Linear and Deep Models Basics with Pytorch, Numpy, and Scikit-Learn

Back cover image design with "Stable Diffusion".

Contents

List of Figures

Preface

This book is an introduction with examples to the python module called "pytorch" for models of regression and classification. The other existing modules for machine learning considered herein are "Scikit-Learn" (also called "sklearn") in a dedicated chapter and "statsmodel" for variance estimation in another dedicated chapter. The module "sklearn" is often involved in projects of machine learning because of its intuitive and powerful processing of the data with many methods implemented often with multicore parallelism. The python module for the graphical outputs is "matplotlib" which allows to draw curves, lines and scatterplots plus the legends and the axis with names.
When the models are not kept linear, the extensions with nonlinearities of the models come from hidden layer(s) within a neural network (nn) framework. "Deep learning" allows to train these nonlinear models despite a high dimensionality and numerous parameters.

NN and GLM

Herein, we are interested on the neural networks called "feedforward neural network" (ffnn) or "multi-layer perceptrons" (mlp). The name mlp has different meanings from an author to another in the literature, because historically the perceptron is for binary classification and not regression. These networks are fully connected neural networks hence with fully-connected layers. All the possible connexions between the nodes of a layer to the nodes of the next layer enter the models. This also induces that the network receives an input "x" at the first layer and produces an output -a prediction approximating the true "y"- at the last layer after a cascade of nonlinear transformations.

The glm are a generalization or many models of classification and regression. They aim at predicting a variable "y" from the knowledge of variables "x", with an unique function with a high level parameterization. Herein, we are interested on only particular cases with python because dealing with this very general unique fonction is not practical, nevertheless the general case is also discussed. These neural networks allow to extend glm in order to improve the training and the modeling. How to define the network and how to train the parameters in order to improve the inference for linear models and reach a relevant prediction.

Training algorithms

The interest in implementing the training algorithms is to understand the mechanisms involved

and to improve some current implementations and models. Thus, the book provides a fully working python code built on the top of the module pytorch and which is completely flexible. On the contrary to the usual python modules for machine learning where the models are already fully implemented the process is different with pytorch: one needs to define and train the parameters for each model or build a general implementation which is able to handle several models as proposed herein. One can use existing programs or existing high level libraries but learning asks for coding.

It is proposed an introduction and implementation to several algorithms for the estimation of the parameters of deep glm. The linear versions are particular cases and the foundation of these models, they are often involved in most of the chapters herein. The algorithms are implemented with numpy and pytorch for several of them while some numerical results come directly from the existing modules that we use for the computations.

Datasets and python codes available for download

The number of datasets processed with python herein is near 20, with a complementary list of some other datasets from repositories. The smaller dataset is 10 rows for pedagogical reasons while the larger ones are about 500000 observation from forestry for one and 11000000 for a second one from physics. The usual dataset "mnist" is included with 60000 images as an usual benchmark in deep learning.

Herein, images datasets are most of the time converted into this same "tabular form" with an unique data file where each row corresponds to an image. The file formats for large tables is also discussed with several examples and exercices. One of the purposes of the document is to be able to handle those large datasets with "pytorch" (and "sklearn") with a limited available quantity of computer memory for training a linear or deep glm. Preprocessing should take only a few seconds or at most a few minutes for millions of rows herein. Obviously, the more as possible of computer memory is probably a wise advice but this is not always possible for a cost reason or for the largest datasets.

Several datasets and python codes are expected to be available for download at the url "github.com/rpriam/book1" from a few days after that the book will appear at the publishing platform website. Otherwise, the reader may consult for any information, the page of the publishing platform for the book, or may directly send a message to "rpriam@gmail.com" for any request about the datasets and the companion programs.

Included (chapters, exercices, solutions) and not included (everything else)

It has been decided to not present the backpropagation in this volume but first-order and second-order derivatives are discussed for the linear glm. The purpose is to get rid of most of the algebra and be happy with the automatic computation of the values of the derivatives of any loss function, thanks to pytorch. The bayesian models have also not been included and are left for a second volume with more advanced models. Neural networks specialized for texts and images processing, called recurrent, convolution or transformer networks are not discussed herein neither but the last

exercice of the book allows to learn about as a perspective.

The document is divided into 9 chapters including a chapter for the introduction and a chapter for the full correction of selected exercices (among the nearly 60 proposed at the end of each chapter) with python codes and several additional large datasets.

The exercices allow to check if the chapter is well understood and allow to go behond the contents of the course. A complementary contents to this book is expected to be proposed as a second volume with more advanced methods and eventually more analytical contents.

How to run the algorithms

The computer used for writing the document is with a quite old cpu with only 4 cores and a small gpu (2 Go) but with some supplementary memory (8 Go). In case of unlikelily memory problem, if freeing the computer memory is not enough the reader may reduce the size of the larger datasets by working only on a sample or adding the missing loop with chunks. But this should not be required, most of the time, as this was handled already and took care of. Linux was the operating system for running the python code, and jupyter notebooks, one per chapter, were initially converted into the text format ".tex" before final updates. This supposes to install all the python packages and libraries (called herein "modules") required in order to access their functionalities. Another solution is jupyter notebooks with google colab for cloud computing or for local computations.

Chapter 1

Introduction

1.1 Overview

In the following chapters we are interested on the estimation of statistical models related to neural networks. The purpose is to predict a variable y_i called "target", "outcome", or "of interest" or "dependent" with the knowledge of a vector x_i, aggregating a set of p variables $\{x_{i1}, x_{i2}, x_{ip}\}$ eventually large, called "predictive", "independent" or "explicative", and otherwise "predictor". One supposes that there exists a relation between x and y, say a function $g(.)$, such that:

$$y_i = g(x_{i1}, x_{i2}, x_{ip}) .$$

The most usual models for this problem are divided into two different cases, one for the prediction of a numeric value and one for the prediction of a categorical value. They are named after the:

- "regression models" when y is a scalar (or sometimes a vector) with its values continuous (real) or discrete (integer ordered). They are called "linear regression" for a continuous outcome, and "simple" for one predictor or "multiple" for several ones.
- "classification models" when y is a category or class label. The category is usually coded with an integer not ordered, just as an index. Thus, they are called[1] "logistic regression" or "multinomial regression" for respectively two classes or more than two classes.

These models depend on unknown parameters θ, because they need to adapt to a sample. Each dataset and related population is different such that the prediction changes. Instead of changing the function, only the parameters need to change. The function is well chosen and enough general in order to be relevant for many different datasets or populations, each one with a different parameter. This is written as follows:

$$y_i = f(x_{i1}, x_{i2}, x_{ip}; \theta) .$$

For each dataset or population, the parameters in the vector θ needs to be found, or say "inferred", "estimated", or "learned" while the model is said to be "fitted" or "trained",

$$\theta = ? .$$

[1] Thus, both modeling cases lead to the prediction of a numerical value with related models. Herein all the models have same foundation, as the (extended) members of the "generalized linear models" most of the time.

1.2 Notion of sample and log-likelihood

1.2.1 Random sample

The population is supposed infinite or enough large to be supposed infinite. Thus a datum is here of the form:

$$(y_i, x_i)\,,$$

with for continuous ($q \geq 1$) or binary ($q = 2$), multinomial ($q > 2$), discrete ($q = \infty$) regressions:

$$x_i = (x_{i1}, x_{i2}, \cdots, x_{ip})^T \in \mathbb{R}^p \text{ and } y_i \in \mathbb{R}^q \text{ (with often } q = 1) \text{ or } y_i \in \{0, 1, \cdots, q\} \text{ (with } q \in \mathbb{N}_*^+).$$

The dataset is a data sample with n observations,

$$s_n = \{(y_1, x_1), (y_2, x_2), \cdots, (y_i, x_i), \cdots, (y_n, x_n)\}\,.$$

Because only a sample is available and not the whole population or even an enough large sample, there exists an (eventually small or big) error in $\hat{\theta} = \hat{\theta}(s_n)$, the value of θ found from the sample. This is not the true parameter θ^* for the whole population which is unknown, hence there is an algebrical and numerical difference between the available value and the true value.

1.2.2 Loglikelihood

The "probabilistic distribution" of the sample comes from a member of the family of "generalized linear models" such that there exists a known parametric form for:

$$f(y_i; x_i, \theta)\,.$$

Thus, the "loglikelihood" is defined as the log of the n products of the density function or probability mass function above, one per sample, such that,

$$\begin{aligned}
\log \mathcal{L}(\theta) &= \log f(y_1, y_2, \cdots, y_i, \cdots, y_n; x_1, x_2, \cdots, x_i, \cdots, x_n, \theta) \\
&= \log \{f(y_1; x_1, \theta) \times f(y_2; x_2, \theta) \times \cdots \times f(y_i; x_i, \theta) \times \cdots \times f(y_n; x_n, \theta)\} \\
&= \log f(y_1; x_1, \theta) + \log f(y_2; x_2, \theta) \cdots + \log f(y_i; x_i, \theta) \cdots + \log f(y_n; x_n, \theta)\,.
\end{aligned}$$

And, with the summation sign sigma or sign sum,

$$\log \mathcal{L}(\theta) = \sum_{i=1}^{n} \log f(y_i; x_i, \theta)\,.$$

The derivatives of the loglikelihood w.r.t. θ allows to write an algorithm in order to find an estimation $\hat{\theta}$ of the parameters, given a sample of n observations.

1.2.3 Optimization

The search for the parameters θ is summarized as follows:

$$\hat{\theta} = \text{argmax}_\theta \log \mathcal{L}(\theta)\,.$$

This notation says that the vector $\hat{\theta}$ to the left is the solution which maximizes the function to the right. This function depends on the vector θ which can take usually a not countable infinite number of possible different values, and we are looking for the one which makes the function maximum.

In neural network, this is a loss function, thus, the optimization is the look for a minimum, for instance of " - $\log \mathcal{L}(\hat{\theta})$ " here. Sometimes, the loss function is instead just a distance between the true target values y_i and their approximations from the network $\hat{y}_i$, as explained later.

There are many algorithms for this task because, the problem is different if the dataset is large or small, the variable space is high dimensional or small dimensional, and for other computational constraints. This is not rare to have different algorithms for differents models, as in computational statistic for bayesian settings for instance. But for neural networks, the algorithms are often more generic because the derivative are not written in closed-form, and often estimated numerically. These derivatives are the horse power of most of the training algorithms because the nonlinear function is approximated locally with a linear or quadratic function before the optimization. The approximation comes from these derivatives, as in any usual Taylor serie. For some rare cases, the solution $\hat{\theta}$ is a closed-formed expression, exact without algorithm, such as for the linear regression.

1.2.4 Bias and variance

As the parameters $\hat{\theta} = \hat{\theta}(s_n)$ comes from a sample s_n of size n, there are several questions about: is it accurate enough for the current or new data. Generally, in a statistical setting for machine learning with smooth functions, the estimator is such that it converges to the true parameter for large samples, $\hat{\theta} \to \theta$. It is relevant if it is unbiased, which means that the average of the estimator $\hat{\theta}$ from many samples is equal to the true parameter unknown θ, $E(\hat{\theta}) = \theta$. Another properties which comes with the estimator is its variance, say $V(\hat{\theta}) > 0$, which tells how near the true parameter is to its approximation via a confidence interval, this is discussed in a next chapter.

1.3 Numerical example with a nonlinear curve

1.3.1 Simple linear regression

This is the simplest case actually, there is no need for a distribution, just a loss function:

$$\ell(\theta) = \frac{1}{n}\sum_i^n (y_i - \beta_0 + \beta_1 x_{i1})^2\,.$$

The difference with a distribution -and its related loglikelihood function- is that the loss function is just a measure of the difference between the true targets and the approximated targets as found from a model, while the distribution has a probabilistic interpretation.

Here, y_i is the true target variable while the prediction from the model is $\hat{y}_i = \beta_0 + \beta_1 x_{i1}$. In the literature this notation is also sometimes after training such that $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_{i1}$, thus this has to be understood according to the context. A way to be clear would be to write $\hat{y}_i(\theta)$ and $\hat{y}_i(\hat{\theta})$ respectively without additional symbol for the target variable.

The derivatives are found with usual algebra as follows.

$$\begin{array}{rcl} \frac{\partial \ell(\theta)}{\beta_0} & = & \frac{\partial}{\beta_0}\left[\frac{1}{n}\sum_i^n (y_i - \beta_0 + \beta_1 x_{i1})^2\right] \\ \frac{\partial \ell(\theta)}{\beta_1} & = & \frac{\partial}{\beta_1}\left[\frac{1}{n}\sum_i^n (y_i - \beta_0 + \beta_1 x_{i1})^2\right] . \end{array}$$

Which leads to:

$$\begin{array}{rcl} 0 & = & \frac{1}{n}\sum_i^n -(y_i - \hat{\beta}_0 + \hat{\beta}_1 x_{i1}) \\ 0 & = & \frac{1}{n}\sum_i^n -(y_i - \hat{\beta}_0 + \hat{\beta}_1 x_{i1}) x_{i1} . \end{array}$$

If not resolved by the more elementary way to write one coefficient as a function from the first equation and replace in the second equation, in order to find the expression of one coefficient before the expression of the second from first equation, the problem is simply written matricially for resolution:

$$\begin{pmatrix} 1 & \frac{1}{n}\sum_i^n x_{i1} \\ \frac{1}{n}\sum_i^n x_{i1} & \frac{1}{n}\sum_i^n x_{i1}^2 \end{pmatrix} \begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{pmatrix} = \begin{pmatrix} \frac{1}{n}\sum_i^n y_i \\ \frac{1}{n}\sum_i^n y_i x_{i1} \end{pmatrix} .$$

This leads to:

$$\begin{array}{rcl} \begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{pmatrix} & = & \begin{pmatrix} 1 & \frac{1}{n}\sum_i^n x_{i1} \\ \frac{1}{n}\sum_i^n x_{i1} & \frac{1}{n}\sum_i^n x_{i1}^2 \end{pmatrix}^{-1} \begin{pmatrix} \frac{1}{n}\sum_i^n y_i \\ \frac{1}{n}\sum_i^n y_i x_{i1} \end{pmatrix} \\ \begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{pmatrix} & = & \frac{1}{\frac{1}{n}\sum_i^n x_{i1}^2 - (\frac{1}{n}\sum_i^n x_{i1})^2} \begin{pmatrix} 1 & -\frac{1}{n}\sum_i^n x_{i1} \\ -\frac{1}{n}\sum_i^n x_{i1} & \frac{1}{n}\sum_i^n x_{i1}^2 \end{pmatrix} \begin{pmatrix} \frac{1}{n}\sum_i^n y_i \\ \frac{1}{n}\sum_i^n y_i x_{i1} \end{pmatrix} \\ \begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{pmatrix} & = & \frac{1}{var(x_1)} \begin{pmatrix} 1 & -\overline{x_1} \\ -\overline{x_1} & \overline{x_1^2} \end{pmatrix} \begin{pmatrix} \overline{y} \\ \overline{yx_1} \end{pmatrix} . \end{array}$$

The usual linear algebra leads to the known analytical usual solution, with an inverse of the matrix which is either analytical for $p = 2$ (as just above) or $p = 3$, either numerical (via a computer

program) for larger values of p. When the function is nonlinear, the problem is different because a closed-form solution is not available yet. Another issue is for large p and large n where the matrix involved is large such as the numerical algorithm is too slow or worse, the matrix does not fit in the computer memory. In both cases, alternative algorithms allow to get a solution directly from the expression of the derivatives above in order to find the unknown parameters (see next chapters).

1.3.2 Polynomial regression

For regression, a polynom can be seen as a neural network, but with an heuristic as a first approach.

A simple way to consider at first an hidden layer in a neural network with a nonlinear transformation is with polynom:

- From unidimensional input $x_i = x_{i1}$ to multidimensional layer

$$x_i \to g_\ell(x_i) = \begin{pmatrix} x_i \\ x_i^2 \\ x_i^3 \\ \vdots \\ x_i^\ell \end{pmatrix} \in \mathbb{R}^\ell .$$

  This is not the traditional layer with a linear transformation, but nowadays, a layer has a generic meaning wich could include such transformation. Because neural networks are most often useful for high-dimensional input vectors instead of a simple scalar, such transformation is most often not considered, and replaced by a nonlinear function as in a next subsection for an example.

- From multidimensional layer to unidimensional output $\hat{y}_i$

$$g_\ell(x_i) = \begin{pmatrix} x_i \\ x_i^2 \\ x_i^3 \\ \vdots \\ x_i^\ell \end{pmatrix} \to \hat{y}_i ,$$

  where the approximation of the true target variable is written as a weighted sum, of the powers of x_i plus a constant term,

$$\hat{y}_i = g(x_i, \theta) = w_0 + w^T g_\ell(x_i) = w_0 + \sum_{k=1}^{\ell} w_k x_i^k .$$

In a matrix format, one gets that:

$$\begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{pmatrix} = \begin{pmatrix} 1 & x_1 & \cdots & \cdots & x_1^\ell \\ 1 & x_2 & \cdots & \cdots & x_2^\ell \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & \cdots & \cdots & x_n^\ell \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_\ell \end{pmatrix} .$$

Note that the simple linear regression is also written with this form, with $\ell = 1$, because the result before is retrieved by the product to the left of the transpose matrix which contains the variable x and called design matrix.

The estimation $\hat{y}_i$ approximates y_i with the help of the knowledge of x_i, and depends on parameters w_k, such that a comparison between the vector of y_i and the vector of $\hat{y}_i$ is the Euclidean distance, or the mean. It is thus optimized the following criterion.

- Cost function with $\theta = (w_k)$, it is obtained by comparing the true input with the approximated output.
$$\ell(\theta) = \frac{1}{n}\sum_i^n [y_i - w_0 - w^T g_\ell(x_i)]^2 .$$

- Optimization, the optimal solution searches for the best parameters.
$$\begin{aligned} \hat{\theta} &= \text{argmin}_\theta \, \ell(\theta) \\ &= \text{argmin}_\theta \, \frac{1}{n}\sum_i^n [y_i - \hat{y}_i(\theta)]^2 \\ &= \text{argmin}_\theta \, \frac{1}{n}\sum_i^n [y_i - w_0 - w^T g_\ell(x_i)]^2 . \end{aligned}$$

The solution is known via a least square or regression problem and with an analytical solution (when there is no problems such as collinearities). Note that an usual trick is to consider the vector x_i by adding the component 1 as its first component, such as the "intercept" (also called "bias" in the computing literature: understood as a bias term for the model without this additive correction) w_0 comes as a component into the vector w for a lighter notation, with "$[w_0, w^T][1, g_\ell(x_i)^T]^T$".

1.3.3 Neural network

For the nonlinear regression, the loss is with $g()$, and,

$$\ell(\theta) = \frac{1}{n}\sum_i^n [y_i - g(x_i, \theta)]^2 .$$

The nonlinear model is approximated with a linear model in the polynomial function but this has some limits for more complex functions which are unknown and not polynomial.

In a neural network, the nonlinearity is modeled with layers which are linear transformations associated to a nonlinear function.

The neural network with one hidden layer, instead of a polynomial transformation, is a linear transformation plus a nonlinear one. The nonlinear transformation is called "activation function", in the sense that each component in the hidden layer is a neuron, and it is activated or not in the sum thanks to the activation function which has often a shape as a "S curve": a smooth switch zero or one. This is not discussed further herein, the idea is biological with signals which are propagated in the brain via neurons which are or not activated according to the received amount of signal.

Formally, this is just written as follows:

- From unidimensional input y_i to multidimensional layer

$$x_i \rightarrow x_i^T W^{(1)} = \begin{pmatrix} x_i w_{11} \\ x_i w_{12} \\ x_i w_{13} \\ \vdots \\ x_i w_{1\ell} \end{pmatrix} \in \mathbb{R}^\ell .$$

  A non linear function $\sigma()$, such as sigmoid or thanh, is applied to each component,

$$\sigma(x_i^T W^{(1)}) = \begin{pmatrix} \sigma(x_i w_{11}) \\ \sigma(x_i w_{12}) \\ \sigma(x_i w_{13}) \\ \vdots \\ \sigma(x_i w_{1\ell}) \end{pmatrix} \in \mathbb{R}^\ell .$$

- From multidimensional layer to unidimensional output $\hat{y}_i$ Another set of weights are required in order to retrieve the dimensionaly of y_i, such that one just writes that:

$$\sigma(x_i^T W^{(1)}) \rightarrow \hat{y}_i ,$$

where,

$$\hat{y}_i = g(x_i;\theta) = w_{20} + \sigma(x_i^T W^{(1)})^T W^{(2)} ,$$

or eventually,

$$\hat{y}_i = g(x_i;\theta) = w_{20} + \sigma\left(\sigma(x_i^T W^{(1)})^T W^{(2)}\right) ,$$

or directly,

$$\hat{y}_i = g(x_i;\theta) = \sigma\left(w_{20} + \sigma(x_i^T W^{(1)})^T W^{(2)}\right) .$$

The later case would be for a target variable in the range $[0;1]$ because the additional term w_{20} would make the infence more cumbersome if left out from the second activation function. while the just before case would be for a target variable in the real line with continuous values. The case without a second activation function is just more linear and may be enough for fitting some simple curves. There exists theorem which prove that neural networks are able to approximate any nonlinear function, but in practice this is not always easy to find a relevant value for their weights.

Here w_{20} is not called "intercept" but "bias" in the literature for neural networks, while,

$$\sigma(x_i^T W^{(1)})^T W^{(2)} = w_{21}\sigma(x_i w_{11}) + w_{22}\sigma(x_i w_{12}) + w_{23}\sigma(x_i w_{13}) + \cdots + w_{2\ell}\sigma(x_i w_{1\ell}) .$$

Thus finally, this is another optimization problem:

- Cost function
$$\ell(\theta) = \frac{1}{n}\sum_{i}^{n}[y_i - w_{20} - \sigma(x_i^T W^{(1)})^T W^{(2)}]^2 .$$

- Minimization w.r. θ

$$\begin{aligned}\hat{\theta} &= \text{argmin}_\theta \ell(\theta) \\ &= \text{argmin}_\theta \frac{1}{n}\sum_i^n [y_i - \hat{y}_i(\theta)]^2 \\ &= \text{argmin}_\theta \frac{1}{n}\sum_i^n [y_i - w_{20} - \sigma(x_i^T W^{(1)})^T W^{(2)}]^2 .\end{aligned}$$

Generally, x_i is not one dimensional but p dimensional and $\ell << p$, instead of here having ℓ larger than the dimensionaly of x_i, because the neural network reduces the dimensionaly in order to add nonlinearities, as the real space for the data is generally smaller than the one given. There is a first transformation associated to the layer, linear and nonlinear, followed by a linear transformation, and eventually a second nonlinear transformation, not considered here. Note that on the contrary, for small dimensional x_i, an hidden layer has more neurons than p in order to increase the dimensionality of the nonlinear hidden space, $\ell > p$, and get a good prediction for the target variable. This replaces the polynomial transformation with order not explicit, as the order is automatically chosen from the found weights. In theory, one hidden layer is probably enough, but two or three are not rare in practice.

In the literature, it was shown that such functions are able to retrieve the true function $g()$ for a well chosen set of weights and activation transformations: this is more general than the polynomial regression, again with weights but also nonlinear transformation. There is a difference between knowing the function as in inference for the nonlinear regression and estimate its parameters, while not knowing the form of the function in neural networks and estimate the weights in order to retrieve the function: $g(., \hat{\theta})$ versus $\hat{g}(.)$. A consequence is that more parameters are to be fitted in neural networks, this induces more problems of generalization not only because the real function is unknown but also because of the numerous unknown parameters. This is more an estimation of function than an estimation of parameters, even if weights are estimated like usual parameters.

1.3.4 Numerical illustration

Let have a work directory where data files, data directories, and temporary files are stored.

```
def towdir(s):
    return (str('./datasets_book/'+s))
```

Let check the computer memory state. There should be remaining some space otherwise the operating system risks to freeze. For neural network, it is a good habit to check the computer memory. The code for this document was written with a laptop with more than 6 GB of memory, thus it might be running in most of recent architecture very well and even for older computers with enough computer memory.

For instance, psutil may be considered here.

```
import psutil
memory = psutil.virtual_memory()
print(f" Memory in used   : {memory.percent} %\n",
      f"Memory available : { round(memory.free / (1024.0 ** 3),2)} GB")
```

```
 Memory in used   : 31.8 %
 Memory available : 5.77 GB
```

For the current chapter, the computer memory is not a concern as the data samples are very small. A sample from a nonlinear function is generated and the purpose is to retrieve this function, or at least retrieve the target variable in order to predict their values for new data. Think for instance, to a function in physics or in biology which is unknown. By the past scientists looked for a way to find an expression for the function according to some theory, and then used to fit its parameters according to some observation available, while nowadays they may use directly the computer in order to find the function. In this paragraph, the function is simple, but it may be more complicated and more difficult to find for real problem with an unknown form for the function.

In this chapter we do not need pytorch which comes after in the next chapters. A sample from a polynomial curve is generated with python via a new function and stored in the computer memory as python objects "x_all" and "y_all", with 80 points. Hence "x_all" and "y_all" are numpy arrays of one dimension, their length is $n = 80$. For instance $y_i = x_i^2$ with x_i the values at equal distance for the inverval $[-1.5; 1.5]$. But here a polynom with a higher degree was chosen. The data sample (or dataset) is defined from the two data "x_all" and "y_all". It may be stored as a data table which is a matrix with two columns, one for "x_all" and one for "y_all". From these data, we want to find the link from "x_all" to "y_all", hence the polynom (coefficients and degree) is unknown. In next chapters, even the information that the function is polynomial may be unknown because neural networks can infer the function whatever its shape: this is where pytorch becomes useful and in particular when the problem becomes high dimensional. See also the exercices at the end of the chapter for a first implementation with the python module "sklearn" for these dataset and function.

Usually, one wants to know how behave the model on a new dataset, thus, it is sampled a part of the dataset to simulate this. Otherwise, as the generating function is available here, it could be sampled directly new observations instead.

```
import numpy as np
import numpy.random as rd

def f_idx_traintest(n, rate_train = 0.8, shuffle = True):
    n_train = int(np.round(rate_train*n))
    idx_all = range(n)
    if shuffle: idx_all = rd.permutation(idx_all)
    idx_train, idx_test = idx_all[:n_train], idx_all[n_train:]
    return idx_train, idx_test

n                   = len(x_all)
idx_train, idx_test = f_idx_traintest(n)
```

```
x_train, x_test  = x_all[idx_train], x_all[idx_test] # here x is
y_train, y_test  = y_all[idx_train], y_all[idx_test] # a vector!

print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

```
(64, 1) (64, 1) (16, 1) (16, 1)
```

The dataset was stored in txt files in order to get the same data at each run and made available the sample.

```
# np.savetxt("./x_all_4degrees_polynom.txt", x_all)
# np.savetxt("./y_all_4degrees_polynom.txt", y_all)
# np.savetxt("./x_train_4degrees_polynom.txt", x_train)
# np.savetxt("./y_train_4degrees_polynom.txt", y_train)
# np.savetxt("./x_test_4degrees_polynom.txt", x_test)
# np.savetxt("./y_test_4degrees_polynom.txt", y_test)
```

Thus, the files are loaded.

```
import numpy as np
import numpy.random as rd
x_train = np.loadtxt(towdir("./x_train_4degrees_polynom.txt"))
y_train = np.loadtxt(towdir("./y_train_4degrees_polynom.txt"))
x_test  = np.loadtxt(towdir("./x_test_4degrees_polynom.txt"))
y_test  = np.loadtxt(towdir("./y_test_4degrees_polynom.txt"))
```

```
n_train = len(y_train)
n_test = len(y_test)

print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

```
(64,) (64,) (16,) (16,)
```

The data sample was splited into two subsamples, one for fitting the model and one for checking the obtained model with new data. This is what happens with real data when one wants to predict the unknown target variables from new values for the explaining variables, this is the ultimate purpose of the modeling. Actually, in deep learning the purpose is most often "prediction" than "explaining" because the nonlinearities make the explaination cumbersome, such as neural networks were often called "black-box" by the past even if now they are more called "artificiel intelligence" or just related to "deep learning" which is a big change actually. There is even some direction of current research in order to make them explainable which is completely the opposit of the former naming.

For the moment, we stay in the linear case, let draw the real curve and the available sample points.

```
def f_polynomial_value(x, *beta):
    fx = beta[0]
    for c,bc in enumerate(beta[1:]):
        fx += bc * np.power(x,c+1)
    return fx
```

```
def fun_plot_truepolynom(x,y,true_coeffs,showlegend=True,
                         namesample="whole",draw=True):
    if draw: plt.scatter(x, y, color='black',marker='.', s=50)
    if draw: axis = plt.axis()
    x_s = np.asarray([xi for xi in range(-15,16)])/10
    y_s = [f_polynomial_value(xi, *true_coeffs) for xi in x_s.ravel()]
    #
    label_poly = "Polynom "
    for p in range(len(true_coeffs)):
        if p>0:
            if true_coeffs[p]>=0:
                label_poly += " + "
            if true_coeffs[p]<0:
                label_poly += " - "
        label_poly += str(abs(true_coeffs[p]))
        if p>0: label_poly += " x^"+str(p)
    #
    if draw:
        plt.plot(x_s, y_s, label=label_poly + " ")
        plt.xlim(-1.6, 1.6)
        plt.ylim(-4.5, 24.5)
        plt.title("Real polynomial curve and available data points for␣
 ↪"+str(namesample)+" sample")
        if showlegend:
            plt.legend(loc='best')#, bbox_to_anchor=(1, 0.5))
    return plt, x_s, y_s

# %matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
true_coeffs = [-1.0, 0.0, 2.0, - 3.0, 1.5]

plt.figure(figsize=(10, 10))
plt.subplot(2,1,1)
plt_train_polyn, x_s, y_s = fun_plot_truepolynom(x_train,y_train,
                                                 true_coeffs,True,"train")
plt.subplot(2,1,2)
```

```
plt_test_polyn, x_s, y_s = fun_plot_truepolynom(x_test,y_test,
                                                true_coeffs,True,"test")
plt.show()
```

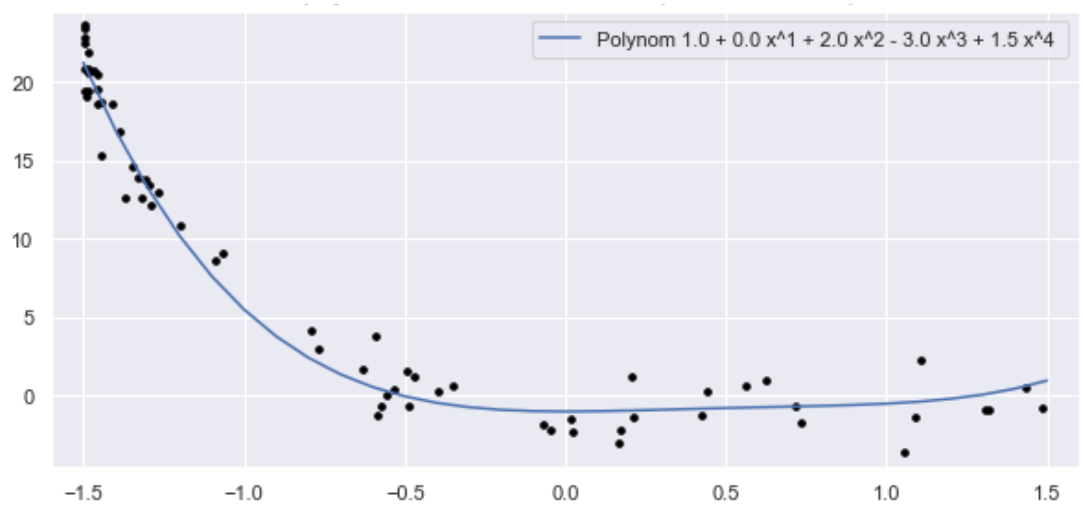

Figure 1.1: True polynomial curve and train sample

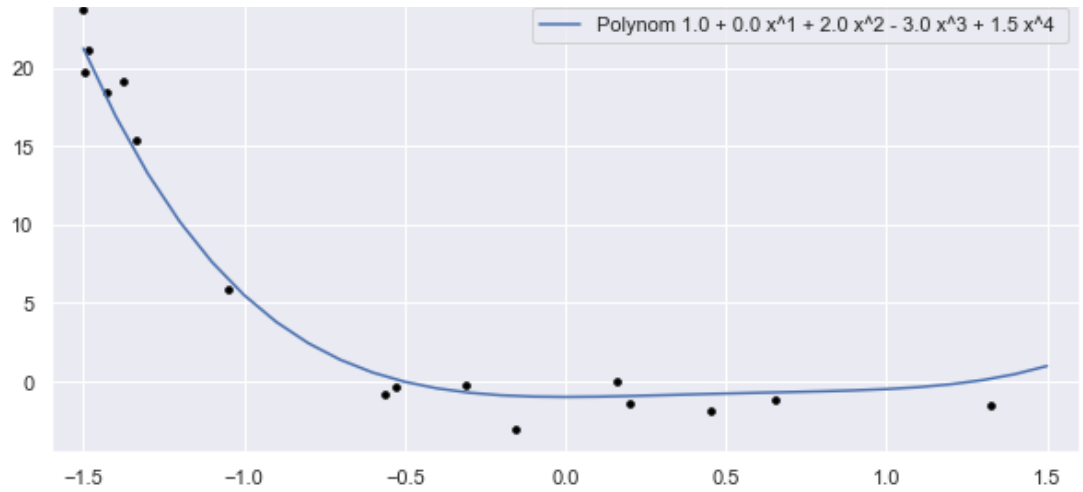

Figure 1.2: True polynomial curve and test sample

Note that here the distribution for the x_i was not taken uniform, hence the reader may use its own example of data for further understanding the polynomial fitting and the sample error associated.

Simple regression fitting

The fitting for the simple regression leads to the following result. The solution is written with the available values in the sample.

With a precision of four digits, the means $\overline{x_{\text{train}}}$, $\overline{x^2_{\text{train}}}$, $\overline{y_{\text{train}}}$, and $\overline{y_{\text{train}} x_{\text{train}}}$, are respectively:

```
-0.5628 1.194 8.0168 -11.7599
```

The derivative of the loss function, $\frac{1}{n}\sum_i^n [y_i - g(x_i, \theta)]^2$, w.r.t. β_0 and β_1 leads to the resolution via a linear system:

$$
\begin{aligned}
\begin{pmatrix} 1 & -0.5628 \\ -0.5628 & 1.1940 \end{pmatrix} \begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{pmatrix} &= \begin{pmatrix} 8.0168 \\ -11.7599 \end{pmatrix} \\
\begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{pmatrix} &= \begin{pmatrix} 1 & -0.5628 \\ -0.5628 & 1.1940 \end{pmatrix}^{-1} \begin{pmatrix} 8.0168 \\ -11.7599 \end{pmatrix} \\
\begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{pmatrix} &= \begin{pmatrix} 1.3610 & 0.6415 \\ 0.6415 & 1.1399 \end{pmatrix} \begin{pmatrix} 8.0168 \\ -11.7599 \end{pmatrix} \\
\begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{pmatrix} &= \begin{pmatrix} 3.3672 \\ -8.2620 \end{pmatrix}
\end{aligned}
$$

This is a good exercice to find the linear system (see also next chapter) with paper and pencil, before writing this solution. With python code, first the matrix is defined, followed by its inverse and its product with the column vector, as follows.

```
A = np.array([ [1,                  np.mean(x_train)],
               [np.mean(x_train) , np.mean(x_train**2)] ])

b = np.array([ np.mean(y_train),
               np.mean(y_train*x_train) ])

beta_hat = np.linalg.inv(A) @ b
```

Note that for a larger number of variables, a more robust solution is to solve for the linear system $A\beta = b$ instead of taking the inverse of A, this is also available in the module numpy with a native function.

Thus, $\hat{\beta}$ is found equal to:

```
print(beta_hat.reshape((2,1)))
```

```
[[ 3.36717979]
 [-8.26200073]]
```

The estimations from the simple linear model are obtained as,

$$
\begin{aligned}
\hat{y}_i &= \hat{\beta}_0 + \hat{\beta}_1 x_i \\
&= 3.36718 - 8.26200 x_i \,.
\end{aligned}
$$

Thus with python, one gets the prediction for the test sample as follows,

```
y_test_hat = beta_hat[0] + beta_hat[1] * x_test
# import pandas as pd
# results_ls = [y_test, y_test_hat]
# results_ls = pd.DataFrame(results_ls).transpose()
# results_ls.columns = ["y_test", "y_test_hat"]
# print(results_ls.head())
```

A measure of the error from the model is the value of the loss function, here the mean square error (mse), which is computed as follows,

```
def mse(y_hat,y):
    return np.sum((y_hat-y)**2,axis=0)/len(y)

print(mse(y_test_hat,y_test))
```

```
29.483941666233978
```

Let solves this problem directly with a numerical algorithm implemented in the module "scipy.optimize" in order to find the unknown parameters as follows.

```
from scipy.optimize import curve_fit

def f_polynomial2_value(x, *beta):
    fx = beta[0] + x * beta[1]
    return fx
```

```
popt, pcov = curve_fit(f_polynomial2_value, x_train.ravel(),
                       y_train.ravel(), p0=np.zeros((2,1)))

# y_test_hat2 = np.zeros(len(x_test))
# for i in range(n_test):
#     y_test_hat2[i] = f_polynomial2_value(x_test[i], *popt)
y_test_hat2 = [f_polynomial2_value(x, *popt) for x in x_test.ravel()]
```

```
print(mse(y_test_hat2,y_test))
```

```
29.4839416885612
```

The same value is slightly obtained as expected for a simple linear model. Anyway, the mse is large because the linear model is not relevant for this dataset, and a nonlinear regression should be fitted. Next, the curve is approximatively retrieved with a better fit than just a linear regression, by changing the function to feed the numerical optimization module.

Polynomial fitting with scipy

The fitting of a curve is illustrated in this paragraph. We are going to compare several polynoms with different degrees because usually the polynomial degree is unknown when only the data sample is available. The purpose is to find the best polynom, say the relevant degree and a relevant estimation for the coefficients.

```
degree_s = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Note that an analytical expression exists already in the linear case such as the simple regression and the polynomial regression. Thus, this example illustrates the fitting of any function with a numerical algorithm and allows to compare with the exact solution. For an optimization of a function with the python module scipy, one writes the nonlinear function and the fitting as follows:

```
from scipy.optimize import curve_fit

def f_polynomial_regression(degree,x_in,y_in,x_new):
    popt, pcov = curve_fit(f_polynomial_value, x_in.ravel(), y_in.ravel(),
                           p0=np.zeros((degree+1,1)))  ##bounds=
    y_new = [f_polynomial_value(xi, *popt) for xi in x_new.ravel()]
    return np.asarray(y_new).reshape(len(y_new),1),popt, pcov

y_train_polyn_hat_s  = [ f_polynomial_regression(degree, x_train,␣
 ↪y_train, x_train) for degree in degree_s ]

y_test_polyn_hat_s  = [ f_polynomial_regression(degree, x_train, y_train,␣
 ↪x_test) for degree in degree_s ]
```

```
for d, degree in enumerate(degree_s):
    np.savetxt(towdir("yhat_train_"+str(degree)+"degrees_polynom.txt"),
               y_train_polyn_hat_s[d][0])
    np.savetxt(towdir("yhat_test_"+str(degree)+"degrees_polynom.txt"),
               y_test_polyn_hat_s[d][0])
```

The resulting estimations are visualized and compared numerically.

