Olivier Sigaud

Regression Labs: Instructions

2020-2021

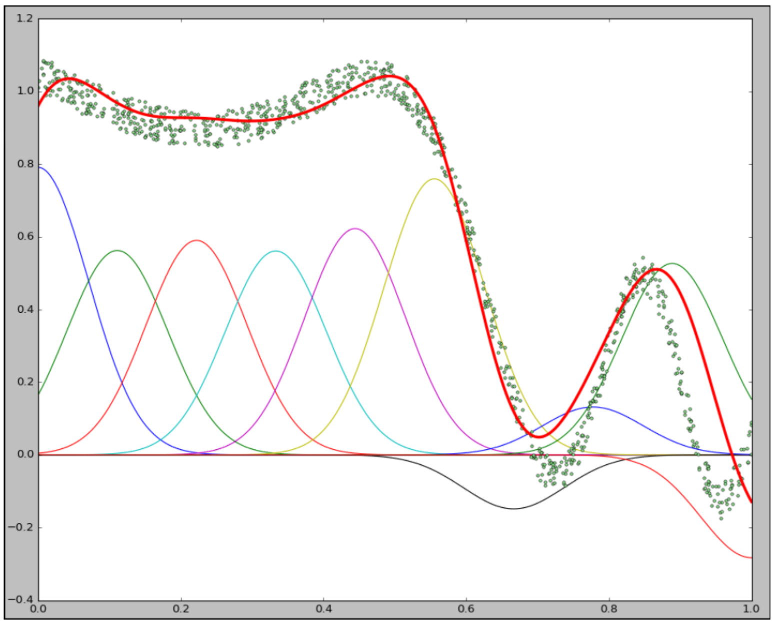

Figure 1: An exemple of regression with RBFNs

1 Final report instructions and evaluation

- Upload you lab report on the Moodle page of the class, following the “Upload your lab report” link.
- You must upload 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).
- For each function required in a question, the corresponding piece of code should be copy-pasted into your report to facilitate evaluation.
- 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).

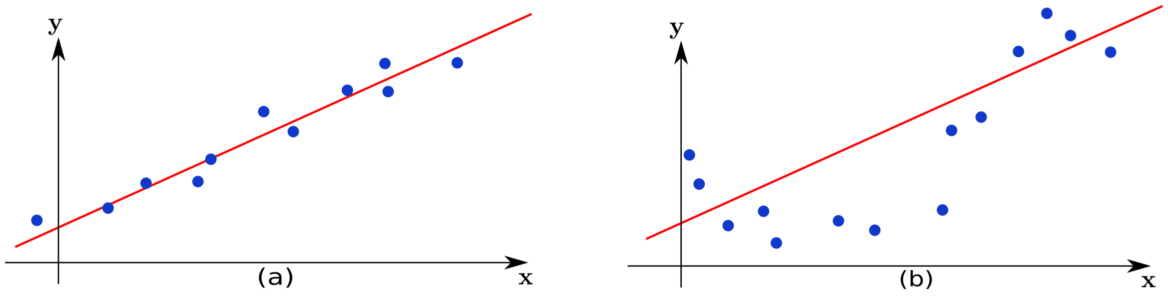

Figure 2: Data more or less adapted to linear regression

Often, linear models are not enough (e.g. Figure 2(b)) 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_i:

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

In particular, we will use Gaussian functions.

2.2 Weighted sums of Gaussian functions

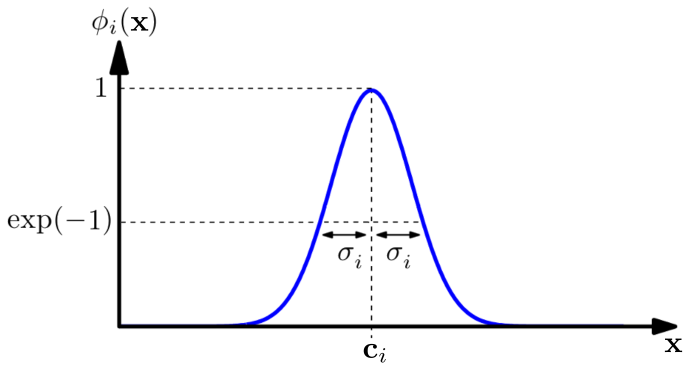

Figure 3: A 1-dimensional Gaussian function.

