FTML practical session 4: 2023/03/31

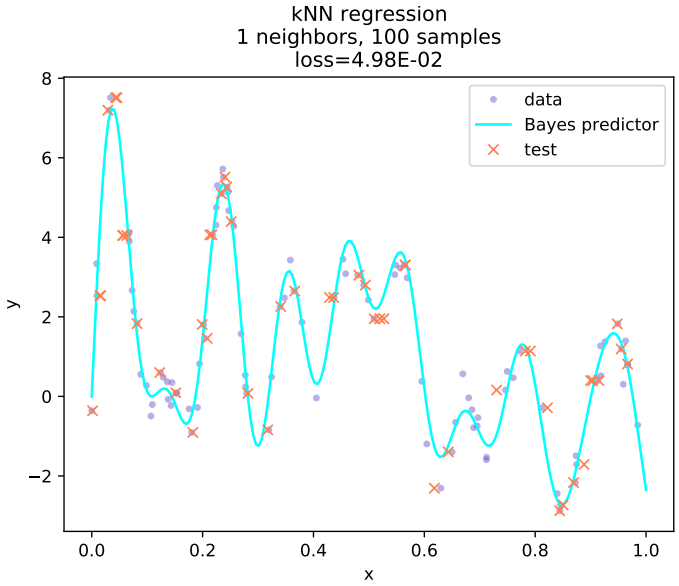


TABLE DES MATIÈRES

INTRODUCTION

In this session we will illustrate the curse of dimensionality with a nearest neighbors algorithm, and then implement a logistic regression manually, performing gradient descent on the empirical risk minimization problem. The two problems are not related.

1 CURSE OF DIMENSIONALITY AND ADAPTIVITY TO A LOW DIMENSIONAL SUPPORT

1.1 Nearest neighbors algorithms

A nearest neighbors algorithm is a method of prediction, based on a dataset, but without any optimization. It consists in averaging the predictions of the nearest neighbors of a sample x in a given dataset. See two examples below, where both input and output spaces are $\mathbb{R}$. The two example differ by the number of samples in the dataset.

