$

where $\|.\|$ is the ℓ_2-norm on $\mathbb{R}^m$. Applying this to our weight update problem, we obtain:

$$\widehat{\mathbf{w}}_m^{(k+1)} = \widehat{\mathbf{w}}_m^{(k)} - \left(\mathrm{H}\xi(\widehat{\mathbf{w}}_m^{(k)}, \mathbf{x}_{k+1}, y_{k+1})\right)^{-1} \nabla\xi\left(\widehat{\mathbf{w}}_m^{(k)}, \mathbf{x}_{k+1}, y_{k+1}\right)$$

Note that this form is very similar to weight update equations for RLS and Kalman filters.
Remark: In our description of the Newton-Raphson method, we implicitly assumed that function g has a unique minimum, to which the sequence of parameters converges. In general, non-linear regression error function does not have a unique minimum, and the iterative sequence $\widehat{\mathbf{w}}_m^{(k)}$ will converge to the closest minimum from $\widehat{\mathbf{w}}_m^{(0)}$, making this method sensitive to its initialization. There are numerical methods to circumvent this flaw, such as **genetic algorithms**.

2.2 Application to linear regression

As an example, let us apply the Newton-Raphson method to the linear regression problem studied in the linear regression document. In this case, $\xi(\mathbf{w}_m, \mathbf{x}_{k+1}, y_{k+1}) = \left(y_{k+1} - \mathbf{x}_{k+1}^\mathsf{T}\mathbf{w}_m\right)^2$, which gives

$$\frac{\partial}{\partial w_i}\xi\left(\mathbf{w}_m, \mathbf{x}_{k+1}, y_{k+1}\right) = -2\left(y_{k+1} - \mathbf{x}_{k+1}^\mathsf{T}\mathbf{w}_m\right)\frac{\partial}{\partial w_i}\left(\mathbf{x}_{k+1}^\mathsf{T}\mathbf{w}_m\right) = -2\left(y_{k+1} - \mathbf{x}_{k+1}^\mathsf{T}\mathbf{w}_m\right)x_{k+1,i}$$

and $\nabla\xi\left(\mathbf{w}_m, \mathbf{x}_{k+1}, y_{k+1}\right) = -2\left(y_{k+1} - \mathbf{x}_{k+1}^\mathsf{T}\mathbf{w}_m\right)\mathbf{x}_{k+1}$. Then

$$\frac{\partial}{\partial w_i w_j}\xi\left(\mathbf{w}_m, \mathbf{x}_{k+1}, y_{k+1}\right) = -2x_{k+1,j}\frac{\partial}{\partial w_i}\left(y_{k+1} - \mathbf{x}_{k+1}^\mathsf{T}\mathbf{w}_m\right) = 2x_{k+1,i}x_{k+1,j}$$

Thus

$$\mathrm{H}\xi\left(\mathbf{w}_m, \mathbf{x}_{k+1}, y_{k+1}\right) = \left(\frac{\partial}{\partial w_i w_j}\xi\left(\mathbf{w}_m, \mathbf{x}_{k+1}, y_{k+1}\right)\right)_{(i,j)\in[\![1,n]\!]^2} = 2\mathbf{x}_{k+1}\mathbf{x}_{k+1}^\mathsf{T}$$

and the weight update equation becomes

$$\widehat{\mathbf{w}}_m^{(k+1)} = \widehat{\mathbf{w}}_m^{(k)} + \left(\mathbf{x}_{k+1}\mathbf{x}_{k+1}^\mathsf{T}\right)^{-1}\mathbf{x}_{k+1}\left(y_{k+1} - \mathbf{x}_{k+1}^\mathsf{T}\widehat{\mathbf{w}}_m^{(k)}\right)$$

We obtain a formula similar to the RLS weight update equation but with a noticeable difference in the gain factor $\left(\mathbf{x}_{k+1}\mathbf{x}_{k+1}^\mathsf{T}\right)^{-1}\mathbf{x}_{k+1}$ instead of $\mathbf{g}_{k+1} = \dfrac{(\mathrm{X}_{k,m}^\mathsf{T}\mathrm{X}_{k,m})^{-1}\mathbf{x}_{k+1}}{1 + \mathbf{x}_{k+1}^\mathsf{T}(\mathrm{X}_{k,m}^\mathsf{T}\mathrm{X}_{k,m})^{-1}\mathbf{x}_{k+1}}$. Note that in this derivation, we did not take into account the error from previous samples, hence the disappearance of the terms $\mathrm{X}_{k,m}$. This is a behavior that we also find with neural networks for which the learning process only cares about the current example input, without taking into account the previous ones.