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:

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

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 Main function

The `main.py` file contains all the necessary code to call all the required function approximation methods. Part of this code can be commented or uncommented to run the desired functions, parameters can be changed, etc.

3.2 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. All visualization functions are in the `plot()` function of each approximator.

3.3 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.

3.4 Vectors of Gaussian functions

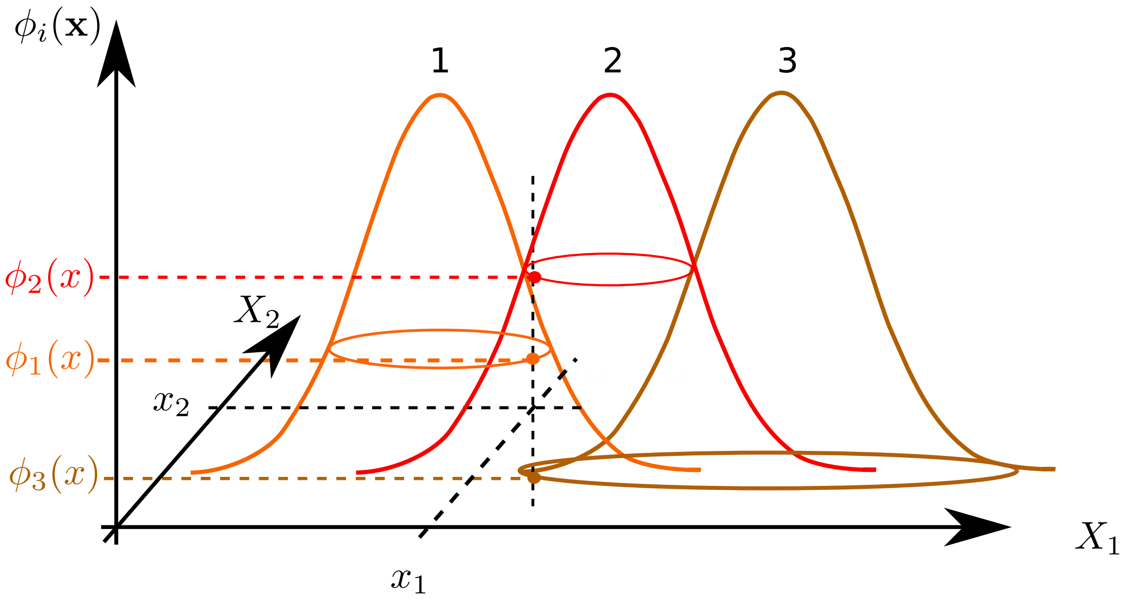

Figure 4: RBFN projection of a 2D point $\mathbf{x}$ into $\phi_i(\mathbf{x})$

A class `Gaussians` is provided in `gaussians.py` to represent a vector of Gaussian feature functions $\phi(\mathbf{x})$ of an input vector $\mathbf{x}$, as used in RBFNs and Locally Weighted Regression (LWR) (see Section 2.2). The number of elements of $\phi(\mathbf{x})$ (i.e. the number k of Gaussian functions) is defined by the `nb_features` attribute.

In regression algorithms, we often use a transpose of $\phi(\mathbf{x})$. But in `numpy`, transposing a standard one-dimensional vector does not work as expected: the result of the tranpose operator is the same vector. As a result, **we choose to code the $\phi(\mathbf{x})$ function so that its output is a vertical vector. When one needs to get the flat horizontal vector corresponding to $\phi(\mathbf{x})$, one needs to use `phi_output(x).transpose()[0]`**.

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 is transformed into a one element vector to keep consistent with the N-dimensional case.

Note that the input of the (multivariate) Gaussian functions is of the same dimension as $\mathbf{x}$, but their output is one-dimensional (see Figure 4).