```
def fun_plot_polynom(x,y_true,y_hat_s,degree_s,showlegend=True,
                     namesample="whole",draw=True):
    if draw: plt.scatter(x, y_true, color='black',marker='.', s=50)
    if draw: axis = plt.axis()
    err_s = np.zeros((len(degree_s),))
    idx_sort = np.argsort(x.ravel())
    for c, (y_hat,degree) in enumerate(zip(y_hat_s,degree_s)):
        y_hat = y_hat[0]
```

```
        y_hat = y_hat.ravel()
        mse = np.sum((y_hat[idx_sort]-y_true[idx_sort])**2,axis=0)/
↪len(y_hat)
        #print(degree, distance_true_fitted)
        if draw:
            plt.plot(x[idx_sort], y_hat[idx_sort],
                     label='degree={0}'.format(degree))
            plt.xlim(-1.6, 1.6)
            plt.ylim(-4.5, 24.5)
            plt.title("Error for the "+str(namesample)+" sample")
            if showlegend:
                plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
        err_s[c] = mse
    return err_s
```

```
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # plot formatting
plt.figure(figsize=(10, 10))
plt.subplot(2,1,1)
err_train_polyn_s = fun_plot_polynom(x_train,y_train,y_train_polyn_hat_s,
                                     degree_s,True,"train")
plt.subplot(2,1,2)
err_test_polyn_s = fun_plot_polynom(x_test,y_test,y_test_polyn_hat_s,
                                    degree_s,False,"test")
plt.show()
```

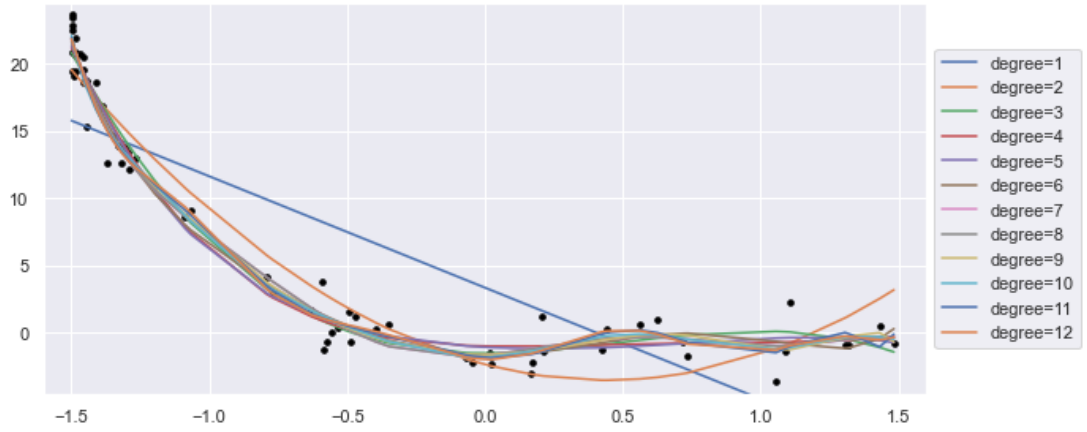

Figure 1.3: Several fitted polynomial curves and train sample

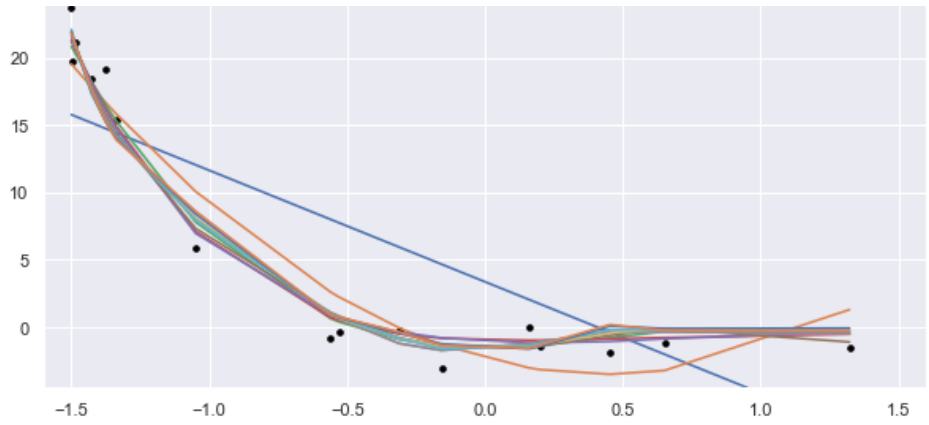

Figure 1.4: Several fitted polynomial curves and test sample

The error here is the usual average quadratic distance between the estimated and the true values.

```
import pandas as pd
results = [degree_s, err_train_polyn_s, err_test_polyn_s]
results_pd = pd.DataFrame(results).transpose()
results_pd.columns    = ["degrees", "err_p3_tr", "err_p3_te"]
results_pd["degrees"] = results_pd["degrees"].astype(np.int8)
print(results_pd)
```

```
    degrees  err_p3_tr  err_p3_te
0         1  26.042915  29.483942
1         2   5.078671   5.864941
2         3   2.134521   1.980844
3         4   1.960133   1.879909
4         5   1.933263   1.968510
5         6   1.774574   2.215584
6         7   1.687690   2.534879
7         8   1.687600   2.531840
8         9   1.654814   2.750692
9        10   1.635598   2.765160
10       11   1.592089   3.051881
11       12   1.574494   3.105261
```

```
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # plot formatting
axis = plt.axis()
plt.ylim(0,10)
plt.xticks(degree_s)
plt.plot(degree_s, err_train_polyn_s, label="train")
plt.plot(degree_s, err_test_polyn_s, label="test")
```

```
plt.title("Error from fitting with polynomial curves of degrees 1-20")
plt.legend(loc='best', bbox_to_anchor=(0.5, 1.05),
           ncol=2, fancybox=True, shadow=True)
plt.show()
```

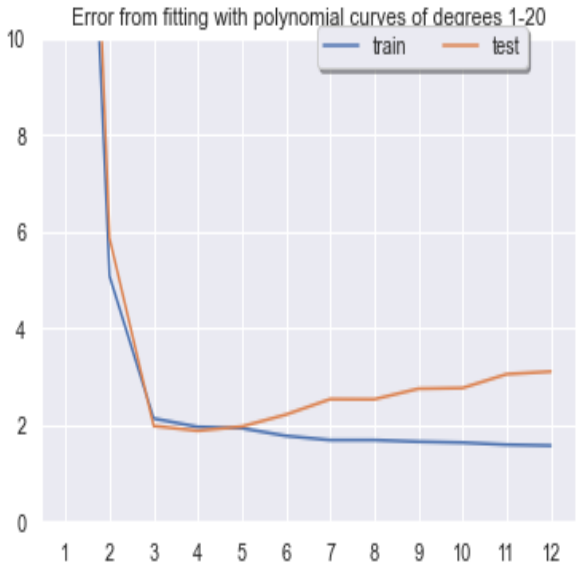

Figure 1.5: Error on the train and test samples

It is clear that for the train sample, increasing the degree of the polynom induces a better fitting, while for new data with the test sample, this is not true, the error increases and even explodes in some cases. The curve learns all the noise of the data instead of learning the underlying smooth real curve. Note that the optimization with scipy allows to introduce bounds on the parameters such as the parameter is no greater than its bounds. In a next chapter, another approach is considered also related to curve fitting with an explicit penalty term added to the objective function.

With the smaller mse for the test sample, hence for new data, the model kept here is the polynomial regression with four degrees. The estimations from this model are obtained as,

$$\begin{aligned} \hat{y}_i &= \hat{\beta}_0 + \hat{\beta}_1 x_i + \hat{\beta}_2 x_i^2 + \hat{\beta}_3 x_i^3 + \hat{\beta}_4 x_i^4 \\ &= -0.97154 - 0.42205\, x_i + 2.55021\, x_i^2 - 2.98778\, x_i^3 + 1.13000\, x_i^4 . \end{aligned}$$

This model is globally not far of the true one, according to the parameter values, but for some datasets, the mse on the test sample may be not minimum exactly for the right degree of the polynom because this way to choose is not optimal or the sample too small. It makes sense that looking just to one new sample is not enough if the test sample is not reflecting the general case, and better approaches are implemented in a next chapter with several test samples instead of just only one. This is related to the problem of generalization which is interested on the quality of the model for a novel data sample, or how make a model enough general for any new similar dataset from the unique available sample.

Next chapters, pytorch is presented for such cases with also some main algorithms for deep learning. Instead of relying on a numerical library for the optimization, the numerical training must be written for each general architecture of neural network. This allows to avoid the traps coming from the nonlinearities and the associated useless local minima of the loss function. In the other hand, neural network models are relevant for many current data and still able to predict an outcome for new data. Less flexible models are more limited and perform lesser, hence extending linear statistical models within the neural framework is appealing when enough data are avaible for training.

1.4 Glm as neural networks

The linear and polynomial regression are not enough because they are not relevant for non continuous target variables and unknown nonlinearities. This is why was defined the family of models called glm or linear generalized model where the distribution functions and the underlying loss functions allows to extend the Gaussian and Euclidean cases, in the statistical literature. Some members of this family are considered in the next chapters.

As a recall to generalized linear models, let denote θ for the natural parameter and ϕ for te positive dispersion parameter. The target variable y_i belongs to an exponential family with density or mass of specific form,

$$p(y|\theta,\phi) = \exp\left\{\frac{y\theta - b(\theta)}{a(\phi)} + c(y,\phi)\right\}.$$

In the generalized linear models the mean $\mu = \mathbf{E}(Y|\mathbf{x})$ is written via a function called link function which is a strictly increasing function, and which is applied to the usual inner product of the parameter vector β with the vectors of explicative variables (plus bias one for the intercept) x_i, such that the link function $\eta(.)$ has an inverse $\eta^{-1}(.)$ and,

$$g(\mu) \quad = \quad \eta(\mathbf{x}\beta).$$

This also induces:

$$y_i \approx \mu(x_i^T\beta).$$

In neural network, the functional relation between y_i and x_i is more complex and it is retrieved the notation of the statistical nonlinear regression. Instead of $\mu(x_i^T\beta)$ in linear generalized models, this is written with weights w,

$$y_i \approx \mu(x_i, w).$$

To be more precise, often at least for classification and regression which are met in deep learning for images and biology, the expression is:

$$y_i \approx \mu(\phi(x_i,w)^T\beta).$$

In this case, y_i is most of the time univariate and continuous as a curve fitting problem like the one in this chapter, or categorical for image classification with advanced layers in the neural network in order to process the spatial information hidden in the images.

Deep regressions can often be interpreted as nonlinear regressions with a particular expression for the nonlinearities: the observed vector of explaining independent variables is transformed via a nonlinear space before applying the usual final weighting and sum towards the target variable. Historically, the Gauss-Newton method was introduced for inference of the parameters of such model and back-propagation which is related to the so-called "chain rule" for the derivatives. With a growing number of variables and large datasets, first-order algorithms are usually more practicable than exact second-order methods even today.

1.5 Why learn pytorch

Pytorch is a module for python which is a powerful interpreted computing langage with hundred of modules available. This module is dedicated to the definition of architectures of neural networks and their training. A way to maximize the likelihood with this module is to minimize a loss as for the three cases of distribution often met: Bernoulli, Multinomial, and Gaussian for binary, discrete and continuous target variables. In the general case, one has to write her/his own loss function, or just consider one existing in the python module. For neural networks, the loss measures the difference between the true target variable y_i and the one found by the network $\hat{y}_i$, by extending the usual Euclidean distance to other measures of discripancy more relevant for robust comparison or binary/multi-category/integer comparisons.

It is possible to consider losses not defined in the python module such as for survival analysis or any likelihood or even function to be optimized. Another example is for a nonlinear regression when the nonlinearities are not polynomial anymore, a main reason to consider neural networks when the nonlinear function is unknown. With this module, many neural networks are already implemented, and available online, thus this makes possible to use and adapt them for our datasets, but also to improve, combine or evaluate. The module allows also to optimize in a fast and reliable way the architectures while finding values of the weights automatically for very large datasets, with diverse computers (laptop, desktop, distributed servers) and diverses ships (cpu, gpu, multi cpu, multi gpu). This module is available with computer clusters or supercomputers for going to scale after prototyping with a simple home computer or workstation.

1.6 Exercices

1. (python) Compare side by side in an array, for ten observations, the true values and estimated values for the target variable when the polynom is fitted with four degrees. Hint, the parameters are found as the second array in the return from the function f_polynomial_regression() and stored in the variable y_test_polyn_hat_s.

2. (sklearn) Consider a neural network with non linear functions instead of the polynomial regression and find suitable values of the weights for this dataset. Compare the predicted/estimated target values and the error with the results from the ones above in the chapter.

3. (python) Implement a "cross-validation" with scipy and propose a method for regularizing the model. Hint, a way to see this problem of regularization is that when the polynomial

degree increases, the regression coefficient increases and the norm of the vector of coefficient should be kept enough small.

4. (stat) How to choose well the training and test samples for best results.
5. (stat) Retrieve the solution for one variable of the linear regression in a matricial notation from the initial matricial expression of the y as a function of the x (as in the polynomial regression above). The solution should be the same than the one written after the derivatives proposed here. The linear system with the analytical expression comes from the derivative w.r.t. β_0 and β_1 before rewriting into a matrix and two vectors by isolating the vector of regression coefficient at each row.
6. (stat) For the polynomial regression (and more generally multiple regression), an analytical exact expression is available for the mean squared error. Retrieve this expression, and its value for the polynomial regression. Statistical or machine learning books treats these questions with a more elaborate developement, out of the scope herein.

Chapter 2

Linear models with numpy and scikit-learn

In this chapter we are interested on finding a linear function for explaining observations y_i depending on p independent variables x_{ij} aggregated in the vector $x_i = (x_{i1}, x_{i2}, \cdots, x_{ip})^T$. This is written with a noise ε_i as,

$$\begin{aligned} y_i &= f(x_i) + \varepsilon_i \\ &= f(x_{i1}, x_{i2}, \cdots, x_{ip}) + \varepsilon_i \,. \end{aligned}$$

The noise comes from the error with the linear approximation. The error may be induced by an inexact measure (for prediction) or unknown variables (for explaining). This model is usually called a regression model while $f(x_i)$ is the regression line. Here in the linear case we have just:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_j x_{ij} + \cdots + \beta_p x_{ip} + \varepsilon_i \,.$$

With real data the noise is often Gaussian, and other noises are possible actually, depending on the distribution of y_i. Note that with pytorch, there is just need to choose a loss function among a list of loss functions already available in the python module, or eventually define the loss by writing a python function, before launching the optimization which does not ask for more analytical work. On the other hand, knowing the underlying mechanism allows a better understanding of the algorithms, their limits and a better extension or improvement the models if one is not happy with the current available contents.

The main purpose is thus to find the coefficients β_j from a sample of n couples (y_i, x_i), compound of the variable y_i called target, outcome or dependent and the variable x_i called the predictor or explanatory or independent (non dependent) variable. The sample is a set of n couples,

$$s_n = \{(y_1, x_1), (y_2, x_2), \cdots, (y_n, x_n)\} \,.$$

The method is called supervised because the variables y_i is available for each vector x_i, hence, the purpose is to "explain" or "predict" y_i as a function of x_i. These two purposes are well separated:

- "to explain" has for meaning that the coefficients β_j can be understood as weights positive or negative for the dependant variable y_i,
- "to predict" has for meaning that given new values of x_i a value for y_i is computed and should be near the true value.

In practical applied statistics, one can explain with a non predictive model. Hence, whatever is the size of the variance of the noise, its distribution just needs to be typically Gaussian. Eventually, one may not even look at the noise, but this becomes mandatory for validating the model and the coefficients of the regression. Herein, we want a relevant model for prediction, and the noise might be not too large otherwise the prediction could be not enough efficient.

The linear algebra provides a solution for the vector of regression coefficients $\beta = (\beta_0, \beta_1, \beta_2, \cdots, \beta_j, \cdots, \beta_p)^T \in \mathbb{R}^{p+1}$ in closed-form, but next chapters will present alternative solutions via diverse algorithms. Finding a value for β from a sample has diverse namings in the literature: infering, fitting, learning or training. They may not have exactly the same meaning from an author to another, but one always gets $\hat{\beta}$ given the sample, and this approximated value shoud be near the true value.

As the sample comes from a whole population often infinite, a consequence is that the obtained solution depends on the sample: $\hat{\beta}$ may be denoted $\hat{\beta}(s_n)$, $\hat{\beta}_s$ or $\hat{\beta}_n$. This estimation comes with some variability in the sense that changing the sample leads to a different estimation $\hat{\beta}$. Thus when possible, it is checked the expectation and the variance of $\hat{\beta}$, at least for simpler models such as linear.

The other quantities which are measured are related to the sample, and compute how good are the estimation $\hat{y}_i$ in comparison to the true values y_i of the target variable.

Next, a more formal presentation leads to the expression for the coefficients and of the diverses criteria discussed for the quality. A probabilistic interpretation of the regression is written with the help of the Gaussian noise. An illustrative example follows then.

2.1 Linear regression

The basic assumption when one derives the equations for linear regression is to assume that the output is determined by a given continuous function $f(x)$ and a random noise ε given by the Gaussian distribution with mean value equal to zero and a spherical variance with undetermined parameter σ^2.

Recall that the derivative of the cost function of the linear regression with respect to the parameters β is equal to zero: $X^T(y - X\hat{\beta}) = 0$. This leads to the solution for $\hat{\beta}$ such that:

$$\begin{aligned} \hat{\beta} &= argmin_\beta \, \ell(\beta) \\ &= \left(X^T X\right)^{-1} X^T y. \end{aligned}$$

where,

$$\ell(\beta) = \frac{1}{n}\|y - X\beta\|^2.$$

Here the loss function is written directly with a quadratic distance in order to have a notation related to a Gaussian distribution.

2.1.1 Probabilistic interpretation and model

It is introduced a term related to a variance such that, the criterion leads to an equivalent criterion for the optimization problem:

$$\begin{aligned}
\tilde{\ell}(\beta,\sigma) &= \frac{n}{2}\log 2\pi\sigma^2 + \frac{n\,\ell(\beta)}{2\sigma^2} \\
&= \frac{n}{2}\log 2\pi\sigma^2 + \frac{||(y-X\beta)||_2^2}{2\sigma^2} \\
&= \frac{n}{2}\log 2\pi\sigma^2 + \frac{\sum_i (y_i - x_i^T\beta)^2}{2\sigma^2} \\
&= \sum_i \frac{1}{2}\log 2\pi\sigma^2 + \sum_i \frac{(y_i - x_i^T\beta)^2}{2\sigma^2} \\
&= -\sum_{i=1}^n \log p(y_i; x_i^T\beta, \sigma) \\
&= -\log \prod_{i=1}^n p(y_i; x_i^T\beta, \sigma) \\
&= -\log p(y; X, \beta, \sigma)\,.
\end{aligned}$$

The two last rows comes from a property of a Gaussian probability function as follows. The probability distribution of the dependent variable y_i conditionally to the independent variables is normally distributed because of this equality just before. Even without this hypothesis of distribution, this results holds. This comes from the expression of this underlying distribution as a Gaussian distribution:

$$y_i \sim \mathcal{N}(x_i^T\beta, \sigma^2) \quad \text{is equivalent to} \quad p(y_i; x_i^T\beta, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_i - x_i^T\beta)^2}{2\sigma^2}\right].$$

After considering such expression, we just need to take the logarithm from the distribution of the i.i.d. variables y_i conditionally to x_i. Conditionally just mean that the variable x_i appears in the mean, hence can be supposed constant, while this is y_i which is random. This makes sense because y_i is depending on x_i and comes after that x_i is given a value. Thus, coming with this distribution, we retrieve our criterion which is optimized for the usual linear regression, despite that there was eventually no distributional hypothesis originally (with just the simple sum of squares for the loss). Indeed:

$$\begin{aligned}
\log p(y; X, \beta, \sigma) &= \log \prod_{i=1}^n p(y_i; x_i^T\beta, \sigma) \\
&= \log \prod_{i=1}^n \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_i - x_i^T\beta)^2}{2\sigma^2}\right] \\
&= \sum_{i=1}^n \log\left\{\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_i - x_i^T\beta)^2}{2\sigma^2}\right]\right\} \\
&= \sum_{i=1}^n \left\{-\frac{1}{2}\log 2\pi\sigma^2 - \frac{(y_i - x_i^T\beta)^2}{2\sigma^2}\right\}
\end{aligned}$$

Hence one can assume that the variables with values y_i are identically and independently distributed according to the above Gaussian distribution. This leads to define the likelihood of an observation y_i with the input variables x_i given the vector of coefficients β which is unknown. Since these events are assumed to be independent and identically distributed we can build the probability distribution function (p.d.f) for all possible observations $(y_1, y_2, \cdots, y_n)$ as the product of the single events for the observations y_i.

To find a value for the unknown parameter β, the more frequent method is the maximum likelihood estimation (MLE). This method estimates the parameters of the parametric probability distribution, given a data sample. At the end, the function is maximized such as the sample maximizes the likelihood function. For analytical reasons and practical reasons, this is the logarithm of the likelihood which is maximized, also called the loglikelihood, this is equivalent because the logarithm is a strictly increasing function: this allows to avoid the maximization of products which is a tricky numerical problem. Note that the loss in neural network is minimized, thus this is the negative of the loglikelihood which is optimized by modules such as pytorch, but they are not limited to loglikelihood and other functions not related to parametric distribution are possible such as for robust losses or more advanced structures of neural networks (recurrent for sequence processing, or convolution for image processing for instance, out of the scope herein). This induces that there is a strong distributional hypothesis in this statistical inference, which is not required in neural networks: they are able to fit any unknown function with a relevant optimization and a suitable architecture (hidden layer(s) and number of neurons). Only more flexible statistical models such as tree-based for instance are able to compete with neural networks.

The consequence of this result is that the solution for β is a maximum likelihood estimator, and there is also required the estimation for the variance parameter:

$$\begin{aligned} \hat{\beta} &= \text{argmax}_{\beta}\, \tilde{\ell}(\beta, \sigma) \\ \hat{\sigma}^2 &= \text{argmax}_{\sigma}\, \tilde{\ell}(\hat{\beta}, \sigma)\,. \end{aligned}$$

The first line comes from the fact that σ cancels out for the first term and remains proportional for the second term, when computing the derivative of the function $\tilde{\ell}(.,.)$ w.r.t. β, while the solution for σ is after the solution for the regression coefficients.

2.1.2 Example of simple linear regression (data)

The work directory is given from the function.

```
def towdir(s):
    return (str('./datasets_book/'+s))

import deepglmlib.utils as utils
import numpy as np
```

```
import importlib
importlib.reload(utils)
```

```
<module 'deepglmlib.utils' from
'/home/rodolphe/Documents/ARTICLEs/BOOK/deepglmlib/utils.py'>
```

In this example, the explaining variable is univariate such that $x_i = x_{i1}$, hence:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i \,.$$

In python, we generate the noise ε_i, the explaining variable x_i, compute y_i and show their linear relation with the line.

The data and true parameters are stored in the computer disk, thus loaded just below from text files.

```
beta = np.loadtxt(towdir("./beta_1d_reglinear.txt"))
xy   = np.loadtxt(towdir("./xy_1d_reglinear.txt"))
x    = xy[:,0].reshape((len(xy),1))
y    = xy[:,1].reshape((len(xy),1))
x.shape,y.shape, beta.shape
```

```
((10, 1), (10, 1), (2,))
```

```
%matplotlib inline
import matplotlib.pyplot as plt
xmin   = 0
xmax   = 1

fig, ax = plt.subplots()

ax.plot(np.array([xmin,xmax]), beta[0]+beta[1]*np.array([xmin,xmax]),␣
 ↪"b-")
ax.plot(x, y ,'bo')
ax.axis([xmin,xmax,0.80*min(y), 1.20*max(y)])
ax.set_xlabel(r'$x$')
ax.set_xlabel(r'$y$')
ax.set_title(r'Sample points and the real linear regression line')

plt.show()
```

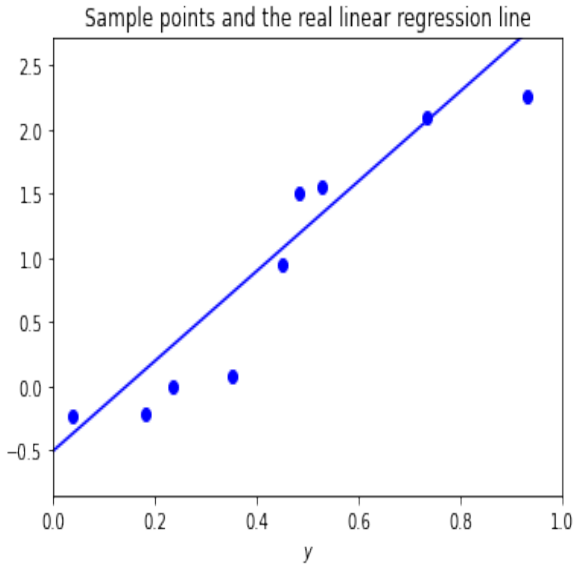

Figure 2.1: True regression line and sample with 10 observations.

2.1.3 Analytical solution for the linear multiple regression

More formally, a regression model is able to model the conditional distribution of the variable y with the given variable x, denoted $p(y|x)$.

As seen just before, a sample of n couples (x_i, y_i) is available with :

- The dependent variables y_i for $i = 1, 2, \ldots, n$
- The independent variables $x_i = (x_{i0}, x_{i1}, x_{i2}, \ldots, x_{ip})^T$ for $i = 1, 2, \ldots, n$ multivariate in R^{p+1} with $x_{i0} = 1$ for the intercept β_0.

For notational reasons, the intercept β_0 is included in β while x_i is also with an additional component x_{i0} in order to deal with this definition of beta. The function $f(.)$ which allows to write y_i as a function of x_i is linear here. For instance, the number of apples produced in an apple tree may be in a linear relation with the season, the average temperature and the height of the tree.

Below, we rewrite the regression with a linear system in order to find an analytical solution.

Linear system of n equations and matricial notation

Let rewrite the n linear equations, one per line as follows, via three different notations. The regression line is true for each observation couple (y_i, x_i), such that:

$y_1 = \beta_0 x_{10} + \beta_1 x_{11} + \cdots + \beta_p x_{1p} + \varepsilon_1$	$y_1 = \sum_{j=0}^{p} \beta_j x_{1j} + \varepsilon_1$	$y_1 = \beta^T x_1 + \varepsilon_1$
$y_2 = \beta_0 x_{20} + \beta_1 x_{21} + \cdots + \beta_p x_{2p} + \varepsilon_2$	$y_2 = \sum_{j=0}^{p} \beta_j x_{2j} + \varepsilon_2$	$y_2 = \beta^T x_2 + \varepsilon_2$
$\vdots$	$\vdots$	$\vdots$
$y_i = \beta_0 x_{i0} + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i$	$y_i = \sum_{j=0}^{p} \beta_j x_{ij} + \varepsilon_i$	$y_i = \beta^T x_i + \varepsilon_i$
$\vdots$	$\vdots$	$\vdots$
$y_n = \beta_0 x_{n0} + \beta_1 x_{n1} + \cdots + \beta_p x_{np} + \varepsilon_n$	$y_n = \sum_{j=0}^{p} \beta_j x_{nj} + \varepsilon_n$	$y_n = \beta^T x_n + \varepsilon_n$

In the system above, it is recognized a column vector to the left for the y_i, a column vector to the right for the noises ε_i, and a scalar product between the vectors x_i and β. A more condensed expression of the system with the sign sum is given at the second column. In a vectorial notation, this linear system is written at the third column. One must notice that the new observed variable x_{i0} was introduced for notational reason, its value is one.

It is easily recognized a product of a matrix with a vector just after the sign equal, which allows a matricial notation as explained next paragraph. The notation for the different vectors and matrices is below, followed by the more condensed notation with a matrix for the independent variables instead of vectors.

Let define the matrix X as follows. The matrix $X \in \mathbb{R}^{n\times(p+1)}$ is called the design matrix. It allows to have a matricial expression of the linear system involving the regression coefficients:

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & \dots & x_{1p} \\ 1 & x_{21} & x_{22} & \dots & \dots & x_{2p} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n1} & x_{n2} & \dots & \dots & x_{np} \end{bmatrix} = \begin{bmatrix} 1 & x_{1;1:p}^T \\ 1 & x_{2;1:p}^T \\ \dots & \\ 1 & x_{n;1:p}^T \end{bmatrix}, \; y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} \text{,and } \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \dots \\ \varepsilon_n \end{bmatrix}$$

Here, $x_{1;1:p}$ is when keeping the p last components of x_i without the first equal to 1, this is the more usual notation for x_i, not preferred here for the matricial computation, otherwise β_0 is separated, leading to more lenghty analytical expressions. Another approach is a separated name, $\tilde{x_i}$ when adding the first constant component, which is not considered here.

After the design matrix, three column vectors are defined, one for the target variable, one for the regression coefficients and one for the noise.

Here $x_i = (x_{i0}, x_{i1}, x_{i2}, \dots, x_{ip})^T \in \mathbb{R}^{p\times 1}$, $y \in \mathbb{R}^{n\times 1}$, $\beta = (\beta_0, \beta_1, \beta_2, \dots, \beta_p)^T \in \mathbb{R}^{(p+1)\times 1}$ and $\varepsilon \in \mathbb{R}^{n\times 1}$. For the three vectors, the dimensionality is obtained such as this leads to rewrite the n equations with an unique matricial equation:

$$y = X\beta + \varepsilon\,.$$

This means that when the data is available, from physical or biological measures for instance, the data are written with this matricial expression, with y and X, in order to find a solution for β. For the example just before, this leads to the following numerical matrix and vector:

```
X = np.hstack([np.ones((len(x),1)),x.reshape(len(x),1)])
y = y.reshape(len(x),1)
# print("X=\n",np.round(X,2))
# print("y=\n",np.round(y,2))
```

Here recall that $p = 1$. In summary, from the sample, it is available the information from the data y, which has a role different from the other variables available. The other variables x are aggregated in the matrix X which is called the design matrix. The noise is latent and included in the observation y_i, such as it is not dealt with in a first approach: with y_i given, and $\hat{y}_i$ estimated, this is just $\varepsilon_i = y_i - \hat{y}_i = y_i - \beta^T x_i$. Note that it is estimated with $\hat{\varepsilon}_i = y_i - \hat{\beta}^T x_i$ when $\hat{\beta}$ denotes the vector of regression coefficients found according to the available sample. The noises are the quantities minimized with a square as it it recognized:

$$\ell(\beta) \quad = \quad \tfrac{1}{n}\textstyle\sum_i \varepsilon_i^2(\beta, X)\,.$$

This is the gap between the observations y_i and their linear approximations $\hat{y}_i$, such that the regression line must minimize these gaps as a solution.

The linear regression model has for purpose to predict or to explain y in terms of X via a linear relation. Other forms of relation may be polynomial for instance, or even more non linear for a function of X involving another analytical approach than this chapter. The linear relation between X and y is relevant when it is acceptable, thus it will be consider how to check such relation. The linear hypothesis leads to the fitting of the linear regression model and to find regression parameters which are estimation of $\beta = (\beta_0, \ldots, \beta_p)^T$ from the available sample.

The linear regression provides a mathematical expression of the relation of y_i as a function of the p variables x_{ij} and the parameters β_j. This leads to explain the variable y_i from the predictors x_{ij} by the sign of the coefficients β_j for a positive or negative influence from the independent variable to the dependent variable. This leads also to predict the variable y_i^{new} from new predictors x_{ij}^{new} by using the linear equation and ignoring the noise such that $\hat{y}_i^{\text{new}}$ is a guess for the true target variable y_i^{new} which is unknown and not available.

This shows the difference between explaination and prediction. In the first one which "explains", the noises may be large as only the linearity and the shape of the noise are checked hence a large variance σ is possible and frequent. In the second one which "predicts", the noises and so the variance must be small such that the values $\hat{y}_i$ found by the model are expected to be near the true values y_i. This is why in clinical research for instance, the analysts may focus on small sets of independent variables which allow to explain and not to predict. This generates a large amount of concurrent results which are compared afterwards with the idea that if the bias are different and the conclusions are identical, an effect is detected. The predictive models may required several tens of variables which are costly to gather each time. They may be difficult to improve because this supposes to look for new relevant variables at the beginning of the study.

After this introduction to the notations and the purpose with practical illustrations, the expressions for the solution are found next.

Optimization for β - mse to be minimized

For the fitting of the model, say find a value for β, it is required to write a criterion and optimize it, the criterion will be a function of β in order to proceed at the optimization (minimization or maximization). The solution is written as a function of the sample, because only the sample is available.

A relevant criterion is one able to minimize the error between the true y_i and the predicted $\hat{y}_i$ from the model. This is available from the ε_i, and for mathematical ease but also in order to remove the sign of ε_i, the square of the noise is usually minimized. This allows nice closed-form solutions with nice statistical properties even if this can be improved as will be seen later.

The loss, also called cost function because it is minimized, is thus written as follows:

$$\begin{aligned}
\ell(\beta) &= \frac{1}{n}\sum_{i=1}^{n}\varepsilon_i^2(\beta,X) \\
&= \frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2 \\
&= \frac{1}{n}\left\{(y-\hat{y})^T(y-\hat{y})\right\} \\
&= \frac{1}{n}\left\{(y-X\beta)^T(y-X\beta)\right\} \\
&= \frac{1}{n}\left\{y^Ty-2y^TX\beta+\beta^TX^TX\beta\right\}.
\end{aligned}$$

The vectorial notation in the criterion at the third line comes from just the norms of the vector $y-\hat{y}$ which is the usual trick here. The fourth line comes from rewriting $\hat{y}$ with its expression as a function of the design matrix and the vector of the regression coefficients.

Optimization for β - analytical solution

Having a cost function, our purpose is now to minimize this loss w.r.t. the parameter vector β. For the estimation of β which leads to a solution $\hat{\beta}$ which depends on the sample, we just minimize the cost function via the optimization problem:

$$\hat{\beta} = \text{argmin}_{\beta\in\mathbb{R}^{p+1}}\ \ell(\beta).$$

The minimization is performed via the derivative of the cost function with respect to β which is equal to zero at the optimum. This gives a mathematical expression as follows in a vectorial approach:

$$\begin{aligned}
\frac{\partial}{\partial\beta}[\ell(\beta)] &= \frac{2}{n}\left\{-2X^Ty+2X^TX\beta\right\} \\
&= \frac{-2}{n}X^T(y-X\beta) \\
&= 0.
\end{aligned}$$

This leads to

$$\begin{aligned} X^T(y - X\beta) &= 0 \\ X^T y &= X^T X\beta \\ \hat{\beta} &= \left(X^T X\right)^{-1} X^T y. \end{aligned}$$

Here the expression given is available only if the matrix $X^T X$ is invertible, otherwise a way around is required, such as making the matrix inverting by inflating the diagonal or prefering a pseudo-inverse, or using a numerical algorithm for the optimization which does not involve the matrix inversion. Note that nowadays for large matrices, such algorithms are more and more wanted and used in practice.

One may also notice that the optimization may be performed component-wise which avoids any knowledge on matrix derivative. This translates in mathematical terms as follows: proceed at the computation of the derivatives component by component for each coefficient β_j with j in $0, 1, \ldots, p+1$. More precisely one may write the expression for each $\frac{\partial}{\partial \beta_j}[\ell(\beta)]$, then one may rewrite the expression from the $p+1$ univariate derivatives via vectors and a matrix: the same matricial solution is obtained as with the direct matricial derivative. Some algorithms proceed also at an optimization component by component or by set of components instead of all the components once, both approaches are out of the scope herein.

As the design matrix is defined with $X \in \mathbb{R}^{n \times p+1}$, its product with the transposed version, $X^T X \in \mathbb{R}^{p+1 \times p+1}$, is a squared matrix of side $p+1$. If for a small example, the matrix may be invertible, for larger values of $p+1$ this becomes quickly intractable because the squared matrix is dense and ask for too much computer memory or because there are just too much useless variables that one needs to remove. The solution above for the regression is not really able to lead to a solution relevant in this case, and other algorithms are required. Improved algorithms and improved regression models are thus available in the literature and python modules such as online training and variables selection. This kind of numerical variants are introduced in the next chapters.

Optimization for β - checking that the cost function is minimized

When a solution from the optimization is obtained, it is usual to check if the optimum is a minimum or a maximum, this is possible by looking at the sign of the eigen values of the second-order derivative. For this regression model, one gets that the second-order derivative of the criterion is:

$$H = \frac{\partial^2 [\ell(\beta)]}{\partial \beta^T \partial \beta} = \frac{2}{n} X^T X.$$

This second-order derivative is usually called the Hessian matrix and denote H. As a product of a matrix with its transpose, it is always positive, hence the optimization is a minimization and the problem is said convex, with X supposed of full rank here. The Hessian matrix and the gradient

which is the first derivative are involved in an approximation of the criterion around the optimal solution, such as:

$$\ell(\beta) = \ell(\hat{\beta}) + (\beta - \hat{\beta})^T \frac{\partial [\ell(\beta)]}{\partial \beta} + \frac{1}{2}(\beta - \hat{\beta})^T \frac{\partial^2 [\ell(\beta)]}{\partial \beta^T \partial \beta}(\beta - \hat{\beta}) + R(\beta)$$

This is the same approximation which is available from Taylor series for univariate functions. Here for a quadratic problem we must have the remaining term $R(\beta)$ equal to zero while the expression is exact. A consequence is that if it exists, there is only one unique solution, which insures the convergence of the algorithms with few conditions. For nonlinear functions, the remaining term is not zero in general and there exist local solutions which are not global (the best), and make the optimization more tricky: today for neural networks, solutions and workarounds have been recently proposed hence this problem is partially solved for some architectures.

Optimization for σ

Let see when we solve for an exact solution,

$$\frac{\partial}{\partial \sigma}\left[\tilde{\ell}(\hat{\beta}, \sigma)\right] = 0$$

Hence, it comes:

$$\begin{aligned}
\frac{\partial}{\partial \sigma}\left[\frac{n}{2}\log 2\pi\sigma^2 + \frac{||(y - X\hat{\beta})||_2^2}{2\sigma^2}\right] &= 0 \\
\frac{\partial}{\partial \sigma}\left[n\log\sigma + \frac{||(y - X\hat{\beta})||_2^2}{2\sigma^2}\right] &= 0 \\
\frac{\partial}{\partial \sigma}\left[n\frac{1}{\sigma} - \frac{||(y - X\hat{\beta})||_2^2}{\sigma^3}\right] &= 0 \\
\hat{\sigma}^2 &= \frac{1}{n}||(y - X\hat{\beta})||_2^2
\end{aligned}$$

Note that this estimation is biased because the expectation is equal to $\frac{n-p}{n}\sigma$, a result which is found in any statistical book for the linear model, such that the unbiased estimation may be preferred for a more advanced concern. This also induces that minimizing the loss minimizes also this expression for the variance term, while the loglikelihood is maximized.

2.1.4 How to check the quality of the linear regression

The residuals are orthogonal to the vectorial space span from the columns of X because the solution is able only to model this space. The remaining space is kept in the noise which is in a pythagorian

triangle. In order to check if the fitting of the linear model is successful diverse methodologies have been proposed in the statistical literature by computing different indicators which tell us how the fit is relevant. One of the most used method is to check the residuals $y_i - \hat{y}_i$ which might be Gaussian distributed, centered at zero and without any structure hidding some correlation among the data. Thus, one can have a look at the (studentized) residuals with a plot, at their distribution with an histogram or at the comparison of their distribution with the Gaussian one with a qq-plot. These plots and the related statistical tests for the linear regression are not considered herein, but are wise complements to the usual numerical indicators. Instead, a plot of y_i against $\hat{y}_i$ will illustrate variables selection in the dedicated chapter for the regression with a continuous target variable while a matrix aggregating the counts for comparing the values of y_i against $\hat{y}_i$ for a discrete target variable is presented at the end of the current chapter with an example of classification.

For the regressions with a continuous outcome, numerical indicators are as follows.

Mean squared error (MSE)

This indicator is formally an empirical expectation over a sample. It is written as the mean of the square of the noise per datum as:

$$MSE(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 .$$

This sum is just the loss or objective function that was minimized in order to find the solution for the regression. When comparing models with different choices of variables, the better fit may be with the smaller MSE but actually one must also care about the error for new data. The MSE is never equal to zero because there is always some random noise with real data. The MSE is written as the following python function,