3 Gradient descent and BFGS algorithm

3.1 Gradient descent

A first solution to circumvent the difficulty of computing the Hessian matrix $\mathrm{H}\xi$ is to replace it by a scalar $\mu_k > 0$ called a **learning factor**. The corresponding method is called **gradient descent**. The weight update equation is then:

$$\widehat{\mathbf{w}}_m^{(k+1)} = \widehat{\mathbf{w}}_m^{(k)} - \mu_{k+1}\nabla\xi\left(\widehat{\mathbf{w}}_m^{(k)}, \mathbf{x}_{k+1}, y_{k+1}\right) \tag{2}$$

This sequences converges linearly to the closest minimum from $\widehat{\mathbf{w}}_m^{(0)}$, i.e. there exists a constant $C \in]0,1[$ such that

$$\forall k \in \mathbb{N} \qquad \left\|\widehat{\mathbf{w}}_m^{(k+1)} - \widehat{\mathbf{w}}_m\right\| \leq C\left\|\widehat{\mathbf{w}}_m^{(k)} - \widehat{\mathbf{w}}_m\right\|$$

We have to be careful in our choice of μ_k: if it is too small, the descent will have a very low convergence, if it is too large, the descent may have stability issues. A good trade-off is to choose $\mu_k = \mu_0\alpha^k$ as a geometric series, with $\alpha \in]0,1[$. Doing so, the learning factor is large at the beginning of the descent and then decreases to 0.

3.2 BFGS algorithm

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm proposes to replace the Hessian matrix by an estimate. This algorithm will be described in an upcoming version of this document.

4 Single neuron classifier

4.1 Affine separation

An **affine hyperplane** is a set of coordinates $\mathbf{x} = (x_1, \dots, x_m) \in \mathbb{R}^m$ satisfying the relation:

$$w_0 + w_1 x_1 + \dots + w_m x_m = \widetilde{\mathbf{x}}^\mathsf{T} \mathbf{w}_m = 0$$

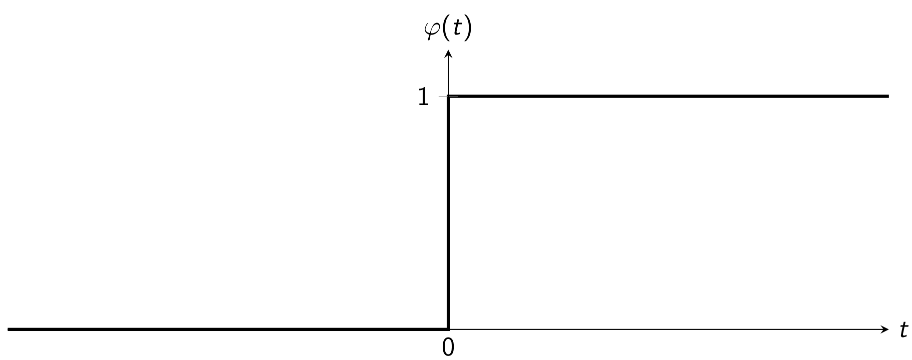
where $\widetilde{\mathbf{x}} = (1, x_1, \dots, x_m)$, and $\mathbf{w}_m = (w_0, w_1, \dots, w_m)$ is a vector of fixed weights. This hyperplane partitions $\mathbb{R}^m$ into two sets: the set S_0 of coordinates $\mathbf{x} \in \mathbb{R}^m$ such that $\widetilde{\mathbf{x}}^\mathsf{T} \mathbf{w}_m < 0$ and the set S_1 of coordinates $\mathbf{x} \in \mathbb{R}^m$ such that $\widetilde{\mathbf{x}}^\mathsf{T} \mathbf{w}_m > 0$. Note that if $\mathbf{w}_m \in \mathbb{R}^m$ is such that $\widetilde{\mathbf{x}}^\mathsf{T} \mathbf{w}_m = 0$, then vector $\alpha \mathbf{w}_m$ with $\alpha \in \mathbb{R}$ satisfies this relation too, hence the weights vector is not unique. However, this aspect has no impact on the following.

