SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Olivier Sigaud

Regression Labs: Instructions

2019-2020

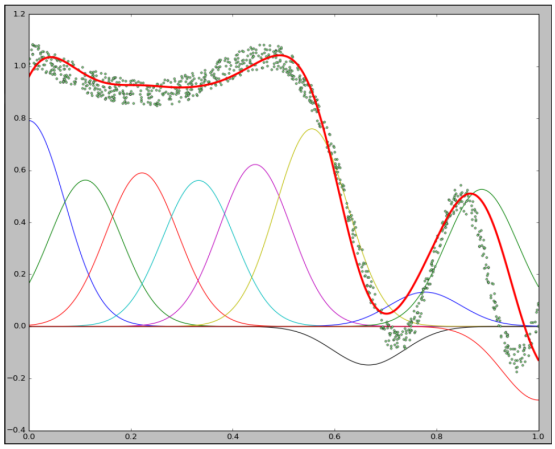

Figure 1: An exemple of regression with RBFNs

1 Final report instructions and evaluation

- An email must be sent to `Olivier.Sigaud@upmc.fr` at the end of the lab (i.e. before 18:00 PM, the day of the lab).
- The email subject must contain: "Regression Lab, NAME1 - NAME2" where NAME1 and NAME2 are the names of the involved students.
- The email must contain your report included as a pdf file and a copy of your source files, altogether included into an archive whose name must be `REG_NAME1_NAME2` (.zip or .tar.gz).
- The evaluation will take into account the quantity of work done, the accuracy of the results and the quality of the content (including written english).

2 General information

2.1 Introduction

The objective of regression or function approximation is to create a model from observed data. The model has a fixed structure with parameters (like the coefficients of a polynomial for instance), and regression consists in adjusting these parameters to fit

the data. In machine learning, it is a very important technique since having a good model enables better predictions and performance.

Generally speaking, given some data $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$, the goal is to adjust a model $\mathbf{y} = f(\mathbf{x})$ so that the *learned* function f accounts for datapoints and generalize well to other unseen points.

The simplest case is linear regression, which assumes a relation of type $\mathbf{y} = A\mathbf{x} + \mathbf{b}$ between the data $\mathbf{x}$ and $\mathbf{y}$. Several methods exist to adjust the parameters A and $\mathbf{b}$, the most well-known being the least squares method (or least norm in the multivariate case).

In this lab, we will always assume that $\mathbf{y}$ is of dimension 1 (and write y instead of $\mathbf{y}$ in what follows).

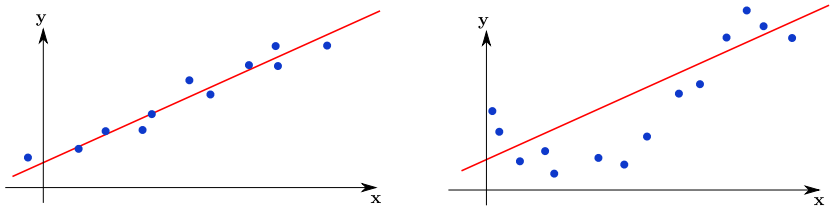

Figure 2: On the left and right: data more or less adapted to linear regression.

Often, linear models are not enough (e.g. Figure 2, on the right) and we must rely on nonlinear models. Here, we focus on the case where f can be written as a sum of k functions parametrized by vectors θ_j:

$$f(\mathbf{x}) = \sum_{j=1}^{k} f_{\theta_j}(\mathbf{x}).$$

In particular, you will use Gaussian functions.

2.2 Weighted sums of Gaussian functions

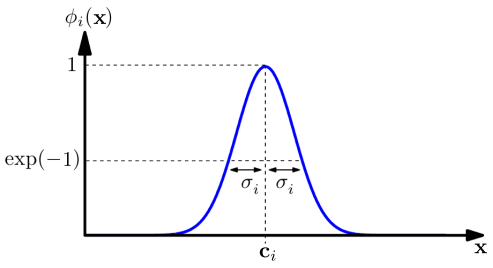

Figure 3: A 1-dimensional Gaussian function.

Gaussian functions $\phi_i(\mathbf{x}) = \exp(-\frac{(\mathbf{x}-\mathbf{c}_i)^2}{\sigma_i^2})$ are almost equal to zero everywhere except in a neighborhood of $\mathbf{c}_i$, which represents the "center" of the Gaussian (see Figure 3). The value of σ_i determines how large this neighborhood is.

Then weighted Gaussian functions f_{θ_i} can be written:

$$f_{\theta_i} = \theta_i \phi_i(\mathbf{x}) = \theta_i \exp\left(-\frac{(\mathbf{x}-\mathbf{c}_i)^2}{\sigma_i^2}\right). \quad (1)$$

In these labs, the centers of the Gaussian functions are fixed in advance, and evenly distributed in the input space. Besides, all the σ_i's are usually set to the same value. Thus weighted Gaussian functions f_{θ_i} have a unique scalar parameter which is their weight θ_i.