```
def MSE_score(y,yhat):
    return np.sum((y-yhat)**2)/len(yhat)
```

The mean absolute error (MAE) is related to the mean squared error by replacing the quadratic distance by an absolute distance, and not often given, on the contrary to the next indicator, without any doubt the preferred indicator in many domains despite its limits.

Coefficient of determination or R^2

This indicator is related to the prediction from new independent variables. The larger is the better with a maximum at 1.0, a more negative value in the worser scenario of bad fitting, and a value at 0.0 for only an intercept β_0 without covariates. The score R^2 is defined as

$$R^2(y,\hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2},$$

where we have defined the mean value of y as

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

It is also known that this statistics is unable to check the linearity of the data, and supposes that the hypothesis of linearity is true, hence may provide erroneous high value for real function which are nonlinear: the residuals should always be checked too. The R^2 is written with a python function,

```
def R2_score(y, yhat):
    return 1 - np.sum((y - yhat) ** 2) / np.sum((y - np.mean(y)) ** 2)
```

Note that these indicators are available with sklearn for some methods such as LinearRegression in sklearn.linear_model, from the sub module metrics, with the names "mean_squared_error", "r2_score", "mean_absolute_error" while "mean_squared_log_error" is an alternative one.

2.1.5 Example of simple linear regression (continued)

In this paragraph, we explain how to compute the regression coefficients and the different scores directly from the formula without using any python module except for basic algebra and then compare with two modules available in python and able to solve the regression problem via an implemented function. The values for the design matrix X and the vector of dependent variables y have been given before in this chapter.

Implementation from python with the algebra

The direct implementation involves a matrix inversion and its multiplication with y in order to get the regression coefficients. The python operator "@" is for a matricial multiplication in its usual definition by summing along the columns for the element coming at the left and along the rows for the element coming at the right. The function ".dot()" allows an equivalent matrix multiplication. The function "linalg.inv()" from the module numpy is for the inversion of a squared matrix.

The solution for the regression is written as follows with python:

```
betahat_np = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
print("betahat_np=")
print(betahat_np.reshape(2,1))
```

```
betahat_np=
[[-0.75708353]
 [ 3.66657469]]
```

For this small dataset, this result comes after a few steps as follows:

$$\hat{\beta} = \left[\begin{pmatrix} 1 & 0.35 \\ 1 & 0.45 \\ 1 & 0.02 \\ 1 & 0.18 \\ 1 & 0.23 \\ 1 & 0.93 \\ 1 & 0.53 \\ 1 & 0.48 \\ 1 & 0.73 \\ 1 & 0.04 \end{pmatrix}^T \begin{pmatrix} 1 & 0.35 \\ 1 & 0.45 \\ 1 & 0.02 \\ 1 & 0.18 \\ 1 & 0.23 \\ 1 & 0.93 \\ 1 & 0.53 \\ 1 & 0.48 \\ 1 & 0.73 \\ 1 & 0.04 \end{pmatrix} \right]^{-1} \begin{pmatrix} 1 & 0.35 \\ 1 & 0.45 \\ 1 & 0.02 \\ 1 & 0.18 \\ 1 & 0.23 \\ 1 & 0.93 \\ 1 & 0.53 \\ 1 & 0.48 \\ 1 & 0.73 \\ 1 & 0.04 \end{pmatrix}^T \begin{pmatrix} 0.07 \\ 0.94 \\ -1.06 \\ -0.21 \\ -0.01 \\ 2.26 \\ 1.55 \\ 1.50 \\ 2.09 \\ -0.23 \end{pmatrix}.$$

From the products $X^T X$ with the matrix X and $X^T y$ with the vector y, this leads to:

$$\begin{aligned} \hat{\beta} &= \begin{pmatrix} 10.000 & 3.950 \\ 3.950 & 2.334 \end{pmatrix}^{-1} \begin{pmatrix} 6.913 \\ 5.567 \end{pmatrix} \\ &= \begin{pmatrix} -0.757 \\ 3.666 \end{pmatrix}. \end{aligned}$$

There is a small difference with the previous direct numerical solution because of the rounding of the numbers in the matrix, which recalls that rounding numbers is suitable only for showing the results, while intermediate expression should be computed without approximation in order to avoid any propagation of the rounding error.

Implementation from numpy

In the module numpy, the regression is implemented with the function "linalg.lstsq()" such that,

```
betahat_np2 = np.linalg.lstsq(X, y, rcond =None)[0]
print("betahat_np2=")
print(betahat_np2.reshape(2,1))
```

```
betahat_np2=
[[-0.75708353]
 [ 3.66657469]]
```

Implementation from sklearn

In the module sklearn, the linear regression is implemented in the function linear_model.LinearRegression(r). We get the fitting, the regression coefficients from the fitting and the prediction for the small dataset as follows.

```
# ?LinearRegression
```

```
#import sklearn as sklearn
from sklearn.linear_model import LinearRegression
fit_skl = LinearRegression(fit_intercept=False).fit(X, y)
#yhat_skl = fit_skl.predict(X)
#betahat_skl = [ fit_skl.coef_, fit_skl.intercept_]
betahat_skl = fit_skl.coef_.reshape(2,1)

print("betahat_skl=")
print(betahat_skl)
```

```
betahat_skl=
[[-0.75708353]
 [ 3.66657469]]
```

Here, x_i has for first component 1, hence there is no need to fit the intercept, with the option "fit_intercept=True", otherwise by default it is fitted and return in the variable "intercept_", while the vector of coefficients is returned in the variable "coef_", which in the present case contents already the intercept.

Indicators with numpy

The R^2 and the MSE from the two functions defined above are equal to:

```
yhat = X @ betahat_skl
print(f" R^2 = {np.round(R2_score(y,yhat),2):2.2f} \
      \n MSE = {np.round(MSE_score(y,yhat),2):2.2f}")
```

```
 R^2 = 0.91
 MSE = 0.11
```

Solution for σ

The solution for the (biased) standard-deviation is computed as,

```
sigmahat = np.sqrt( np.sum((y-np.matmul(X,betahat_skl))**2) / len(y) )
print("sigmahat=",np.round(sigmahat,5))
```

```
sigmahat= 0.32934
```

Let plot for the small example, the cost function for σ before having the equation solve. The loglikelihood is computed for several values of the standard-deviation. Note that the design matrix

could be an entry for the function instead of a global python variable. Then we plot the cost error for the variance quantity.

```
import numpy as np
def f_loglik_gauss(y,beta,sigma):
    n = len(y)
    f = +n/2*np.log(2*np.pi*sigma**2) + \
         np.sum((y-np.matmul(X,beta))**2) /(2*sigma**2)
    return -f
sigma_min    = sigmahat*0.99 # 0.05
sigma_max    = sigmahat*1.01 # 0.09
sigma_grid   = np.mgrid[sigma_min:sigma_max:100j]
logLik_grid  = f_loglik_gauss(y,betahat_skl,sigma_grid)
logLik_min   = f_loglik_gauss(y,betahat_skl,sigmahat)
```

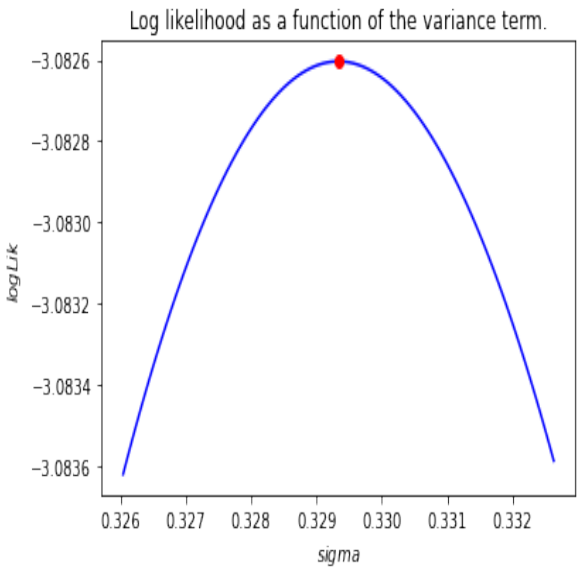

Figure 2.2: Loglikelihood and variance parameter

We see that there is a clear maximum, which can be checked with the second-order derivative, $\partial^2 \left[-\tilde{\ell}(\hat{\beta},\sigma)\right]/\partial^2\sigma$, which is positive. The estimation of the regression coefficients does not depend on this variance hence the later is generally not involved for neural networks.

The functions for drawing scatterplots and curves in the graphical output above are just below.

```
def f_draw(x,y,markercolor,xlabel,ylabel,title,ax):
    ax.plot(x,y, markercolor)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_title(title)
```

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
f_draw(sigma_grid,logLik_grid,"b-",r'$sigma$',r'$logLik$',
       str(r'Log likelihood as a function of the variance term.'),ax)
ax.plot(sigmahat, logLik_min, "ro")
plt.show()
```

Most of the functions defined in the chapters are written in a python file in order to be available after. Note that the functions in the file "deepglmlib.utils.py" does not need to refer "utils" because the functions are already defined locally, thus, this is removed in every functions in the file.

Indicators with sklearn

For the regression, the indicators are computed in a function with optional printing of the indicators and drawing of the scatterplot between true and predicted targets.

```
from sklearn.metrics import mean_squared_error,r2_score
from sklearn import metrics
def nprd(x,ndec=3): return np.round(x,ndec)

def f_metrics_regression(y,yhat,printed=False,drawn=False,
                         ax=None,ndec=2,samplename="sample"):
#     np.set_printoptions(precision=3)
    np.set_printoptions(suppress=True)

    mse_yyhat = mean_squared_error(y.ravel(), yhat.ravel())
    r2_yyhat  = r2_score(y.ravel(), yhat.ravel())
    #cor_yyhat = np.corrcoef(y.ravel(),yhat.ravel())[0,1]

    if printed:
        print("MSE = ", nprd(mse_yyhat,ndec), "\n"
              "R2  = ", nprd(r2_yyhat,ndec),)
              #"COR_train : ", "{:.3f}".format(cor_yyhat))

    if drawn==True:
        f_draw(y,yhat,"b.","y (true target)","y_hat (predicted target)",
               str("Scatterplot for regression from "+samplename), ax)

    return mse_yyhat,r2_yyhat  #,cor_hat
```

It it recognized the mean square error (for linear and nonlinear regression) and the coefficient of determination (for linear regression).

```
mse_yyhat,r2_yyhat = f_metrics_regression(y,yhat,True,False,None,
                                          ndec=3,samplename="sample")
```

```
MSE =  0.108
R2  =  0.906
```

2.1.6 Remarks

A main purpose of the next chapters is to consider nonlinear models by changing the linear processing of the explaining/predicting variables from a sum into some nonlinear function, i.e. $\beta^T x_i$ is changed into a more general function $g(x_i, \beta)$ where $g(.,.)$ also depends on linear transformations associated to non linear ones which follow each other. The target variables y_i shall also be eventually not continuous, such as binary or integer, associated to different loss functions. The training of such deep models will be handled via pytorch where $g(.,.)$ comes from a particular expression explained in the next chapter after: one gets powerful and accurate regression and classification models when the dataset is enough large. If for usual expressions of $g(.)$, python modules for machine learning such as sklearn are able to process the related model fitting, when the dataset increases and the model evolves, more flexible approaches are required such as offered with the module pytorch for running the optimization.

This concludes the introduction to the linear regression. The reader may have noticed that the linear classification is also a regression with a different distribution for the outcome, in the framework called generalized linear models. Next, instead of a continuous variable for the outcome, this is a binary one which is involved.

2.2 Logistic Regression (2 classes)

In this chapter, a classification model called logistic regression for a binary dependent variable y_i is presented with its analytical formulation and its implementation available from a python module. The case of more than two classes is discussed at the end of the chapter.

With binary values, the previous Gaussian distribution in the linear regression is changed into the Bernoulli one. This is an usual approach in research, when one changes something and observes what happens. In an experiment, this would be a condition which is altered. For instance, in the medical world, there would be "new medecine" or "no medecine" for ill people, and the analyst would count after how many people (from a sample) survived to each hypothesis (with or without medecine) in order to decide if the drug can be given to more people when it performs better than a previous one. In statistical modeling this may be the data distribution as here which is changed in order to check if the new model is more suitable for the prediction, otherwise the previous model is kept, similarly.

2.2.1 Probabilistic interpretation

Let present the new modeling involved in a logistic regression. The dependent variables y_i is binary,

$$y_i = \begin{cases} 0 & \text{for the class of failure} \\ 1 & \text{for the class of success} \end{cases} .$$

As before y is a vector aggregating all the observations for the outcomes, X is the $n \times p$ design matrix while β is the $p \times 1$ vector of unknown parameters or regression coefficients.

The usual linear regression predicts values on $\mathbb{R}$ while we are interested on predicted values in the set $\{0,1\}$. We have seen that the linear regression involves a Gaussian distribution, then changing the distribution into a Bernoulli one makes sense but its parameter needs a bounded probability.

Recall that the parameter of the Bernoulli distribution is the probability of success, say the probability that the observation is equal to one. For this reason, the logistic regression is based on this distribution but instead of just implementing the inner product $x_i^T \beta$ directly as the parameter, it supposes a nonlinear transformation of this inner product:

$$y_i \sim \mathcal{B}(\sigma(x_i^T \beta)) \quad \text{is equivalent to} \quad p(y_i; x_i^T \beta) = \left(\sigma(x_i^T \beta)\right)^{y_i} \left(1 - \sigma(x_i^T \beta)\right)^{1-y_i} .$$

This model is the one we want because $\sigma(x_i^T \beta) \in \{0,1\}$ and it is defined for binary observations as expected. Note that the usual notation $\sigma()$ is just a coincidence, and not related to any variance quantity with the scalar σ from the Gaussian likelihood.

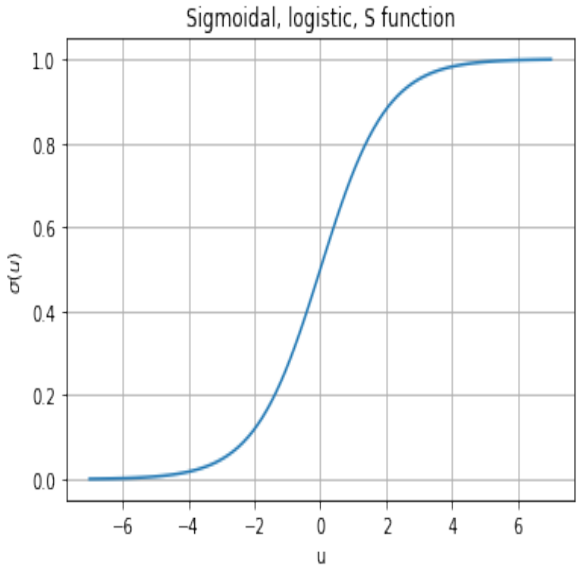

Figure 2.3: Sigmoid function $u \to \sigma(u)$ with "S shape"

Here, $\sigma()$ is an increasing smooth function with bounds in order to model probabilities.

Sigmoid function

The sigmoid function (or the logistic curve) is useful in neural networks for assigning weights on a relative scale. The value u is the weighted sum of parameters involved in the learning algorithm, with $1-p(u)=p(-u)$. This function takes any real number, u, and outputs a number in $]0;1[$.

$$\sigma(u)=\frac{\exp u}{1+\exp u}=\frac{1}{1+\exp -u}\,.$$

Note that the logistic model leads to a smooth version of a neural network called "perceptron", not studied herein, as it performs less well. There are some theoritical concerns (such as number of steps before convergence of the algorithm) with this simple network which are out of the scope.

Recall that it is assumed to have two classes with y_i either 0 or 1. Furthermore it is assumed also to have an unknown vector of parameters β in our fitting with the sigmoid function. With that, it is defined probabilities:

$$\begin{aligned} p(y_i=1|x_i,\beta) &= \frac{\exp(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2})}{1+\exp(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2})}=\sigma(x_i^T\beta), \\ p(y_i=0|x_i,\beta) &= \frac{1}{1+\exp(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2})}=1-\sigma(x_i^T\beta)\,, \end{aligned}$$

where β are the weights we want to guess from data, in our case β_0, β_1, and β_2, in the example below, for instance.

Note that, as $\{y_i=0\}$ and $\{y_i=1\}$ are complementary events,

$$p(y_i=0|x_i,\beta)+p(y_i=1|x_i,\beta)=1\,.$$

Likelihood function

It is defined the total likelihood for all possible outcomes from the available sample or dataset $\{(y_i,x_i),1\geq i\geq n\}$, with the binary labels $y_i\in\{0,1\}$ and where the data points are drawn independently.

$$\begin{aligned} \log p(y;X,\beta) &= \log\textstyle\prod_{i=1}^n p(y_i;x_i^T\beta,\sigma) \\ &= \textstyle\prod_{i=1}^n \left(p(y_i=1|x_i,\beta)\right)^{y_i}\left(1-p(y_i=1|x_i,\beta)\right)^{1-y_i} \end{aligned}$$

The purpose is thus to maximize this probability: the probability of the observed data for this model. The likelihood is approximated in terms of the product of the individual probabilities of each outcome y_i, because of the hypothesis of independence.

This allows to implement the Maximum Likelihood Estimation (MLE) principle where the optimization leads to solutions for the unknown vector of parameters. This solution has known properties, all exactly the same for any form of likelihood from Gaussian, Bernoulli, etc distributions. In neural network, this is a loss which is minimized, the opposit of the loglikelihood.

Loss function

After that, one applies the logarithm from the mass distribution of the i.i.d. variables y_i, such that:

$$\begin{aligned} \log p(y;X,\beta) &= \log \prod_{i=1}^{n} p(y_i;x_i^T\beta) \\ &= \log \prod_{i=1}^{n} \left(\sigma(x_i^T\beta)\right)^{y_i} \left(1-\sigma(x_i^T\beta)\right)^{1-y_i} . \end{aligned}$$

Since the events for the observations were assumed to be independent and identically distributed the probability mass function for all possible event y is a product of the single events, such that after the logarithm, the loglikelihood is obtained:

$$\begin{aligned} \ell(\beta) &= \log p(y;X\beta) \\ &= \sum_{i=1}^{n} \left(y_i \log p(y_i=1|x_i,\beta) + (1-y_i)\log\left[1-p(y_i=1|x_i,\beta)\right]\right) . \end{aligned}$$

Reordering the logarithms, one can rewrite the sum as, with an example for two independent variables,

$$\begin{aligned} \ell(\beta) &= \sum_{i=1}^{n} \left(y_i(\beta^T x_i) - \log\left(1+\exp\left(\beta^T x_i\right)\right)\right) \\ &= \sum_{i=1}^{n} \left(y_i(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2}) - \log\left(1+\exp\left(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2}\right)\right)\right) . \end{aligned}$$

The maximum likelihood estimator is defined as follows: this is the set of parameters that maximize the loglikelihood, say maximize $\ell(\beta)$ with respect to β.

In neural networks, since the cost (error) function is just the negative loglikelihood, for logistic regression the sign would be changed for a minimization (of a loss). The correponding loss is named the "cross-entropy".

2.2.2 Derivatives

The cross-entropy is a convex function of the weights β and, therefore, any local minimizer is a global minimizer.

Minimizing this cost function with respect to the two parameters β_0, β_1 and β_2 we obtain,

$$\begin{aligned} \frac{\partial \ell(\beta)}{\partial \beta_0} &= -\sum_{i=1}^{n} \left(y_i - \frac{\exp(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2})}{1+\exp(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2})}\right) \\ \frac{\partial \ell(\beta)}{\partial \beta_1} &= -\sum_{i=1}^{n} \left(y_i x_{i1} - x_{i1}\frac{\exp(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2})}{1+\exp(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2})}\right) \\ \frac{\partial \ell(\beta)}{\partial \beta_2} &= -\sum_{i=1}^{n} \left(y_i x_{i2} - x_{i2}\frac{\exp(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2})}{1+\exp(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2})}\right) . \end{aligned}$$

Let us now define a vector y with n elements y_i, an $n \times p$ matrix X which contains the x_i values and a vector p of fitted probabilities $p(y_i|x_i,\beta)$. We can rewrite in a more compact form the first derivative of the cost function as,

$$\frac{\partial \ell(\beta)}{\partial \beta} = -X^T (y-p) \text{ and } \frac{\partial^2 \ell(\beta)}{\partial \beta \partial \beta^T} = X^T \Omega X .$$

Here it is denoted a diagonal matrix Ω with elements $\omega_i = p(y_i|x_i,\beta)(1-p(y_i|x_i,\beta)$, which leads to a compact expression of the second derivative too.

2.2.3 Predicted labels $\hat{y}_i$

Let see how translate the modeling with the sigmoidal function in order to have the new predicted values from β. This is now for a threshold s in $]0,1[$ such that we get that:

$$\hat{y}_i = \begin{cases} 0 & \text{if } \sigma(x_i^T \hat{\beta}) > s \\ 1 & \text{if } \sigma(x_i^T \hat{\beta}) \leq s \end{cases} .$$

The difference with the linear regression is that the output from the regression was directly used for $\hat{y}_i$ while for the logistic regression this is a probability which needs to mutate into a binary value. The choice for s is according to diverse criteria in order to insure errors balanced or unbalanced for the two possible values of outcomes. Here, an error happens when $y_i \neq \hat{y}_i$ as explained next.

2.2.4 How to check the quality

In order to check if the fitting of the linear model is successful the same methodology than for the regression is not anymore relevant because the target variable is discrete. Hence the indicators which tell us how the classification is relevant are different. Note that residuals are defined for the logistic regression but rarely considered in practice. Actually, the mse from the regression is retrieved nextafter but the theory is different here. The linear regression can be relevant and not exact for all the data while the linear classification can only be relevant if it is exact for enough data. The main indicators are as follows.

Classification errors - how $\hat{y}_i$ compares with y_i

Because the problem is a classification and the variable y is not continuous, the purpose is to retrieve exactly the same class for each datum. Indeed, $y_i \in \{0,1\}$ and $\hat{y}_i \in \{0,1\}$, then the errors happen whenever y_i is not equal to $\hat{y}_i$. The four different cases are summarized just below:

- $y_i = 1$ and $\hat{y}_i = 1$ is True Positive or TP
- $y_i = 0$ and $\hat{y}_i = 0$ is True Negative or TN
- $y_i = 1$ and $\hat{y}_i = 0$ is False Negative or FN
- $y_i = 0$ and $\hat{y}_i = 1$ is False Positive or FP

□

Confusion matrix

A matrix allows a visual representation of the four cases listed above. They are summarized with the cells of the "confusion matrix" which shows the counts for the correspondence or non correspondence between the true y_i and the predicted $\hat{y}_i$. The matrix is thus defined as follows:

$$\begin{array}{cc} & \hat{y}=1 \quad \hat{y}=0 \\ y=1 & \#\text{TP} \quad \#\text{FN} \\ y=0 & \#\text{FP} \quad \#\text{TN} \end{array}$$

The cells with #TP, #FN, #FP and #TN denote the sizes of the set by couting the number of TP, FN, FP and TN in the sample when comparing the true target y_i with the predicted target $\hat{y}_i$. We are happy if the diagonal of the matrix is large, and the non diagonal is small, but other alternative indicators exist (and also for unbalanced classification not considered here). The value of the indicators depends on the cut with s for choosing $\hat{y}_i$, this explains why the cut may be decided for maximizing an indicator of interest, even if often the threshold 0.5 is selected by default.

□

Accuracy

This indicator is written as follows,

$$ACC(y,\hat{y}) = \frac{\#\text{TP}+\#\text{TN}}{\#\text{TP}+\#\text{FN}+\#\text{FP}+\#\text{TN}}.$$

The value of the indicator is related to the criterion or loss function that was minimized in order to find the solution for the logistic regression. This quantity is not directly optimized because it is not smooth on the contrary to the loss of the regression, and non smooth functions are difficult to optimize. The expression is exactly equal to the previous mean square error, because $(y_i-\hat{y}_i)^2$ is equal to zero (with $(1-1)^2=0$ and $(0-0)^2=0$) if there is no error and equal to one if there is an error (with $(0-1)^2=1$ and $(1-0)^2=1$), while the denominator is the size of the sample.

```
def ACC_score(y,yhat):
    return np.sum(y==yhat)/len(y)
```

□

Error rate

This indicator is on the contrary to the accuracy the percentage of missclassified units in the sample, such that:

$$ERR(y,\hat{y}) = 1-ACC(y,\hat{y}).$$

□

Precision and recall

The "precision" is the percentage from the number of true positive against the number of true positive and false positive, while the "recall" is the percentage from the number of true positive against the number of true positive and false negative

$$PREC(y,\hat{y}) = \frac{\#\text{TP}}{\#\text{TP+\#FP}}\,.$$

$$RECALL(y,\hat{y}) = \frac{\#\text{TP}}{\#\text{TP+\#FN}}\,.$$

This is only a measure for the the success, conditionally to the true target equal to one, which is mesured here, as a percentage from the count of predicted positive w.r.t. the count of true positive. This reflects the purpose to only retrieve the value 1 of the target variable, otherwise the true negative would be in the numerator.

□

ROC curve

This curve is a representation of the points (precision by recall) when the threshold changes within the interval $[0;1]$ in order to find the best threshold which maximizes both:

$$s \to (PREC_s(y,\hat{y}), RECALL_s(y,\hat{y}))\,.$$

This curve is not so much considered because the accuracy is more often a concern with a symmetric error, such that, one may plot eventually this curve as a complement, named here the "ACC curve",

$$s \to \left(\frac{\#\text{TP}_s}{\#\text{TP}_s + \#\text{FN}_s + \#\text{FP}_s + \#\text{TN}_s}, \frac{\#\text{TN}_s}{\#\text{TP}_s + \#\text{FN}_s + \#\text{FP}_s + \#\text{TN}_s} \right)\,.$$

Here it is denoted the index s when the counts come from a value s as the level of cut for finding the target variable $\hat{y}_i$ from the the continuous quantity $\sigma(x_i^T\hat{\beta})$, instead of just selecting $s = 0.5$ from the usual and not optimal first choice, see "sklearn.metrics.roc_curve" for instance.

□

AUC

The quantity from the "Area Under the Curve" ROC or "AUC" allows to judge the quality of a classification. With 0.5 the random choice for the predicted labels, a larger value towards 1 leads to the best model. A visual representation associated to this area or just the value of area is a complement to the other indicators. Note that in image classification, a wide domain of research in neural networks, authors focus on the accuracy most of the time, while this area appears more frequently in machine learning for tabular data for instance.

□

F1 score

The F1 score is a particular case of F-measure (with parameter equal to 1), an harmonic mean:

$$F_1(y,\hat{y}) = \frac{2\#\text{TP}}{2\#\text{TP}+\#\text{FP}+\#\text{FN}} = \frac{2}{\frac{1}{PREC(y,\hat{y})}+\frac{1}{RECALL(y,\hat{y})}}.$$

□

MCC

The "Matthews Correlation Coefficient" is related to the Pearson correlation for a contingency matrix, such as a phi coefficient or mean square contingency coefficient,

$$MCC(y,\hat{y}) = \frac{\#\text{TP}\ \#\text{TN} - \#\text{FP}\ \#\text{FN}}{(\#\text{TP}+\#\text{FP})(\#\text{TP}+\#\text{FN}) + (\#\text{TN}+\#\text{FP})(\#\text{TN}+\#\text{FN})}.$$

□

All these indicators for two classes are generalized for a muli-classes model, with the confusion matrix which required the K classes instead of just two at the level of the rows and columns, while the other indicators are also extended by counting the errors per class and summing.

Herein, the implementation of the indicators comes from the module sklearn. These indicators are available from the sub module metrics, with the names "accuracy_score", "precision_score" and "recall_score" and "confusion_matrix", with the first argument for the true target variable and the second one for the predicted one.

2.2.5 Example of bivariate classes

For an artificial dataset, the Gaussian distribution from the beginning of the chapter is replaced with a Bernoulli one, let generate a sample and have a look to the obtained data. Actually, if we have two classes with the covariates x_i, this means that these variables themselves have a partitioning. Hence, the independent variables are generated in order to come from a mixture of distributions with two modes, say two different groups more or less separated. Without such underlying classes, there is no reason that y_i codes two classes. Because of the supposed dependence, this must be caused by the independent variables.

```
def f_sigmoid(a):
    return np.exp(a)/(1+np.exp(a))
```

```
# import numpy as np

# n1 = n2 = 50
# n  = 2 * n1
```

```
# beta = np.array([-0.5,3.5,2.0]).reshape((3,1))

# x = np.vstack([ np.random.normal(1,1,n).reshape((n1,2)) ,
#                 np.random.normal(-1,1,n).reshape((n2,2)) ])

# p             = f_sigmoid( np.hstack([ np.ones((n,1)), x]) @ beta )
# y             = np.random.binomial(1,p)
```

Let save the dataset with the python module numpy in text format.

```
# np.savetxt("./xy_2d_reglogistic.txt",np.hstack([x,y]))
# np.savetxt("./beta_2d_reglogistic.txt",beta)
```

```
import numpy as np
xy   = np.loadtxt(towdir("./xy_2d_reglogistic.txt",))
beta = np.loadtxt(towdir("./beta_2d_reglogistic.txt"))

x    = xy[:,[0,1]]
y    = xy[:,2]
n    = len(y)

print(xy.shape,x.shape,y.shape)
```

```
(100, 3) (100, 2) (100,)
```

From the data sample, the generation process in unknown, and only the observed x and observed y from the sampling process are available. To be clear, the "true class labels" will be the "observed class labels" herein and available in the dataset, not the class labels of the independent variables x which are unknown, not modeled, and out of the scope for our analysis.

```
n0 = np.sum(y==0)
n1 = np.sum(y==1)
n0,n1
```

```
(51, 49)
```

The format for y as a list with only one dimension is often preferred in some python modules, otherwise a not related exception may be received without further explaination, such that checking if it is a (python) vector or a list-like format is a good way to do, before all, in order to avoid tricky computer messages from python.

```
y = y.ravel()
```

Let plot the dataset with the known labels.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot(x[y.ravel()==0,0], x[y.ravel()==0,1], 'bx',
         x[y.ravel()==1,0], x[y.ravel()==1,1], 'bo')
ax.set_xlabel(r'$x1$')
ax.set_ylabel(r'$x2$')
ax.set_title(r'Sample points from two classes')

plt.show()
```

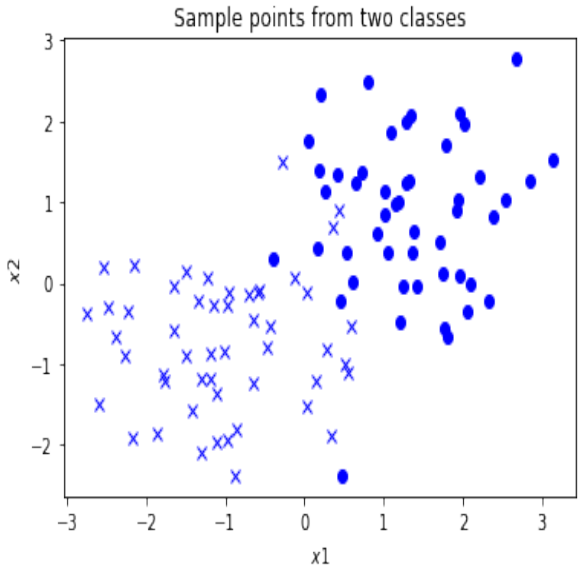

Figure 2.4: Data sample for two classes with a linear frontier

From the graphic, we get two classes almost separated as expected because the independent variables are generated separately for each class with a distance between the means larger than the standard-deviations. The logistic regression will be able to find the frontier between the two classes with again a line as in linear regression but in a different way. Actually, this is the reason why these two models are called "linear models". Note that there is not a closed-form solution on the contrary to the linear regression, hence, an iterative algorithm is required whatever the size of the dataset. This is common to many methods in statistics, machine learning and even neural network which will be studied in a next chapter. Note that the algorithm will ask for a computation of the first-order derivative of the cost function. While the second-order will be not required for all the numerical algorithms. If this is the case in this chapter by default for the used implementations, second-order ones are less used in deep learning because of the size of the networks and the large number of weights or unknown parameters to find in order to fit the model to the sample.

```
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(solver='lbfgs')
logreg = logreg.fit(x, y.ravel())
```

Here it has been chosen the algorithm called 'lbfgs' by sklearn, but other algorithms are available such as 'newton' for instance, they are both second-order methods, and thus better to use for small datasets.

The parameters and the accuracy after training follow.

```
betahat_rg=np.append(logreg.intercept_,logreg.coef_)
betahat_rg.reshape((x.shape[1]+1,1))
```

```
array([[-0.64925705],
       [ 2.27302541],
       [ 0.99991908]])
```

```
yhat = logreg.predict(x)
print("acc=",np.sum(y==yhat)/len(y))
```

```
acc= 0.94
```

An hyperplane is the frontier between the two classes such as $p(\hat{\beta}^T x_{\text{fr}}) = 0.5$, thus with two dimensions where $p = 2$ this leads to the equation:

$$\begin{aligned} \hat{\beta}_0 + x_{\text{fr1}}\hat{\beta}_1 + x_{\text{fr2}}\hat{\beta}_2 &= \ln\left(\tfrac{0.5}{1-0.5}\right) \\ &= 0.0 \end{aligned}$$

This line is added to the graphic with the two classes as colored points in order to illustrate the resulting classification. Such function is presented for drawing several lines useful next chapter while only one solution from sklearn appears below.

```
def f_vizu2d_beta(ax,x1_yeql1,x1_yeql2,x2_yeql1,x2_yeql2,
                  beta_list, marker_list, xlim=None, ylim=None):
    x1 = np.append(x1_yeql1,x1_yeql2)
    x2 = np.append(x2_yeql1,x2_yeql2)
    x1_min=min(x1)
    x1_max=max(x1)

    ax.plot(x1_yeql1, x2_yeql1, 'bx', x1_yeql2, x2_yeql2, 'bo')

    for beta, marker in iter(zip(beta_list,marker_list)):
```

```
        x1_ = sigma_grid   = np.mgrid[x1_min:x1_max:1000j]
        x2_ = -(beta[0]+x1_*beta[1])/beta[2]
        plt.plot(x1_,x2_,marker)

    if xlim is not None: ax.set_xlim(xlim)
    if ylim is not None: ax.set_ylim(ylim)

    ax.set(xlabel=r'$x1$', ylabel=r'$x2$')
    ax.set_title(r'Sample points and separation(s) line(s)')
```

```
import matplotlib.pyplot as plt

fig, (ax1) = plt.subplots(1, 1, figsize=(8,5))

f_vizu2d_beta(ax1,x[y.ravel()==0,0],x[y.ravel()==1,0],
              x[y.ravel()==0,1],x[y.ravel()==1,1],[betahat_rg],['c-'],
              xlim=[min(x[:,0]),max(x[:,1])],ylim=[min(x[:,0]),max(x[:
 ↪,1])])
```

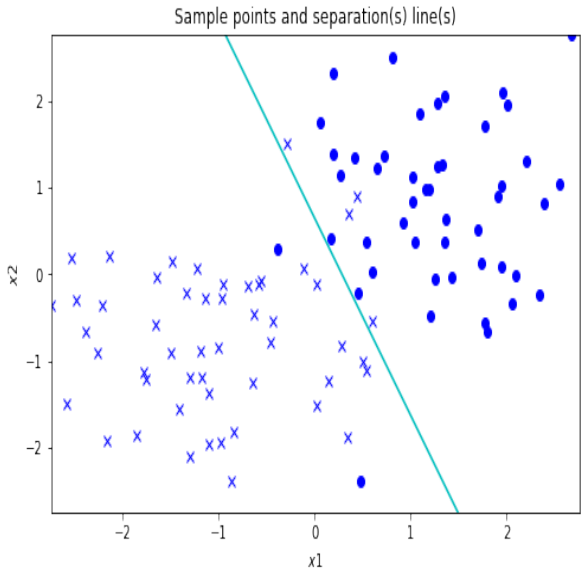

Figure 2.5: True frontier between the two classes

The number of errors is recognized visually (when some points does not overlap) by counting the points from one side and the other side of the line, which does not belong to the corresponding class from each side. This also underlines that for a nonlinear frontier, a line is not enough, which explains why nonlinear models are required such as with neural networks. Some errors are removable by improving the frontier between the two classes.

2.2.6 Indicators with sklearn

For the classification, the indicators are computed in a function with optional printing of the indicators and drawing of the scatterplot (not implemented here) between true and predicted outcome.