A **classifier** is a system inputting coordinates $\mathbf{x}$ and whose goal is to determine the label $i \in \{0, 1\}$ such that $\mathbf{x} \in S_i$. This system can be modeled by a non linear-regression:

$$y = \varphi\left(\widetilde{\mathbf{x}}^\mathsf{T} \mathbf{w}_m\right)$$

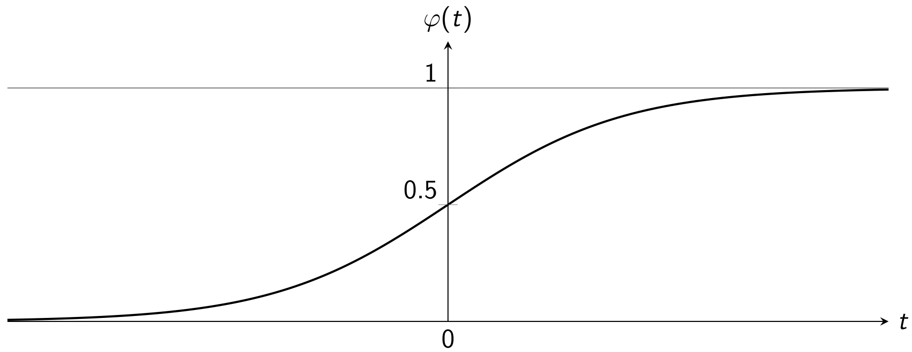
where φ is a mapping such that $y = 0$ if $\mathbf{x} \in S_0$ and $y = 1$ if $\mathbf{x} \in S_1$. Such a mapping φ is called an **activation function** in machine learning, because it corresponds to neuron activation in neural networks. An obvious choice for φ will be:

$$\varphi(t) = \chi_{[0,+\infty[}(t) = \begin{cases} 1 & \text{if } t \geq 0 \\ 0 & \text{if } t < 0 \end{cases}$$



Although this function provides the required output, it presents a major issue: it has a discontinuity in 0, which is annoying to compute derivatives and apply gradient descent. A good alternative, often used in machine learning, is the **sigmoid function**:

$$\forall t \in \mathbb{R} \qquad \varphi(t) = \frac{1}{1 + e^{-t}}$$



It rapidly goes to 0 when t tends to $-\infty$, and to 1 when t tends to $+\infty$, it is infinitely derivable, and its derivative is given by the relation:

$$\forall t \in \mathbb{R} \qquad \varphi'(t) = \varphi(t)\,(1 - \varphi(t))$$

Using the derivative of the error ξ found in Equation (1) and the gradient descent update Equation (2), we get:

$$\begin{aligned}
\widehat{\mathbf{w}}_m^{(k+1)} &= \widehat{\mathbf{w}}_m^{(k)} - 2\mu_{k+1}\left(\varphi\left(\widetilde{\mathbf{x}}_{k+1}^{\mathsf{T}}\widehat{\mathbf{w}}_m^{(k)}\right) - y_{k+1}\right)\varphi'\left(\widetilde{\mathbf{x}}_{k+1}^{\mathsf{T}}\widehat{\mathbf{w}}_m^{(k)}\right)\widetilde{\mathbf{x}}_{k+1} \\
&= \widehat{\mathbf{w}}_m^{(k)} - 2\mu_{k+1}\left(\varphi\left(\widetilde{\mathbf{x}}_{k+1}^{\mathsf{T}}\widehat{\mathbf{w}}_m^{(k)}\right) - y_{k+1}\right)\varphi\left(\widetilde{\mathbf{x}}_{k+1}^{\mathsf{T}}\widehat{\mathbf{w}}_m^{(k)}\right)\left(1 - \varphi\left(\widetilde{\mathbf{x}}_{k+1}^{\mathsf{T}}\widehat{\mathbf{w}}_m^{(k)}\right)\right)\widetilde{\mathbf{x}}_{k+1}
\end{aligned}$$