3 Provided code

The numpy reference guide is here:
`http://docs.scipy.org/doc/numpy/reference/`

3.1 Data generation

Two functions to generate points corresponding to a noisy linear and a noisy nonlinear model are provided in the `SampleGenerator` class in the `sample_generator.py` file.

3.2 Function approximators

We consider three families of approximation models: linear models (given in `line.py`), radial basis function networks (given in `rbfn.py`) and locally weigthed regression models (given in `lwr.py`). In all these models, the `theta` attribute represents the vector of parameters to be optimized and the `f(self,x)` function represents the function approximator output for a given input `x`, to be optimized. The parameters used by f are either the `theta` attribute or a value `*usertheta` given as input. The number of elements of $\phi(\mathbf{x})$ (i.e. the number k of Gaussian functions) is defined by the `nb_features` attribute.

These function approximation models come with various `train_...(self,...)` functions that you have to fill.

3.3 Vectors of Gaussian functions

A class `Gaussians` is provided in `gaussians.py` to represent a vector of Gaussian feature functions, as used in RBFNs and Locally Weighted Regression (LWR) (see Section 2.2).

We consider a vector of Gaussian functions $\phi(\mathbf{x}) = (\phi_1(\mathbf{x})\ \phi_2(\mathbf{x}) \cdots \phi_k(\mathbf{x}))^\mathsf{T}$, a vector of weights $\theta = (\theta_1\ \theta_2 \cdots \theta_k)^\mathsf{T}$, and a vector of weighted Gaussian functions $f(\mathbf{x}) = \phi(\mathbf{x})^\mathsf{T}\theta$.

Given some input vector $\mathbf{x}$, the function `phi_output(x)` returns the vector of the output of (non-weighted) Gaussian functions applied to $\mathbf{x}$. In the particular case where x is a scalar, it simply returns the vector of one-dimensional Gaussian functions applied to x.

Note that the (multivariate) Gaussian functions are of the same dimension as $\mathbf{x}$, but their output is one-dimensional.

3.4 Visualization

For all the regression methods described below, as illustrated in Figure 1, after executing it, the observed data are shown by points, and the red curve is the *learned* function f corresponding to the parameters θ that have been incrementally adjusted. When this applies, the other curves correspond to the $f_{\theta_i}(\mathbf{x})$ functions, they show the decomposition of f, which is the sum of all these functions.

3.5 Main function

The `main.py` file contains all the necessary code to call all the function approximation methods that you have to implement. You can comment or uncomment part of this code to run the functions you want, change the parameters, etc.

4 Questions

In Section 4.1, you will first study four different algorithms. Completing this part is mandatory. In Section 4.2, you will go beyond these basic algorithms and compare different approaches.

4.1 Families of regression algorithms

We first study four basic regression algorithms: the linear least squares, Radial Basis Function Networks (RBFNs) using various fitting algorithms, and Locally Weighted Regression.

4.1.1 Linear Least squares

The linear least squares method finds the best linear model from a batch of data, using

$$\theta^* = \min_{\theta} \underbrace{\|\mathbf{y} - \theta^\intercal \mathbf{X}\|^2}_{L(\theta)}. \tag{2}$$

The parameters for the optimal model are given by:

$$\theta^* = (\mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{y}. \tag{3}$$

In the file `line.py`, the `Line` class provides the `train_from_stats(self,x_data,y_data)` function for getting the linear least square model using the `stats.linregress(x_data,y_data)` function.

Question 1: Fill the `train_ls(self,x_data,y_data)` function so as to perform the same linear least square computation using (3). Do you get exactly the same results as with the `train_from_stats(self,x_data,y_data)` function? Add a screenshot of your results in your report.

4.1.2 Radial Basis Function Networks

We now study Radial Basis Function Networks (RBFNs). You will implement several regression techniques with these models: batch least squares, gradient descent, and optionnally recursive least squares.