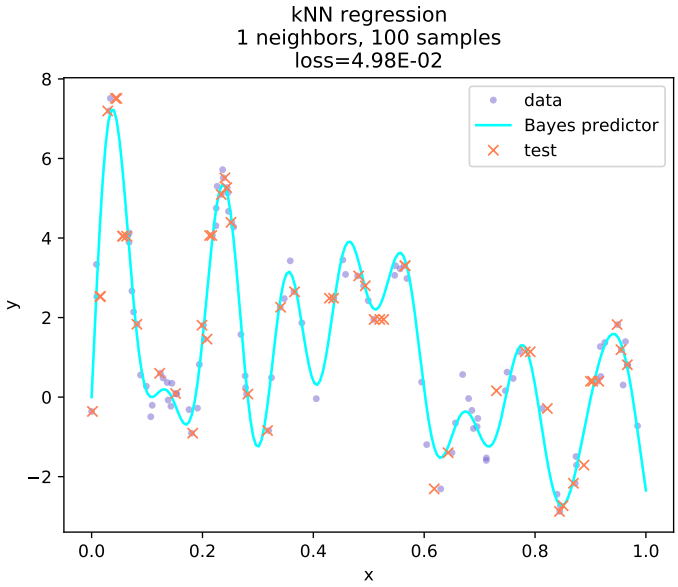


FIGURE 1 – knn

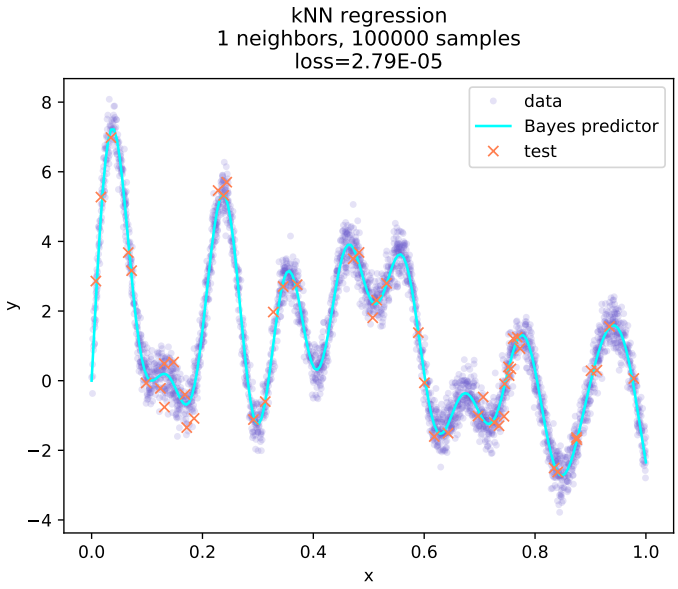


FIGURE 2 – knn

A natural question would be : why do we not use this method all the time ? The answer is that as soon as input space dimension is not very small (for instance not < 10), the number of samples that would be required to obtain a satisfactory error is simply too large.

1.2 Curse of dimensionality

We will study the influence of the input space dimension on the possibility to approximate a function with a nearest neighbors approach. Our setting is :

— input space : $\mathcal{X} = [0,1]^d$. Hence, here the dimensionality of the problem is d.

— output space : $\mathcal{Y} = \mathbb{R}$

— toy function to approximate : $f : x \in \mathbb{R}^d \mapsto \|x\|$ (euclidean norm).

In the general case, it is possible to show that if the target function f^* is Lipshitz-continuous, and if the n samples in the dataset are on a lattice that divides $[0,1]^d$ in cubes of equal size, then the error made by a k-NN esimator can be upper bounded by a bound of the form $\mathcal{O}(n^{-1/d})$.

Note that this is a rough bound. It is possible to have more subtle results, and the constant in the $\mathcal{O}$ depends on d and on the Lipshitz constant of the target function However, it is not possible to have a bound that is "way better" (i.e. "way smaller") than this bound.

This means that if d is large, then $1/d$ is small, and the decrease of the error is very slow. To be more precise, in order to reach an error ϵ, it is possible to show that the number of sampled needed n_ϵ verifies

$$n_\epsilon \geqslant \frac{\epsilon^{-d} d^{d/2}}{\alpha^{d/2}} \tag{1}$$

where $\alpha >$ is a constant. n_ϵ hence grows exponentially with $1/\epsilon$, more than exponentially with d which is a example of curse of dimensionality.

We note that our target function is indeed Lipshitz continuous and thus respect the hypothesis to obtain equation 1. If a function is not Lipshitz continuous, the situation is even worse !

1.2.1 *Simulation*

Run a simulation that computes the test error of a kNN estimator, for different values of n and d, in order to observe the curse of dimensionality. You can try several values of k, for instance 2.

You can use the template file **knn_1.py**. Note that as we know the function that we try to approximate, it is possible to perfeclty evaluate the error made by the k-NN algorithm for each sample. For instance, you can observe images like the one in figure 3. In the simulation that generated the figure, the samples (both the dataset and the test samples) were drawn randomly in $\mathcal{X}$, with a uniform distribution (hence the dataset is not on a regular lattice, although you can do use a regular lattice in your simulation).