Figure 1 illustrates the result of a single neuron classifier for linear separation on a set of 300 training examples. Our points belong to $\mathbb{R}^2$ and the separating hyperplane of equation $-6 + 2x + 3y = 0$ is represented by the black line. Hence we want to estimate the weights vector $\mathbf{w}_m = (-6, 2, 3)$. Blue points have coordinates $\mathbf{x} = (x, y)$ satisfying $\widetilde{\mathbf{x}}^{\mathsf{T}}\mathbf{w}_m < 0$ and magenta points have coordinates $\mathbf{x} = (x, y)$ satisfying $\widetilde{\mathbf{x}}^{\mathsf{T}}\mathbf{w}_m > 0$. The red line corresponds to the weights estimated by the classifier. Figure 2 displays the evolution of training and test errors versus the increasing number of training examples. Given a number of training examples between 100 and 2000 with a step of 100, we train the classifier 200 datasets with this number of examples followed by a test on 10,000 examples, and we averaged these training and test errors over the 200 trials.

4.2 Quadratic separation

An **ellipsoid** is a set of coordinates $\mathbf{x} = (x_1, \dots, x_m) \in \mathbb{R}^m$ satisfying the relation:

$$\widetilde{\mathbf{x}}^{\mathsf{T}}\mathrm{Q}_m\widetilde{\mathbf{x}} = 0$$

where $\widetilde{\mathbf{x}} = (1, x_1, \dots, x_m)$, and Q_m is a fixed upper triangular matrix. This ellipsoid partitions $\mathbb{R}^m$ into two sets: the set S_0 of coordinates $\mathbf{x} \in \mathbb{R}^m$ such that $\widetilde{\mathbf{x}}^{\mathsf{T}}\mathrm{Q}_m\widetilde{\mathbf{x}} < 0$, i.e. the points inside the ellipsoid, and the set S_1 of coordinates $\mathbf{x} \in \mathbb{R}^m$ such that $\widetilde{\mathbf{x}}^{\mathsf{T}}\mathrm{Q}_m\widetilde{\mathbf{x}} > 0$, i.e. the points outside the ellipsoid. As mentioned in linear regression, vector $\mathbf{x}$ and matrix Q_m are flatten as follows

$$\begin{aligned}
\tilde{\mathbf{x}} &= (x_1^2, x_1x_2, \dots, x_1x_m, x_2^2, \dots, x_jx_k, \dots, x_m^2)^{\mathsf{T}} \\
\widetilde{\mathrm{Q}}_m &= (q_{1,1}, q_{1,2}, \dots, q_{1,m}, q_{2,2}, \dots, q_{i,j}, \dots, q_{m,m})^{\mathsf{T}}
\end{aligned}$$

The single neuron provides the output

$$y = \varphi\left(\tilde{\mathbf{x}}^{\mathsf{T}}\widetilde{\mathrm{Q}}_m\right)$$

where φ is again the sigmoid function.

Figure 3 illustrates the result of a single neuron classifier for ellipsoid separation on a set of 800 training examples. Our