Batch Least Squares We consider a batch of data consisting of N $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{1 \leq i \leq N}$ pairs. We want to minimize the following error:

$$\varepsilon(\theta) = \frac{1}{2N} \sum_{i=1}^{N} \left(y^{(i)} - f_\theta\left(\mathbf{x}^{(i)}\right) \right)^2.$$

If θ is a local minimum of the function ε, it means that the gradient is equal to zero in θ. So we try to solve:

$$\nabla \varepsilon(\theta) = \mathbf{0}$$

The gradient of a sum is the sum of the gradients, and we can use the following formula to calculate the gradient of a product of two functions g and h: $\nabla(gh) = g\nabla h + h\nabla g$ (in particular $\nabla(g^2) = 2g\nabla g$). Therefore:

$$\nabla \varepsilon(\theta) = \frac{1}{N} \sum_{i=1}^{N} - \left(y^{(i)} - f_\theta\left(\mathbf{x}^{(i)}\right) \right) \phi(\mathbf{x}^{(i)}) = -\frac{1}{N} \sum_{i=1}^{N} \phi(\mathbf{x}^{(i)}) \left(y^{(i)} - \phi(\mathbf{x}^{(i)})^\mathsf{T} \theta \right).$$

To make the gradient equal to zero, we want to have:

$$\left(\sum_{i=1}^{N} \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^\mathsf{T} \right) \theta = \sum_{i=1}^{N} \phi(\mathbf{x}^{(i)}) y^{(i)}.$$

This can be rewritten $A\theta = b$. A is not necessarily an invertible matrix, and the general solution is obtained by using the "pseudo-inverse" (`np.linalg.pinv(A)`) $A^\sharp$, using $\theta = A^\sharp b$.

> **Question 2:** In `rbfn.py`, fill the `train_ls(self,x_data,y_data)` training function that computes θ using the least squares method. In the `main.py` file, try to find values of `nb_features` leading to good results. Put a screenshot into your report.

Gradient Descent Let the vector $\theta^{(t)}$ be the value of the parameters at iteration t. We observe some new data $(\mathbf{x}^{(t+1)}, y^{(t+1)})$. The error of the current model on this pair is:

$$\varepsilon^{(t+1)} = y^{(t+1)} - f_{\theta^{(t)}}(\mathbf{x}^{(t+1)}).$$

The idea of the gradient descent is to slightly modify θ to improve the error obtained on $(\mathbf{x}^{(t+1)}, y^{(t+1)})$. To do so, we consider the function $\theta \mapsto f_\theta\left(\mathbf{x}^{(t+1)}\right)$ and compute its gradient in $\theta^{(t)}$, which we denote by $\nabla_\theta^{(t+1)}$:

$$\nabla_\theta^{(t+1)} = \phi(\mathbf{x}^{(t+1)}).$$

Remark: To verify that the gradient is equal to $\phi(\mathbf{x}^{(t+1)})$, one can use the following formula:

$$g(\mathbf{a}+\lambda\mathbf{b}) = g(\mathbf{a}) + \lambda(\nabla g(\mathbf{a}))^\mathsf{T}\mathbf{b} + o(\lambda).$$

The gradient is oriented towards the direction of steepest increase. This means that it gives the direction in which a small modification of the vector of inputs leads to the largest increase of the function output. The opposite direction is the one of steepest decrease. So if the goal is to increase $f_{\theta^{(t)}}\left(\mathbf{x}^{(t+1)}\right)$, θ should be modified in the direction defined by $\nabla_\theta^{(t+1)}$. In this case, $\varepsilon^{(t+1)}$ is positive. Conversely, when $\varepsilon^{(t+1)}$ is negative, θ should be modified in the direction defined by $-\nabla_\theta^{(t+1)}$. The formula used is the following one:

$$\theta^{(t+1)} = \theta^{(t)} + \alpha\varepsilon^{(t+1)}\nabla_\theta^{(t+1)} = \theta^{(t)} + \alpha\varepsilon^{(t+1)}\phi(\mathbf{x}^{(t+1)}),$$

where $\alpha > 0$ is a coefficient called the "learning rate".

Question 3: In `rbfn.py`, fill the `train_gd(self,x,y)` training function that improves θ using gradient descent. In the `main.py` file, try to find values of `maxIter,nb_features` and the learning rate leading to good results. Put a screenshot into your report.

4.1.3 Locally Weighted Least Squares

The third family of models is Locally Weighted Least Squares (LWLS). The LWLS algorithm uses a weighted sum of local linear models, parametrized by vectors θ_i such that $\dim(\theta_i) = \dim(\mathbf{x}) + 1 = d + 1$:

$$f(\mathbf{x}) = \sum_{i=1}^{k} \frac{\phi_i(\mathbf{x})}{\sum_{j=1}^{k}\phi_j(\mathbf{x})} m_{\theta_i}(\mathbf{x}),$$

with $m_{\theta_i}(\mathbf{x}) = w(\mathbf{x})^\mathsf{T}\theta_i$ and $w(\mathbf{x}) = (\mathbf{x}_1\ \mathbf{x}_2 \cdots \mathbf{x}_d\ 1)^\mathsf{T}$.