```
from sklearn.metrics import accuracy_score,precision_score,
                            recall_score,confusion_matrix
from sklearn import metrics

def f_metrics_classification(y,yhat,printed=False,drawn=False,
                             ax=None,ndec=3,samplename="sample"):
    cm_yyhat = metrics.confusion_matrix(y, yhat)
    acc_yyhat = metrics.accuracy_score(y, yhat)
    prc_yyhat = metrics.precision_score(y, yhat)
    rcc_yyhat = metrics.recall_score(y, yhat)
    if printed:
        print("Confusion matrix")
        print(cm_yyhat)
        print("Accuracy  = ", nprd(acc_yyhat,ndec))
        print("Precision = ", nprd(prc_yyhat,ndec))
        print("Recall    = ", nprd(rcc_yyhat,ndec))
    if drawn:
        pass

    return acc_yyhat, prc_yyhat, rcc_yyhat, cm_yyhat
```

It it recognized the accuracy, precision and recall, plus the confusion matrix, which are described previously in the chapter. For the bivariate example just before, the resulting error is as follows.

```
acc_yyhat, prc_yyhat, rcc_yyhat, cm_yyhat = f_metrics_classification(y,
                          yhat,True,False,None,ndec=3,samplename="sample")
```

```
Confusion matrix
[[47  4]
 [ 2 47]]
Accuracy  =  0.94
Precision =  0.922
Recall    =  0.959
```

Here, we are mainly interested on the accuracy which may be also computed during the training if a loop is implemented. As expected from the visual inspection of the classes, the classification error is small, 0.06, because the classes are linearly separable.

2.3 Multinomial regression (3 or more classes)

When the classification is for more than two classes, in this case, either a rule is set from binary classifiers, either the model is updated for multi-classes. The first way asks for combinations of binary classifiers, while the second way asks for a new distribution for the classes. Following the modeling approach of the chapter, this results to:

$$y_i \in \{1, 2, \cdots, K\}.$$

Here the number of classes may be equal to 3 or more. The distribution is not anymore Gaussian or bernoullian, this is a multinomial law which allows a label with more than two values.

The sigmoid function is replaced with a softmax function for the probabilities which generalized the sum to one for K probabilities instead of just 2 probabilities, with the expression:

$$\begin{aligned} p(y_i = k|x_i, \beta_k) &= \frac{\exp(\beta_{k0} + \beta_{k1}x_{1i} + \beta_{k2}x_{2i} + \cdots + \beta_{kp}x_{pi})}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \beta_{l1}x_{1i} + \beta_{l2}x_{2i} + \cdots + \beta_{lp}x_{pi})} \\ &= \frac{\exp(\beta_k^T x_i)}{1 + \sum_{l=1}^{K-1} \exp(\beta_l^T x_i)}. \end{aligned}$$

The method is now named "multinomial regression" or "softmax regression". Instead of one linear regression transformed by a non linear function, there are K linear regressions with each its own coefficients and they are embedded in the softmax function. Thus, this model remains linear as the two previous models.

The target variable y_i is rewritten with binary vectors with components y_{ik}, here the value is one if $y_i = k$ and zero otherwise.

The new likelihood is now:

$$P(y|X, \beta) = \prod_{i=1}^{n} \prod_{k=1}^{K} [p(y_i = k|x_i, \beta_k)]^{y_{ik}}.$$

The usual training algorithm remains sligthly similar than for the logistic regression, via gradient descent (Newton's method for instance). Note that in the literature, an advice is to substract the max of the K regressive parts in each probability of class separately in order to avoid numerical issues. The code from the python module sklearn is as previously for two classes with the same function "LogisticRegression()" which is able to handle a multicategory target variable.

2.4 Exercices

1. (sklearn) Instead of the logistic regression, let test the linear regression for the binary target variables, and compare the resulting classification on this (bad) model. We will choose 0 and 1 as values for y_i. This is convenient because the linear regression will give us a continuous value for the prediction $\hat{y}_i$, hence a first choice is just greater or lower to 0.5 for deciding of

the class, $y_i = 1$ if $x_i^T \hat{\beta} > 0.5$, and $y_i = 0$ if $x_i^T \hat{\beta} \leq 0.5$. Compare with the logistic regression for the (very) small dataset of the chapter, or other data. Which model is performing better for predicting the target variable.

2. (sklearn) Test a linear regression or classification for two or three small datasets from the uci machine learning repository, kaggle or sklearn. These datasets are generally pre-processed. Such that mostly missing values may be removed or imputed for some of them. While categorical independent variables may be binarized in order to enter the models. For instance, the following datasets are usual benchmarks.

Dataset	#rows	#vars	#classes
Iris	150	4	3
Breast cancer wisconsin	569	30	2
Wine Quality	4898	11	NA
Credit Card Default	30000	22	NA
Pima Indians Diabetes	768	20	2
Ionosphere	351	33	2
Glass Identification	214	9	6
Ecoli	336	7	8
Abalone	4177	8	NA
KDD Cup 1998	191779	481	NA

3. (sklearn) Test a regression or a classification for one large dataset, with for the training: a) the full sample for a batch algorithm and b) the subsamples for a minibatches algorithm via cycling chunks from the datafile. For instance, the "Higgs dataset from the communication "Searching for Exotic Particles in High-energy Physics with Deep Learning" (2014, Nature) with 11000000 (11M) rows and 28 variables while y is binary, available at the uci in csv format, or in hdf5 format also online. Another dataset larger, is available at "https://developer.ibm.com/data/airline/" in a gzip format from a csv file with 200 million (200M) "domestic US flights" to "predict the likelihood of a flight arriving on time", or its subsample of 2000000 (2M) row samples.

4. (sklearn) Test a regression to predict the cover types in forest from the dataset available from the uci machine learning repository at the entry "Covertype Data Set", with 581012 data vectors and 54 variables (attributes or features). This dataset is from "Comparative Accuracies of Artificial Neural Networks and Discriminant Analysis in Predicting Forest Cover Types from Cartographic Variables" (1999, Elservier).

Chapter 3

First-order training of linear models

Pytorch allows to infer the weights of neural networks by automatically computing the gradient of a cost function and implement the optimization loop without requiring something else than the definition of the network and the definition of the cost function. All the burden for the optimization is almost avoided except some settings in order to help the optimization to perform well with the data. This means that for the linear regression or the logistic regression where the vector β is unknown, one just needs to define a neural network which is corresponding to one of the linear or logistic model, add the cost function, choose an option for the optimization and launch the learning procedure which finds a numerical value for $\hat{\beta}$. The weights of the network w are identified to the coefficients of regression while the structure of the network is very simple in this case, one of the first to appear in the history of neural networks.

For minimizing the cost function $\ell(w)$ the idea of learning parameters for the network is found with an iterative algorithm called gradient descent, from an initial value $w^{(0)}$, and a choice for the learning rate α_t often constant and equal to a small value α such that it is iterated until stabilization:

$$w^{(t+1)} = w^{(t)} + \alpha_t \left.\frac{\partial}{\partial w}\Big(\ell_b(w)\Big)\right|_{w^{(t)}}.$$

Here b is a (mini)batch of indexes i, with s_b a subset of $\{1,\cdots,n\}$. Choosing the whole set or only one single index i is possible but the later often leads to a slower converge. Here we will first choose just one datum and iterate over the whole sample. This approach adds randomness to the learning algorithm which is more efficient, and also avoid an heavy computation on the whole dataset of the gradient at each step which is quicker. When s is the set of indexes $\{1,2,\cdots,n\}$ with $\cup_b s_b = s$ the main variants are defined as follows for each iteration of the algorithm:

- when a singleton $b = i \in s$ from the sample is chosen not randomly by cyle, the algorithm is called "gradient descent"
- when a subset $s_b \in s$ from the sample is chosen not randomly by cyle, the algorithm is called "minibatch gradient descent"
- when a singleton is chosen randomly from the whole sample, the algorithm is called[1]

[1] Robbins–Monro procedure.

"stochastic gradient descent"
- when a subset is chosen randomly from the whole sample, the algorithm is called "stochastic minibatch gradient descent"
- when the full sample $s_b = s$ is chosen, the algorithm is a "batch gradient descent".

This also supposes a good choice for the learning rate α_t which must change according to the size of the minibatch. The stochastic (minibatch) gradient descent approaches are able to reach outstanding solutions for deep neural networks with the rise of the size of the dataset which insures generalization and nearly optimal solutions. For a stochastic procedure, one just need to shuffle, say randomly change the order of, the units in the dataset before each epoch, where an epoch is a full cycle on the dataset. Despite huge volumes of data and huge number of weights, the trainind has become possible nowadays. The batches iterations are generally very slow to compute and eventually prone to bad solutions for nonlinear functions where they are trap in local minima.

There exists some hypothesis required for the convergence of the gradient descent not discussed here. Convexity is one which insures to reach the optimal solution, but relaxing this condition becomes possible in the case of the stochastic version. These algorithms are relevant event when the cost function is nonlinear with several local minima instead of just one minimum. It is also suitable for large datasets or some non invertible matrices for instance for the linear regression. The loss functions of linear regression and logistic regression are convex which allows non stochastic gradient descents to converge fast to the optimum. The choice for α_t is important for insuring the convergence where a stable value is obtained at the final steps but also a good convergence where the solution is relevant at the last step. These two conditions are required in order to get a useful numerical value for $\hat{\beta}$ otherwise another function or value for α_t is wanted until a suitable result is obtained. Changing the initialization is also important in order to change the local minimum of the nonlinear function when the optimum is not reach at the end of the iterations. Think about an example of a polynomial curve, there are several minima and the algorithm is able to go to the minimum at each valley but not always change of valley because this would suppose enough big changes during the first iterations, which is only possible when the gradient is not already small near zero as at the convergence. The gradient of the function needs to be smaller and smaller during the algorithm in order to insure the convergence, and according to the formula above for the update, a large change of value needs either a large learning rate either a large gradient.

Next, such algorithm is implemented with numpy before using the power of pytorch. The optimization is performed with numpy by writing the closed-form expression for the derivative from linear algebra first, before the automatic derivative with pytorch after: with numpy and pytorch for the linear regression and directly with pytorch for the logistic regression.

3.1 Linear regression

The algorithm with iterative updates for the inference of a linear regression is explained in this subsection. Some derivatives are presented partly with only one dependent variable but the last matricial expression is relevant for several variables (by eventually replacing the design matrix by the more complete one). The algorithms are not completely stochastic because it is preferred to cycle the dataset in the same order and several times instead of drawing randomly the selected

rows, as this lead to nearly similar result here.

3.1.1 Gradient descent with one vector and numpy

For this chapter, the simple linear model is defined with the coefficients $\beta_0 = -0.5$ and $\beta_1 = 3.5$ and the noises ε_i, such as the regression function is defined with $x_i = x_{i1}$ as:

$$y_i = \beta_0 + \beta_1 x_{i1} + \varepsilon_i \,.$$

The set of observations or data sample is $(x_i, y_i)_{i\in\{1,\dots,n\}}$. We want to estimate the regressions coefficients with a vector $\hat{\beta}$. The loss function called "mean squared error" is minimized such that,

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}}\, \ell(\beta)\,,$$

where for only one independent variable,

$$\begin{array}{rcl} \ell(\beta) & = & \sum_{i=1}^{n} \ell_i(\beta) \\ & = & \sum_{i\in s} (y_i - \beta_0 - \beta_1 x_{i1})^2 \,. \end{array}$$

Here n may be very large, hence it is better to have an algorithm which does not need the full sample at each iteration.

We have seen previously that the gradient of the cost function with respect to the weights is written as a vector which will enter the iterative procedure. As explained before, we add the component one to the vector in order to get $x_i = (1, x_{i1})^T$ while aggregating all the vectors in the design matrix X, such as at the end, the gradient is written as follows,

$$\frac{\partial}{\partial w}\Big(\ell_i(\beta)\Big) = \begin{bmatrix} \frac{\partial}{\partial \beta_0}\Big(\ell_i(\beta)\Big) \\ \frac{\partial}{\partial \beta_1}\Big(\ell_i(\beta)\Big) \end{bmatrix} = -2 \begin{bmatrix} (y_i - \beta_0 - \beta_1 x_{i1}) \\ x_{i1}\,(y_i - \beta_0 - \beta_1 x_{i1}) \end{bmatrix} = -2 \begin{bmatrix} 1 \\ x_{i1} \end{bmatrix} \left[y_i - x_i^T \beta\right] \,.$$

Now we just need to choose a value or a function for $\alpha_t = \alpha(t)$ that we choose constant equal to $\alpha_t = \alpha = .01$ here for instance. This is implemented just belows, recall that:

$$s = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}\,.$$

Next we follow the update rule of β after a random initialization of the vector, hence we implement:

$$\begin{bmatrix} \beta_0^{(t+1)} \\ \beta_1^{(t+1)} \end{bmatrix} = \begin{bmatrix} \beta_0^{(t)} \\ \beta_1^{(t)} \end{bmatrix} + \alpha_t \times (-2) \begin{bmatrix} 1 \\ x_{i1} \end{bmatrix} \left[y_i - x_i^T \beta^{(t)}\right] \,.$$

The algorithm stop at the final step $t = T$. In general it must be several cycles over all the data sample, each cycle is called an epoch. This translates in python as follow below, in a vectorized way because vectors are handled faster than scalar components with loops in pytorch.

Dataset from the files

The work directory is given from the function.

```
import deepglmlib.utils as utils
import numpy as np

def towdir(s):
    return (str('./datasets_book/'+s))
```

```
import importlib
importlib.reload(utils)
```

```
<module 'deepglmlib.utils' from
'/home/rodolphe/Documents/ARTICLEs/BOOK/deepglmlib/utils.py'>
```

```
import numpy as np

beta = np.loadtxt(towdir("./beta_1d_reglinear300.txt"))
xy   = np.loadtxt(towdir("./xy_1d_reglinear300.txt"))

beta = beta.reshape((2,1))
x    = xy[:,0].reshape((len(xy),1))
y    = xy[:,1].reshape((len(xy),1))
n    = len(y)
X    = np.hstack([np.ones((n,1)), x])

X.shape, x.shape, y.shape, beta.shape
```

```
((300, 2), (300, 1), (300, 1), (2, 1))
```

```
print("beta_true=")
print(beta)
```

```
beta_true=
[[-0.5]
 [ 3.5]]
```

Solution with the sample from algebra

The regression coefficients are estimated from the sample with algebra as:

```
betahat_np = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
betahat_np = betahat_np.reshape((2,1))
print("betaha_np=")
print(betahat_np)
print()
print("loss_np=",np.round(np.sum((y-X@betahat_np)**2)/n,5))
```

```
betaha_np=
[[-0.47946113]
 [ 3.48042302]]

loss_np= 0.04026
```

Loop with data one by one and parameters update via numpy

A first implementation is with a simple loop then a second implementation is more oriented neural network before a third implementation using pytorch for the derivative and a pure pytorch implementation before a final version oriented object with a parent class from pytorch which will be the way to do in the next chapters.

```
import copy
#beta_init = np.random.randn(2,1) * 2
```

```
#beta_st    = copy.deepcopy(beta_init)
beta_st    = 5.0 * np.ones((2,1))
alpha_t     = 1e-2
n_epoqs     = 20
loss_st_s   = []
grad2_st_s   = []
beta_st_s   = []
beta_st_s.append( copy.deepcopy(beta_st.squeeze()) )

n = len(y)
for epoch in range(0,n_epoqs):
    for i in range(0,n):
        xi = X[i,:].reshape((2,1))
        yi = y[i]
        gradient1  = -2*xi@(yi-xi.T@beta_st)
        beta_st  -= alpha_t * gradient1
        beta_st_s.append( copy.deepcopy(beta_st.squeeze()) )
    #print("beta_st=\n",beta_st.ravel())
    loss_st_s.append( np.sum((y-X@beta_st)**2)/n )
    grad2_st_s.append( np.sum((gradient1)**2) )
```

```
tmax_st=epoch
```

The results are the final estimation for the regression coefficients and the loss. It is wise to show the form of the loss after training, in order to check the convergence, as a not too fast and not too slow decrease generally inform for a relevant choice of the learning rate, otherwise another choice is wanted. Anyway, the gradient should be near zero in order to confirm that an optimum (here unique for the linear regression) was reached at the end of the loop. The resulting curve validates the learning rate.

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
utils.f_draw(range(len(loss_st_s)), loss_st_s, "b-", r'$epoch$',r'$mse$',
             "Fitting linear regression (per 1 unit, numpy)",ax)
```

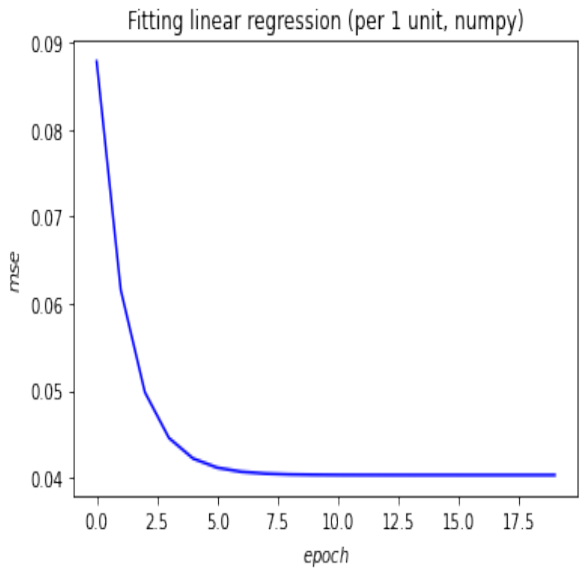

Figure 3.1: Decreasing loss function during training, per epochs

The algorithm performs well even if there was no an advanced tuning of the learning rate, just a few tries (not presented, but the reader might try other learning rates and compare the obtained curves, see exercices). Note that some stochastic algorithms would sample the data vector in some case, this is not considered here. The values for α_t among 10^u with u integer between -6 to 3 are an usual try, as there does not exist any formula yet in order to find it automatically.

3.1.2 Gradient descent with a minibatch and numpy

Next the learning with a minibatch is presented for the linear regression. As the loss function is less noisy during the training, this will allow visual comparisons by inspection of the learning curves.

With a number of B minibatches, the loss function is now written as B sums,

$$\begin{aligned}\ell(\beta) &= \sum_{i=b}^{B} \ell_b(\beta) \\ &= \sum_{b=1}^{B} \left\{ \sum_{i \in s_b} (y_i - \beta_0 - \beta_1 x_{i1})^2 \right\} .\end{aligned}$$

The new update formula for $\beta^{(t)}$ is thus written not from a single data vector but from the whole subset in the minibatch, roughly n/B, as:

$$\frac{\partial}{\partial w}\Big(\ell_b(\beta)\Big) = \begin{bmatrix} \frac{\partial}{\partial \beta_0}\Big(\ell_b(\beta)\Big) \\ \frac{\partial}{\partial \beta_1}\Big(\ell_b(\beta)\Big) \end{bmatrix} = \sum_{i \in s_b} -2 \begin{bmatrix} 1 \\ x_{i1} \end{bmatrix} \left[y_i - x_i^T \beta \right] = -2 X_b^T \left(y_b - X_b \beta_b \right) .$$

This is implemented just below, with a size $|s_b|$ for the batch b equal to 10. For the training, the same batches are cycled at each epoch, but shuffling is also possible (by reordering constant minibatches, by sampling new minibatches with or without replacement) at each epoch:

$$s = s_1 U s_2 U \cdots U s_B .$$

This is instead of changing their order or sampling batches at each step. This is convenient for a fast access to datafiles for example, as the aim is often to process large datasets which cannot fit in the computer memory. The new update formula is thus as follows,

$$\begin{bmatrix} \beta_0^{(t+1)} \\ \beta_1^{(t+1)} \end{bmatrix} = \begin{bmatrix} \beta_0^{(t)} \\ \beta_1^{(t)} \end{bmatrix} + \alpha_t \times (-2) X_b^T \left(y_b - X_b \beta_b \right) .$$

Thus,

```
#beta_mb = copy.deepcopy(beta_init)
beta_mb = -5.0 * np.ones((2,1))
alpha_t  = 1e-2
n_epoqs  = 20

size_batch=10
i_s       = np.array(np.mgrid[0:n:
 ↪complex(real=0,imag=size_batch)],dtype=int)

loss_mb_s = []
grad2_mb_s = []
beta_mb_s = []
beta_mb_s.append( copy.deepcopy(beta_mb.squeeze()) )

n = len(y)
for epoch in range(0,n_epoqs):
    for l in range(0,len(i_s)-1):
```

```
        b = range(i_s[l]+1*(l>0),i_s[l+1])
        Xb = X[b,:]
        yb = y[b]
        gradientb = -2*np.transpose(Xb)@(yb-Xb @ beta_mb)
        beta_mb -= alpha_t * gradientb
        beta_mb_s.append( copy.deepcopy(beta_mb.squeeze()) )
    loss_mb_s.append( np.sum((y-X@beta_mb)**2)/n )
    grad2_mb_s.append( np.sum((gradientb)**2) )
tmax_mb=epoch

print("beta_mb=\n",beta_mb)
print()
print("loss_mb=",np.round(loss_mb_s[tmax_mb-1],5))
```

```
beta_mb=
 [[-0.46612023]
 [ 3.48484567]]

loss_mb= 0.04051
```

There is a nice decreasing curve for the loss during the training because the cost function is a smooth convex parabola and the learning rate was chosen not too lage or not too small even if probably not optimally. A better setting would lead to a solution nearer to the optimum for $\hat{\beta}$. With this setting, the convergence may be slower than the stochastic algorithm.

3.1.3 Gradient descent with a minibatch and pytorch

According to the tutorial[2] of pytorch, tensors are used to encode the data to process, but also the internal states and parameters of models. Basically, tensors extend the usual two dimension matrix or one dimensional vector into more dimensions such as cubes with three dimensions. These mathematical objects are also called multi-array or just array in numpy. This is with a more general definition from the point of view of the storage as they may be stored in the computer memory near the cpu or directly in the gpu memory near its highly parallel processors. Thus arrays are involved herein, in a similar way than with numpy with similar computations even if the storage and the syntax may be somewhat different in some case or extended. It is possible to change the size of an array, append arrays, get the shape and apply usual mathematical operations (addition, substraction and multiplication).

With the module pytorch, we are not using anymore arrays from numpy but tensors from pytorch. This allows to implement the power of the gpu by moving the arrays/tensors in the memory of the gpu and perform in parallel the computations of the derivatives for nonlinear functions automatically. This means that the gradient is directly computed by the module whatever the complexity of the network. We just need to follow the logic implemented in the module. Note that for small

[2] "https://pytorch.org/tutorials/index.html",
"https://pytorch.org/tutorials/beginner/former_torchies/tensor_tutorial.html"

dataset, the cpu is enough and the gpu supposes to install cuda before being available. In this chapter we stay in the computer memory and with the cpu. For small datasets, an implementation with numpy may be faster because more direct but asks for explicit expressions.

Loading the python module

```
import torch
```

```
torch.__version__
```

```
'1.10.0+cu102'
```

In order to get the data in a format relevant for pytorch, it is usual to use a class called "DataLoader", which is optional as one may have its own loop for cycling a source of data. This python class gives an iterator in order to cycle the dataset with minibatches, whatever are the data places: computer memory, gpu memory, hard disk or even from a remote server via the network because it is possible to write its own class. Hence, it is first imported the dataset in a dataloader object, and then implemented the autograd function with calculates a value of the gradient for a function automatically.

3.1.4 Dataloader, autograd and training loop

Here we need first to create an object of class "Datataset" (see documentation of pytorch) which takes a simple numpy array in the present case. This class is very general and when implemented, it is able to deal also with multimedia files for instance while performing parallel processing.The Dataloader provides the iterator for reading the Dataset and loading the minibatches with the vectors y_b and the matrices X_b or x_b in order to train the model.

Dataloader

There are several ways to implement this class, for instance by creating our own class, but here we just create new objects of the class torch.Tensor from the whole dataset because the numpy array is very small and fit in the computer memory. After that, the iterator is created by instanciating a new object of class Dataloader, while the operator function "iter()" leads to get the X and y from each minibatch (of size 10).

```
from torch.utils.data import DataLoader, TensorDataset

dataset     = TensorDataset( torch.Tensor(X), torch.Tensor(y) )
dataloader  = DataLoader(dataset, batch_size= 10, shuffle=False)
```

We have still access to the dataset in a numpy format.

```
print(dataloader.dataset.tensors[0].numpy().shape,
      dataloader.dataset.tensors[1].numpy().shape)
```

```
(300, 2) (300, 1)
```

The main interest of these classes appears in next chapters to load a datafile sequentially and partially from the disk, directly from the classes Dataset and Dataloader, without loading the full data in memory for any format implemented in the class like compressed binary format or multimedia files in several directories while applying some transformation (mutate into a tensor format or into a different shape for instance).

Autograd and loop for the updates

The automatic computation from pytorch with tensors via cpu (or gpu if added) is then written as a function here, but this is for later use, otherwise an implementation with no function is easier to debug if required.

First an object of class Dataset reprenting a pointer to the data in memory or computer disk is created, and for cycling the data sample, an object of class DataLoader which is like a python "iterator", they are named respectively "dataset" and "dataloader", as explained in the pararaph above.

Then, the function for the optimization with pytorch is defined as follows. The iterator from dataloader provides the minibatches of size 10 here, leading to the objects X_b and y_b which are part of the whole design matrix X and the whole vector y aggregating the target variables y_i. The vector for the regression coefficients is declared requiring gradient, with the command "requires_grad_(True)" such as the automatic derivative is processed at the call from the function "backward()". In short, a separated function here was implemented for the implementation of the updates, with the gradient retrieved with the variable "grad" from the Tensor object which contains the regression coefficients, "betat_ag", and it is reinitialised before new data. The call "with torch.no_grad()" allows to avoid any update to the gradient for any other computations than gradient or parameters updates.

```
def loss_mse_Xb_yb_beta(Xb,yb,beta): #loss to minimize
    return (yb-Xb@beta).pow(2).sum()

def f_minimizeloss_update_torch(beta_ag,alpha_t,Xb,yb):
    beta_ag -= alpha_t * beta_ag.grad

def f_minimizeloss_loop_torch(dataloader,loss,f_update,
                              alpha_t=1e-3,n_epoqs=180,
                              betainit=None):
    loss_ag_s  = []
    beta_ag_s = []
    n          = dataloader.dataset.tensors[0].numpy().shape[0]
```

```
    p          = dataloader.dataset.tensors[0].numpy().shape[1]

    if betainit is None: betainit = np.zeros((p,1))
    beta_ag    = torch.Tensor(betainit)
    beta_ag.requires_grad_(True)          #required for autograd
    beta_ag_s.append(betainit.squeeze())  #keep init beta in a list

    for epoch in range(0,n_epoqs):
        loss_epoch=0
        #for l in range(0,len(i_s)-1):
        for Xb,yb in iter(dataloader):
            loss_b = loss(Xb,yb,beta_ag)
            loss_b.backward()
            #(^--- update gradient of loss from autograd)
            with torch.no_grad():
                f_update(beta_ag,alpha_t,Xb,yb)
                beta_ag.grad.zero_()
                #beta_ag -= alpha_t * beta_ag.grad
                #beta_ag.grad.zero_()
            loss_epoch += loss_b
            #keep intermediate value of beta in a list:
            beta_ag_s.append(copy.deepcopy(
                   beta_ag.detach().numpy().squeeze()))
        loss_ag_s.append(loss_epoch/n)
    tmax_ag = epoch
    return beta_ag, tmax_ag, loss_ag_s, beta_ag_s
```

The call to this function for updating the parameters after one step is as follows:

```
loss     = loss_mse_Xb_yb_beta
alpha_t  = 1e-2
n_epochs = 1 #20
f_update = f_minimizeloss_update_torch

beta_ag, tmax_ag, loss_ag_s, beta_ag_s = \
   f_minimizeloss_loop_torch(dataloader,loss,
                             f_update,alpha_t,
                             n_epochs)

print("beta_ag=\n",beta_ag)
print()
print(f"loss={loss_ag_s[tmax_ag]:1.5f}")
```

```
beta_ag=
 tensor([[0.5787],
```

```
        [1.5517]], requires_grad=True)

loss=0.69634
```

```
# f_draw(range(len(loss_ag_s)), loss_ag_s, "b-", r'$epoch$',r'$mse$',\
#        "Fitting linear regression (per 10 units, pytorch)")
```

It must be noticed that changing the loss obviouly change the gradient, but more importantly, there is a difference between the mean and the sum, with a factor 1/n, thus this must be keep in mind when choosing the learning rate which is proportional to this factor by linearity and definition of the gradient. One can compare the value of the numerical gradient in "betat_ag.grad" with the analytical expression written with numpy, they are exactly the same (with an eventual very small numerical difference actually, often neglectable with real data).

```
beta_ag = beta_ag.detach().numpy()
```

To compare the numerical and analytical gradient, let rewrite the update by computing the difference between both, as follows.

```
diff2_grad_s = []
def f_minimizeloss_update_torch(beta_ag,alpha_t,Xb,yb):
    gradient = -2*np.transpose(Xb)@(yb-Xb @ beta_ag)
    diff_grad = gradient - beta_ag.grad
    diff2_grad_s.append( torch.sum( diff_grad **2) )
    beta_ag -= alpha_t * beta_ag.grad
    beta_ag.grad.zero_()
```

```
f_update = f_minimizeloss_update_torch

beta_ag, tmax_ag, loss_ag_s, beta_ag_s = \
   f_minimizeloss_loop_torch(dataloader,loss,f_update,alpha_t,n_epochs)
```

```
np.min(diff2_grad_s), np.max(diff2_grad_s)
```

```
(0.0, 0.0)
```

This avoids to find the mathematical expression for the derivative which is eventually not even possible to compute exactly in closed-form with paper and pencil for more complicated loss functions. A verification of the gradient is required at least at the end of the training in order to check if the numerical procedure has performed well (which should be the case most of the time) but also if the gradient is enough small as required.

3.1.5 Visualization of the parameters trajectory

Let draw the intermediate solutions for $\beta^{(t)}$ at each update from each epoch in order to further understand what the algorithms do. This supposes to have run "f_minimizeloss_loop_torch" with "n_epochs=20". There are three different implementations for the training algorithm, hence three curves for three trajectories appear on the graphic below.

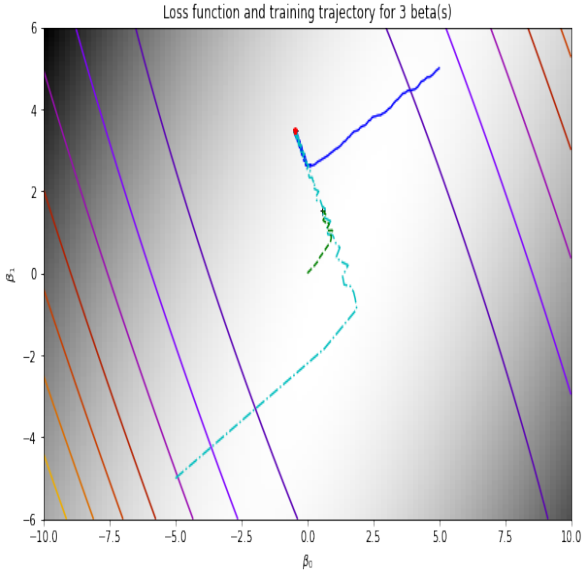

Figure 3.2: Loss and intermediate values of beta during convergence

The graphs just above shows the cost function as a function of its parameters, as follows:

$$\beta \rightarrow f(\beta) = MSE(y, X\beta)$$

This allows to show the shape of the loss which is convex as a 3d parabola, but also the intermediate solutions from each epoch until the convergence. The surface is quadratic as expected from the expression of the loss, hence the algorithms need to do a descent along the surface until the minimum. The better algorithm with the better settings would be the one with fastest descent. The initial positions are different for the algorithms hence a fair comparison is not in stake. If the three trajectories were shown per epoch instead of per update, they would be more smooth and more identical considering the fast convergence near the solution.

It is observed that the algorithms converge towards almost the same solution near the exact one from linear algebra. This is without inverting a matrix, only with the gradient of the loss function which is computed only partially with only one data vector or a few ones at each iteration.

The python function for this graphic is as below.

```
import matplotlib.pyplot as plt
```

```
def f_MSE(betahat):
    yhat = X @ betahat
    return np.sum((y-yhat)**2) / len(yhat)

def f_draw_mse2d_betas(Xmin,Xmax,Ymin,Ymax,f_,betahat,
                       beta_s_list, marker_list):
    XX, YY = np.mgrid[Xmin:Xmax:100j, Ymin:Ymax:100j]
    ZZ=0*XX
    for k in range(0,100):
        for l in range(0,100):
            betakl = [XX[k,l],YY[k,l]]
            ZZ[k,l] = f_(betakl)
    plt.imshow(ZZ.T, cmap=plt.cm.gray_r,
               extent=[Xmin, Xmax, Ymin, Ymax],) #origin='lower')
    plt.contour(XX, YY, ZZ, cmap=plt.cm.gnuplot)
    plt.xlabel(r'$\beta_0$')
    plt.ylabel(r'$\beta_1$')
    plt.title(f'Loss function and training trajectory for '+\
              str(len(beta_s_list)) +' beta(s)')
    for beta_s in iter(beta_s_list):
        plt.plot(np.asarray(beta_s)[-1,0],
                 np.asarray(beta_s)[-1,1], 'k+', markersize=5)
    for beta_s, marker in iter(zip(beta_s_list,marker_list)):
        plt.plot(np.asarray(beta_s)[:,0], np.asarray(beta_s)[:,1],
                 marker, markersize=0.8)
    plt.plot(betahat[0], betahat[1], 'rp', markersize=5)
    plt.show()
plt.figure(figsize=(20,6))
f_draw_mse2d_betas(-10.0,10.0,-6.0,6.0, f_MSE, betahat_np,
      [beta_st_s, beta_mb_s, beta_ag_s],
      ['b-', 'c-.', 'g--'])
```

Next, these python functions are further illustrated with an example for classification.

3.2 Logistic regression

Recall in the second chapter, the small example of classification model is defined with the coefficients $\beta_0 = -0.5$ and $\beta_1 = 3.5$ and $\beta_1 = -2.5$ such as,

$$E(y_i|x_i) = \sigma(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2})\,.$$

Here σ is usually the sigmoidal function, otherwise the regression is called with another name.

The set of observations or data sample is $s = (x_i, y_i)_{i\in\{1,\dots,n\}}$, with y_i binary. We want to estimate the coefficients with a vector $\hat{\beta}$. The loss function called cross-entropy loss is minimized such that,

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}}\, \ell(\beta)\,.$$

We are direcly interested with the minibatches, thus, with a number of B minibatches, the loss function is now written as B sums from subsets of s such that,

$$\begin{aligned} \ell(\beta) &= \sum_{i=b}^{B} \ell_b(\beta) \\ &= \sum_{b=1}^{B} -\left\{ \sum_{i \in s_b} \left(y_i (\beta^T x_i) - \log\left(1 + \exp\left(\beta^T x_i\right)\right)\right)\right\}. \end{aligned}$$

The new update formula for $\beta^{(t)}$ is thus written not from a single data vector but the whole subset in the minibatch, roughly n/B, and from the formula of the previous chapter just for the subset,

$$\frac{\partial}{\partial \beta}\left(\ell_b(\beta)\right) = \begin{bmatrix} \frac{\partial}{\partial \beta_0}\left(\ell_b(\beta)\right) \\ \frac{\partial}{\partial \beta_1}\left(C_b(\beta)\right) \end{bmatrix} = -X_b^T \left(y_b - p_b\right).$$

The algorithm stops at the final step $t = T$. The new update formula at each iteration is thus as for the linear regression but with a different gradient function because the loss function has changed. This is a first-order algorithm,

$$\begin{bmatrix} \beta_0^{(t+1)} \\ \beta_1^{(t+1)} \end{bmatrix} = \begin{bmatrix} \beta_0^{(t)} \\ \beta_1^{(t)} \end{bmatrix} - \alpha_t \left.\frac{\partial}{\partial \beta}\left(\ell_b(\beta)\right)\right|_{\beta^{(t)}}.$$

This translates in python as follow below, in a vectorized way because vectors are manipulated instead of scalar components.

3.2.1 Dataset with bivariate classes (continued)

Let first load the dataset.

```
import numpy as np
xy   = np.loadtxt(towdir("./xy_2d_reglogistic.txt",))
beta = np.loadtxt(towdir("./beta_2d_reglogistic.txt"))

x    = xy[:,[0,1]]
y    = xy[:,2]
n    = len(y)

print(xy.shape,x.shape,y.shape)
```

```
(100, 3) (100, 2) (100,)
```

```
sum(y==1), sum(y==0)
```

```
(49, 51)
```

3.2.2 Loss minimization with pytorch

The implementation for minimizing the cost function is thus as follows.

```
X    = np.hstack([np.ones((n,1)), x])
X.shape
```

```
(100, 3)
```

```
beta_init  = np.random.randn(3,1) * 2
```

```
print("loglik_init=",y.dot(X@beta_init) - np.sum(np.log(1 + np.
 ↪exp(X@beta_init))))
```

```
loglik_init= [-77.83109693]
```

```
# print("loglik_skl=",y.dot(X@betahat_skl) - np.sum(np.log(1 + np.
 ↪exp(X@betahat_skl))))
```

The training is implemented with pytorch, from the previous function, by just changing the loss and the parameters for the inference.

```
import numpy as np
import torch # from torch import Tensor
from torch.utils.data import DataLoader, TensorDataset

def loss_crossentropy_Xb_yb_beta(Xb,yb,beta): #loss for a minibatch
    yb=yb.reshape((len(yb),1))
    beta=beta.reshape((len(beta),1))
    XbTbeta = Xb@beta
    l  = torch.sum(torch.log(1+torch.exp(XbTbeta))) - yb.T@XbTbeta
    return l.squeeze()
```

```
batch_size = 10

dataset     = TensorDataset( torch.Tensor(X), torch.Tensor(y) )
dataloader  = DataLoader(dataset, batch_size= batch_size, shuffle=False)
```

```
loss     = loss_crossentropy_Xb_yb_beta
alpha_t  = 7e-3
n_epochs = 500
```

```
f_update = f_minimizeloss_update_torch

betahat_ag, tmax_ag, loss_ag_s, betahat_ag_s = \
   f_minimizeloss_loop_torch(dataloader,loss,f_update,alpha_t,
                             n_epochs,betainit=beta_init)

print("betahat_ag=\n",betahat_ag)
print()
print(f"loss={n*loss_ag_s[tmax_ag].detach():1.5f}")
```

```
betahat_ag=
 tensor([[-1.1000],
        [ 3.6104],
        [ 1.1798]], requires_grad=True)

loss=14.68434
```

```
betahat_ag = betahat_ag.detach().numpy()
betahat_ag
```

```
array([[-1.1000439],
       [ 3.6103797],
       [ 1.1797962]], dtype=float32)
```

```
# plt = f_draw(range(len(loss_ag_s)), loss_ag_s, "b-",␣
  ↪r'$epoch$',r'$mse$',
#              "Fitting logistic  regression (per B units, pytorch)")
```

3.2.3 Comparison of the solutions

Let compare the different losses (minus the loglikelihood) from the algorithm above, plus also the classification error. It is remembered that the classification error or error rate is the quantity to be minimized, but as a non continuous quantity, the optimization is for a smooth function instead and with a similar behaviour than the wanted minimization of the error.