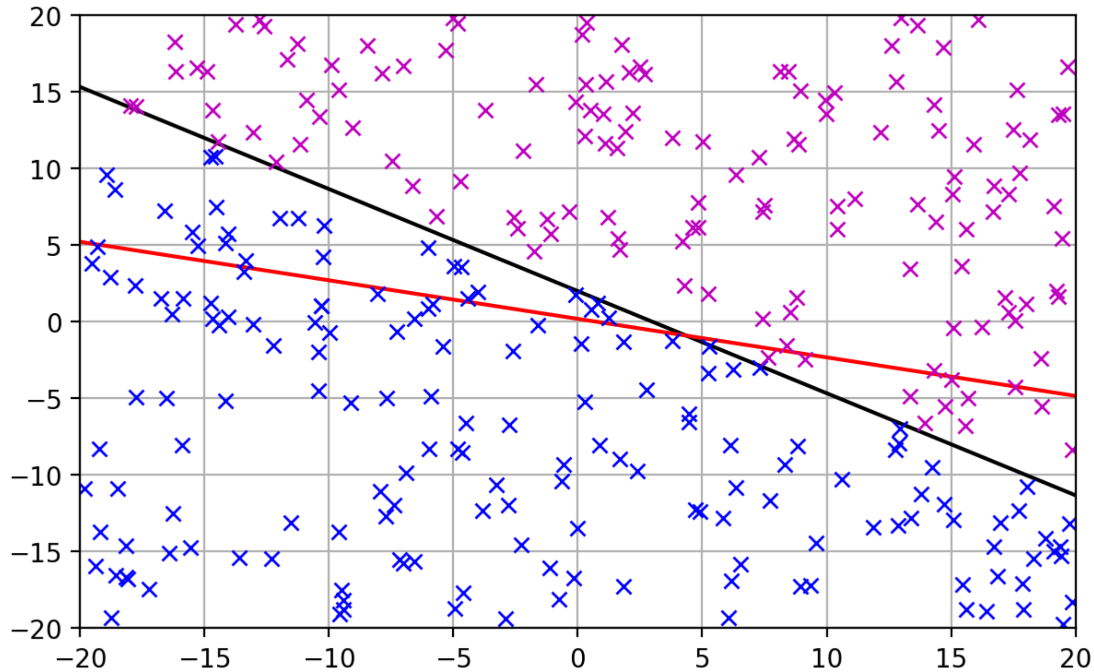


Figure 1: Illustration of Single Neuron Classifier for affine separation

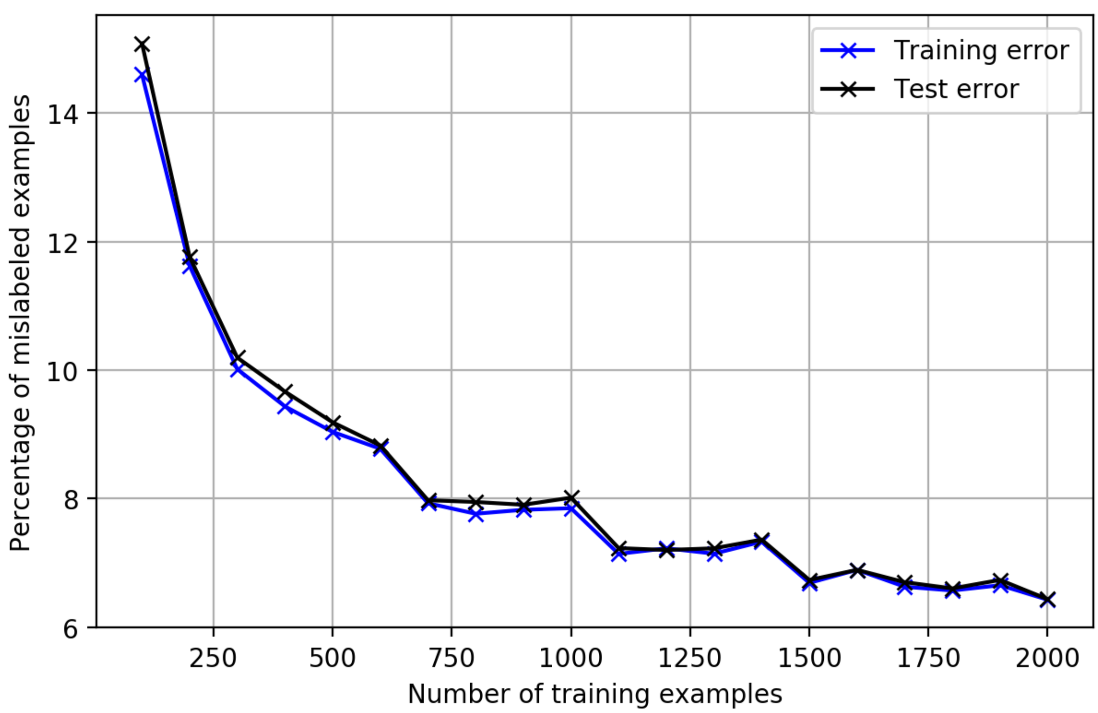


Figure 2: Evolution of training and test errors versus the number of training examples for affine separation