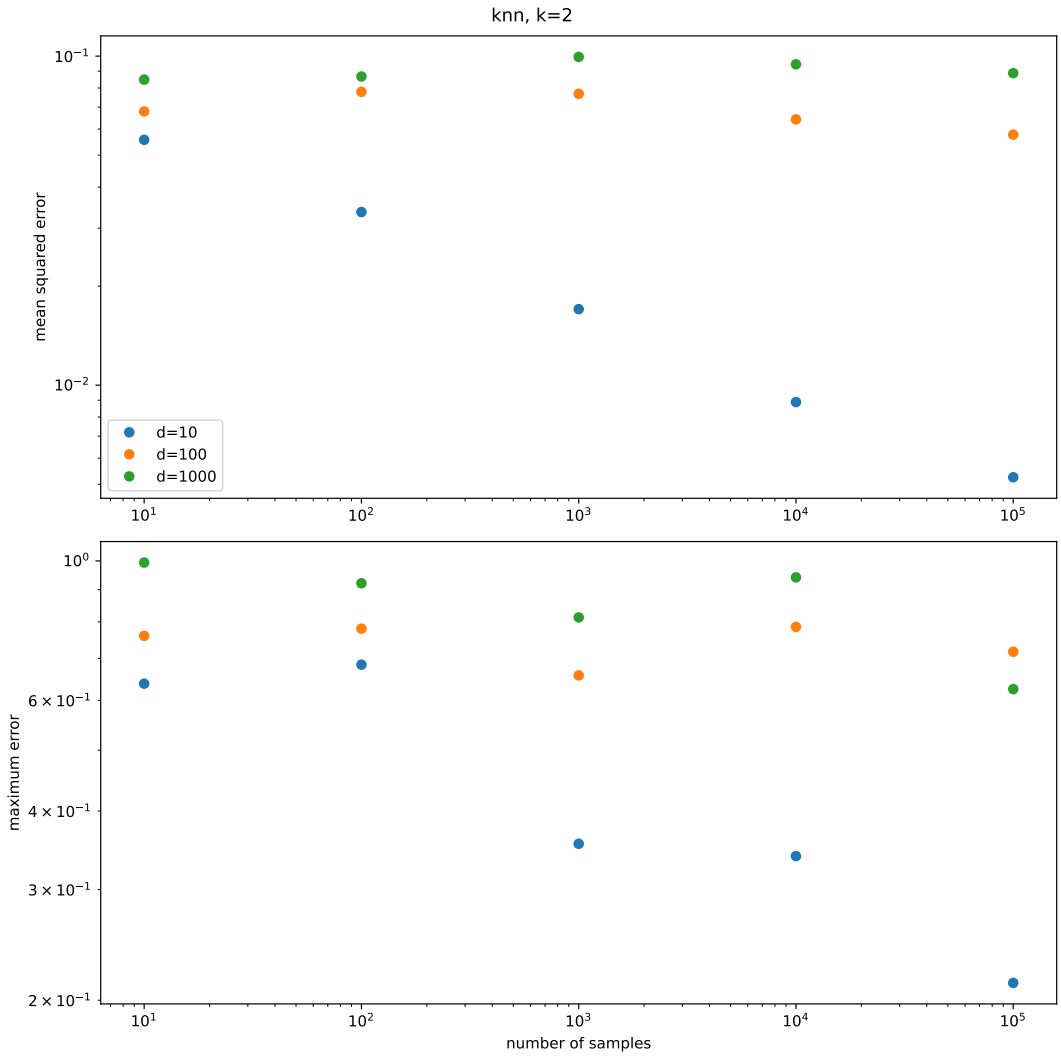


FIGURE 3 – Curse of dimensionality. We observe that the logarithm of the error is linear in the logarithm of the number of samples, with a slope that depends on the dimension d.

1.3 Adaptivity

However, the situation is not always that catastrophic in high dimensions and we will illustrate this with an example. We keep exactly the same setting, but now the

input dataset X is given and fixed, and we use $d = 30$.

Perform a k-NN estimation using the data stored in **data_knn/** and use a varying number of samples n that you use from the dataset. Plot the error as a function of n.

You can use the template file **knn_2.py**. You should observe something like figure 4. We note that the learning rate is way better than $\mathcal{O}(n^{-1/d})$.

— What could be an explanation of this phenomenon ?