Each local model is computed using the following locally weighted error:

$$\varepsilon_i(\theta_i) = \frac{1}{2N}\sum_{j=1}^{N}\phi_i\left(\mathbf{x}^{(j)}\right)\left(y^{(j)} - m_{\theta_i}\left(\mathbf{x}^{(j)}\right)\right)^2 = \frac{1}{2N}\sum_{j=1}^{N}\phi_i\left(\mathbf{x}^{(j)}\right)\left(y^{(j)} - w\left(\mathbf{x}^{(j)}\right)^\mathsf{T}\theta_i\right)^2.$$

As with the least squares method, we try to cancel out the gradient, which amounts to solving:

$$-\frac{1}{N}\sum_{j=1}^{N}\phi_i(\mathbf{x}^{(j)})w\left(\mathbf{x}^{(j)}\right)\left(y^{(j)} - w\left(\mathbf{x}^{(j)}\right)^\mathsf{T}\theta_i\right) = 0.$$

Therefore, we pose $\theta_i = A_i^\sharp b_i$, with:

$$A_i = \sum_{j=1}^{N} \phi_i(\mathbf{x}^{(j)}) w\left(\mathbf{x}^{(j)}\right) w\left(\mathbf{x}^{(j)}\right)^\mathsf{T}$$
$$b_i = \sum_{j=1}^{N} \phi_i(\mathbf{x}^{(j)}) w\left(\mathbf{x}^{(j)}\right) y^{(j)}.$$

The `f(x)` function is now different, it calls `weight(x)` wich computes the $w(\mathbf{x})$ for one or several values of $\mathbf{x}$. Notice that now θ is a matrix resulting from the concatenation of the θ_i which are vectors of 2 parameters (when $dim(\mathbf{x}) = 1$).

Question 4: Implement the `train_lwls(self,x_data,y_data)` function that computes θ. Show in your report the obtained results.

4.2 Going further

In this section you can study various extensions to what you have done before, and perform comparisons between the various methods.

4.2.1 Back to linear least squares

Question 5: In the `train_ls(self,x_data,y_data)` function that you coded in Question 1, add some code to compute the `slope,intercept,r_value` as in `train_from_stats(self,x_data,y_data)`.

4.2.2 Ridge Regression

Ridge Regression is the other name of regularized linear least squares. This time, we want to minimize

$$\theta^* = \arg\min_\theta \frac{\lambda}{2}\|\theta\|^2 + \frac{1}{2}\|\mathbf{y} - \mathbf{X}^\intercal\theta\|^2, \tag{4}$$

and the analytical solution is:

$$\theta^* = (\lambda\mathbf{I} + \mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}. \tag{5}$$

Question 6: Fill the `train_regularized(self,x_data,y_data,coef)` function to compute regularized linear least squares using (5), where the variable `coef` stands for the regularization factor λ. Add a screenshot of your results in your report.

Question 7: For a batch of 50 points, study in the `train_regularized(self,x_data,y_data,coef)` function how the residuals degrade as you increase the value of `coef`.

4.2.3 Recursive Least Squares

Recursive Least Squares is the incremental version of the batch Least Squares method. In this variant, A and b are recomputed everytime some new data pair is obtained, with the following equations:

$$A^{(t+1)} = A^{(t)} + \phi(\mathbf{x}^{(t+1)})\phi(\mathbf{x}^{(t+1)})^{\mathsf{T}},$$
$$b^{(t+1)} = b^{(t)} + \phi(\mathbf{x}^{(t+1)})y^{(t+1)}.$$

The parameters are computed as follows: $\theta^{(t+1)} = (A^{(t+1)})^{\sharp}b^{(t+1)}$, but they can also be estimated using the Sherman-Morrison formula:

$$(A + \mathbf{x}^{(t)}y^{(t)\top})^{\sharp} = A^{\sharp} - \frac{A^{\sharp}\mathbf{x}^{(t)}y^{(t)\top}A^{\sharp}}{1 + y^{(t)\top}A^{\sharp}\mathbf{x}^{(t)}}.$$

Remark: to apply this formula we must start with a non-zero value for $A^{(0)}$.

Question 8: In `rbfn.py`, fill the `train_rls(self,x,y)` training function that computes θ using the recursive least squares method (without the Sherman-Morrison formula). In the `main.py` file, try to find values of `maxIter,nb_features` leading to good results. Put a screenshot into your report.

Question 9: Do the same with the `train_rls2(self,x,y)` training function, which does the same with the Sherman-Morrison formula.

4.2.4 Comparisons

Question 10: For both RBFNs and LWR methods, try to modify the amount of noise in the generated data (in `sample_generator.py`), and comment your results. Which method is the fastest? Which gives the best results, and why? What are the main differences between these methods, for example if `nb_features` is increased?

Question 11: Using RBFNs, comment on the main differences between incremental and batch methods. What are their main advantages and disadvantages? Explain how you would choose between an incremental and a batch method, depending on the context.

Question 12: Compare both recursive variants (with and without the Sherman-Morrison formula). Which is the most precise? The fastest? You can use computation time measurements to support your answer.