points belong to $\mathbb{R}^2$ and the separated ellipse of equation $x^2 + 2y^2 - 2xy - x + y - 5 = 0$ is represented by the black line. Hence we want to estimate the weights vector $\widetilde{Q}_m = (1, 2, -2, -1, 1, -5)$. Blue points inside the ellipse have coordinates $\mathbf{x} = (x, y)$ satisfying $\widetilde{\mathbf{x}}^T \widetilde{Q}_m < 0$ and magenta points outside the ellipse have coordinates $\mathbf{x} = (x, y)$ satisfying $\widetilde{\mathbf{x}}^T \widetilde{Q}_m > 0$. The red line represents the ellipse corresponding to the weights estimated by the classifier.

Remark: When we flattened matrix Q_m, we transformed the space of representation (also called **feature space**) by changing 2-dimensional coordinates (x, y) into 6-dimensional coordinates $(x^2, y^2, xy, x, y, 1)$. Doing so, we implicitly perform what is known in machine learning as the **kernel trick**. This method will be studied in further details when dealing with Support Vector Machines.

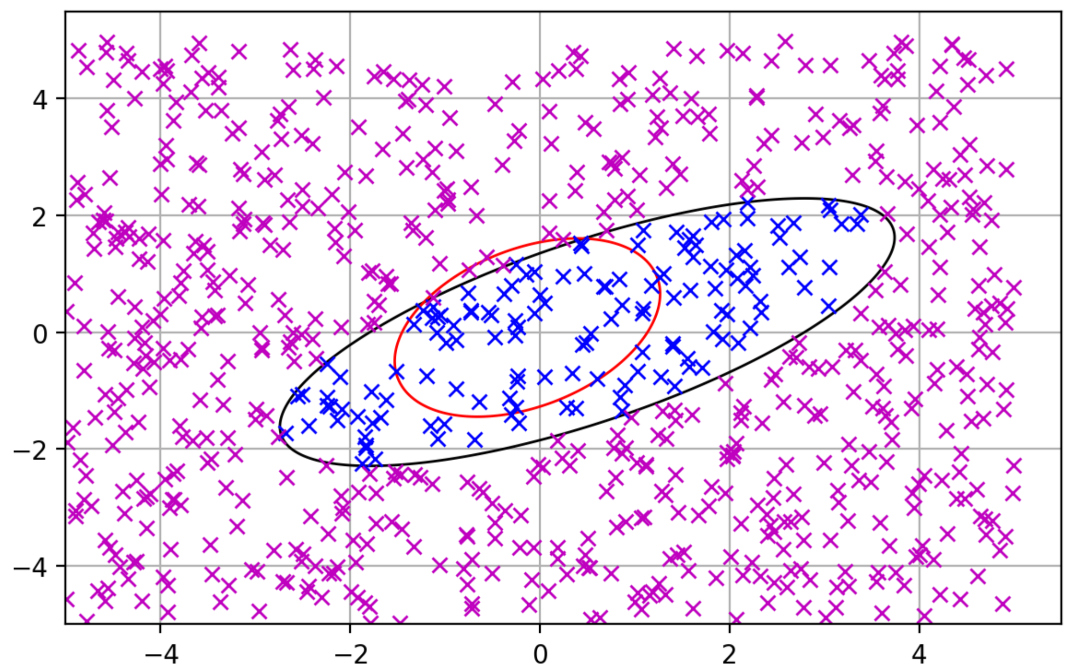


Figure 3: Illustration of Single Neuron Classifier for quadratic separation

Figure 4 displays the evolution of training and test errors versus the increasing number of training examples. Given a number of training examples between 100 and 2000 with a step of 100, we train the classifier 200 datasets with this number of examples followed by a test on 10,000 examples, and we averaged these training and test errors over the 200 trials.