— How could we test this hypothesis ? (we have seen a way during the class)

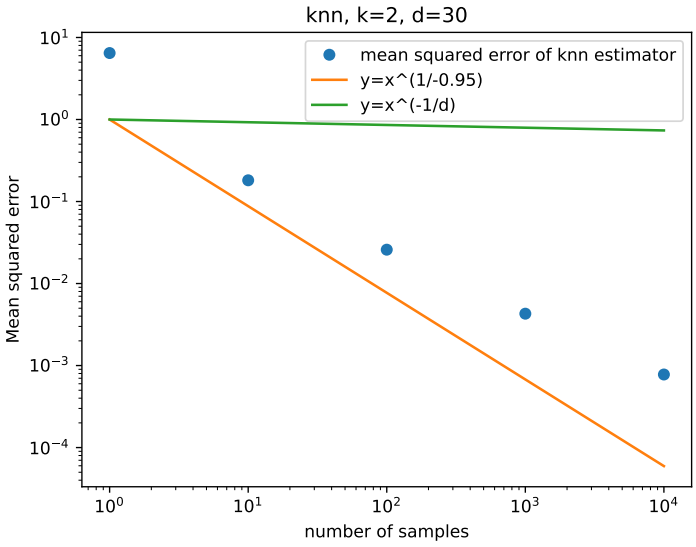
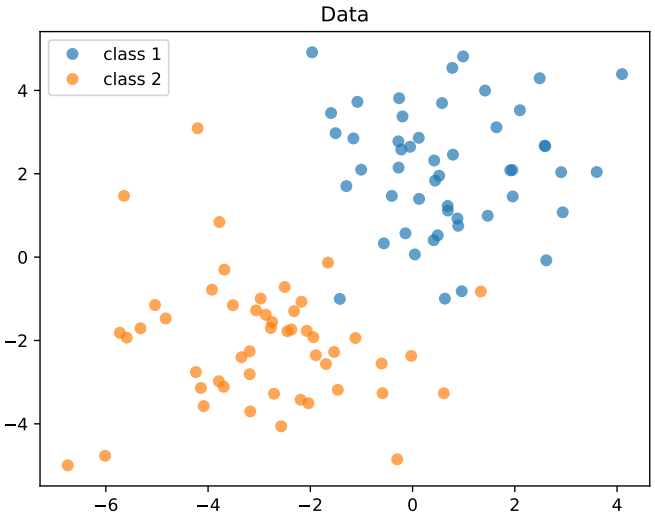


FIGURE 4 – Nearest neighbors estimation on a different dataset.

2 LOGISTIC REGRESSION AND GRADIENT DESCENT

The goal of this exercise is to study some first examples of first-order algorithms, namely GD and SGD. Some binary classification data are stored in the **lr/data/** folder.

Implement a regularized logistic regression on this problem, optimized by GD and then SGD.



— $\mathcal{X} = \mathbb{R}^2$

— $\mathcal{Y} = \{-1, 1\}$

— Logistic loss :

$$l(\hat{y}, y) = \log(1 + e^{-\hat{y}y}) \tag{2}$$

With the usual L2 regularization, the empirical risk writes :

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(x_i^T\theta, y_i) + \frac{\lambda}{2}\|\theta\|^2 \tag{3}$$

We note $g_i(\theta) = l(x_i^T\theta, y_i) + \frac{\lambda}{2}\|\theta\|^2$. Then

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} g_i(\theta) \tag{4}$$

We admit the expressions of the gradients for now :

$$\nabla_\theta g_i = -y_i\sigma(-x_i^T\theta y_i)x_i + \lambda\theta \tag{5}$$

and

$$\nabla_\theta R_n = \frac{1}{n}\sum_{i=1}^{n} -y_i\sigma(-x_i^T\theta y_i)x_i + \lambda\theta \tag{6}$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is a sigmoid function :

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{7}$$

— you have some template files in **lr/**

— use scikit-learn first to perform a logistic regression (without GD or SGD) and have an idea of your target empirical risk after optimization, and of the target test accuracy to aim for.

— use scikit-learn to split the dataset into train / test

— explore different values of λ, of the learning rates, and of the various HPs of the problem.

— you may also automate the search for HPs like in the previous session