3.5 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. The `f(self,x)` function represents the function approximator output for a given input `x`, to be optimized. The parameters used by f is the `theta` attribute.

For instance, for RBFNs, we consider a vector of k 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 $f(\mathbf{x}) = \phi(\mathbf{x})^\mathsf{T}\theta$.

These function approximation models come with various `train_...(self,...)` functions that must be filled.

4 Questions

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

We consider either a batch approach or an incremental one. In the batch approach, we consider a batch of data consisting of N $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{1 \le i \le N}$ pairs. In the incremental one, we get an additional pair of data $(\mathbf{x}^{(t)}, y^{(t)})$ at each time step t.

In the general case, the input data $\mathbf{x}$ is of dimension d, and we have $\mathbf{x} = (x_1\ x_2 \cdots x_d)^\mathsf{T}$. Under algebraic form, the batch of data is rewritten $(\mathbf{X}, \mathbf{y})$ where $\mathbf{X}$ is a $N \times d$ matrix and $\mathbf{y}$ is a N dimensional vector.

To solve a linear least square problem, we want to use a model of the form $f_\theta(\mathbf{X}) = \theta^\mathsf{T}\mathbf{X} + \mathbf{b}$. To deal more elegantly with the intercept $\mathbf{b}$, we extend the vectors $\mathbf{x}$ with an additional dimension using $\bar{\mathbf{x}} = (x_1\ x_2 \cdots x_d\ 1)^\mathsf{T}$, and we redefine the Gram matrix as

$$\bar{\mathbf{X}} = \begin{pmatrix} \mathbf{x}_{1,1} & \mathbf{x}_{1,2} & \cdots & \mathbf{x}_{1,d} & 1 \\ \mathbf{x}_{2,1} & \mathbf{x}_{2,2} & \cdots & \mathbf{x}_{2,d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{x}_{N,1} & \mathbf{x}_{N,2} & \cdots & \mathbf{x}_{N,d} & 1 \end{pmatrix}.$$

From the list of vectors `x_data` provided by the sample generator, $\bar{\mathbf{X}}$ can be obtained by first transforming `x_data` into a `numpy array` and transposing using `x_data=numpy.array(x_data).transpose()`, and then adding the ones with `numpy.hstack((x_data,numpy.ones((x_data.shape[0],1))))`.

4.1 Batch Linear Least squares

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

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

The parameters for the optimal model are given by:

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

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.

Code question 1: Fill the `train(self,x_data,y_data)` function so as to perform the same linear least square computation using (3). Add some code to compute the `slope,intercept,r_value` as in `train_from_stats(self,x_data,y_data)`. Does it provide exactly the same results as with the `train_from_stats(self,x_data,y_data)` function? Add a screenshot of your results and the code of your `train(self, x_data,y_data)` function in your report.

4.1.1 Ridge Regression

Ridge Regression is the other name of regularized linear least squares, or Tikhonov regularization. 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

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

Code question 2: 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 and the code of your `train_regularized(self,x_data,y_data,coef)` function in your report.

Study question 3: For a batch of 50 points, study with the `train_regularized(self,x_data,y_data,coef)` function how the residuals degrade as you increase the value of `coef`.

4.2 Radial Basis Function Networks

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

4.2.1 Batch Least Squares

There are two perspectives about the batch least squares method.

The first perspective consists in considering that the RBFN is used to project from an input space into a feature space, and then we perform the standard linear least square calculation in this projected space.
Thus, from the batch of data we build the Gram matrix **G**, projecting each datapoint $x^{(i)}_{1\leq i\leq N}$ into its output vector for all the features $\phi_j(\mathbf{x}^{(i)})_{1\leq j\leq k}$. Then we apply the standard linear least square method, replacing matrix **X** with matrix **G**:

$$\theta^* = (\mathbf{G}^\intercal\mathbf{G})^{-1}\mathbf{G}^\intercal\mathbf{y}. \tag{6}$$

This is the easiest approach to code in `python`.