Loss and loglikelihood

```
X_torch = dataloader.dataset.tensors[0]
y_torch = dataloader.dataset.tensors[1]
```

The losses from the initialization and the solution with pytorch are compared.

```
# loss_skl = loss(torch.Tensor(X),torch.Tensor(y),torch.
 ↪Tensor(betahat_skl))
# print("loss_skl=",np.round(loss_skl.detach().numpy(),5))
```

```
loss_betainit = loss_crossentropy_Xb_yb_beta(X_torch,y_torch,torch.
 ↪Tensor(beta_init))
print("loss_betainit=",np.round(loss_betainit.detach().numpy(),5))
```

```
loss_betainit= 77.83112
```

```
loss_torch = loss_crossentropy_Xb_yb_beta(X_torch,y_torch,torch.
 ↪Tensor(betahat_ag))
print("loss_betatorch=",np.round(loss_torch.detach().numpy(),5))
```

```
loss_betatorch= 14.64568
```

Accuracy

About the accurracy, one gets:

```
import deepglmlib.utils as utils

yhat_init  = (utils.f_sigmoid((X@beta_init).ravel())>=0.5).astype(int)
yhat_torch = (utils.f_sigmoid((X@betahat_ag).ravel())>=0.5).astype(int)
# yhat_sk    = logreg.predict(x)

print("acc_init=",np.mean(y==yhat_init))
# print("acc_sk=",np.mean(y==yhat_sk))
print("acc_torch=",np.mean(y==yhat_torch))
```

```
acc_init= 0.63
acc_torch= 0.95
```

The solution with pytorch is slighly performing less well for the loss and the accuracy in comparison to sklearn for this dataset and the chosen hyperparameters (see exercices). Note that for a large dataset, the design matrix (the x_i and also the y_i) are stored on the computer disk, thus a loop is required in order to retrieve the rows of the matrix (with minibatches), and compute incrementally the predicted target in order to compare with the true one and compute the accuracy, as explained in the next chapter.

Just below, the frontier is visualized in a graphical representation.

Frontiers

Let plot again the frontier corresponding to the cut from the probability that a data vector belongs to one class or another, with the threshold 0.5 as in the previous chapter but here with several frontiers (from initial and intermediate values with pytorch plus true value).

```
# %matplotlib inline
import deepglmlib.utils as utils
import matplotlib.pyplot as plt
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8.5,11))

# First plot with frontiers from three vectors beta
utils.f_vizu2d_beta(ax1,x[y.ravel()==0,0],x[y.ravel()==1,0],
    x[y.ravel()==0,1],x[y.ravel()==1,1],
    [beta_init,betahat_ag, beta],['k-.','c--', "g-"],
    xlim = [min(x[:,0]),max(x[:,0])], ylim=[min(x[:,1]),max(x[:,1])])

# Second plot with frontiers from intermediate beta during training
beta_s    = [beta for t,beta in enumerate(betahat_ag_s) \
 if t in [0,10,50,100,500,1000,2000,3000,4000,len(betahat_ag_s)-1] ]
marker_s  = ['k-' for i in range(len(beta_s))]
utils.f_vizu2d_beta(ax2,x[y.ravel()==0,0],x[y.ravel()==1,0],
    x[y.ravel()==0,1],x[y.ravel()==1,1],beta_s,marker_s,
    xlim = [min(x[:,0]),max(x[:,0])], ylim=[min(x[:,1]),max(x[:,1])])
plt.show()
```

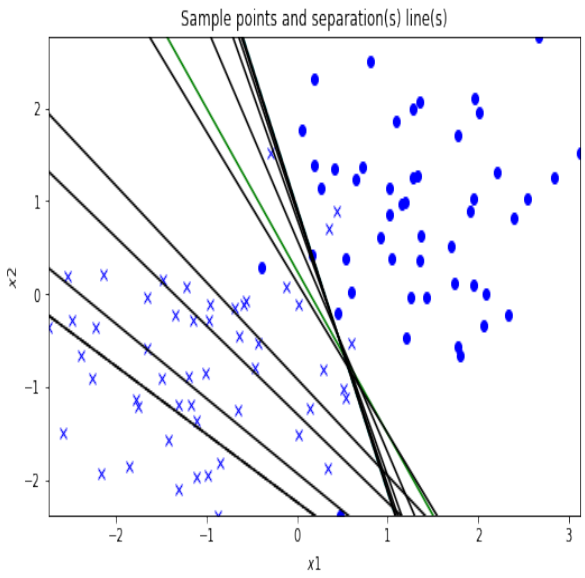

Figure 3.3: Intermediate frontiers for the two classes during convergence

The solution is slightly different to the one from the the module of machine learning but the frontier looks almost the same with nearly a same error rate. Another difference is that we have a custom

algorithm and will be able to update it as most as required on the contrary to existing modules which are often cumbersome to update by the end user.

3.3 Exercices

1. (sklearn) Retrieve the regression coefficients with sklearn. There is a penalty term added to the loglikelihood by defaut, $\frac{1}{C}(\beta_1^2+\beta_2^2)$ with C=10, which is removed here with the option penalty='none', but this term should be kept and tuned for better results with new data (see a next chapter about).
2. (sklearn) Test the algorithms with real data as the exercice in previous chapter, for the same datasets, and compare the results with sklearn.
3. (python) Test other values of the learning rate in the presented set of candidate values, and around the best one in order to improve the estimation for β.
4. (python) Test for also other sizes of minibatches and change accordingly the learning rate, how is the change.
5. (python) Test other settings for the learning rate as a decreasing function of the epoch instead of a constant.
6. (python) Draw the trajectory of the training for beta in the logistic regression, with two graphics, one for the coefficients β_0 and β_1, one for the coefficients β_0 and β_2. After a vectical disposition of the two graphics, propose a way to get both graphics side by side, one to the left and one with an horizontal disposition. It is possible to improve these graphics, by showing the mse and the real beta from the true distribution instead that an estimated one.
7. (stat) How to get a nonlinear frontier from the usual logistic classification instead of the linear one, in any available python implementation.
8. (stat+python) Propose a better strategy for choosing the minibatches, or for initializing the weights for the simple linear regression, and check graphically by comparing the learning curves.
9. (python) Rewrite the python code by selecting random batches (a. one random single unit from the sample at each epoch instead of a cycle or, b. again n/B subsamples after shuffling the index of the rows in the data table at the beginning of each epoch) instead of the constant non stochastic subsamples, and compare the two learning curves.
10. (stat+python) Test the algorithms in "variance reduction for stochastic gradient optimization", "SAGA" and also the "multi-point bandit feedback" approach (among other) by implementing the algorithms with numpy or pytorch. Is the inference improved, how and why.
11. (stat) Justify analytically why the gradient descent converges to the optimum for a convex function.

Chapter 4

Neural networks for (deep) glm

In the three first chapters it was recalled the fondations for the linear regression and the linear classification, then a first approach was introduced in order to perform the parameters learning with pytorch. In this chapter, pytorch is now able to model and train [1] neural networks via implemented python classes. The target variable y_i is the output of the network while the vectors x_i feeding the network are the input variables. The unknown nonlinear transformation which takes x_i and provides y_i must be captured by the neural network during the training of its weights in order to offer an accurate prediction for current and also future inputs.

4.1 From linear models to neural networks

A neural network in a simplest definition can be seen as taking an input vector x_i and providing an output y_i via nested weighted nonlinear functions. For more generality, we will denote the output as a vector even if a scalar is also possible. A neural network uses weights denoted W or w which needs to be found during what is called the training of the neural network. Like fitting a regression or classification model for computing an estimate of β, training the neural network leads to compute an estimate of W from a sample. In the case of the linear regression and linear classification, $W = \beta$, but of course neural networks are more general. This is written as follows:

$$y_i = f(x_i)$$

Hence, the true and unknown function $f(.)$ is approximated by a neural network:

$$y_i = g(x_i|W)$$

We train the model in order to get $\hat{W}$ such that,

[1] See also "https://pytorch.org/tutorials/beginner/nlp/deep_learning_tutorial.html".

$$\begin{aligned}\hat{f}(x_i) &= g(x_i|\hat{W}) \\ &\approx f(x_i)\,.\end{aligned}$$

Thus, the fonction $f(.)$ may be very general and anyway, it was shown in the literature that any functional relationship between x_i and y_i can be learnt by some neural network, presented herein, called MLP which are deep neural network with what are called hidden layers. The first layer is for the input variable, the last layer is for the output variables, hence they are not hidden as the values are read in the sample, while, between both, latent virtual observations in higher or smaller spaces than the input, are called the hidden layers. The components are the nodes and they are linked with weights, in the idea of the brain for signal propagation, from the input towards the output.

For the linear cases with no hidden layers, this is depicted just as,

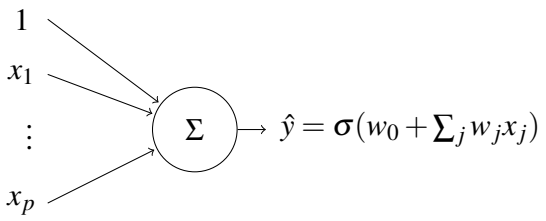

Figure 4.1: Simple neural network for a linear model.

For the linear regression and the linear classification we have two layers, the input for x_i and the output for the y_i, while the weights are defined from the edges between the nodes of the input to the nodes of the output. When some layers are defined between the input and the ouput, the neural network becomes a deep neural network able to model a nonlinear function.

The loss function compares y_i and $\hat{y}_i$ for training purpose.

4.1.1 Loss functions from glm

The output $\hat{y}_i$ is as follows for y_i continuous, binary or categorical:

- Linear regression

$$\hat{y}_i = w_0 + w_1 x_{i1} + w_1 x_{i2} + \cdots + w_p x_{ip} = identity(w^T x_i)\,.$$

- Logistic regression,

$$\hat{y}_i = \frac{exp(w_0 + w_1 x_{i1} + w_1 x_{i2} + \cdots + w_p x_{ip})}{1 + exp(w_0 + w_1 x_{i1} + w_1 x_{i2} + \cdots + w_p x_{ip})} = sigmoid(w^T x_i)\,.$$

- Softmax regression,

$$\hat{y}_i = \begin{pmatrix} \hat{y}_{i1} \\ \hat{y}_{i2} \\ \vdots \\ \hat{y}_{iK} \end{pmatrix} = softmax(w^T x_i)\,.$$

$$\hat{y}_{ik} = \frac{\exp\left(w_{k0} + w_{k1}x_{i1} + w_{k2}x_{i2} + \cdots + w_{kp}x_{ip}\right)}{1 + \sum_{l=1}^{K-1} \exp\left(w_{l0} + w_{l1}x_{i1} + w_{l2}x_{i2} + \cdots + w_{lp}x_{ip}\right)}\,.$$

Here, for the third case, w is a matrix of size $(K \times 1 + p)$ in order to obtain K real values.

Thus, $\hat{y}_i = E(y_i)$, with:

- Linear regression

$$\hat{y}_i = w^T x_i\,.$$

- Logistic regression,

$$\hat{y}_i = \frac{exp(w^T x_i)}{1 + exp(w^T x_i)} \qquad = p(y_i = 1)\,.$$

- Softmax regression,

$$\hat{y}_{ik} = \frac{\exp\left(w_k^T x_i\right)}{1 + \sum_{l=1}^{K-1} \exp\left(w_l^T x_i\right)} = p(y_{ik} = 1 | x_i, \beta_k)\,.$$

A simplified way to understand how the network performs is that the outputs are crude estimations of the dependent variable in the three cases. An objective function is computed between the real observations y_i and the prediction from the network $\hat{y}_i$. Such that:

$$\ell(W) = \sum_i LOSS(y_i, NN(x_i))\,.$$

Where

$$\begin{aligned} NN(x_i) &= \text{"the output from the neural netwok"} \\ &= \hat{y}_i\,. \end{aligned}$$

4.1.2 Post-processing of the output

We have seen that for the classification problem, the output of the network are probabilities which are approximations of expectations in a probabilistic model, hence, the true $\hat{y}_i$ is obtained after comparing the expectations with a threshold in order to decide which class the input vector belongs. Another interpretation is that the network aims at approaching the binary values of y_i or y_{ik} with smooth values inside the interval $[0;1]$ in order to construct the approximation of the true function $f(.)$.

It must be noticed that in the python code here, yhat has not the same meaning in the training loop and outside, after for processing the results, thus, this is to understand as follows (with $\leftarrow$ as an affectation to the value of a variable, here yhat, in the computer meaning). After training, the actual predictions are obtained:

- Linear regression (one output in $]-\infty;\infty[$ from nn, keep the same after the loop) with predicted output $\hat{y}_i$ in the real line.

$$\hat{y}_i \leftarrow NN(x_i) \in \mathbb{R}\,.$$

- Logistic regression (one output in $]0;1[$ from nn compared to s a threshold) with predicted output $\hat{y}_{ik} \in \{0,1\}$ after "cuting",

$$\begin{array}{rcl} \hat{y}_i & \leftarrow & NN(x_i) \in]0;1[\\ \hat{y}_i & \leftarrow & \left\{ \begin{array}{ll} 1 & \text{if } sigmoid(\hat{y}_i) > \text{s} \\ 0 & \text{if } sigmoid(\hat{y}_i) \leq \text{s} \end{array} \right. . \end{array}$$

- Softmax regression (several output in $]0;1[$ compared between them to choose the predicted $\hat{y}_i$ in $\{1,2,\cdots,K\}$) after "argmax",

$$\begin{array}{rcl} \hat{y}_i & = & \left(\begin{array}{c} \hat{y}_{i1} \\ \hat{y}_{i2} \\ \vdots \\ \hat{y}_{iK} \end{array} \right) \leftarrow NN(x_i) \in]0;1[^K \text{ and } \sum_k \hat{y}_{ik} = 1 \\ \hat{y}_i & \leftarrow & argmax_k \hat{y}_{ik}\,. \end{array}$$

  Here, *argmax* finds the index of the largest probability among the K ones found from the softmax transformation.

If this is not clear to the reader, an external explaination on glm may be required. Also one may consult the python code coming nextafter which proposes a complement of information before coming back to this paragraph if required.

Note also that in some implementations, for classification, the outputs $\hat{y}_i$ or $\hat{y}_{ik}$ are in $\mathbb{R}$, hence the function sigmoid or softmax may be required eventually. This is because the loss may treat directly continous values if these functions are already included, see the table next subsection.

The postprocessing is implemented in "f_test_glmr()" while the training is implemented in "f_train_glmr()". Next, an implementation via pytorch is presented before going deeper with hidden layers just after.

4.1.3 Loss functions in pytorch

A way to maximize the likelihood in pytorch is to minimize a loss as seen before for three cases of distribution, Bernoulli, Multinomial and Gaussian. In the general case, one has to write its own loss function, or just consider another one existing in the python module.

In pytorch, the loss measures the difference between the true target variable y_i and the one found by the network $\hat{y}_i$, by extending the usual Euclidean distance to another measure of comparison more relevant for robust comparison or binary/multi-category/integer comparisons.

Some are extracted from the documentation, "https://pytorch.org/docs/stable/nn.html", at the Loss section. There are equivalent definitions in "torch.nn.functional" too.

Here, the optional argument for the loss "reduction" is put at 'sum', and only a minibatch is in stake, with size B, not the full sample, while $\hat{y}_i = \hat{y}_i(x_i)$ is the output from the network after feeding it with the minibatch of independent variables x_i, such that:

Loss	Distribution(*)	Expression
MSELoss	Gaussian	$\sum_{i=1}^{B}(y_i - \hat{y}_i)^2$
GaussianNLLLoss	Gaussian	$\sum_{i=1}^{B}\left(log(max(var, eps)) + \frac{(y_i-\hat{y}_i)^2}{max(var,eps)}\right) + const$
L1Loss	Laplace	$\sum_{i=1}^{B}\lvert y_i - \hat{y}_i\rvert$
BCELoss	Bernoulli	$\sum_{i=1}^{B} -w_i[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$
BCEWithLogitsLoss	Logistic	$\sum_{i=1}^{B} -w_i[y_i \log \sigma(\hat{y}_i) + (1-y_i)\log(1-\sigma(\hat{y}_i))]$
CrossEntropyLoss	Multinomial	$\sum_{i=1}^{B} -w_k \sum_{k=1}^{K} y_{ik} \log(\frac{\exp(\hat{y}_{ik})}{\sum_l^K \exp(\hat{y}_{il})})$
PoissonNLLLoss	Poisson	$\sum_{i=1}^{B} \exp(\hat{y}_i) - y_i \times \hat{y}_i + 0.5\log(2\pi y_i)$
KLDivLoss	KL divergence	$\sum_{i=1}^{B} y_i(\log \hat{y}_i - \log \hat{y}_i)$

Here KL is for Kullback-Leibler, and for the Poisson loss, the options are log_input at True (as by default) and full at True (instead of False by default). Note that this is the negative loglikelihood as for the Gaussian distribution and the Bernoulli one, but some losses do not have a correspondence with a loglikelihood.

Some loss functions accept weights for the observations which are equal to one in our case herein. Some loss functions are for robust Euclidean one, with HuberLoss and SmoothL1Loss. Some other losses are HingeEmbeddingLoss, SoftMarginLoss, MultiLabelMarginLoss, MultiMarginLoss which are variants for classification, and described in the documentation. The loss to be chosen may be the one with better results.

In the previous chapter, it has been explained how to define a python function for such loss, with two arguments, one for the vector of true target y_i and one for the vector of estimated target $\hat{y}_i$, thus the same idea is required here for other losses. It is possible to consider losses not defined in pytorch such as for a variant of the Poisson distribution for counts.

In this chapter, we are interested on implementing the neural network for regression or classification with the class nn from pytorch. At the same time, we will use the existing loss function:

- For the linear regression, this is the mean square error, with the function "torch.nn.MSELoss()" which computes the usual sum of squares.
- For the logistic regression, this is the cross-entropy with the function "torch.nn.BCEWithLogitsLoss". According to the documentation, this loss combines BCELoss with a sigmoid layer, this is "more numerically stable" than writing its own loss by combining a sigmoid at the output of the network with a BCELoss just after: the final nonlinear transformation is already included in the loss.
- For the softmax regressions, this is the cross-entropy with the function "torch.nn.CrossEntropyLoss()". As for the two classes problem, this loss is more stable by taking as an input just the non normalized output of the network and adding the softmax transformation.

For the BCEWithLogitsLoss, a sigmoidal layer is added automatically, thus there is no need to

add it in the neural network. Similarly for the CrossEntropyLoss, the softmax layer is included in the loss. There is no need to redefine them except for better understanding or creating new losses eventually.

The new python code uses a main class of torch, "torch.nn" which allows to define a function on the weights. For the logistic regression, the sigmoid is already defined at the level of the loss for stability reasons, hence, in both case, regression and classification, we just have to define for the linear models, a linear function for the weights!

```
model = torch.nn.Sequential( torch.nn.Linear(nb_nodes_input,
                                             nb_nodes_output,
                                             bias=True),
                           )
```

As explained before, we have a layer for the input and one for the output, hence with the bias, there are p nodes at the input, and:

- one node for the output for the linear and logistic regressions,

  nb_nodes_input = p

  nb_nodes_output = 1

- more than one for the softmax regression, with K>2,

  nb_nodes_input = p

  nb_nodes_output = K

The bias (intercept) is automatically added (otherwise, we would add one to each independent variable and add a node to the input layer). Note that in the binary case, we just need one node and not two because we can decide of the class with only a threshold from one output.

The parameters are retrieved from the python object model by calling the function "model.parameters()", this object is required by the optimizer.

This leads to finally, with the weights defined implicitely from pytorch, to the call to the function "model.train()" in order to put in "train mode" the neural network with gradient computations.

From now we adopt an approch more generic with python classes and function if possible, in order to avoid to repeat similar python code. This is justified because the fondations have been studied in the previous chapters and until here. This implies that most of the time we just need the next python code for proceeding with regression or classification, with the only change at the level of the layers. We introduce also a class called "Monitor" which will be useful later in the chapter for processing the indicators from the test sample.

4.2 Dataset with nonlinear frontiers

As a baseline, a linear logistic regression is fitted with a new data sample despite that the problem of classification asks for a more complex frontier between the classes. A deep binary regression model is trained in order to improve the classification just after fitting this first model.

The work directory is given from the function.

```
def towdir(s):
    return (str('./datasets_book/'+s))

import deepglmlib.utils as utils
import numpy as np
```

```
import importlib
importlib.reload(utils)
```

```
<module 'deepglmlib.utils' from
'/home/rodolphe/Documents/ARTICLEs/BOOK/deepglmlib/utils.py'>
```

Let load the data file.

```
import numpy as np

xy = np.loadtxt(towdir("./xy_2d_diskandnoise_reglogistic.txt"))
x  = xy[:,[0,1]]
y  = xy[:,2].reshape((xy.shape[0],1))
X  = np.hstack([ np.ones((len(x),1)), x])
n = len(x)
```

Let randomly shuffle the rows for avoiding any row structure before cycling the minibatches. Note that for some files, the classes are not randomly found in the file, they are organized separately, but for the training algorithm, it seems better to have some diversity between the elements of the minibatches. The shuffling may be performed also with the dataloader eventually.

```
ids_random = np.random.permutation(len(y))
np.take(x,ids_random,axis=0,out=x)
np.take(y,ids_random,axis=0,out=y)
np.take(X,ids_random,axis=0,out=X)
x.shape, y.shape, X.shape
```

```
((100, 2), (100, 1), (100, 3))
```

The vector y is kept flat for python, this reduces any risk of automatic "broadcasting" non wanted for algebra with numpy or sklearn for instance.

```
y  = y.ravel()
```

```
import matplotlib.pyplot as plt
import deepglmlib.utils as utils
# true frontier from data generation
```

```
theta = np.linspace(0, 2*np.pi, 20)
x1_circle = 0.45*np.cos(theta)
x2_circle = 0.45*np.sin(theta)
fig, (ax1) = plt.subplots(1, 1, figsize=(5,5))
utils.f_vizu2d_beta(ax1,x[y.ravel()==0,0],x[y.ravel()==1,0],
                    x[y.ravel()==0,1],x[y.ravel()==1,1], [], [],
                    xlim=[min(x[:,0]),max(x[:,0])],
                    ylim=[min(x[:,1]),max(x[:,1])],
                    samplename="Whole sample")
ax1_ = ax1.plot(x1_circle,x2_circle,color='m',label="true frontier")
plt.legend(fancybox=True, framealpha=0.2, loc="lower left")
plt.show()
```

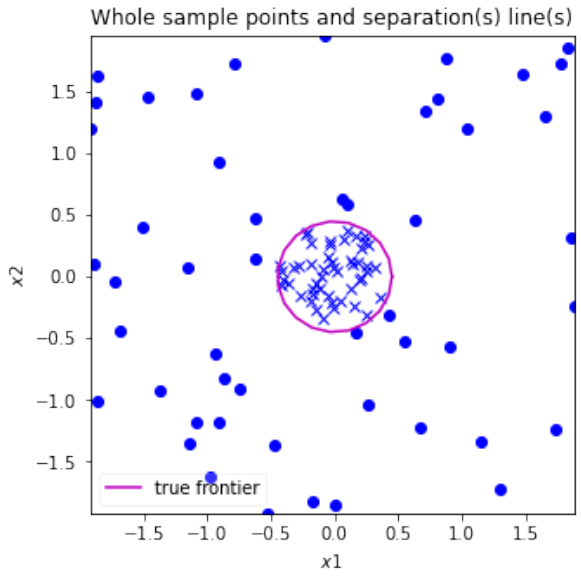

Figure 4.2: Data sample for two classes with non linear frontier

This is the visualization of the artficial dataset in stake with the true frontier between the two classes, a circle for this example. The training of a neural network aims at guessing this unknown frontier from the available data. An evaluation of the quality of the estimated frontier is available with the error rate.

4.3 Train, test and validation subsamples

For prediction purpose, it becomes required to check how would behave the estimated model with new data. A new data sample is a new set of couples (y_i, x_i) different from the given data sample used for the training of the parameters. It is usually observed that checking the loss on the training data is optimistic because the model may overfit for this sample. This means that the model

becomes ideally good for the dataset by learning any of its particularities without being enough general. The problem comes from the fact that a (not enough large) sample is never a true image of the whole population, but only a view of it, with defaults and bias. Training a neural network with these mistaken informations is source of error, and this is even worse when the number of parameters in the model increases as it becomes even more flexible and better able to learn too much all the details and particularities of the train sample.

A real life example is when someone learn to how to make the differences between cars brands but with all their defaults (sunken body, dirt, rust, tires ribs, repaired with parts not from origin for older ones): the human may not be able to recognize new vehicules because of these informations [2]. Another example is for anyone who wants to learn a new activity and is not good at it at the beginning of the learning, hence there is needed more training. To train the biological neural network from the brain, the human has to capture enough data and during an enough amount of time. An approach to check if there is a good understanding is exercices with new contents: they are the new data here where it becomes possible to validate the knowledge for the activity. Without new challenges, this is not sure if the activity is known as expected.

A way to check the quality is to validate the model with a new dataset. This is the "validation set", while the "test set" is used during the training for tuning. The idea of the test set is to have an independent sample different to the training one and to change the (hyper)parameters update in consequence. For a real example, the test set is the exercices which does not count for the final mark and are given during the course for the activity, while the validation set is the exercices from the exam with a mark.

This means that we need to separate the train sample into two parts, one for the training again, and the other one for the test. For enough large samples, a validation sample may be also obtained at this stage. Note that other more elaborated methods exist for insuring a good model, or even dividing the sample (see next chapters). This approach is the more usual way to do before implementing eventually more advanced ones.

In the case when two datasets are already available, we just need to declare two different objects from the class DataLoader, dataloader_train and dataloader_test, from whatever is the place of the data, in memory, on the computer disk or even on a remote server.

The subsamples are drawn randomly without replacement: the test set is found first via sampling because smaller and then substracted from the whole sample to get the train set. This is equivalent to a function in sklearn with more available options, but written in two rows here. Note this is "with suffle" hence the former order of the indexes is not kept. The indexes are the working backbone of processing with a large dataset because this is not possible to replicate or load the full dataset in the computer memory. The problem of some python modules is to not work with indexes but directly with the whole dataset loaded in computer memory, which is not always possible.

```
# Selection of the train and test sample without sklearn
import random as random
```

[2] Note that this can also been seen as additional variables which are useless and prone to induce errors even on the train sample, this is discussed in another chapter.

```
def f_splitIndex(n,percentage=0.25):
    n_test          = int(n*percentage)
    n_train         = int(n-n_test)

    ids_all         = range(n)
    ids_test        = random.sample(ids_all,n_test)
    ids_train       = list(set(ids_all)-set(ids_test))
    return ids_train, ids_test, ids_all

ids_train, ids_test, ids_all = f_splitIndex(n)

# ids_train = np.random.permutation(ids_train)
# ids_test = np.random.permutation(ids_test)

#check if the two sets are complementary
print(len(ids_all), len(ids_train), len(ids_test))
print(set(ids_all)-set(ids_test)-set(ids_train))
```

```
100 75 25
set()
```

As explained in first chapter, we need two samples seen by pytorch through a class "Dataset", one for the training and one for the testing in order to check how behaves the model with new data. We create our two (or three) subsamples, for "train" and "test" (plus eventually "validation" if the test sample is used to tune the training and the sample size is enough large). Then, the iterators with the class "DataLoader", with eventually some shuffling,

```
from torch.utils.data import DataLoader, TensorDataset

dataset = TensorDataset( torch.Tensor(x), torch.Tensor(y) )

def f_splitDataset(dataset,percentage=0.8,batch_size=10):
    n = len(dataset)
    n_train = int(percentage * n)
    n_test = n- n_train
    dataset_train, dataset_test = torch.utils.data.random_split(dataset,
                                                        [n_train, n_test])

    dl_train = DataLoader(dataset_train, batch_size= batch_size,
                          shuffle=False)
    dl_test  = DataLoader(dataset_test, batch_size= batch_size,
                          shuffle=False)
    return dl_train, dl_test, n, n_train, n_test

dl_train, dl_test, n, n_train, n_test = f_splitDataset(dataset)
```

```
n, n_train, n_test
```

```
(100, 80, 20)
```

4.4 Training traditional linear models

In this section, we propose to consider functionalities of pytorch which bring a more advanced training for the parameters of the model. The training loop needs the following python objects:

- a python instance named "model" implementing a pytorch class,
- a pytorch function named "optimizer()" for the gradient step with a learning rate,
- a pytorch function named "loss()" for the objective function to minimize.

We require the module torch.

```
import torch
```

4.4.1 Neural network definition and weights

The neural networks are defined with the class "torch.nn.Module", see "torch.nn.Linear" more specifically. In a generalized model, the linear part is a weighted sum of the independent variables with weights equal to the regression coefficients. This linear transformation is embedded in the class just below which inherits from "torch.nn.Module" in order to define a neural network (nn). Here the net is with fully connected nodes from each x_{ij} for the first layer as the entry of the nn to one unique node (or eventually several nodes for multinomial regression) for the output or last layer. This is written $\sum_j x_{ij} w_{jk}$ for k varying with the index of the component of the target variable. The sigmoid or softmax are included in the loss function such that, only the part before the last transformation coming from the glm is required in this class.

```
class GLMRegression(torch.nn.Module):
    def __init__(self,name,nb_nodes_in,nb_nodes_out):
        super().__init__()
        self.lin = torch.nn.Linear(nb_nodes_in, nb_nodes_out, bias=True)
        self.name = name

    def forward(self, x):
        return self.lin(x)
```

The three neural networks for linear regression, logistic regression and multinomial regression have same architecture, as only the loss function is changed because the link function is included there for more stability. This is similar to glm with a generic definition of the model except that the distribution changes, hence the loss changes too as minus the loglikelihood.

From an instance of the class above, it is possible to extract the weights of the corresponding neural network in order to retrieve the values of the parameters before, during or after training. In the case

of only two layers (input and output), without hidden layers, this is as follows for instance.

The names for the key may change from one modeling to another, thus one must be cautious with this naming. An alternative is to ask for the weight and the bias separately but also only the weights which enter the derivative if some links are chosen constant (see the documentation of pytorch).

```
model     = GLMRegression("LogisticRegression",2,1)

print("model.state_dict().keys()=", model.state_dict().keys())
```

```
model.state_dict().keys()= odict_keys(['lin.weight', 'lin.bias'])
```

The weights of the neural network, here the initial values, are combined as a vector of regression coefficients with the following function. For this case, the naming is used but no naming may be better for more generality. This is written:

```
# def extract_weights_0(model):
#     weight_ = bias_ = None
#     for param_tensor in model.state_dict():
#         if (param_tensor=="0.weight"):
#             weight_= model.state_dict()[param_tensor]
#         if (param_tensor=="0.bias"):
#             bias_= model.state_dict()[param_tensor]
#     return np.append(bias_,weight_)
```

```
def extract_weights_lin(model):
    weight = bias = None
    for param_tensor in model.state_dict():
        if (param_tensor=="lin.weight"):
            weight = model.state_dict()[param_tensor]
        if (param_tensor=="lin.bias"):
            bias = model.state_dict()[param_tensor]
    return np.append(bias, weight)
```

```
beta_init = extract_weights_lin(model)
print(beta_init)
```

```
[ 0.62017655 -0.6241597  -0.6997418 ]
```

Several methods exist for initializing the neural network, as a bad initialization may likely induce a bad convergence, this step must not be forgotten for best results and to avoid to loose time on tuning a model with a bad initialization. In this part, the standard and by default way from pytorch is implemented as it is suitable.

4.4.2 Preparation for the optimization

In this subsection, the dataloader is not only needed, other instances of python classes (from pytorch) are required for the optimization procedure.

Early stopping

An eventual way to improve the training is to compute the accuracy at each epoch, and stop when for the test sample, it begins to drop (or the loss also begins to raise) . Because there is a moment during the training process that the model is well fitted before it begins to learn too much particular details from the train sample.

Stopping at this moment is the decision that we may want called "early stopping". This is the loss which is generally preferred here for checking, the accuracy is less smooth. This is not the only way to monitor the quality of generalization of the model. This explains the proposed class called "MyMonitor" which is able to give a signal to the training loop in order to break its cycle via a stopping rule at the end of each epoch, just below.

Monitoring during the training of the nn

In order to monitor the convergence and compute indicators, first we create a python class which is involved in a passive way during the training. This is optional and our choice for the implementation, alternative ways to do could ask for more advanced python code, because we define just one unique generic python function after that for training the models.

```
class MyMonitor:
    def␣
 ↪__init__(self,model,dataloader_train=None,nbmax_epoqs=1e4,device=None):
        self.model            = model
        self.dataloader_train = dataloader_train
        self.nbmax_epoqs      = nbmax_epoqs
        self.device           = device
    def stop(self,t,loss_train=None):    # not implemented yet
        return False

class MyMonitorTest(MyMonitor):
    def __init__(self,model,loss,
                 dataloader_train=None,dataloader_test=None,
                 nbmax_epoqs=1e4,debug_out=1e2,device=None,
                 transform_yb=None,transform_xb=None):
        super().__init__(model,dataloader_train,nbmax_epoqs,device)
        self.loss            = loss
        self.dataloader_test = dataloader_test
        self.loss_train_s    = np.zeros(nbmax_epoqs)
```

```
        self.loss_test_s     = np.zeros(nbmax_epoqs)
        self.step_test_s     = np.zeros(nbmax_epoqs)
        self.debug_out       = debug_out
        self.transform_yb    = transform_yb
        self.transform_xb    = transform_xb
    def stop(self,t):
        if t%int(self.debug_out)==0 or t==0:
            with torch.no_grad():
                for Xb,yb in iter(self.dataloader_test):
                    if self.device is not None:
                        yb = yb.to(self.device)
                        Xb = Xb.to(self.device)
                        if self.transform_xb is not None:
                            Xb=self.transform_xb(Xb)
                    yhatb = self.model(Xb)
                    yb    = transform_yb(yb,self.model.name,self.device)
                    yhatb = transform_yhatb(yhatb,self.model.name)
                    loss_b = self.loss(yhatb, yb)
                    self.loss_test_s[t] += loss_b
                    self.step_test_s[t] = t
                    #torch.cuda.empty_cache()
                    #gc.collect()
        return False
```

Here, this version finds the value of the loss for the test sample, this computation is asked inside the training loop at the end of each epoch.

List of the required python objects

Let create all the objects which are involved during the training of a neural network:

- the "dataset" for accessing to the sample from the computer memory or computer disk
- the "dataloader" for cycling the minibatches through the dataset object
- the "model" of neural network itself which contains the unknown weights to find
- the "optimizer" for updating the weights with the values of the gradients while eventually updating also the learning rate for an optimal minimization
- the "monitor" for computing the indicators from the test sample and check the convergence for the stopping rule
- the "loss" function to minimize.

We also define the "number of epochs" and the "learning rate" (constant or initial, and eventually a function for its decrease during training). This learning rate remains non automatic and must be choosen by the human most of the time, except for some optimizers available for pytorch and sometimes performing well, not considered here (see the chapter on "autoencoder").

```
debug_out_   = 100
alpha_t_     = 1e-3
nbmax_epoqs_ = 1000
name_model = "LogisticRegression"
# name_model = "LinearRegression"
# name_model = "SoftmaxRegression"

if name_model == "LogisticRegression" :
    loss      = torch.nn.BCEWithLogitsLoss(reduction='sum')
else: #"LinearRegression"
    loss      = torch.nn.MSELoss(reduction='sum')

model     = GLMRegression(name_model,2,1)
model     = model.train()
optimizer = torch.optim.SGD(model.parameters(), lr=alpha_t_, momentum=0.0)
monitor   = MyMonitorTest(model,loss,dl_train,dl_test,
                          nbmax_epoqs_,debug_out_)
```

This optimizer is equivalent to the one used with numpy in the chapter just before, a constant learning rate. The argument "momentum" is able to improve and speed-up the training when well chosen.

4.4.3 Parameters training

The training is still a loop with the minibatches, except that for training the neural network with the loss function, a "backpropagation" is implemented within pytorch which allows faster computations for the gradient when there are hidden layers, this is the automatic computation of the gradient, as seen in the previous chaper.

When defining the function for the main loop, the target variable from the sample y_b or the neural network $\hat{y}_b$ for a current minibatch, may be transformed in order to get the same shape of array for both.

```
def transform_yb(yb,name_model,yhatb=None,device=None):
    if name_model == "LinearRegression"  or name_model == "MLP" \
        or name_model == "LogisticRegression" or name_model == "LMLP" \
        or name_model == "PoissonRegression" or name_model == "PMLP":
        yb = yb.ravel()
    elif name_model == "SoftmaxRegression" or name_model == "SMLP":
        ze = torch.zeros(len(yb), yhatb.shape[1]) # computed before?
        yb =yb.to(torch.long)
        ze[range(ze.shape[0]), yb]=1.0
        yb=ze
        if device is not None:
            yb = yb.to(device)
```

```
    return yb

def transform_yhatb(yhatb,name_model):
    if name_model == "LinearRegression"  or name_model == "MLP" \
        or name_model == "LogisticRegression" or name_model == "LMLP" \
        or name_model == "PoissonRegression" or name_model == "PMLP":
        yhatb = yhatb.ravel()
    return yhatb
```

For the regressions where the output is a simple scalar, the target variable from minibatch is a simple vector with one unique dimension. When the output is a vector such as in the softmax regression, some losses ask for a binarized version of the categorical values, the integers between 1 and K. Ideally, this could be treated before at the level of the dataset or dataloader, but if not, some transformation is required here.