Remarks:

- We see that for both affine and quadratic separations, training and test errors decrease as the number of training examples goes up, but they seem to get rapidly stuck to a non-zero limit. A way to improve this error bound is to extend the concept of single-neuron classifiers to more general structures called **Multiple Layer Perceptrons** (MLP) or **Artificial Neural Networks** (ANN), that we will describe in another document.

- Affine separation can be seen as a special case of quadratic separation where the weight coefficients corresponding to x^2, y^2 and xy are zero. In its current form, the quadratic separation has very poor performance on affine separation with an average error around 50% (same as a coin toss), no matter the number of training samples. This question will be treated in underdetermined systems techniques.

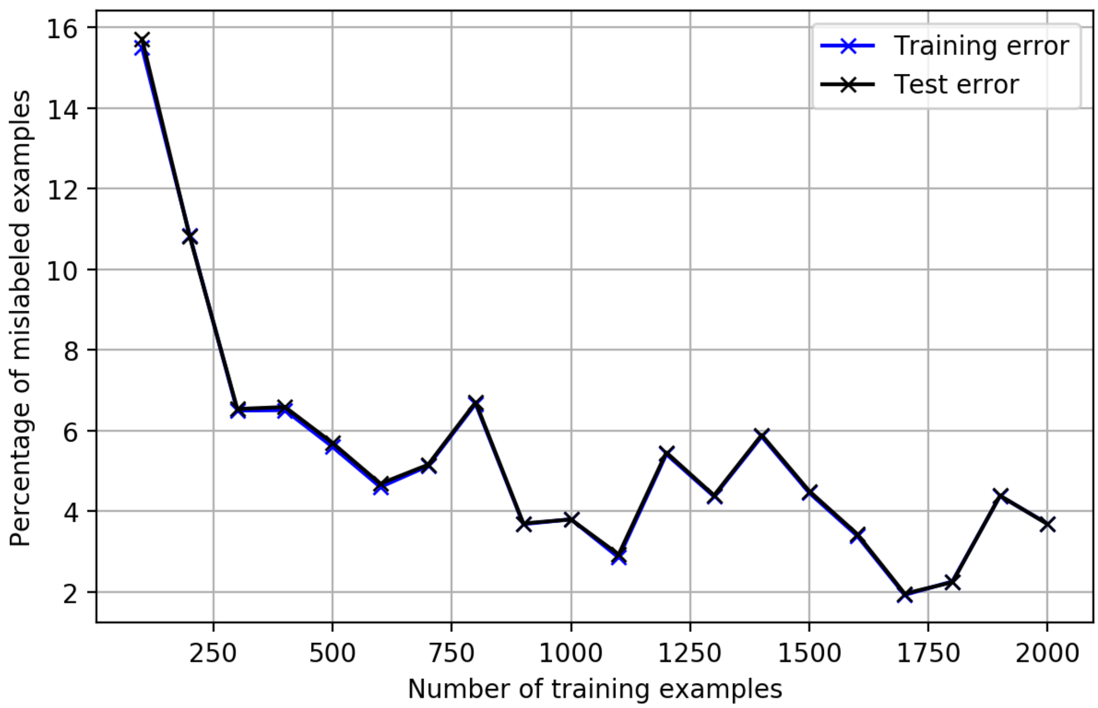


Figure 4: Evolution of training and test errors versus the number of training examples for quadratic separation

5 Upcoming subjects

Here is a list of subjects that will be added in future versions of this document:

- One major issue with non-linear optimization techniques, such as Newton-Raphson method, is that they converge to the local optimum closest to their initial values. **Genetic algorithms** propose an alternative to this problem by generating a population of weight vectors instead of a single and update this population by selecting and mixing the ones exhibiting the best error performances.

- **Support Vector Machines** (SVM) are a class of supervised learning algorithms dealing with the affine separation problem and adding a notion of maximal **margin**. They can also be adapted to more general problems through the **kernel trick**, based on **Mercer's theorem**.