In the second perspective, we perform the whole calculation from scratch. We want to minimize the following error:

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

To get a local minimum over θ of the function $\varepsilon(\theta)$, we need to solve $\nabla_\theta\varepsilon(\theta) = \mathbf{0}$. To compute the gradient, we use $\nabla(g^2) = 2g\nabla g$. Therefore, we have

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

Since $f_\theta(\mathbf{x}) = \phi(\mathbf{x})^\intercal\theta$, we have $\nabla_\theta f_\theta(\mathbf{x}^{(i)}) = \phi(\mathbf{x}^{(i)})$ and we get

$$\nabla_\theta\varepsilon(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - \phi(\mathbf{x}^{(i)})^\intercal\theta\right)\phi(\mathbf{x}^{(i)})$$

To make the gradient $\nabla_\theta\varepsilon(\theta) = \mathbf{0}$, we get:

$$\frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - \phi(\mathbf{x}^{(i)})^\top\theta\right)\phi(\mathbf{x}^{(i)}) = 0$$

$$\frac{1}{N}\sum_{i=1}^{N}\left(\phi(\mathbf{x}^{(i)})y^{(i)} - \phi(\mathbf{x}^{(i)})\phi(\mathbf{x}^{(i)})^\top\theta\right) = 0$$

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

Let us set

$$\mathbf{A} = \left(\sum_{i=1}^{N}\phi(\mathbf{x}^{(i)})\phi(\mathbf{x}^{(i)})^\top\right)$$

and

$$\mathbf{b} = \sum_{i=1}^{N}\phi(\mathbf{x}^{(i)})y^{(i)}.$$

We then have $\mathbf{A}\theta = \mathbf{b}$. $\mathbf{A}$ is not necessarily an invertible matrix, and the general solution is obtained as $\theta = A^\sharp b$, by using either the "pseudo-inverse" $A^\sharp$ (`np.linalg.pinv(A)`) or using `theta = np.linalg.solve(A,b)`.

Code question 4: 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 and the code of your `train_ls(self,x_data,y_data)` function into your report.

4.2.2 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

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

The idea of gradient descent is to slightly modify θ to decrease $\varepsilon^{(t+1)}$. To do so, we consider the function $\theta \mapsto y^{(t+1)} - f_\theta\left(\mathbf{x}^{(t+1)}\right)$ and compute its gradient in $\theta^{(t+1)}$, which we denote by $\nabla_\theta^{(t+1)}$. Since we have $f_\theta(\mathbf{x}) = \phi(\mathbf{x})^\top\theta$, we get $\nabla_\theta^{(t+1)} = -\phi(\mathbf{x}^{(t+1)})$.

The gradient of a function 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. Here, the goal is to decrease $\varepsilon^{(t+1)}$ thus $\theta^{(t+1)}$ should be modified in the direction defined by $-\nabla_\theta^{(t+1)} = \phi(\mathbf{x}^{(t+1)})$. The resulting update formula is

$$\begin{aligned}\theta^{(t+1)} &= \theta^{(t)} + \alpha\varepsilon^{(t+1)}\nabla_\theta^{(t+1)} \\ &= \theta^{(t)} + \alpha(y^{(t+1)} - f_\theta(\mathbf{x}^{(t+1)}))\phi(\mathbf{x}^{(t+1)})\end{aligned}$$

where $\alpha > 0$ is a coefficient called the "learning rate". Using $f_\theta(\mathbf{x}) = \phi(\mathbf{x})^\top\theta$, we can also write it:

$$\theta^{(t+1)} = \theta^{(t)} + \alpha(y^{(t+1)} - \phi(\mathbf{x}^{(t+1)})^\top\theta^{(t)})\phi(\mathbf{x}^{(t+1)}).$$

Code question 5: 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 `alpha` leading to good results. Put a screenshot and the code of your `train_gd(self,x,y)` function into your report.

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:

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

The parameters are then computed as $\theta^{(t+1)} = (\mathbf{A}^{(t+1)})^{\sharp} b^{(t+1)}$. One may also use `numpy.solve(A,b)` as before, but this can fail when **A** is singular.