The main loop is defined as follows, it required the python objects defined above.

```
def f_train_glmr(dl_train,model,optimizer,loss,monitor,device=None,
                 printed=1, loss_yy_model = None, transform_Xb=None,
                 transform_yb=None, transform_yhatb=None,
                 update_model=None,):
    if device is not None: model.to(device)
    model.train()
    nbmax_epoqs    = monitor.nbmax_epoqs
    debug_out      = monitor.debug_out
    loss_train_s   = np.zeros(nbmax_epoqs)
    n_sample       = len(dl_train.dataset)
    monistop       = False
    t=0 #epoch
    while True:
        loss_epoch=0
        for b,(Xb,yb) in enumerate(dl_train):
            # data from minibatch into cpu or gpu
            if device is not None: Xb = Xb.to(device)
            if device is not None: yb = yb.to(device)
            # eventually transform the data
            if transform_yb is not None:
                yb = transform_yb(yb,model.name,device)
            if transform_Xb is not None:
                Xb = transform_Xb(Xb)
            yhatb = model(Xb) # predicted target from nn
            if transform_yhatb is not None: # eventually to shape yb
                yhatb = transform_yhatb(yhatb,model.name)
            optimizer.zero_grad()  # gradient set to zero
            if loss_yy_model is None: # loss computation
```

```
            lossb = loss(yhatb, yb)
        else:
            lossb = loss_yy_model(loss(yhatb, yb),model)
        # gradient computation of loss + nn by backpro.
        lossb.backward()                    # backward step
        #
        if update_model==None:
            optimizer.step()                # update of parameters
        else: # own function for update of parameters (one worker)
            # warning: usual pytorch function may be preferred here
            update_model(model,loss,optimizer,device,b,Xb,yb)
        #
        loss_epoch += lossb                             # loss accumulation
    monitor.loss_train_s[t] = loss_epoch                # loss at epoch
    monistop = monitor.stop(t)                          # eventual stop
    # begin printing terminal output for information
    if debug_out>0:
        debug_out=int(debug_out)
        if printed==1:
            if t%debug_out==0 or t==0:
                print("loss=",round(monitor.loss_train_s[t],5),
                      " \t t=",t,"/", nbmax_epoqs,
                      " \t (",round(100*t/nbmax_epoqs,2),"%)")
        if printed==2 and t==0:
                print("loss=",round(monitor.loss_train_s[t],5),
                     " \t t=",t,"/",  nbmax_epoqs,
                     " \t (",round(100*t/nbmax_epoqs,2),"%)")
    # end printing terminal output for information
    t=t+1 # increases epoch counter
    # check if loop must end (max steps or stopping rule true)
    if monistop or t==nbmax_epoqs:
        if printed==2:
            print("loss=",round(monitor.loss_train_s[(t-1)],5),
                  " \t t=",(t),"/", nbmax_epoqs,
                  " \t (",round(100*(t)/nbmax_epoqs,2),"%)")
        break

tmax= t
return monitor.loss_train_s, tmax, monistop
```

This function is generic in order to avoid copying the same code into several functions, thus there must be some time to read it and isolate the important parts:

- The "loop" is infinite with a growing number of epochs, but stops whenever the maximum number of epochs is reached or some stopping rule was defined and became true in the

function monitor.stop() .

- Three "transformations" are defined for yb, Xb and yhatb, for instance insuring that yb and yhatb have same shape for the loss. The transformation for Xb may be a reduction such as a principal compnent analysis or a random projection for instance. But it may be kept in mind that any transformation is costly during the training, and for yb or Xb this could be included in the dataloader in order to speed up the training, while ideally pre-computed and stored in computer disk for loading with the dataset, this avoids any costly transformation during the running time.
- The eventual function "update_model()" for updating the weights from the model differently than from the optimizer, this is a short-cut here.
- The "device" is for computing the gradient and for updating the weights within the cpu or the gpu. The data and the neural network must be put in the same device, otherwise the computation is not possible with the usual current computer architectures, and this would actually decrease the interest for having a fast gpu.
- There is three levels of "printing" the loss during the training, more or less informative: no loss, first and last losses or all losses during training.
- The "monitor" allows to compute the loss for the test sample while the main function just above computes the loss for the train sample. The monitor contains already the variable for keeping the values of the loss during the training. The losses for the test and the train samples are available in the object monitor at the end of the training for graphics and checking the convergence.

The same idea of the algorithm than from the previous chapter is recognized, except that the gradient is always automatic, the gpu is available natively for this python module and a few options are added here for later use. If required, a way to better understand this function is to rewrite it in a simpler way without all the options. This is left as an exercice to the reader: practicing the language is the best way to master the language. After a simplified version, the optional features can be added one by one, in a different way eventually. This proposed example of implementation looks enough flexible for the purpose of almost the whole document.

Training

We have already created an instance from the class of neural network, one for the optimization, one for the monitor. We have defined the loss function, the number of epochs and the learning rate. Now we train the model with the function defined above "f_train_glmr()" which returns the loss, the number of iterations and the state of the object monitor. This leads to:

```
beta0   = extract_weights_lin(model);

loss_train_s,tmax,monistopc  = \
     f_train_glmr(dl_train,model,optimizer,loss,monitor,printed=1,
                  transform_yb=transform_yb,
                  transform_yhatb=transform_yhatb,)
```

```
betahat = extract_weights_lin(model);
print()
print("beta_init=",beta0)
print("beta_hat=",betahat)
```

```
loss= 56.79406          t= 0 / 1000     ( 0.0 %)
loss= 54.82794          t= 100 / 1000           ( 10.0 %)
loss= 54.74583          t= 200 / 1000           ( 20.0 %)
loss= 54.74141          t= 300 / 1000           ( 30.0 %)
loss= 54.74113          t= 400 / 1000           ( 40.0 %)
loss= 54.74111          t= 500 / 1000           ( 50.0 %)
loss= 54.74111          t= 600 / 1000           ( 60.0 %)
loss= 54.74111          t= 700 / 1000           ( 70.0 %)
loss= 54.74111          t= 800 / 1000           ( 80.0 %)
loss= 54.74111          t= 900 / 1000           ( 90.0 %)

beta_init= [ 0.29660398 -0.06700447  0.27895743]
beta_hat= [-0.04427463 -0.31265295 -0.0087152 ]
```

4.4.4 Post-processing

Loss curves after training the nn for test and train samples

We have considered the test step which is actually more a validation step to be more precise even if the literature can missname them sometimes. In the class MyMonitorTest, a loop cycles the test (or validation) sample without updating the weights of the neural network which were found during the training step.

The output is shown.

```
def f_draw_s(x_list,y_list,markercolor_list,xlabel,ylabel_list,title,
             ax,ishow=True,legendloc="best", legendsize=20):

    for (x,y,markercolor,ylabel) in␣
 ↪iter(zip(x_list,y_list,markercolor_list,ylabel_list)):
        ax.plot(x,y, markercolor,label=ylabel)

    ax.set_title(title)
    ax.set_xlabel(xlabel)
    # plt.ylabel(ylabel)
    ax.legend(loc=legendloc, prop={'size': legendsize})
```

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
```

```
f_draw_s([ range(len(loss_train_s)),
           monitor.step_test_s[monitor.loss_test_s>0].astype(int) ],
          [ loss_train_s/n_train,
           monitor.loss_test_s[monitor.loss_test_s>0]/n_test],
          ["b-", "r-"] ,"t",[ "loss train", "loss test"], " ", ax)
```

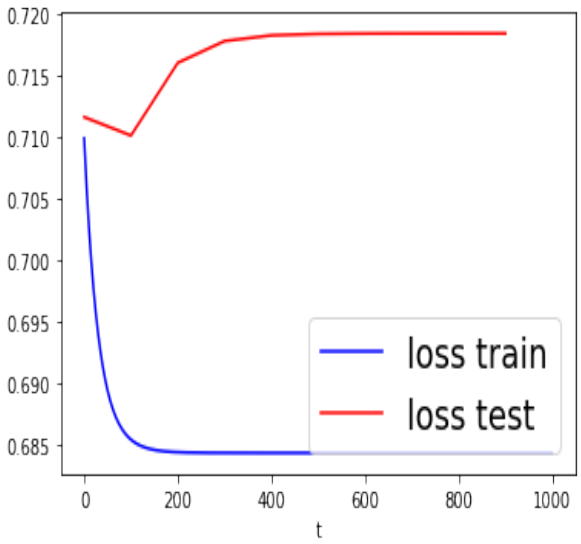

Figure 4.3: Loss function for train and set samples, per epochs

Comparing the two curves allows to decide if the neural network suffers from a generalization problem, with overfitting or underfitting, but also if the learning rate is (at least approximatively) accurate. Typically, the training loss is lower than the test loss because the model was learnt on the train sample such that it better adapts to this sample. Ideally the test loss curve should be no too far at least at the end of the procedure. This is not the case here because a linear model is not enough for nonlinear frontiers between the classes.

Accuracy with pytorch

The neural network has one scalar output for the linear regression and logistic regression while a vector of K outputs for the softmax/multinomial regression. These outputs are obtained from the dataset by feeding the neural network with the independent variables x_i and then processing into probabilities for the classification in order to compute the estimated label $\hat{y}_i$ with a threshold. These final values allow to find the accuracy by comparing with the true values if they exist, otherwise, this is our prediction for the classes when the information is unknown.

The accuracy is computed in this part after finding the labels, it is also discussed an accuracy per class and visualized the frontiers because for this dataset a 2d representation is available. To compute the quality of the regression or classification, we need the estimated labels or predicted

target variables which are obtained in the function below by looping the dataset with minibatches or chunks.

```
def f_test_glmr(model,dataloader,is_yhat=False,threshold=0.5):
    stm = torch.nn.Softmax(dim=1)
    yhat = None
    y = None
    acc = 0
    with torch.no_grad():
        for Xb, yb in iter(dataloader):
            # Forward pass
            yhatb = model(Xb)
            if model.name == "LinearRegression":
                acc += torch.sum((yhatb == yb).float())
            elif model.name == "LogisticRegression" or model.name ==␣
 ↪"LMLP":
                yhatb = (torch.exp(yhatb)/(1+torch.exp(yhatb))>threshold).
 ↪float()
                yb    = yb.reshape((len(yb),1))
                acc += torch.sum((yhatb == yb).float())
            elif model.name == "SoftmaxRegression" or model.name ==␣
 ↪"SMLP":
                yhatb = stm(yhatb)
                yhatb = torch.argmax(yhatb, dim=1)
                acc += torch.sum((yhatb == yb).float())
            if is_yhat:
                if yhat is None:
                    yhat = yhatb
                    y = yb
                else:
                    yhat = np.append(yhat,yhatb)
                    y    = np.append(y,yb)
    return (float)(acc/len(dataloader.dataset)), yhat, y
```

```
acc_train, yhat_train, y_train = f_test_glmr(model,dl_train,True)
acc_test, yhat_test, y_test = f_test_glmr(model,dl_test, True)
print("acc_train_tch=",utils.nprd(acc_train),
      "acc_test_tch=",utils.nprd(acc_test))
```

```
acc_train_tch= 0.587 acc_test_tch= 0.6
```

```
acc_train_tch, prc_train_tch, rcc_train_tch, cm_train_tch = \
    utils.f_metrics_classification(y_train,yhat_train,False)
acc_test_tch, prc_test_tch, rcc_test_tch, cm_test_tch = \
```

```
    utils.f_metrics_classification(y_test,yhat_test,False)

print("acc_train_tch = ",utils.nprd(acc_train_tch),
      "acc_test_tch  = ",utils.nprd(acc_test_tch))
```

```
acc_train_tch =  0.588 acc_test_tch  =  0.6
```

Accuracy per class instead of the usual aggregated one

The aggregated indicator is not so much informative as the conditional one per class, but generally, it is preferred because this is the overall error which is of first interest. For medical questions for instance or asymetric costs related to the classification, the error should be computed w.r.t. each class for a better understanding of the performance of the classifier. An error has not the same consequence if this is a "FP" or a "FN", and also unbalanced classes induce an unfair accuracy for which the larger class biases the rate.

```
 yhat_train.shape, y_train.shape, yhat_test.shape, y_test.shape
```

```
((80,), (80,), (20,), (20,))
```

```
acc_test_tch_0 = np.sum(y_test[y_test==0]==yhat_test[y_test==0])/
 ↪len(y_test[y_test==0])
acc_test_tch_1 = np.sum(y_test[y_test==1]==yhat_test[y_test==1])/
 ↪len(y_test[y_test==1])

print("acc_test_tch_for_yeq1=", utils.nprd(acc_test_tch_1))
print("acc_test_tch_for_yeq0=", utils.nprd(acc_test_tch_0))
```

```
acc_test_tch_for_yeq1= 0.4
acc_test_tch_for_yeq0= 0.8
```

The errors are not symmetrical for this dataset and for the implementation chosen for the training. The two accuracies are not both near 0.5, the half or the expected value which is the worse prediction by chance (one out two). This result teaches several points:

- This underlines that looking at the accuracy is probably not enough for a deeper evaluation of the model, and other indicators are to be checked anyway. The error from one class may be preferred to be minimized for instance, but the threshold may be also chosen differently for a different final error.
- This underlines that a linear model is not always enough and a nonlinear model is sometimes mandatory in classification (and regression). The usual statistical approach for explaining with the model is limited for usual machine learning question suchs as prediction.
- The usual way in statistics is to add some interaction and power terms (two most often) which is related to polynomial regression, but the frontier here is not just a simple nonlinear

version of the line (or hyperplan) such that a polynomial transformation may be not always enough.

Visualization of the frontier between the classes

Let check the solution graphically in order to retrieve the conclusion about the linear model for this dataset. The frontier is checked visually by computing the predicted label $\hat{y}_i$ for the whole plane with discretization, such that we get:

```
def f_plot_2d_boudary_MLP(ax,model,x,y,nbs=100,threshold=0.5):
    x1_s =  np.mgrid[min(x[:,0]):max(x[:,0]):complex(real=0, imag=nbs)]
    x2_s =  np.mgrid[min(x[:,1]):max(x[:,1]):complex(real=0, imag=nbs)]
    x1_x2_s = np.zeros((nbs*nbs,2))

    k=0
    for i in range(nbs):
        for j in range(nbs):
            x1_x2_s[k,:] = (x1_s[i],x2_s[j])
            k+=1
    x1_x2_s = torch.Tensor(x1_x2_s)

    with torch.no_grad():
        y12_s = model(x1_x2_s)
        y12_s = transform_yhatb(y12_s,model.name)
        y12_s = torch.exp(y12_s) / (1+torch.exp(y12_s))
        y12_s = (y12_s > threshold).int().detach().numpy() + 1
    ax.scatter(x1_x2_s[:,0], x1_x2_s[:,1],c=y12_s[:] , cmap=plt.cm.Accent)
    ax.scatter(x[:,0], x[:,1],c=y[:] , cmap=plt.cm.autumn)

    #plt.show()
```

Thus,

```
# %matplotlib inline
import deepglmlib.utils as utils
import matplotlib.pyplot as plt

f_plot_2d_boudary_MLP(plt,model,x,y,300)

plt.plot(x1_circle,x2_circle,color='m',label="true frontier")
plt.legend(fancybox=True, framealpha=0.2, loc="lower left")
plt.show()
```

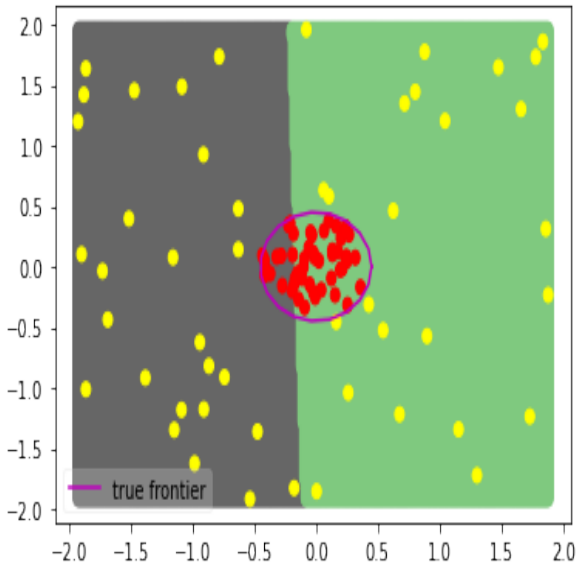

Figure 4.4: Linear frontier from neural network without hidden layers

The resulting frontier is linear as expected, hence the model is misktaken. The frontier becomes nonlinear by introducing in the neural networks hidden layers and nonlinear activation functions.

Such neural network is implemented next in order to solve this issue. This allows to extend glm, otherwise there exists other methods for nonlinear classification with trees and ensembles for instance but out of the scope herein.

4.5 Training deep nonlinear models

In this section, after discovering functionalities of pytorch for a more advanced training of linear models, additional latent layers and nonlinear functions are introduced in the python class for the definition of neural networks. The new resulting model is trained with the same loop embedded in our python function from the section just before and the accuracies are compared.

4.5.1 Hidden layers in deep models

The deep neural model is defined by adding the "activation functions", which are nonlinear functions to apply to the output of a node, and by adding "hidden layers" between the input layer and the output layer. The hidden layers are like latent observations, but they result of a transformation from the layer just before in the neural network.

The output is obtained by a sequential computation from the input layer to the first hidden layer, then from the first hidden layer to the second hidden layer, and so on until the last layer. Hence, each transformation leads to the next layer, until the last layer for the output.

Formal definition of the model

Recall that the input layer is for the features or independent variables x_{ij} plus the bias or intercept 1, with $p+1$ nodes, while the output layer is for the target or dependent variable y_i with 1 or K nodes. This induces to defined intermediate functions $g_\ell()$ such that when counting the last transformation for the final layer $L+1$, we get:

$$f(x_i) = g_{L+1}(g_L(...g_\ell(...g_2(g_1(x_i)))$$

Thus each function tranforms the previous current transformation by applying p_ℓ (= the size of the current layer) times the same function for the hidden layer ℓ, hence to each of its p_ℓ nodes. This is associated to a linear mapping with a linear matrix or weights, such that each transformation (function to each node+mapping to next layer) is summarized as a multidimensional transformation from a space of $p_{\ell-1}$ dimensions to a space of p_ℓ dimensions, when denoting p_0 for the first layer:

$$g_\ell : u \to g_\ell(u)$$

with,

$$\begin{array}{rcl} u & = & (u_1, \cdots, u_{p_{\ell-1}})^T \\ g_\ell(u) & = & (g_{\ell,1}(u), \cdots, g_{\ell,p_\ell}(u)) \, . \end{array}$$

There are several possible functions implemented in deep learning. Three have been already seen, thus they are for instance, - linear for linear regression - sigmoid for binary linear classification - softmax for multicategorical classification. The hidden layers are able to transform nonlinearly the vectors x_i, before the regression or classification which happens at the last layer.

Example of hidden layers with pytorch

The linear transformation for hidden layers remains exactly the same than before. Thus, let have a look at the type of activation functions which can be considered in the class we define next. They are from the list available from "torch.nn" such that, the functions "Sigmoid", "Softmax", "ReLU", "CELU", "LeakyReLU", "PReLU", "Tanh", have each a different shape, while "Dropout" and "AlphaDropout" select a percentage of nodes randomly.

Implementation from module "torch.nn"	# Parameters
Sigmoid()	0
Softmax(dim: Union[int, NoneType] = None)	0 (always dim=1)
ReLU(inplace: bool = False)	1
CELU(alpha: float = 1.0, inplace: bool = False)	2
LeakyReLU(negative_slope: float = 0.01, inplace: bool = False)	2
PReLU(init: float = 0.25)	1
Tanh()	0
Dropout(p: float = 0.5, inplace: bool = False)	2
AlphaDropout(p: float = 0.5, inplace: bool = False)	2

Many other layers or activation functions have been proposed until even recently. Some layers are useful for regularization by dropping some weights i.e. canceling out a percentage of neurons. Some layers are for normalization such as batch normalization and layer normalization in order to standardize each minibatch which allows a better convergence in some cases. Such advanced layers like for image classification (i.e. neighboors layer called convolution layer) or clustering are not considered herein neither, they may be tested as an exercice.

4.5.2 Definition and model training

In this subsection, deep models are illustrated by adding an hidden layer. We are interested on improving the results from linear classication or linear regression when these models are not performing enough well on a dataset. This is the case when the frontier is nonlinear, such as a circle instead of a simple line.

There are several ways to define a network with pytorch, and the one above may be less convenient because one needs to write by hand all the nested transformations coming from all the hidden layers in the "forward" function. A more powerful notation uses a list of layers as an input variable for the constructor of the class.

Generic python class with hidden layers and activation functions

According to the current implementation, we just need to change the class "GLMRegression" into a more general class which is able to handle our new settings. Note that the layers and activation functions can be added in several ways in the class with pytorch, one after the other in the forward function or all at the same time with a function called "sequential()". For more generality we use lists and leave as an exercice the definition of particular models, mostly required for advanced variants of networks such as recurrent ones with time varying variables: they are out of the scope and often defined without such list. Typically in sklearn for instance, for such network, one just has just to define a list with the sizes of the hidden layers, and the list of the activation functions. Pytorch is more flexible hence asks for more details thanks to the additional functionalities, if required.

From here, it will be preferred a class constructor from a list of layers and functions as proposed in pytorch. This avoids to rewrite the full class each time when the structure of the network changes. Recall that the last activation function may be included in the loss function in some cases, such as for sigmoid and softmax, but for instance the sigmoid may appear from an other (hidden) layer in order to induce nonlinearities. The last activation function might be linear here.

```
import torch.nn as nn
class GNLMRegression(nn.Module):
    def __init__(self, name, layers):
        super().__init__()
        self.name = name
        self.layers = layers
        self.net = nn.Sequential(*layers)
```

```
        #torch.nn.init.xavier_uniform_(self.net[1].weight)
    def forward(self, x):
        return self.net(x)
```

The main interest of the parameterization of this class is a higher level access to the pytorch library without dealing with python classes, but this is mostly for documentation of the layers that are in used in this chapter. Otherwise, writing directly this class, and considering our already defined other classes or functions remains possible, at least for a different situation such that advanced image processing.

Anyway, let check how behaves this new class added to our implementation of deep learning with pytorch.

```
nb_nodes_in  = 2
nb_nodes_out = 1
nb_nodes_hid1 = 10

layers = []
layers.append(nn.Linear(nb_nodes_in,nb_nodes_hid1, bias=True))
layers.append(nn.Tanh())
layers.append(nn.Linear(nb_nodes_hid1, nb_nodes_out, bias=False))
```

It has been added in the class, the object net which is required for the function forward.

```
#not used after (included in a python class)
net = nn.Sequential(*layers)
print(net)
```

```
Sequential(
  (0): Linear(in_features=2, out_features=10, bias=True)
  (1): Tanh()
  (2): Linear(in_features=10, out_features=1, bias=False)
)
```

Note that there is also available the class named "ModuleList" for dealing with a list of layers, this is left to the reader as an exercice to access the documentation of pytorch in order to learn more about.

```
print(layers)
```

```
[Linear(in_features=2, out_features=10, bias=True), Tanh(),
Linear(in_features=10, out_features=1, bias=False)]
```

Next we copy the object "layers", otherwise, each time the training is performed from this object, it will update the same weights. The python variable "layers" is not just a description of the model: it contains the weights. For large models, a shallow copy may be preferred, depending on the computer memory.

Thus, from the module named "copy", we use "deepcopy()" for creating our object, and then run the training: the resulting classification should be better than before.

```
import copy
model =  GNLMRegression("LMLP",copy.deepcopy(layers))
nbmax_epoqs=6000
alpha_t= 1e-3
debug_out=100
loss      = torch.nn.BCEWithLogitsLoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=alpha_t, momentum=0.0)
monitor   = MyMonitorTest(model,loss,dl_train,dl_test,
                          nbmax_epoqs,debug_out)
loss_train_s,tmax,monistopc  = f_train_glmr(dl_train,model,
                                optimizer,loss,monitor,
                                device=None,printed=2,
                                loss_yy_model = None,
                                transform_Xb=None,
                                transform_yb=transform_yb,
                                transform_yhatb=transform_yhatb,)
```

```
loss= 56.75431            t= 0 / 6000     ( 0.0 %)
loss= 2.40058    t= 6000 / 6000           ( 100.0 %)
```

The model is saved for later.

```
torch.save(model,towdir("deepmodel_diskandnoise.pth"))
torch.save(model.state_dict(),towdir("deepmodelw_diskandnoise.pth"))
```

Finally the metrics are computed.

```
acc_train, yhat_train, y_train = f_test_glmr(model,dl_train,True)
acc_test, yhat_test, y_test = f_test_glmr(model,dl_test, True)

print("acc_train=",utils.nprd(acc_train,4), " acc_test=",utils.
  ↪nprd(acc_test,4))
```

```
acc_train= 1.0  acc_test= 1.0
```

The model is very accurate for this sample, but the frontier is not perfect according to the true one. Some regularization may improve further the model but a model without error on the population is difficult to achieve with a small sample for the training.

The two losses have same shape and are almost equal, there is no "underfitting" or "overfitting", the "generalization" looks relevant here. This is the opposite to the curves without nonlinearities where dramatic and pathological differences are detected via a visual checking of the two losses.

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
plt = f_draw_s([ range(len(loss_train_s)),
                 monitor.step_test_s[monitor.loss_test_s>0].astype(int) ],
               [ loss_train_s/n_train,
                 monitor.loss_test_s[monitor.loss_test_s>0]/n_test],
               ["b-", "r-"] ,"t",[ "loss train", "loss test"], " ",ax)
```

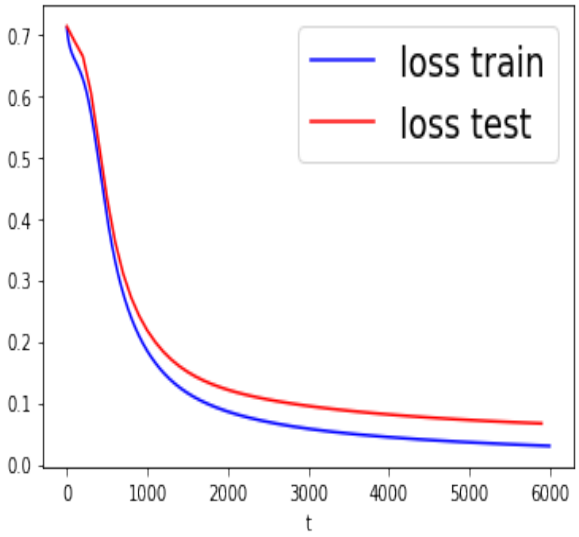

Figure 4.5: Loss function for train and set samples, per epochs

Let check the frontier, the circle is expected to not be retrieved exactly because the separation between the two classes has not this shape for these samples.

```
import deepglmlib.utils as utils
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
f_plot_2d_boudary_MLP(ax,model,x,y,300)
ax.plot(x1_circle,x2_circle,color='m',label="true frontier")
ax.legend(fancybox=True, framealpha=0.2, loc="lower left")
```

```
<matplotlib.legend.Legend at 0x7f18bab2c250>
```

It appears that small samples can be optimistic even on the test sample, but this is not always the case. Anyway, only one train sample and one test sample is not the best approach as explained next chapter. This underlines also why more data is that one wants for training the neural network. This would fill the empty space here which induces the ellipsoidal shape for the frontier instead of a perfect circle.

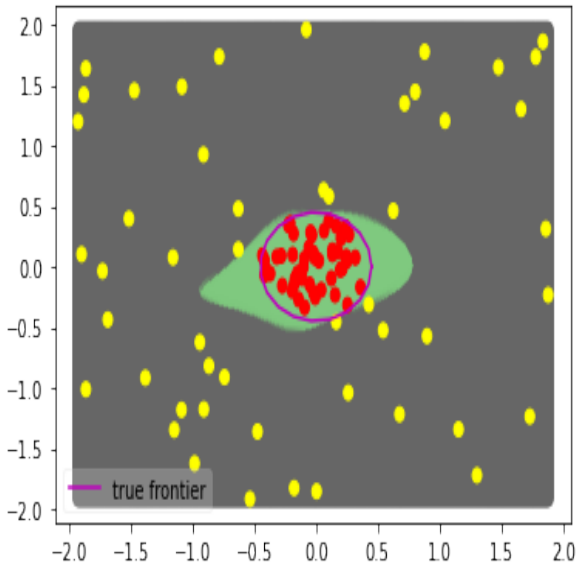

Figure 4.6: Nonlinear frontier from neural network wit hidden layer

There is even for such a small dataset, the evidence of overfit -for some initialization- because the curve may grow after the decrease. Early stopping could be implemented here, or reducing the number of nodes could be even required. Note also with the ReLU, the overfiting seems reduced as this layer is defined for this purpose in the literature by canceling some nodes: a large part of the activation function is null. The number of hidden nodes is chosen high for demonstration purpose, thus may be reduced with an additional hidden layer.

This small example underlines a pitfall of the deep neural networks where a bad training can hurt the results for new incoming data, because the model may learn too much particular details of the empirical distribution without remaining enough general. In statistics, the idea is the occam razor which advises to prefer simpler models to more complex ones, with the usual "bic criterion" in regression, but also the selection of hyperparameters via "cross-validation" for more complex models with an average decision and a sampling of new data.

Next chapter, the model is considered in a more general situation for more complex datasets. Regularization and training will be discussed further.

4.6 Exercices

1. For the dataset of the chapter,

   a) (pytorch) Change the learning rate and the batch size in order to accelerate the training, while keeping a decreasing loss during the loop.

   b) (pytorch) Chech the accuracy for a range of numbers of nodes by drawing a curve of the accuracy versus their number, with and without the ReLU transformation. separately. Is the ReLU transformation useful and how ?

c) (pytorch) Idem for other layers (see the documentation)

d) (pytorch) Do the same for the regression model with an hidden layer, check visually the overfitting. What is observed when the number of nodes grows? Is it related to polynomial regression, why.

e) (pytorch) Test the accuracy and the overfitting with two layers with several number of nodes.

f) (pytorch) Test another approaches in order to avoid overfitting.

g) (sklearn) Fit the logistic regression with sklearn for the dataset in this chapter. How this model compares to the neural network with pytorch.

2. (pytorch) Find a neural network for the classification of the following dataset with XOR structure from the datafile "foursquares_xor.txt" with labels at last column, after vizualizing the points with the classes.

3. (pytorch) Propose a neural network for the classification from these datasets generated from a submodule for datasets of sklearn for the three first ones,

   a) x, y = datasets.make_classification(n_samples=n,n_features=d, n_redundant=0, n_informative=2,random_state=1, n_clusters_per_class=1)

   b) x, y = datasets.make_gaussian_quantiles(n_samples=n, n_features=2, n_classes=g, random_state=42)

   c) x, y = datasets.make_moons(n,noise=.1)

   d) x, y = a two dimensional dataset with three spirals.

   The data samples are stored in the files "2gauss_xy_2d.txt", "gaussq_xy_2d.txt", "2moon_xy_2d.txt", and "threespiral2d.npy" respectively.

4. (pytorch) Test linear and nonlinear regressions or classifications via neural networks and pytorch with the datasets from the exercices in the previous chapter and compare with the logistic regression. Some datasets are more difficult than other: next chapters will explain how improve the choice for the hyperparameters (learning rate, number of neurons in the hidden layers). The iris dataset is a good candidate here with a small size and three classes, for a linear classification.

5. (pytorch) Test the code for the 20000 newgroups (see dataset from sklearn) with only four classes for instance.

6. (pytorch) Test the code for the 60k digits handwritten images mnist with ten classes. (stat) How to reduce the number of weights in the model for images classification.

7. (stat+pytorch) Propose a method in order to improve the training and reduce the classification error for the logistic regression and its nonlinear version.

8. (nn) Propose a graphical representation for the architectures of the neural networks in the chapter, including the exercices.

Chapter 5

Lasso selection for (deep) glm

In the previous chapters training linear and deep models with pytorch has been presented with the loss, the related maximum likelihood and algorithms. For improving the models, the number of parameters may be reduced for instance by penalization as explained below. When some variables are useless or the model over-parameterized, it improves dramatically the accuracy or the mse for the test sample as illustrated with an example. An hyperparameter is set for the strength of the regularization, its value is found with a method of cross-validation in order to choose the value among a set or an interval.

5.1 Brief recall on regularized regression

Instead of constructing new variables in a new reduced space, some parameters are just removed from the equation because they are useless. More precisely, the approch considered here is called "lasso", and the penalization is the "L_1 distance", the sum of the absolute values of the weights. This is very competitive in practice, the added penalty term to the loss is also even able to reduce the values of the weights when they are not just cancel out with very small values for the less relevant. Note that using the "L_2 distance" instead leads to a "ridge model" with also a reduction but not a removal of the weights, thus often less efficient. This is implemented by default in sklearn for regression models as a first step against overfitting. Associating the two distances is called the "elastic net model", which generalizes both with a linear combination of the lasso model and the ridge model. This also means that one just needs to minimize a new loss function by adding a simple term to the former one, which explains why so much applications of this idea has been published by scientists. But it has been also invented analytical theories in order to explain and train in a better way the resulting loss.

More formally, the approach is described as follows for the linear case, with different possible expressions for ℓ and R with $\hat{y}_i(\beta) = \beta^T x_i$, otherwise for a deep model with $\hat{y}_i(\beta, W) = \beta^T g_L(...g_\ell(...g_2(g_1(x_i))$ for instance. Instead of directly minimizing the objective function or loss $\ell()$, a penalization $R()$ is added in order to avoid inflating values during the training of the components of the parameters vector, see the documentation of "sklearn" for instance.

$$\begin{aligned}\ell_\lambda(\beta) &= \ell(\beta)+\lambda R(\beta)\\ &= \frac{1}{n}\sum_{i=1}^{n} L(y_i,\hat{y}_i(\beta))+\lambda R(\beta)\,.\end{aligned}$$

The first term is the usual loss function $\ell()$, they are defined for regression, the least-squares or the robust least-squares (Huber) for instance. Otherwise there is also defined other ones for classification, the logistic regression for instance.

The second term allows to decrease the overfitting from the regression and improve the generalization. For the regularization term $R = R(\beta)$, they are, with ρ in $[0;1]$ for the parameter of linear combination.

the L_2 norm (ridge)	$R_2 = \frac{1}{p}\sum_j \lvert\beta_j\rvert^2$
the L_1 norm (lasso)	$R_1 = \frac{1}{p}\sum_j \lvert\beta_j\rvert$
the Elastic Net	$R_e = \frac{\rho}{2}R_2 + (1-\rho)R_1$

This is visualized below, the lasso penalty induces smaller values than the ridge one, for two components β_1 and β_2 for which the solution lies on a circle or a square respectively, while the solution for elastic net is a combination of both.

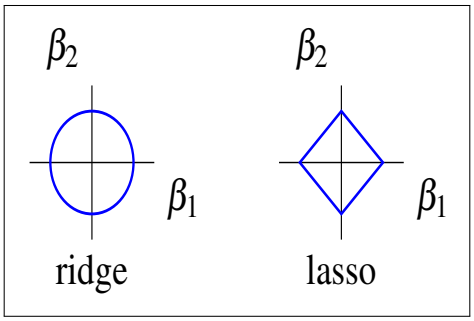

Figure 5.1: Solutions for each penalty function alone with $p = 2$.

Note also a probabilistic interpretation of these penalizations is to add a prior on the parameters, a Gaussian density for the ridge penalty, and a Laplace density for the lasso one, say $\pi(\beta|\sigma^2) = \prod_{j=1}^{p} \frac{\lambda}{2\sqrt{v}} \exp(-\frac{\lambda|\beta_j|}{2\sqrt{v}})$, with $v > 0$, but this is out of the scope herein. Bayesian inference remains possible with pytorch[1] but ask for more advanced implementations. Note that kind of model is often presented as able to find a relevant value for the regularization parameters with only the train sample, but involving a test sample may perform better actually.

Finally, this induces to add to the previous derivative $\nabla\ell$, λ as a product with the derivative of $R(\beta)$, say $\lambda\nabla R$. With "sklearn" some functions are available to perform this new optimization and even find the best value for the new unknown parameter λ, called "hyperparameter" in general,

[1] See the external python module "pyro" with such models for instance.

in order to get the best generalization as possible from the available sample. When the hyperparameter is very large, only the penalization is minimized, hence the weights are all zero, while when it becomes to decrease, the former loss is also minimized such that the previous solution is regularized with this additional term.

This is implemented with pytorch in this whole chapter, just nextafter.

5.2 Dataset with uninformative variables

The work directory is given from the function.

```
def towdir(s):
    return (str('./datasets_book/'+s))

import deepglmlib.utils as utils
import numpy as np
```

```
import importlib
importlib.reload(utils)
```

It is imported the functions from previous chapters, all saved in a python file as a custom module.

An artificial dataset is generated in order to perform the comparisons. It is added to a design matrix some useless columns for the variables which are not meaningful, for testing the model selection. The columns are normalized. The datasets for train and test samples are saved for later use into separated files in a compressed numpy format.

The dataset generated is loaded in order to check the contents.

```
import numpy as np
for filenamefix in {'','_train','_test'}:
    with open("".join([towdir('x_y'),filenamefix,'_450d_lasso.npz']),
              'rb') as f:
        xy = np.load(f)
        namex_ = "".join(['x',filenamefix]); x_ = xy[namex_]
        namey_ = "".join(['y',filenamefix]); y_ = xy[namey_]
        print(namex_, x_.shape, namey_, y_.shape)
```

```
x (450, 255) y (450, 1)
x_train (338, 255) y_train (338, 1)
x_test (112, 255) y_test (112, 1)
```

Let train the lasso model with pytorch as a neural network and compare with the regression without penalization. For lasso, the computer program is very similar from the previous chapter, except that the loss function is altered, and the optimization becomes more difficult. This supposes to add

the penalization at the level of the update of the weights of the neural network. This is performed next with the module torch after training the regular regression.

For the dataloader, the previous functions are already available. The train and set samples are already separated here and available from the computer files.

```
from torch.utils.data import DataLoader, TensorDataset
import torch
x_y_train_450d_lasso = np.load(towdir('./x_y_train_450d_lasso.npz'))
x_y_test_450d_lasso  = np.load(towdir('./x_y_test_450d_lasso.npz'))
x_train              = x_y_train_450d_lasso['x_train']
y_train              = x_y_train_450d_lasso['y_train']
x_test               = x_y_test_450d_lasso['x_test']
y_test               = x_y_test_450d_lasso['y_test']
dataset_train = TensorDataset(
                         torch.from_numpy(x_train.astype(np.float32)),
                         torch.from_numpy(y_train.astype(np.float32)) )
dataset_test  = TensorDataset(
                         torch.from_numpy(x_test.astype(np.float32)),
                         torch.from_numpy(y_test.astype(np.float32)) )
```

```
n_train, p_train = x_train.shape
n_test, p_test   = x_test.shape
```

The argument "shuffle" is at False, this supposes that the suffling was performed before once, and is not repeated during the training.

```
import torch
from torch.utils.data import TensorDataset, DataLoader
dl_train = DataLoader(dataset_train,shuffle=False,batch_size=10)
dl_test  = DataLoader(dataset_test,shuffle=False,batch_size=10)
```

Note that "num_workers" for parallel processing with several cpu cores and "pin_memory" for better management of the memory buffer are two options worth to try for some computers.

```
print("cuda.is_available()     = ", torch.cuda.is_available())
print("cuda.get_device_name(0) = ",torch.cuda.get_device_name(0))
```

```
cuda.is_available()     =  True
cuda.get_device_name(0) =  NVIDIA GeForce 940MX
```

The training function is now implemented with the gpu device for faster training, when available.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

5.3 Multiple regression without lasso

Let train the model without regularization for this new dataset, even if some independent variables are just noise. This is likely to be the case with real data too. Useless variables detected by the lasso model may be also just correlated to useful ones. In both cases, the usual regression is expected to not perform well because its lack of robustness.