Another incremental approach avoids the computation of the pseudo-inverse $(\mathbf{A}^{(t+1)})^{\sharp}$ by computing this inverse incrementally, using the Sherman-Morrison formula:

$$u = v^T = \phi(\mathbf{x}^{(t)})$$
$$(\mathbf{A}^{(t+1)})^{\sharp} = \mathbf{A}^{(t)\sharp} - \frac{\mathbf{A}^{(t)\sharp}uv\mathbf{A}^{(t)\sharp}}{1 + v\mathbf{A}^{(t)\sharp}u},$$
$$\mathbf{b}^{(t+1)} = \mathbf{b}^{(t)} + \phi(\mathbf{x}^{(t)}).$$

Remark: to apply this formula, we must start with a non-zero value for $A^{(0)\sharp}$, e.g. the Identity matrix.

In `rbfn.py`, the `train_rls(self,x,y)` and `train_rls2(self,x,y)` training functions are two variants for computing θ using the recursive least squares method (without the Sherman-Morrison formula). The `train_rls_sherman_morrison(self,x,y)` function computes θ using the recursive least squares method with the Sherman-Morrison formula.

Study question 6: In the `main.py` file, look for values of `maxIter,nb_features` for those functions leading to good results. Put a screenshot into your report.

Study question 7: Compare both recursive variants (with and without the Sherman-Morrison formula) and gradient descent. Which is the most precise? The fastest? You can use computation time measurements to support your answer.

Study question 8: 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.

4.3 Locally Weighted Least Squares

The third family of models is Locally Weighted Least Squares (LWLS). The LWLS algorithm uses a weighted sum of M local linear models. In RBFNs, the size of the vector θ was the number of features. Here, for each local model θ_k, it is the number of dimensions of $\bar{\mathbf{x}}$[1] to take the intercept into account. Thus $\dim(\theta_k) = \dim(\mathbf{x}) + 1 = d + 1$. The model of the global latent function can be written

$$f(\mathbf{x}) = \sum_{k=1}^{M} \frac{\phi_k(\mathbf{x})}{\sum_{j=1}^{M} \phi_j(\mathbf{x})} m_{\theta_k}(\mathbf{x}),$$

with $m_{\theta_k}(\mathbf{x}) = \bar{\mathbf{x}}^{\mathsf{T}}\theta_k$.

[1] Remember that $\bar{\mathbf{x}} = (x_1\ x_2 \cdots x_d\ 1)^{\mathsf{T}}$. The function to compute $\bar{\mathbf{x}}$ is given in `lwr.py` under the name `bar(x)`.

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

$$\varepsilon_k(\theta_k) = \frac{1}{2N}\sum_{i=1}^{N}\phi_k\left(\mathbf{x}^{(i)}\right)\left(y^{(i)} - m_{\theta_k}\left(\mathbf{x}^{(i)}\right)\right)^2 = \frac{1}{2N}\sum_{i=1}^{N}\phi_k\left(\mathbf{x}^{(i)}\right)\left(y^{(i)} - (\bar{\mathbf{x}}^{(i)})^T\theta_i\right)^2.$$

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

$$-\frac{1}{N}\sum_{i=1}^{N}\phi_k(\mathbf{x}^{(i)})\bar{\mathbf{x}}^{(i)}\left(y^{(i)} - (\bar{\mathbf{x}}^{(i)})^T\theta_k\right) = 0.$$

Therefore, we pose $\theta_k = A_k^\sharp b_k$, with:

$$\mathbf{A}_k = \sum_{i=1}^{N}\phi_k(\mathbf{x}^{(i)})\bar{\mathbf{x}}^{(i)}(\bar{\mathbf{x}}^{(i)})^T$$

$$\mathbf{b}_k = \sum_{i=1}^{N}\phi_k(\mathbf{x}^{(i)})\bar{\mathbf{x}}^{(i)}y^{(i)}.$$

This calculation gives us the parameters θ_k of each local linear model. The global parameter θ is now a matrix resulting from the concatenation of all local models θ_k.

Code question 9: The code of the `train_lwls(self,x_data,y_data)` function that computes θ is provided. Run it and show in your report the obtained results.

Study 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?