5.3.1 Training

```
import torch.nn as nn
import copy
px = p_train #px = dataset_train.x.shape[1]
layers_regress = []
layers_regress.append(nn.Linear(px,1,bias=True))

model =  utils.GNLMRegression("LinearRegression",
                        copy.deepcopy(layers_regress))
model.train()
nbmax_epoqs = 500
alpha_t     = 1e-4
debug_out   = 5
loss      = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=alpha_t, momentum=0.0)
monitor   = utils.MyMonitorTest(model,loss,dl_train,dl_test,nbmax_epoqs,
                                debug_out,device=device)
```

```
loss_train_s,tmax,monistopc  = utils.f_train_glmr(dl_train,model,
                           optimizer,loss,monitor,device=device,printed=2)
```

```
loss= 1737.45862          t= 0 / 500      ( 0.0 %)
loss= 3.22817    t= 500 / 500    ( 100.0 %)
```

5.3.2 Post-processing

According to the graphic for the losses, the model looks fine with a convergence of the algorithm and even no issue for the generalization with new data. But from the generation process it is clear that this model is likely to be mistaken and to need improvements. This is confirmed below via numerical indicators based on the predictions $\hat{y}_i$. For computing these predictions from the model, a function is defined below.

```
def f_get_yhat(model,dl_,device=None,f_transform_x=None):
    with torch.no_grad():
        i_ae = []
        t=0
```

```
        for step, (xb, yb) in enumerate(dl_):
            if device is not None:
                xb = xb.to(device)
                if f_transform_x is not None:
                    xb = f_transform_x(xb)
                yhat_b = model(xb).detach().cpu()
            else:
                if f_transform_x is not None:
                    xb = f_transform_x(xb)
                yhat_b = model(xb).detach().numpy()
            if t == 0:
                yhat = yhat_b
                y = yb.detach().numpy()
            else:
                yhat = np.append(yhat, yhat_b, axis=0)
                y    = np.append(y,yb.detach().numpy())
            t+=1
    return yhat.squeeze(), y.squeeze()
```

Here, "device" is an argument in order to be able to proceed at the level of the gpu. In pytorch, there are several ways to use the gpu: load all the dataset in the gpu when this is possible or load each minibatch during the training loop in the gpu. Here the model must be also in the gpu in order to access the data. The command is just ".to(device)" where device is an object depending on the computer hosting pytorch. By the way, if required, an optional argument for a function "f_transform_x()" was added but it may be preferred the dataloader for such transformation like in image processing. This leads to.

```
yhat_train, y_train = f_get_yhat(model,dl_train,device=device)
yhat_test, y_test   = f_get_yhat(model,dl_test,device=device)

yhat_train.shape, yhat_test.shape, y_train.shape, y_test.shape
```

```
((338,), (112,), (338,), (112,))
```

```
mse_test, r2_test  = utils.f_metrics_regression(y_test,yhat_test,False)
cor_test           = np.corrcoef(y_test,yhat_test)[0,1]

print("mse_test = ", utils.nprd(mse_test))
print("r2_test  = ", utils.nprd(r2_test))
print("cor_test = ", utils.nprd(cor_test))
```

```
mse_test =  0.205
r2_test  =  0.587
cor_test =  0.84
```

```
mse_train, r2_train = utils.f_metrics_regression(y_train,yhat_train,False)
cor_train           = np.corrcoef(y_train,yhat_train)[0,1]

print("mse_train = ", utils.nprd(mse_train))
print("r2_train  = ", utils.nprd(r2_train))
print("cor_train = ", utils.nprd(cor_train))
```

```
mse_train =  0.009
r2_train  =  0.979
cor_train =  0.99
```

The model performs very well on the train sample because this is a regression line fitted with data generated after a nested model but the model is less convincing for the test sample which induces a problem of overfitting. The poor generalization of the model is corrected by adding some regularization (removing nodes and weights in the neural network which are not needed), as explained next after.

5.4 Multiple regression with lasso

Let add the regularization. A first approximation of the optimal value of the hyperparameter λ was found by trying several values. This is not shown here, see next part for an automatic setting, the reader may try its own candidate values.

```
lambda_l1 = 0.020 # lambda_l1 = 0.006
```

The loss is altered by adding the absolute value of the weights to the previous loss function, the mse. By this way the penalization appears in the update rule at each optimization step during the training, like if this was a new loss function. This is the role of the following function which is an argument of the generic training procedure. Thus, the training loop keeps the same, it is just introduced a wrapping function which adds the regularization to the previous one.

```
def loss_yy_model(lossb,model):
    lossb_b_rg = lossb
    lossb_b_l1 = (torch.abs(list(model.parameters())[0])+0.000001).sum()
    loss_b     = lossb_b_rg + lambda_l1 * lossb_b_l1
    return loss_b
```

This leads to new weights or regression coefficients after training as follows.

```
nbmax_epoqs = 350
alpha_t     = 0.001
model     = utils.GNLMRegression("LinearRegression",
                                 copy.deepcopy(layers_regress))
loss      = torch.nn.MSELoss(reduction='mean')
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=alpha_t, momentum=0.0)
monitor   = utils.MyMonitorTest(model,loss,dl_train,dl_test,
                                nbmax_epoqs,debug_out,device)
```

The computer code is nearly the same than without regularization, only the function for the loss is implemented instead of remaining at "None". This allows the training without any deep knowledge of pytorch, but also the perspective to change a generic code for the purpose of the reader, such as for instance, any other regularization functions, loss functions, optimizer functions, etc. without requiring to reinvent the wheel. Note that an additional generic python function with less arguments is written in a next chapter with also an additional member from the generalized linear models for discrete regressions. The training is launched.

```
loss_train_s,tmax,monistopc = \
    utils.f_train_glmr(dl_train,model,optimizer,loss,
                   monitor,device=device,
                   loss_yy_model=loss_yy_model,printed=2)
```

```
loss= 235.8985            t= 0 / 350      ( 0.0 %)
loss= 2.1845     t= 350 / 350    ( 100.0 %)
```

5.4.1 Post-processing and mean square error

For the usual multiple regression, the mse on the test sample was found not enough small in comparison with the one from training: this suggests that the first model cannot perform very well with new data. Let check the indicators for the new model with a penalization. The new model leads to new mse for the train and the test samples as follows.

```
yhat_train_l1, y_train_l1 = f_get_yhat(model.cpu(),dl_train)
yhat_test_l1, y_test_l1   = f_get_yhat(model.cpu(),dl_test)

y_train_l1     = y_train_l1.squeeze()
y_test_l1      = y_test_l1.squeeze()
yhat_train_l1  = yhat_train_l1.squeeze()
yhat_test_l1   = yhat_test_l1.squeeze()
```

```
_,_, = utils.f_metrics_regression(y_test_l1,yhat_test_l1,True)
_,_, = utils.f_metrics_regression(y_train_l1,yhat_train_l1,True)
```

```
MSE =  0.03
R2  =  0.93
MSE =  0.03
R2  =  0.94
```

The new model with the regularization performs better than the one without. This confirms the needs for the penalizing term or at least some regularization in order to improve the usual model.

5.4.2 Interpretation of the obtained model

The following graphic with the predicted target variable against the true target variable for both samples, test and train, confirms a higher error for the test sample when the penalization is removed.

```
import matplotlib.pyplot as plt

def f_plot_save_regression_yyhat(y_tt, yhat_tt, label_yy, filename):
    fig, ax = plt.subplots()
    utils.f_draw_s(y_tt, yhat_tt, ["b+", "ro"] ,"ytrain",
                   label_yy, " ", ishow=False, ax = ax)
    plt.show()
    plt.savefig(filename)
    plt.close()

f_plot_save_regression_yyhat([ y_train, y_test ],
                             [ yhat_train, yhat_test],
                             [ "yhat_train", "yhat_test"],
                      towdir("pytorch_nn_450d_lasso_without_l1.png"))

f_plot_save_regression_yyhat([ y_train, y_test ],
                             [ yhat_train_l1, yhat_test_l1],
                             [ "yhat_train_l1", "yhat_test_l1"],
                      towdir("pytorch_nn_450d_lasso_with_l1.png"))
```

This leads to the two following graphics. This is on the left without lasso and on the right with lasso, there is a huge improvement for the test sample which makes almost mandatory such kind of procedure for prediction.

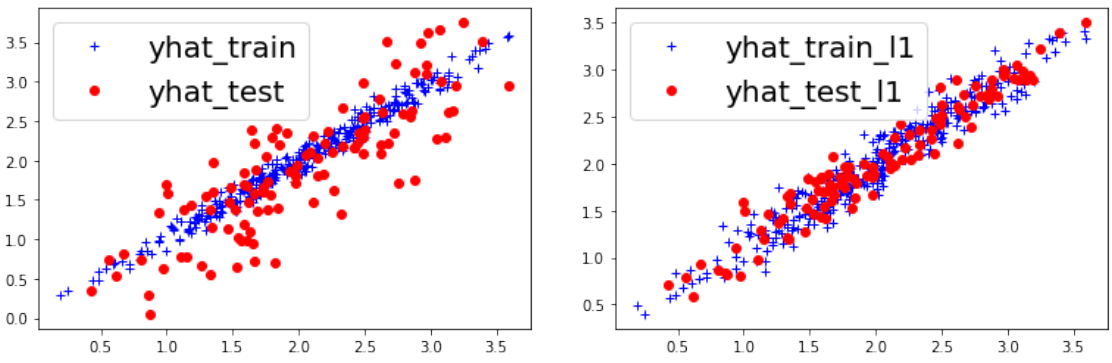

Figure 5.2: Scatterplot of y_i and $\hat{y}_i$: without L_1 (left) and with L_1 (right)

The weights or regression coefficients $\hat{\beta}_j$ come with some variance and variability, this explains why it is not better retrieved the true target variable from the model. The penalization improves the

generalization for new data by reducing the quality of the prediction for the training set. Adding the regularization performs quite well here, and without this term the model appears not useful for new data. Retrieving same values for the indicators for train and test sample confirms the idea of a model able to generalize and to perform exactly the same for any data, as if it was estimated from the whole population, but this is not always the case actually.

The training algorithm may be not optimal above for the optimization with a nonlinear regularization term: for the lasso regression more advanced inferential procedures may be preferred for faster training. But they ask for specialized python programs (see the literature on lasso for such implementations) while the presented solution is via numerical derivatives.

5.4.3 Other regularization methods

The lasso regression may be improved for real data by alternative or more advanced models, in particular the ridge regression assumes another penalization with the term $\sum_j |\beta_j|^2$. Elastic net, with the term $\sum_j \frac{\rho}{2}|\beta_j|^2 + (1-\rho)|\beta_j|$, is expected to share the good properties of both: it is worth to try, this depends on the data. There is also the power from the q-norm with the term $\sum_j \rho_j |\beta_j|^q$, or the clustered one with the term $\sum_k \sum_{j \in J_k} \rho_k |\beta_j|^q$ with some structure on the regression coefficients or the neural network parameters, for instance per layer. Here the quantities ρ_j are either for an adaptative penalization or to just focus on some parameters more than other ones. Some of these different ways to regularize are considered in the exercices, but one may keep in mind that for neural networks, specialized hidden layers are often now available for this purpose and more advanced methods were proposed recently. For instance the "dropout layer", the activation function "relu" (and their smooth variants) and the "batch normalization" provide some kind of regularizations. Some most recent ways are in real competition against the lasso method in order to remove the weights which are not wanted, for a parsimonious and lighter model.

Next after, it is explained how to find in an automatic way the hyperparameters such as for the regularization. It is also retrieved if the proposed value for lambda_l1 in the lasso, 0.02, was actually suitable for this dataset via "cross-validation" with pytorch.

5.5 Setting of the hyperparameters

Hyperparameters are the parameters which are not trained via the gradient, they do not enter the set of weights of the neural network but are related to its structure or its training: "learning rate", "batch size", "number of hidden layers", "number of neurons per layers", "regularizing constant", etc. For the setting of the values of the hyperparameters, the best common advice according to the literature (and any serious practical experience) would be to search for these values before any automatic procedure in order to reduce the "search space". But in the other hand, this could reduce the capability to reach the optimal one in the case of a bad first choice, due to human error.

The cross-validation next after should be considered for this task, because only one test sample is not enough. Using only one test sample is prone to optimistic or pessimistic decisions and a biased result when comparing between two candidate values for an hyperparameter. But cross-validation should be used only when the hyperparameters were not find manually because this is very time

consuming and suppose to repeat the training 5 or 10 times for all the possible candidate values for each hyperparameters. This is times the product of the number of eventual values for each, which becomes quickly a huge integer value when the number of hyperparameters $\#hyper$ grows. If there are 15 candidate values for each hyperparameter, this is around:

$$\text{"Total number of models fitted for cross-validation"} \approx 7.5 \times 15^{\#hyper}.$$

A saving time would be to prefer these approaches around a good value found before with a human search via several empirical tries or a first naive choice of a few values. Otherwise the train and test samples need to be selected very cautiously in order to be similar, via stratified sampling with enough strata and eventually fine quantization or clustering of the data space. Note that in the documentation of sklearn several cross-validation procedures (and lasso algorithms) are available. Here only the more usual procedure is considered.

5.5.1 How to improve the learning

The final model which generalizes better is also with an optimal learning rate because minimizing the loss, even with conditions, is the first purpose before all. There exists now many variant ways for updating the learning rate during the training loop with minibatches. Herein, we have considered a constant learning rate because for convex problems or with few and sometimes many nonlinearities, such approach is often enough. When the objective function is more complicated another learning rate schedule may be preferred. Hence, two questions are risen here: how to select the better function for updating the learning rate and how to guess its better initial value.

- For the varying learning rate, $\alpha_t = \alpha(t)$ as an alternative to a simple constant, this is a decreasing function (linear or exponential) which needs the setting of some parameters (speed of the decrease, initial value), see "torch.optim.lr_scheduler". Because the training updates are with $\alpha_t \nabla \ell$, the risk with a bad setting for the varying learning rate is to cancel out the gradient update before that the gradient $\nabla \ell$ is null with α_t null before that the minimum is reached. This is another good reason to check the gradient at the end of the training. An alternative approach is a specialized optimizer, such as the "Adam optimizer", which is a powerful candidate to replace the previous gradient descent. With this kind of optimizer, the learning rate is updated automatically according to the objective function and its gradient, see the documentation of pytorch for the list of such optimizers.
- For the initial learning rate, α_0, trying several values (among powers of 10), say 10^s for $s \in \{-5,-4,-3,-2,-1,0,1,2,3\}$ and their between values, remains possible but for very large datasets, the training might be stopped after a few steps for making it possible. Another way to do is choose a first high value for the learning rate, and then try a smaller one while checking if this improves the final results. By dividing at each time by two or ten, this should provide some good value at the end. But there is since a few years ago, a more automatic approch called "Cyclical Learning Rates" which is implemented for pytorch, sometimes able to provide the wanted value with a graphical proof.

A too fast training may also suppose a too large learning rate while a too slow one may suppose a too small learning rate: for constant values or varying ones. At this moment, there is a clear

understanding that the training of neural network models ask for some manual settings which may be cumbersome for non experts, as there seems to be missing the real receip which works all the time. Recent automatic implementations are based on bayesian inference, but are time consuming, see "automl" in the literature for instance, and the last section of the chapter as examples.

5.5.2 Grid search for optimal hyperparameters

Several hyperparameters require an a priori value: learning rate α, learning rate decay, momentum β, other hyperparameters for some learning rate function (in Adam, etc), number of hidden layers and their number of nodes, and also the size of the minibatch for which the learning rate may need to change.

Two main approaches are available (beside bayesian and genetic ones not considered here), the grid one and the sampling one. Note that if the sampling is not adaptative by looking better regions in some intervals, the two approaches are almost equal, once the sample values are obtained before in some array. They are called in sklearn tuning by "grid-search" and tuning by "randomization-search" (see also for an implementation "Ray Tune" available with pytorch). A set of values is avaible in order to try the model on each value of the set, and compare a resulting indicator of quality in order to keep the best value. The grid seach is also eventually replaced by some chosen set of candidate value before training, instead of dividing an interval into values with equal distance for two neiboors as the grid supposes. Some python libraries are dedicated ("optuna") or have functionalities (sklearn) for such settings.

As an illutration, let revise the example with the regression and useless variables. The cross-validation via grid search is implemented for pytorch. This is the whole data sample which is loaded (instead of the test and train samples from two files, because the cross-validation takes its own train and test subsamples, several times, from the available data in order to propose an average and robust decision: the resulting indicator is not depending anymore of some biased choice for the test and train samples. This is as follows.

```
from torch.utils.data import DataLoader, TensorDataset
import torch
x_y_all_450d_lasso=np.load(towdir('./x_y_450d_lasso.npz'))
x           = x_y_all_450d_lasso['x']
y           = x_y_all_450d_lasso['y']
dataset = TensorDataset( torch.from_numpy(x.astype(np.float32)),
                         torch.from_numpy(y.astype(np.float32)) )
```

The cross-validation (cv) provides nearly unbiased indicators, while the same code than before is used except that the train and test samples are not unique anymore. For the "k-fold cv", k train samples and k test samples are required, with often $k = 5$ or $k = 10$. The average of the k resulting indicators of quality replaces the previous unique value. The k samples are found by partitioning the whole sample into k equal parts. With these k subsamples, k pairs of training and test samples are obtained by removing one of the k parts from the sample, each one after the other one. At each removal, the removed part is the test sample while the remaining $(k-1)$ parts become the train sample.

First the k samples of the cross-validation for testing and training are found from the available whole sample. They are generated via the new python function just below which extends the previous one, with the indexes for each fold and each subsample (test and train). These indexes are then used to fed two dataloader of pytorch in order to cycle the minibatches from the two subsamples.

```
#n_all, d_all = dataset.x.shape
n_all, d_all = x.shape
print( n_all, d_all )
```

```
450 255
```

The number of folds is 5 here.

```
k_fold = 5
```

Thus, five pairs of train and test samples are put in a dictionary as follows:

```
import numpy.random as rd

def f_idx_traintest_kfolds(n,k_fold=5,shuffle = True):
    if not shuffle : idx_all = range(n)
    if shuffle     : idx_all = rd.permutation(range(n))
    idx_s = dict()
    for k,idx_test in enumerate(np.array_split(idx_all,k_fold)):
        idx_train = [e for e in idx_all if e not in idx_test]
        idx_s[str(k)] = dict({"train":np.asarray(idx_train),
                              "test":np.asarray(idx_test)})
    return idx_s
```

```
idx_s = f_idx_traintest_kfolds(n_all,k_fold=k_fold)
```

Note that some functions are available in python modules such that sklearn (with the function KFold for instance) for a similar result but with more options and alternative sampling strategies for the selection, while the one presented is the most common. In the proposed function just above, the result is coded with a python variable of type dictionnary (dict) with the fold number minus one as the key, and two corresponding values as two lists for the two samples, and names "train" and "test". For this dataset, the sizes for each fold from the dictionnary is given just after, with a verification of the sets:

The sizes for each fold are as follows:

```
# for key in idx_s.keys():
#     n_train = len(idx_s[key]["train"])
#     n_test  = len(idx_s[key]["test"])
#     print("Fold", key," --> n_train =", n_train, " n_test =", n_test)
```

```
set_idx_test_s = []
for key in idx_s.keys():
    print("fold=", key,
          " tr",idx_s[key]["train"].shape,
          " tt",idx_s[key]["test"].shape,
          "all-(train+test)= ",
          set(range(n_all)) - set(idx_s[key]["train"]) - \
           set(idx_s[key]["test"]) )
    set_idx_test_s.append(idx_s[key]["test"])

print("check test set from validation sets are complementary: ", end="")
set_idx_check = set(range(n_all))
for set_idx_test in iter(set_idx_test_s):
    set_idx_check = set_idx_check - set(set_idx_test)
print(set_idx_check)
```

```
fold= 0  tr (360,)  tt (90,) all-(train+test)=  set()
fold= 1  tr (360,)  tt (90,) all-(train+test)=  set()
fold= 2  tr (360,)  tt (90,) all-(train+test)=  set()
fold= 3  tr (360,)  tt (90,) all-(train+test)=  set()
fold= 4  tr (360,)  tt (90,) all-(train+test)=  set()
check test set from validation sets are complementary: set()
```

For a number of folds which does not divide exactly the sample size, all the folds have same size, except the last one. This is different of sklearn with more balanced subsample sizes, hence this is better to chek these sizes anyway. Thus, the folds are in motion next by just repeating several times the previous usual strategy of training: one sample for model fitting and one for the evaluation of the quality. And, at the end averaging.

The training is a loop over the folds in order to gather the metrics and aggregate them. The function is defined in the file "utils.py" and has the following header and return list.

```
#optimizer implemented is sgd() one
def f_reg_l1_nn_cv(idx_s,namefile_s,dataset,model_,loss_,batch_size,
    alpha_t=1e-5,nbmax_epoqs=5000, debug_out=50, device=None,
    transform_yb=None, transform_yhatb=None, transform_Xb=None,
    loss_yy_model=None, printed=2, hyperparameter_to_print=None):
    [...]
    return  loss_train_s_s, loss_test_s_s, yhat_train_s, \
            y_train_s, yhat_test_s, y_test_s
```

The function takes as argument either the sets in "idx_s" for the cross-validation plus the dataset either the filenames in hdf5 format in "namefile_s". After training, the output is the list from each fold from the function "utils.f_train_glmr", as a dataloader is created for each couple of train and set samples. This allows after to compute the averages and compare the results from each fold.

The objects called model, loss, optimizer and monitor are copied for each fold with this function, thus they are created first but the initial python objects are not altered.

The parameters for the learning are as follows.

```
alpha_t     = 0.001
lambda_l1   = 0.02
nbmax_epoqs = 350
batch_size  = 10
```

The call to the function comes after creating the required python objects:

```
import torch.nn as nn
import copy
# nn architecture and initialization kept for all folds
layers_regress = []
layers_regress.append(nn.Linear(px,1,bias=True))
# copied at each fold within the function for cross-validation
model_ = utils.GNLMRegression("LinearRegression",
                              copy.deepcopy(layers_regress))
loss_  = torch.nn.MSELoss(reduction='sum')
```

```
loss_train_s_s, loss_test_s_s, yhat_train_s, \
y_train_s, yhat_test_s, y_test_s = \
    f_reg_l1_nn_cv(idx_s,None,dataset,model_,loss_,batch_size,alpha_t,
                   nbmax_epoqs,debug_out,device=device,
                  #loss_yy_model=loss_yy_model,printed=2,
                  #hyperparameter_to_print=str(f"lambda_l1={lambda_l1}")
                  )
```

```
processing fold n° 1 / 5
loss= 295.82355          t= 0 / 350      ( 0.0 %)
loss= 4.79923    t= 350 / 350    ( 100.0 %)
hyperparameter: None alpha_t= 0.001  mse_train= 0.014  mse_test= 0.114
processing fold n° 2 / 5
loss= 287.93915          t= 0 / 350      ( 0.0 %)
loss= 5.61531    t= 350 / 350    ( 100.0 %)
hyperparameter: None alpha_t= 0.001  mse_train= 0.014  mse_test= 0.094
processing fold n° 3 / 5
loss= 290.78146          t= 0 / 350      ( 0.0 %)
loss= 4.53559    t= 350 / 350    ( 100.0 %)
hyperparameter: None alpha_t= 0.001  mse_train= 0.011  mse_test= 0.132
processing fold n° 4 / 5
loss= 282.48309          t= 0 / 350      ( 0.0 %)
loss= 4.92806    t= 350 / 350    ( 100.0 %)
```

```
hyperparameter: None alpha_t= 0.001  mse_train= 0.014  mse_test= 0.106
processing fold n° 5 / 5
loss= 303.02301          t= 0 / 350      ( 0.0 %)
loss= 5.52593    t= 350 / 350    ( 100.0 %)
hyperparameter: None alpha_t= 0.001  mse_train= 0.02  mse_test= 0.102
```

```
loss_train_s_s, loss_test_s_s, yhat_train_s, \
y_train_s, yhat_test_s, y_test_s = \
    f_reg_l1_nn_cv(idx_s,None,dataset,model_,loss_,batch_size,alpha_t,
                   nbmax_epoqs,debug_out,device=device,
                  loss_yy_model=loss_yy_model,printed=2,
                  hyperparameter_to_print=str(f"lambda_l1={lambda_l1}")
                  )
```

```
processing fold n° 1 / 5
loss= 337.02509          t= 0 / 350      ( 0.0 %)
loss= 14.63122           t= 350 / 350    ( 100.0 %)
hyperparameter: lambda_l1=0.02 alpha_t= 0.001  mse_train= 0.018  mse_test=␣
 ↪0.058
processing fold n° 2 / 5
loss= 298.60675          t= 0 / 350      ( 0.0 %)
loss= 15.15463           t= 350 / 350    ( 100.0 %)
hyperparameter: lambda_l1=0.02 alpha_t= 0.001  mse_train= 0.02  mse_test= 0.
 ↪047
processing fold n° 3 / 5
loss= 306.45609          t= 0 / 350      ( 0.0 %)
loss= 13.26368           t= 350 / 350    ( 100.0 %)
hyperparameter: lambda_l1=0.02 alpha_t= 0.001  mse_train= 0.016  mse_test=␣
 ↪0.064
processing fold n° 4 / 5
loss= 293.34348          t= 0 / 350      ( 0.0 %)
loss= 14.40112           t= 350 / 350    ( 100.0 %)
hyperparameter: lambda_l1=0.02 alpha_t= 0.001  mse_train= 0.018  mse_test=␣
 ↪0.056
processing fold n° 5 / 5
loss= 370.0127           t= 0 / 350      ( 0.0 %)
loss= 13.51316           t= 350 / 350    ( 100.0 %)
hyperparameter: lambda_l1=0.02 alpha_t= 0.001  mse_train= 0.019  mse_test=␣
 ↪0.051
```

This leads to an improvement of the results but less than expected in comparison to the previous two samples (one train and one test).

A way to improve further would be to set better the learning rate by running the training for each of candidate values and keep the best. Example of such values are as follows.

```
alpha_t_s = np.array([0.1,0.075,0.05,0.01,0.0075,0.005,
                      0.003,0.001,0.00075,0.0005,0.0001])[::-1]
```

Instead, let check if the lambda_l1 is really optimal, considering this increase of the mse for the cross-validation. First the datasets are stored on the computer disk.

```
import tables
import h5py

def f_save_dataloader_to_h5py(dataloader,filename,d,transformx=None):

    file = tables.open_file(filename, mode='w')
    x_atom = tables.Float64Atom()
    y_atom = tables.Int16Atom()
    x_ds = file.create_earray(file.root, 'x', x_atom,(0,d))
    y_ds = file.create_earray(file.root, 'y', y_atom,(0,1))

    for step, (xb, yb) in enumerate(dataloader):
        xb = xb.detach().numpy()
        if transformx is not None:
            xb = transformx(xb)
        xb = xb.reshape((len(xb),d))
        yb = yb.detach().numpy()
        yb = yb.reshape((len(yb),1))
        x_ds.append(xb)
        y_ds.append(yb)

    file.close()
```

```
def f_save_cvdatasets_to_h5py(idx_s,dataset,filename_base,
                              d_aftertransformx,transformx=None,
                              show=0):

    namefile_s = dict()
    for k, key_k in enumerate(idx_s.keys()):
        if show: print("processing fold n° "+str(k+1),"/",str(len(idx_s)))

        idx_train, idx_test = idx_s[key_k]["train"], idx_s[key_k]["test"]
        n_train, n_test     = len(idx_train), len(idx_test)

        subsampler_train = torch.utils.data.SubsetRandomSampler(idx_train)
        subsampler_test  = torch.utils.data.SubsetRandomSampler(idx_test)

        dl_train = torch.utils.data.DataLoader(dataset,
```

```
                    batch_size=batch_size,sampler=subsampler_train)

        dl_test  = torch.utils.data.DataLoader(dataset,
                    batch_size=batch_size,sampler=subsampler_test)

        filename_train = filename_base+str("_")+str(k)+str("_")+"train.h5"
        filename_test = filename_base+str("_")+str(k)+str("_")+"test.h5"

        f_save_dataloader_to_h5py(dl_train,filename_train,
                                  d_aftertransformx,transformx)
        f_save_dataloader_to_h5py(dl_test,filename_test,
                                  d_aftertransformx,transformx)

        namefile_s[str(k)] = dict({"train":filename_train,
                                   "test":filename_test})

    return  namefile_s
```

The compressed files are stored.

```
namefile_s = f_save_cvdatasets_to_h5py(idx_s,dataset,
                                       towdir("x_y__450d_lasso__"),
                                       x.shape[1], ####dataset.x.shape[1],
                                       transformx=None)
```

Thus this is the input format for the dictionnary with the file names for the cross-validation: instead of indexes, there is a characters string with the name (and path).

```
namefile_s
```

```
{'0': {'train': './datasets_book/x_y__450d_lasso___0_train.h5',
  'test': './datasets_book/x_y__450d_lasso___0_test.h5'},
 '1': {'train': './datasets_book/x_y__450d_lasso___1_train.h5',
  'test': './datasets_book/x_y__450d_lasso___1_test.h5'},
 '2': {'train': './datasets_book/x_y__450d_lasso___2_train.h5',
  'test': './datasets_book/x_y__450d_lasso___2_test.h5'},
 '3': {'train': './datasets_book/x_y__450d_lasso___3_train.h5',
  'test': './datasets_book/x_y__450d_lasso___3_test.h5'},
 '4': {'train': './datasets_book/x_y__450d_lasso___4_train.h5',
  'test': './datasets_book/x_y__450d_lasso___4_test.h5'}}
```

The call is almost similar than before except that the argument "dataset" in the function call is set to "None", such as also "idx_s", while "namefile_s" is now given a value. Note that for a small dataset, this is better to keep the data in the computer memory for faster training.

The better regularization parameter and the better learning rate are found via cross-validation, among the following candidate values.

```
alpha_t_s   = np.array([0.00005,0.000075,0.001,0.00125,0.00150])
lambda_l1_s = np.array([0.001,0.005,0.01,0.015,0.02,0.025,
                        0.03,0.04,0.05,0.1,0.2,0.3]) #[::-1]
```

```
nbmax_epoqs = 100

resu_s = []
para_s = []

for lambda_l1 in iter(lambda_l1_s):
    for alpha_t in iter(alpha_t_s):

        print(f"lambda_l1={lambda_l1} alpha_t={alpha_t}", end='')

        loss_train_s_s, loss_test_s_s, yhat_train_s, \
        y_train_s, yhat_test_s, y_test_s = \
        f_reg_l1_nn_cv(idx_s,None,dataset,model_,loss_,batch_size,alpha_t,
                       nbmax_epoqs,debug_out,device=device,
                       loss_yy_model=loss_yy_model,printed=0,
                  hyperparameter_to_print=str(f"lambda_l1={lambda_l1}")
                       )

        mse_train_s = []
        for y_train_, yhat_train_ in iter(zip(y_train_s,yhat_train_s)):
            mse_,r2_ = utils.f_metrics_regression(y_train_,
                                                  yhat_train_,False)
            mse_train_s.append(mse_)

        mse_test_s = []
        for y_test_, yhat_test_ in iter(zip(y_test_s,yhat_test_s)):
            mse_,r2_ = utils.f_metrics_regression(y_test_,
                                                  yhat_test_,False)
            mse_test_s.append(mse_)

        resu_s.append([mse_train_s,mse_test_s,yhat_train_s,
                       y_train_s,yhat_test_s,y_test_s])
        para_s.append([lambda_l1,alpha_t])

        msetr = round(np.mean(mse_train_s),4)
        msete = round(np.mean(mse_test_s),4)
```

```
        print(" mse_tr_mean=", msetr, end='')
        print(" mse_te_mean=", msete, end='\n')
```

```
lambda_l1=0.001 alpha_t=5e-05 mse_tr_mean= 0.2603 mse_te_mean= 0.394
lambda_l1=0.001 alpha_t=7.5e-05 mse_tr_mean= 0.1864 mse_te_mean= 0.3377
lambda_l1=0.001 alpha_t=0.001 mse_tr_mean= 0.018 mse_te_mean= 0.1128
lambda_l1=0.001 alpha_t=0.00125 mse_tr_mean= 0.0182 mse_te_mean= 0.1069
lambda_l1=0.001 alpha_t=0.0015 mse_tr_mean= 0.076 mse_te_mean= 0.1614
lambda_l1=0.005 alpha_t=5e-05 mse_tr_mean= 0.2593 mse_te_mean= 0.3928
lambda_l1=0.005 alpha_t=7.5e-05 mse_tr_mean= 0.1861 mse_te_mean= 0.3379
lambda_l1=0.005 alpha_t=0.001 mse_tr_mean= 0.0208 mse_te_mean= 0.0998
lambda_l1=0.005 alpha_t=0.00125 mse_tr_mean= 0.0204 mse_te_mean= 0.0955
lambda_l1=0.005 alpha_t=0.0015 mse_tr_mean= 0.0716 mse_te_mean= 0.1444
lambda_l1=0.01 alpha_t=5e-05 mse_tr_mean= 0.2585 mse_te_mean= 0.392
lambda_l1=0.01 alpha_t=7.5e-05 mse_tr_mean= 0.1847 mse_te_mean= 0.3348
lambda_l1=0.01 alpha_t=0.001 mse_tr_mean= 0.0171 mse_te_mean= 0.0873
lambda_l1=0.01 alpha_t=0.00125 mse_tr_mean= 0.0323 mse_te_mean= 0.0952
lambda_l1=0.01 alpha_t=0.0015 mse_tr_mean= 0.044 mse_te_mean= 0.108
lambda_l1=0.015 alpha_t=5e-05 mse_tr_mean= 0.2576 mse_te_mean= 0.3886
lambda_l1=0.015 alpha_t=7.5e-05 mse_tr_mean= 0.1835 mse_te_mean= 0.3312
lambda_l1=0.015 alpha_t=0.001 mse_tr_mean= 0.0182 mse_te_mean= 0.0753
lambda_l1=0.015 alpha_t=0.00125 mse_tr_mean= 0.0242 mse_te_mean= 0.0778
lambda_l1=0.015 alpha_t=0.0015 mse_tr_mean= 0.0488 mse_te_mean= 0.0929
lambda_l1=0.02 alpha_t=5e-05 mse_tr_mean= 0.2569 mse_te_mean= 0.3883
lambda_l1=0.02 alpha_t=7.5e-05 mse_tr_mean= 0.183 mse_te_mean= 0.3292
lambda_l1=0.02 alpha_t=0.001 mse_tr_mean= 0.0198 mse_te_mean= 0.0677
lambda_l1=0.02 alpha_t=0.00125 mse_tr_mean= 0.0329 mse_te_mean= 0.0789
lambda_l1=0.02 alpha_t=0.0015 mse_tr_mean= 0.056 mse_te_mean= 0.0982
lambda_l1=0.025 alpha_t=5e-05 mse_tr_mean= 0.2557 mse_te_mean= 0.3856
lambda_l1=0.025 alpha_t=7.5e-05 mse_tr_mean= 0.182 mse_te_mean= 0.3265
lambda_l1=0.025 alpha_t=0.001 mse_tr_mean= 0.0222 mse_te_mean= 0.0626
lambda_l1=0.025 alpha_t=0.00125 mse_tr_mean= 0.0295 mse_te_mean= 0.0653
lambda_l1=0.025 alpha_t=0.0015 mse_tr_mean= 0.0581 mse_te_mean= 0.0968
lambda_l1=0.03 alpha_t=5e-05 mse_tr_mean= 0.255 mse_te_mean= 0.3839
lambda_l1=0.03 alpha_t=7.5e-05 mse_tr_mean= 0.1822 mse_te_mean= 0.3251
lambda_l1=0.03 alpha_t=0.001 mse_tr_mean= 0.0243 mse_te_mean= 0.061
lambda_l1=0.03 alpha_t=0.00125 mse_tr_mean= 0.0247 mse_te_mean= 0.0577
lambda_l1=0.03 alpha_t=0.0015 mse_tr_mean= 0.0655 mse_te_mean= 0.0974
lambda_l1=0.04 alpha_t=5e-05 mse_tr_mean= 0.2534 mse_te_mean= 0.381
lambda_l1=0.04 alpha_t=7.5e-05 mse_tr_mean= 0.1803 mse_te_mean= 0.3208
lambda_l1=0.04 alpha_t=0.001 mse_tr_mean= 0.0277 mse_te_mean= 0.0552
lambda_l1=0.04 alpha_t=0.00125 mse_tr_mean= 0.0365 mse_te_mean= 0.06
lambda_l1=0.04 alpha_t=0.0015 mse_tr_mean= 0.1115 mse_te_mean= 0.1329
lambda_l1=0.05 alpha_t=5e-05 mse_tr_mean= 0.252 mse_te_mean= 0.3768
```

```
lambda_l1=0.05 alpha_t=7.5e-05 mse_tr_mean= 0.1795 mse_te_mean= 0.3158
lambda_l1=0.05 alpha_t=0.001 mse_tr_mean= 0.0305 mse_te_mean= 0.0527
lambda_l1=0.05 alpha_t=0.00125 mse_tr_mean= 0.0327 mse_te_mean= 0.0517
lambda_l1=0.05 alpha_t=0.0015 mse_tr_mean= 0.0423 mse_te_mean= 0.0621
lambda_l1=0.1 alpha_t=5e-05 mse_tr_mean= 0.2467 mse_te_mean= 0.3606
lambda_l1=0.1 alpha_t=7.5e-05 mse_tr_mean= 0.1767 mse_te_mean= 0.2936
lambda_l1=0.1 alpha_t=0.001 mse_tr_mean= 0.0364 mse_te_mean= 0.0435
lambda_l1=0.1 alpha_t=0.00125 mse_tr_mean= 0.051 mse_te_mean= 0.0598
lambda_l1=0.1 alpha_t=0.0015 mse_tr_mean= 0.4194 mse_te_mean= 0.4336
lambda_l1=0.2 alpha_t=5e-05 mse_tr_mean= 0.2426 mse_te_mean= 0.337
lambda_l1=0.2 alpha_t=7.5e-05 mse_tr_mean= 0.1833 mse_te_mean= 0.2622
lambda_l1=0.2 alpha_t=0.001 mse_tr_mean= 0.0598 mse_te_mean= 0.0608
lambda_l1=0.2 alpha_t=0.00125 mse_tr_mean= 0.0758 mse_te_mean= 0.0756
lambda_l1=0.2 alpha_t=0.0015 mse_tr_mean= 0.3214 mse_te_mean= 0.3232
lambda_l1=0.3 alpha_t=5e-05 mse_tr_mean= 0.2446 mse_te_mean= 0.3185
lambda_l1=0.3 alpha_t=7.5e-05 mse_tr_mean= 0.1968 mse_te_mean= 0.2502
lambda_l1=0.3 alpha_t=0.001 mse_tr_mean= 0.0654 mse_te_mean= 0.0674
lambda_l1=0.3 alpha_t=0.00125 mse_tr_mean= 0.0831 mse_te_mean= 0.0863
lambda_l1=0.3 alpha_t=0.0015 mse_tr_mean= 0.629 mse_te_mean= 0.6351
```

Note that the function "loss_yy_model" performs with a global variable, hence a more secure way to do could be preferred, such as a method from a class where lambda_l1 is a variable. The training may be slow for some computer machine because the cross-validation asks for five training runs which are repeated by the number of learning rate candidates multiply with the number of regularizing parameters candidates: $5 \times 5 \times 12 = 300$, thus fortunately the gpu was used here.

The number of epochs may be limited otherwise the running time may be too high for some computers, here 100 instead of 350 because the test loss was converging faster according to the previous curves. This is a way to do for any neural network actually, even larger ones and larger datasets. Another strategy is to randomly sample candidate values for alpha_t and lambda_l1 in an interval or according to some distribution, but with no better insurance to get the best parameters. Here, it may be also implemented some strategy to fine tune the search by looking around a good candidate value. These strategies are costly and may be possible only for serious computing availability. This underlines the need for gpu and parallel processing in order to even search the good values for the hyperparameters. This is anyway true for running any numerical python code: a better computer gets the faster result but for deep learning a slow computer may not be able to proceed in a practical time or might at least induce a delay for the results.

From the numerical output just above, the resulting statistics for the computed metrics are thus summarized in a table in order to compare the quality of the results, even if the convergence was eventually not reached.

```
import pandas as pd

meanmsetrain = np.asarray([np.mean(resu_[0]) for resu_ in resu_s])
stdmsetrain  = np.asarray([np.std(resu_[0]) for resu_ in resu_s])
```

```
meanmsetest  = np.asarray([np.mean(resu_[1]) for resu_ in resu_s])
stdmsetest   = np.asarray([np.std(resu_[1]) for resu_ in resu_s])

results = [ [t[0] for t in para_s],[t[1] for t in para_s],
           meanmsetrain, meanmsetest, stdmsetrain, stdmsetest,
           stdmsetrain/meanmsetrain, stdmsetest/meanmsetest]
results_pd = pd.DataFrame(results).transpose()
results_pd.columns = ["lambda_l1","alpha_t",
                      "mse_tr_mean", "mse_te_mean",
                      "mse_tr_std", "mse_te_std", "mse_tr_coeffvar",
                      "mse_te_coeffvar",]
```

The plot is with seaborn here, a module which make matplotlib more powerful to use with less command lines, see the figure and code just below.

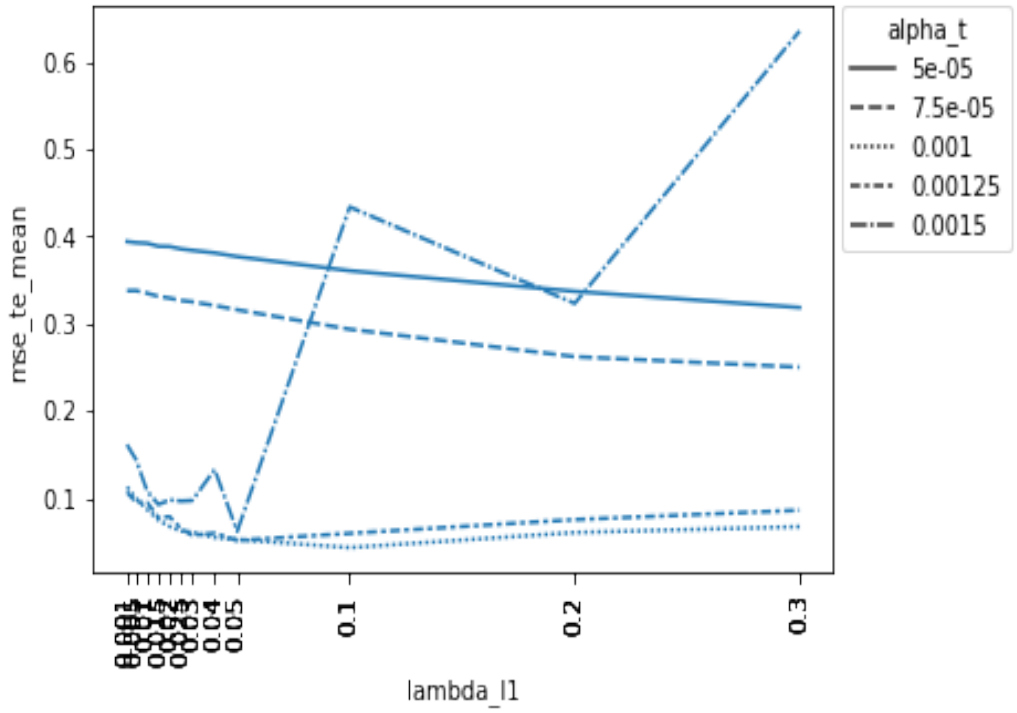

Figure 5.3: Mse averages on test set after CV+L_1 and grid search

```
import seaborn as sns
import matplotlib.pyplot as plt

g = sns.lineplot(data=results_pd, x='lambda_l1', y='mse_te_mean',␣
 ↪style='alpha_t')
```

```
# g.set(xscale='log')
g.set(xticks=results_pd['lambda_l1'])
g.set(xticklabels=results_pd['lambda_l1'])
plt.xticks(rotation=90)
plt.legend(bbox_to_anchor=(1.01, 1), borderaxespad=0, title = "alpha_t")
```

```
<matplotlib.legend.Legend at 0x7f21578b9700>
```

Here, lambda_l1 equal to about 0.1 performs well, but a different alpha_t looks better than the one from before, 0.001, this is the row from the results:

lambda_l1=0.1 alpha_t=0.001 mse_tr_mean= 0.0354 mse_te_mean=0.0451

The difference with the manually chosen parameters above may appear nearly neglectable. Anyway, checking if the choice of the parameters is well suited looks like required in order to insure the best training and settings as possible. For more parameters, parallel plots are suggested in the literature, while a bayesian search for the optimal parameters looks like a better solution in order to minimise the number of runs.

5.5.3 Bayesian search for optimal hyperparameters

The method just before looks like slow for larger networks and more parameters because it needs to compute the solution for every candidate values from all the hyperparameters. A method which is able to search in a more clever way is able to choose better the area to search for the optimal parameters by sampling the best place from the interval where the hyperparameters are suggested to belong.
This is the "bayesian search" which uses probabilitic distributions on the hyperparameters in order to improve the understanding of the search space. Several methods have been proposed in the literature, the following one is tested, it is called "smac". Note that it is also implemented in "autotorch", a module to choose automatically the architecture for a neural network as herein, a feed forward neural network. There exists also modules for the "grid search" or "random search", but this module "smac" tries to look for an optimal path towards the best solution given a space of search.

To proceed with the module, we need:

- The definition of the function to minimize with an unique argument named "config", this variable is a dictionary for the parameters in order to be callable from the module.
- The definition of the "configuration space" with an interval for the sampling of each parameter. A configuration is a unique value of the hyperparameters while the space is the set of interval where the parameters are to be found plus their distribution, such as uniform, gaussian, etc. This may be with a logarithmic transformation if required.
- The definition of a "scenario" for the search, for instance, the total number of configurations tested, total running time, etc plus the configuration space.
- The call to the module for the search, with for instance the function "SMAC4HPO()" which takes as arguments: the function to minimize and also the scenario.

This is as follows for our cross-validation function. First the embedding function is defined.

```
nbmax_epoqs = 100
def f_reg_l1_nn_cv__from_cfg(config):
    #print(config["alpha_t"], config["lambda_l1"],end=" ")
    alpha_t   = config["alpha_t"]
    lambda_l1 = config["lambda_l1"]
    with warnings.catch_warnings():
        warnings.filterwarnings("ignore")
        loss_train_s_s, loss_test_s_s, yhat_train_s, \
        y_train_s, yhat_test_s, y_test_s = \
        f_reg_l1_nn_cv(idx_s,None,dataset,model_,loss_,batch_size,alpha_t,
                       nbmax_epoqs,debug_out,device=device,
                       loss_yy_model=loss_yy_model,printed=0,
                  hyperparameter_to_print=str(f"lambda_l1={lambda_l1}")
                       )
        mse_test_s = []
        for y_test_, yhat_test_ in iter(zip(y_test_s,yhat_test_s)):
            mse_,r2_ = utils.f_metrics_regression(y_test_,
                                                  yhat_test_,False)
            mse_test_s.append(mse_)
        msete = round(np.mean(mse_test_s),4)
    return msete
```

The optimization proceeds with the python module "smac" in order to find the best hyperparameters automatically by defining their intervals (or range) and their distributions.

```
import numpy as np

import warnings
import logging
logging.basicConfig(level=logging.INFO)

from ConfigSpace import ConfigurationSpace
from ConfigSpace.hyperparameters import UniformIntegerHyperparameter, \
                                        UniformFloatHyperparameter

from smac.facade.smac_bb_facade import SMAC4BB
from smac.facade.smac_hpo_facade import SMAC4HPO

from smac.scenario.scenario import Scenario


# Definition for the hyperparameters
configspace = ConfigurationSpace()
```

```
configspace.add_hyperparameter(UniformFloatHyperparameter("lambda_l1", \
                                0.0001, 0.03, default_value=0.01) )

configspace.add_hyperparameter(UniformFloatHyperparameter("alpha_t", \
                                0.00005, 0.00150, default_value=0.001) )
scenario_dict = {'run_obj': "quality", 'deterministic': True}
scenario_dict['cs'] = configspace
scenario_dict['runcount-limit'] = 50   # 50 configurations evaluated
scenario_dict['wallclock-limit'] = 600  # time limit for all evaluations

scenario = Scenario(scenario_dict)

# smac = SMAC4BB(scenario=scenario,
smac = SMAC4HPO(scenario=scenario,
               tae_runner=f_reg_l1_nn_cv__from_cfg)
```

```
INFO:smac.utils.io.cmd_reader.CMDReader:Output to
smac3-output_2022-10-01_16:22:12_461756
INFO:smac.facade.smac_hpo_facade.SMAC4HPO:Optimizing a deterministic␣
 ↪scenario
for quality without a tuner timeout - will make SMAC deterministic and only
evaluate one configuration per iteration!
INFO:smac.initial_design.sobol_design.SobolDesign:Running initial design␣
 ↪for 12
configurations
INFO:smac.facade.smac_hpo_facade.SMAC4HPO:<class
'smac.facade.smac_hpo_facade.SMAC4HPO'>
```

After the definition of the python objects for "smac", the search for the optimal solution is lauched with the following call to "optimize()", a dedicated method from the module.

```
best_found_config = smac.optimize()
```

After running the call, the python variable "best_found_config" has received the optimal parameter found. For debugging, one may call:

```
# %tb
```

In this example of configuration, there was no improvement from the grid search. Choosing 10 folds for the cross-validation would lead to more precise estimations of the hyperparameters with only double running time. Note that with this example of implementation if the loss becomes nan or infty, the module may stop with an exception, hence this should be fixed if required by cautious bounds for the intervals in the configuration space. Early stopping would be interesting here to implement instead of a not full training with a constant number of epochs kept the same for every

choices of hyperparameters candidate. Other implementations of such automatic search for the hyperparameters exist in python modules and are left as an exercice for comparison: this one is implemented in auto-pytorch an auto-ml member which allows to find the architecture of neural networks automatically, even if some setting by an human expert is required for the choice of the intervals but also the final validation and the data preparation with also the recoding for instance.

In this chapter, we have studied how to implement the lasso regression with pytorch and compared several approaches for finding the hyperparameters for the penalization and the learning rate. When well chosen, the regularization (lasso, dropout, ...) allows to reduce the overfitting in order to improve the generalization for the prediction with new data.

5.6 Exercices

1. (sklearn) Retrieve the value lambda_l1 with sklearn and compare the resulting mse and r2 from the result with pytorch. It is noticed here that the parameter for the regularization may be a product by two or an half because in sklearn, it is usual to have a factor two in the mse, see the documentation of sklearn. The regression model in sklearn performs often well by default because a regularization is included with a quadratic norm for the parameters vector, this is a ridge hence regularized regression.
2. (pytorch) Implement a lasso for the classification with a real or artificial dataset from one of the previous chapters herein.
3. (sklearn) Test the stochastic regressor [2] from *sklearn* with different hyperparameters for the number of steps and the batch sizes. Similar behaviour may be observed in comparison to the pytorch implementation, when a regularizing term is added.
4. (pytorch) Add a stopping rule in the class MyMonitor when the training loss is enough stable in order to reduce the running time, and also when the test loss increases during several steps for reducing overfitting.
5. (pytorch) Test the lasso for the deep regression model from the chapter 1, for fitting the data from a polynomial regression. Count the number of parameters which are near or zero and the number of parameters which are kept. Check if this regularization reduces the mean squared error on the test sample. Test other regularizations.

[2] "https://scikit-learn.org/stable/modules/sgd.html"

Chapter 6

Hessian and covariance for (deep) glm

In the previous chapter we have studied neural networks and pytorch for regression and classification. In this chapter, we are interested on numerical "variance estimation" by computing values for $\widehat{\text{Var}}[\hat{\beta}_j]$. As a statistical model, neural networks have parameters which are estimated from a sample. This induces some variability of the resulting estimates. Hence the usual approach is to find the "hessian matrix", say $\nabla^2\ell = \nabla\nabla^T\ell$, with the "second-order derivatives" of the loss function $\ell()$ in order to deduce the variance. For a large number of parameters and small samples, this approch is incomplete and asks for a more advanced method (in case of no tractable numerical "boostrap"), hence out of the scope here. Otherwise such as in the linear case, this is the classical way to do which is presented next after with the modules numpy, statsmodels and pytorch.

6.1 Parameters variance without pytorch

An usual linear regression is fitted and the parameters variance is estimated with python before retrieving the results with numpy and then pythorch.

6.1.1 Dataset from linear regression

The work directory is given from the function.

```
def towdir(s):
    return (str('./datasets_book/'+s))

import deepglmlib.utils as utils
import numpy as np
```

```
import importlib
importlib.reload(utils)
```

Let generate a dataset from a linear model in order to perform the comparisons between the implementations. This is a toy dataset with thirty rows and five variables, the dataset is without useless

columns.

The resulting two samples are standardized.

```
import numpy as np
beta      = np.array([-0.5,1.0,0.8,0.7,0.4,0.2]).reshape((6,1))
beta_true = beta
n         = 30
x         = np.random.uniform(0,1,n*5).reshape((n,5))
X         = np.hstack([np.ones((n,1)), x])
e         = np.random.randn(n).reshape((n,1))/5
y         = X @ beta + e

x_train, x_test, y_train, y_test = utils.f_splitData(x,
                                        y,percentage=0.25)
# def f_normalizeData(x_train,x_test):
#     meanx_train = np.mean(x_train)
#     sdx_train   = np.sqrt(np.var(x_train))

#     x_train = (x_train-meanx_train)/sdx_train
#     x_test  = (x_test-meanx_train)/sdx_train
#     return x_train, x_test
x_train, x_test = utils.f_normalizeData(x_train,x_test)
```

```
n_train, p_train = x_train.shape
n_test, p_test   = x_test.shape
X_train          = np.hstack([np.ones((n_train,1)), x_train])
X_test           = np.hstack([np.ones((n_test,1)), x_test])
```

```
n_train, p_train, n_test, p_test
```

```
(23, 5, 7, 5)
```

6.1.2 Variance estimates from statsmodels

Because the sample is not the population, the regression coefficients have a variance. This is available from the module "statsmodels" which computes the coefficients vector for the linear regression, plus diverses statistical indicators, including the standard-deviation of each component.

```
import statsmodels.api as stm
ols           = stm.OLS(y_train, X_train)
fit_ols_train = ols.fit()
olssumy       = fit_ols_train.summary()
print(olssumy.tables[1])
```

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.9631      0.040     23.779      0.000       0.878       1.049
x1             0.3205      0.051      6.267      0.000       0.213       0.428
x2             0.2742      0.049      5.628      0.000       0.171       0.377
x3             0.2523      0.046      5.499      0.000       0.156       0.349
x4             0.0088      0.047      0.189      0.852      -0.090       0.108
x5             0.0601      0.046      1.298      0.211      -0.038       0.158
==============================================================================
```

The module offers a more advanced view of the table than the usual function "print()" for a numpy array.

This is the usual contents for statistics with data, as follows:

- The regression coefficients at the column named "coef" are the values $\hat{\beta}_j$.
- The "standard-errors" (std) at the column "std err" are the "square root of the variance". These values $(\widehat{\text{Var}}[\hat{\beta}_j])^{0.5}$ are useful because they enter the definition of an interval instead of just an unique value for each component.
- The bounds of the confidence intervals (left at 0.025 and right at 0.975) for the true parameters, say "$\beta_j \in \hat{\beta}_j \pm z_{0.975} \times$ std" where it is recognized $z_{0.975}$ as the percentile of the Gaussian distribution for a two side test.
- A Student test of nullity of the coefficients (value "t" and p-value).

Without the information from the variance, the interpretation of the estimations $\hat{\beta}_j$ for small samples are not sure: with another small sample from the same population, the value of the estimations may change dramatically, such that the conclusion would change completely too. For larger samples, the variance is enough small to be neglected in practice, this is common in machine learning literature.

Let retrieve the results for the variance via the algebra by writing a new python function. This is after a brief recall about the classical analytical expressions from the statistical literature next after.

6.1.3 Algebra for the parameters variance

The variance of the parameters is obtained from the statistical theory as follows for two separated cases, the linear one and the nonlinear one. An underlying hypothesis is a solution from maximum likelihood estimation.

1. For linear models (regression and classification)

- For linear regression.

$$\begin{aligned} \ell(\theta) &= \sum_i^n (y_i - \alpha - x_i^T \beta)^2 \\ &= \sum_i^n (y_i - \tilde{x}_i^T \theta)^2 . \end{aligned}$$

It is recalled that the criterion above comes without distributional hypothesis, even if one is hidden here. This is a Gaussian noise which can be supposed and implicit above. This leads to the matricial system:

$$y = \tilde{X}_n\theta + \varepsilon \text{ where } \theta = (\alpha, \beta^T)^T \text{ and } \varepsilon = (\varepsilon_i)_{i=1}^n \text{with } \varepsilon_i \sim \mathcal{N}(0, \sigma)\,.$$

The solution is written:

$$\hat{\theta} = (\tilde{X}_n^T\tilde{X}_n)^{-1}\tilde{X}_n^T y\,.$$

This estimator is unbiased, $E(\hat{\theta}) = \theta$ where θ is the true parameter for an infinite sample, say the whole population. An estimator of the variance of $\hat{\theta}$ is thus,

$$\hat{V}(\hat{\theta}) = \frac{\hat{\sigma}^2}{n}(\tilde{X}_n^T\tilde{X}_n)^{-1} \text{ where } \hat{\sigma}^2 = \frac{1}{n-p-1}\ell(\hat{\theta})\,.$$

The estimator for the σ^2 is here corrected at the denominator because the one from maximum likelihood is biased (and with fraction n^{-1} instead). As expected when n becomes large, the variance vanished to zero, but for small n it may be large.

- For logistic regression.

$$\begin{aligned}\ell(\theta) &= \sum_i^n y_i\sigma(\alpha + x_i^T\beta) + (1-y_i)(1-\sigma(\alpha + x_i^T\beta)) \\ &= \sum_i^n y_i\sigma(\tilde{x}_i^T\theta) + (1-y_i)(1-\sigma(\tilde{x}_i^T\theta))\,.\end{aligned}$$

With $\omega_i = \sigma(\tilde{x}_2^T\hat{\theta})(1-\sigma(\tilde{x}_2^T\hat{\theta}))$, an estimator of the variance of $\hat{\theta}$ is,

$$\hat{V}_\theta = -(\tilde{X}_n^T\Omega_{\hat{\theta}}\tilde{X}_n)^{-1} \text{ where } \Omega_{\hat{\theta}} = \text{Diag}(\omega_1, \omega_2, \cdots, \omega_n)\,.$$

2. With $\frac{\partial\ell}{\partial\theta} = \sum_i \frac{\partial\ell_i}{\partial\theta}$, the more general way to consider the variance estimation is via the hessian (second-order derivative) or its approximation:

$$\ell(\theta) = \sum_i^n \ell(x_i, \theta) \text{ such that, } \hat{V}_\theta = \left(\sum_i \frac{\partial^2\ell_i}{\partial\theta\partial\theta^T}\right)^{-1} \approx -\left(\sum_i \frac{\partial\ell_i}{\partial\theta}\frac{\partial\ell_i}{\partial\theta^T}\right)^{-1}$$

It is retrieved the logistic regression as a special case. More advanced estimators such as the sandwich one are not considered herein.

6.1.4 Variance estimates with numpy

The variance of the parameters is obtained as follows with python for the linear model.

```
def f_varthetahat(X,y,printed=False):
    n            = X.shape[0]
    p            = X.shape[1] #(p+1, with intercept)
    beta_hat     = np.linalg.inv(X.T @ X) @ X.T @ y
```

```
    y_hat        = X @ beta_hat
    residual     = y - y_hat
    sigma2_hat   = np.sum(residual**2) / (n - p)
    var_beta_hat = sigma2_hat * np.linalg.inv(X.T @ X)
    if printed:
        for p_ in range(p):
            standard_error = var_beta_hat[p_, p_] ** 0.5
            print(f"SE(beta_hat[{p_}]): {standard_error}")
    return beta_hat.ravel(), var_beta_hat, sigma2_hat

beta_hat_train, var_beta_hat_train, sigma2_hat_train = \
   f_varthetahat(X_train,y_train)
```

The estimate of the covariance matrix is equal to:

```
Cov_betahat = var_beta_hat_train

print("Cov_betahat=\n", np.round(Cov_betahat,4))
```

```
Cov_betahat=
 [[ 0.0016 -0.     -0.     -0.      0.      0.    ]
 [-0.      0.0026  0.0005 -0.0008 -0.001   0.0005]
 [-0.      0.0005  0.0024  0.0004 -0.0008 -0.0008]
 [-0.     -0.0008  0.0004  0.0021 -0.      0.    ]
 [ 0.     -0.001  -0.0008 -0.      0.0022  0.0002]
 [ 0.      0.0005 -0.0008  0.      0.0002  0.0021]]
```

The resulting table with the regression coefficients and the standard-deviations is shown just below in a pretty view very similar to the output from the above module, with several columns.

```
from prettytable import PrettyTable
import scipy

def f_print_output_from_varthetahat(beta_hat_train,var_beta_hat_train):
    t = beta_hat_train.ravel()/np.diag(var_beta_hat_train).ravel()** 0.5
    tab = PrettyTable()
    columnnames = ["const"]
    for j in range(len(beta_hat_train)-1):
        colj = "variable_%d" % (j+1,)
        columnnames.append(colj)
    tab.add_column("variable",columnnames)
    tab.add_column("coef",np.round(beta_hat_train.ravel(),3))
    tab.add_column("std err",
                   np.round(np.diag(var_beta_hat_train).ravel()** 0.5,3))
    tab.add_column("c.v.",np.round(beta_hat_train.ravel()/ \
```

```
                    np.diag(var_beta_hat_train).ravel()** 0.5,3))
    tab.add_column("t",np.round(t,4))
    tab.add_column("P>|t|",np.round(2*(1-scipy.stats.t.cdf(np.abs(t),
                    X_train.shape[0]-X_train.shape[1])),3))
    tab.align["variable"] = "l"
    tab.align["coef"] = "r"
    tab.align["std err"] = "r"
    tab.align["c.v."] = "r"
    tab.align["t"] = "r"
    tab.align["P>|t|"] = "r"
    print(tab)

f_print_output_from_varthetahat(beta_hat_train,var_beta_hat_train)
```

```
+------------+-------+---------+--------+---------+-------+
| variable   |  coef | std err |   c.v. |       t | P>|t| |
+------------+-------+---------+--------+---------+-------+
| const      | 0.963 |    0.04 | 23.779 | 23.7794 |   0.0 |
| variable_1 | 0.321 |   0.051 |  6.267 |  6.2666 |   0.0 |
| variable_2 | 0.274 |   0.049 |  5.628 |  5.6282 |   0.0 |
| variable_3 | 0.252 |   0.046 |  5.499 |  5.4993 |   0.0 |
| variable_4 | 0.009 |   0.047 |  0.189 |  0.1889 | 0.852 |
| variable_5 |  0.06 |   0.046 |  1.298 |  1.2984 | 0.211 |
+------------+-------+---------+--------+---------+-------+
```

It was added the variation coefficient, c.v., at the last column to the right, and also the t-test with the related probability. Here, the Student test of equality t_j of the regression coefficient is equal to the variation coefficient because $t_j = \frac{\hat{\theta}_j - 0}{\widehat{\text{var}}(\hat{\theta}_j)} = \text{cv}_j$, while the probability comes from a Student distribution. This is not discussed further herein as this is the variance which is in stake.

When the number of variables increases, this becomes nearly "impossible" to invert the hessian matrix with the current available software, thus it has been proposed solutions to solve this issue for neural networks, mainly by only considering the diagonal part of the hessian matrix because this is not the variance which is of first interest, just the training (see next chapter and the exercices part). Anyway, this brief introduction to the numerical variance estimation is not always relevant for some models where more advanced analytical expressions may be required.

6.2 Hessian and variance for the linear regression

We want to find the second-order derivatives instead of just the first-order derivatives for the loss and w.r.t. the weights of the neural network.

6.2.1 First-order training with pytorch

```
import torch
from torch.utils.data import TensorDataset, DataLoader

dataset_train = TensorDataset( torch.Tensor(x_train),
                               torch.Tensor(y_train) )
dataset_test  = TensorDataset( torch.Tensor(x_test),
                               torch.Tensor(y_test) )
dl_train = DataLoader(dataset_train,shuffle=True,batch_size=10)
dl_test  = DataLoader(dataset_test,shuffle=True,batch_size=10)
```

```
print("cuda.is_available()     = ", torch.cuda.is_available())
print("cuda.get_device_name(0) = ",torch.cuda.get_device_name(0))
```

```
cuda.is_available()     =  True
cuda.get_device_name(0) =  NVIDIA GeForce 940MX
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device
```

```
device(type='cuda', index=0)
```

```
import torch.nn as nn
import copy

px = x_train.shape[1]
nbmax_epoqs = 150
alpha_t     = 1e-3
debug_out   = 10

layers_regress = [ nn.Linear(px,1,bias=True) ]

model     =  utils.GNLMRegression("LinearRegression",
                                  copy.deepcopy(layers_regress))
loss      = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=alpha_t, momentum=0.0)
monitor   = utils.MyMonitorTest(model,loss,dl_train,dl_test,nbmax_epoqs,
                                debug_out,device=device)

loss_train_s,tmax,monistopc  = \
  utils.f_train_glmr(dl_train,model,optimizer,loss,
                     monitor,device=device,printed=1)
```

```
loss= 64.19253           t= 0 / 150      ( 0.0 %)
loss= 27.73732           t= 10 / 150     ( 6.67 %)
loss= 12.85577           t= 20 / 150     ( 13.33 %)
[...]
loss= 0.65766    t= 140 / 150    ( 93.33 %)
```

6.2.2 First computation of hessian with pytorch

Here it is not considered the quality of the regression. Instead, this is the variance of the parameters which is in stake. An implementation is recently available with pytorch for this purpose, with the function:

- torch.autograd.functional.jacobian(func, x)
- torch.autograd.functional.hessian(func, x)

This is written as follows. First it is retrieved the numerical solution for the parameters. The function "extract_weights_lin" in the module "deepglmlib" has the name of the layers as arguments in order to insure more flexibility as the keys may change.

Thus, the vector of regression coefficients is written,

```
def extract_weights_lin(model,keybias="lin.bias",keyweight="lin.weight"):
    weight_ = bias_ = None
    for param_tensor in model.state_dict():
        if (param_tensor==keyweight):
            weight_= model.state_dict()[param_tensor]
        if (param_tensor==keybias):
            bias_= model.state_dict()[param_tensor]
    return np.append(bias_,weight_)
```

```
# tuple([_.view(-1) for _ in model.parameters()])
thetahat_torch = extract_weights_lin(model.cpu(),
                     keybias="net.0.bias",keyweight="net.0.weight")
print( thetahat_torch )
```

```
[ 9.6193182e-01  3.1918368e-01  2.9256883e-01  2.5692576e-01
 -1.9638549e-04  4.3414962e-02]
```

For the first and second-order derivatives, say gradient and hessian, it must be defined the function and the value where the derivative is computed. Here it is built a new fonction based on the loss. This is very near the training or testing, but the argument is now the parameter. First, let retrieve the values of the standard-deviation from the algebra.

```
beta_hat_train2, var_beta_hat_train2, sigma2_hat_train2 = \
   f_varthetahat(X_train,y_train)
```

```
std_beta_hat_train2 = np.sqrt(np.diag(var_beta_hat_train2))
std_beta_hat_train2
```

```
array([0.04049972, 0.05114935, 0.04872171, 0.04588041, 0.04682142,
       0.04630771])
```

The more direct way to retrieve the hessian (for the variance) is to define the likelihood and implement the call to the corresponding pytorch function. In the linear case, without a native loss function from pytorch, this is written as follows.

```
import torch
theta_train_ = \
    torch.Tensor(thetahat_torch).reshape((len(thetahat_torch),1))
theta_train_.requires_grad=True
sigma_train_ = np.sqrt(sigma2_hat_train *(n_train-p_train-1)/n_train)
xx           = torch.Tensor(X_train)
yy           = torch.Tensor(y_train).reshape((len(y_train),1))

def log_lik(theta):
    return -0.5*(xx@theta-yy).T @ (xx@theta-yy) / n_train
```

```
theta_hat = (theta_train_) #, sigma_train_)
I = -torch.Tensor(torch.autograd.functional.hessian(log_lik, theta_hat))
I = I.squeeze()
I
```

```
tensor([[ 1.0000e+00, -1.0259e-08, -1.2050e-08, -1.3931e-08, -5.9545e-10,
         -1.4307e-08],
        [-1.8626e-09,  1.0000e+00, -2.5571e-01,  4.3566e-01,  3.7016e-01,
         -3.4481e-01],
        [ 1.9092e-08, -2.5571e-01,  1.0000e+00, -2.7998e-01,  2.2621e-01,
          4.1054e-01],
        [-2.7940e-08,  4.3566e-01, -2.7998e-01,  1.0000e+00,  1.1472e-01,
         -2.2728e-01],
        [-0.0000e+00,  3.7016e-01,  2.2621e-01,  1.1472e-01,  1.0000e+00,
         -7.5867e-02],
        [ 4.6566e-10, -3.4481e-01,  4.1054e-01, -2.2728e-01, -7.5867e-02,
          1.0000e+00]])
```

This is just that the analytical hessian, divided by n_train, is retrieved numerically for this simple example, because the solution is known in exact closed-form for the linear case:

```
xx.T @ xx / n_train
```

```
tensor([[ 1.0000e+00, -2.0732e-08, -1.3767e-08, -1.0366e-08,  5.1830e-09,
          1.5549e-08],
        [-2.0732e-08,  1.0000e+00, -2.5571e-01,  4.3566e-01,  3.7016e-01,
         -3.4481e-01],
        [-1.3767e-08, -2.5571e-01,  1.0000e+00, -2.7998e-01,  2.2621e-01,
          4.1054e-01],
        [-1.0366e-08,  4.3566e-01, -2.7998e-01,  1.0000e+00,  1.1472e-01,
         -2.2728e-01],
        [ 5.1830e-09,  3.7016e-01,  2.2621e-01,  1.1472e-01,  1.0000e+00,
         -7.5867e-02],
        [ 1.5549e-08, -3.4481e-01,  4.1054e-01, -2.2728e-01, -7.5867e-02,
          1.0000e+00]])
```

This comes from that:

$$\frac{0.5}{n}\frac{\partial^2 (X\theta - y)^T (X\theta - y)}{\partial\theta\partial\theta^T} = \frac{1}{n} X^T X\,.$$

6.2.3 Variance estimates with pytorch

After applying the pytorch function "torch.autograd.functional.hessian", the obtained value of the hessian matrix leads to the numerical estimation of the variance as follows:

```
std_beta_hat_train2_directI = \
   np.sqrt(np.diag(np.linalg.inv(I)*sigma2_hat_train2)/n_train)
np.round(std_beta_hat_train2_directI,3)
```

```
array([0.04 , 0.051, 0.049, 0.046, 0.047, 0.046], dtype=float32)
```

The estimator from algebra for this solution has the same value as expected, which validates the illustration with the dedicated pytorch numerical function here. These solutions are also identical to the one before from the python module stamodels (see a few paragraphs above).

```
np.round(std_beta_hat_train2,3)
```

```
array([0.04 , 0.051, 0.049, 0.046, 0.047, 0.046])
```

Actually it is required to retrieve the same solution for the regression coefficients because unbiasness is supposed here for computing the term $\hat{\sigma}$. In the nonlinear case, there is no need for this proportional variance.

Anyway the hessian matrix for neural networks is more useful for training a small network with a small dataset nowadays. But ill defined hessians and very large dimensional variable spaces seem a problem not solved in the literature even currently considering the increasing sizes of the models in used today for images and texts. For moderated larger matrices, an approximation of the hessian matrix (such as from quasi-newton approaches) may be preferred as explained in the next chapter.

This ends the introduction on variance estimation with pytorch through an example for the linear regression model. It must be added that a resampling algorithm, say "boostrap", is an alternative not considered here. The next chapter will consider further the hessian matrix for training (deep) neural networks (with a few hundreds or thousands of variables or weights).

6.3 Hessian computation for (small) neural networks

A nonlinear loglikelihood with the logistic regression is presented as a more advanced example where the hessian is computed from minibatches (for larger datasets) instead of the full matrix before, thus the data may come from the computer disk for these new python functions.

In summary from above, the derivatives are computed with the module torch via the automatic numerical function for the derivatives, in this section. The main idea is to apply the gradient computation to each component of the gradient vector, following the component-wise definition of the hessian, $H = H_{jk}$, while for the variance $\hat{V} = H^{-1}$. The cells of the matrix H are thus,

$$H_{jk} = \frac{\partial^2 \sum_i \ell_i}{\partial \beta_j \partial \beta_k} = \frac{\partial}{\partial \beta_k}\left[\frac{\partial \ell}{\partial \beta_j}\right] \quad \text{or approximatively} \quad H_{jk} \approx -\sum_i \frac{\partial \ell_i}{\partial \beta_j} \times \frac{\partial \ell_i}{\partial \beta_k}.$$

- For the expression to the left, the gradient function is applied to each component of $\frac{\partial \ell}{\partial \beta} = (\frac{\partial \ell}{\partial \beta_j})_j$. It leads to the wanted matrix, but may ask for many computer memory and may be slow to compute for large values of p and n. With also a related second-order optimization prone to local minima, this explains why first-order training algorithms are often preferred.
- For the expression to the right, recall that $\frac{\partial \ell}{\partial \beta} = \frac{\partial \sum_i \ell_i}{\partial \beta} = \sum_i \frac{\partial \ell_i}{\partial \beta}$. The approximate expression just sums the cross-products of the gradients computed for each element of the sample, say the vectors $\frac{\partial \ell_i}{\beta} = (\frac{\partial \ell_i}{\partial \beta_j})_j$. Recall that this alternative expression approximates the exact hessian because the two expectations from each expression are exactly equal according to maximum likelihood estimation.

First let have a short recall of the formula for logistic regresion with: the expression of the log-likelihood, the derivatives (gradient, hessian) followed by the expression of the information matrix. Then, the variance is computed with pytorch after fitting the regression coefficients from the model for real data, the "abalone dataset".

6.3.1 Hessian expression for logistic regression

In the logistic regression, let rembember that the target or dependent variable is binary, $y_i = 1$ or $y_i = 0$, associated to the explicative variables which enter the model with weights called coefficients regression in the vector θ including the intercept. This leads to the model and the expectations as follows:

$$f(y_i|p_i) \quad = \quad p_i^{y_i}(1-p_i)^{1-y_i} \text{ where } \mathbf{E}(y_i) = p_i = \frac{e^{x_i^T \beta}}{1+e^{x_i^T \beta}}.$$

The (inverse) logit link function insures a linear regression after transformation. Let denote the diagonal matrix $\Omega = \text{Diag}(p_1(1-p_1),\dots,p_n(1-p_n))$, and the dataset available as the couples (x_i, y_i) with the loglikelihood written:

$$\ell(\beta) = \sum_{i=1}^n y_i \ln p_i + (1-y_i)\ln(1-p_i)\,.$$

Thus,

$$\begin{aligned}
\ell(\beta) &= \sum_{i=1}^n \left[y_i x_i^T \beta - \ln(1+e^{x_i^T\beta}) \right] \\
\nabla\ell(\beta) &= \sum_{i=1}^n (y_i - p_i) x_i \\
\nabla^2\ell(\beta) &= \sum_{i=1}^n -p_i(1-p_i) x_i x_i^T \\
\\
I_n(\beta) &= \mathbf{E}[-\nabla^2\ell(\beta)]\,.
\end{aligned}$$

Note that as a result, the Fisher information matrix is equal to the hessian, because the hessian is not random, and do not depend on the y_i,

$$\hat{I}_n(\beta) = -\nabla^2\ell(\beta) = I_n(\beta)\,.$$

In matricial notation, this is rewritten:

$$\begin{aligned}
\nabla\ell(\beta) &= X^T(y-p) \\
\nabla^2\ell(\beta) &= -X^T\Omega X \\
\\
\hat{I}_n(\beta) &= X^T\Omega X\,.
\end{aligned}$$

Thus,

$$\begin{aligned}
I_n(\beta) &= \mathbf{E}[\hat{I}_n(\beta)] \\
&= \hat{I}_n(\beta)
\end{aligned}$$

With the gradient per unit equal to $\nabla\ell_i = (y_i - p_i)x_i$, this also induces that the approximation for the hessian matrix is as follows,

$$\begin{aligned}
\tilde{I}_n(\beta) &= \sum_i \nabla\ell_i^T \nabla\ell_i \\
&= \sum_i (y_i - p_i)^2 x_i^T x_i \\
\\
\mathbf{E}[\tilde{I}_n(\beta)] &= I_n(\beta)\,.
\end{aligned}$$

As an exercice the reader may show the equality of the expressions from the first-order derivatives and from the second-order derivatives. Thus it is retrieved the general result for this example of a member of the generalized linear model. This is an expectation thus by the law of the large numbers, the equality holds for the sum only for enough large samples. This expression from the gradients has been implemented for neural networks by the past before the rise of first-order approaches.

6.3.2 Real dataset with two classes

First the dataset is loaded with the module "pandas" because it is in format "csv". This module allows also a nice processing of the data table for naming and accessing the columns with character strings, plus also for the rows and columns the selection of subsets or the computation of descriptive statistics (sum, mean, variance) for instance. The chosen target variable is the "count of rings" which takes integer values, it is binarized in order to be suitable for a binary classification.

```
import pandas as pd
abalone = pd.read_csv(towdir("./abalone_prep.csv"))
x       = abalone.drop(columns="rings")
y       = abalone["rings"].values - 1
y       = (y>np.median(y)).astype(int)

x.shape, y.shape
```

```
((4177, 6), (4177,))
```

Let select two subsamples for the training, the train and set samples.

```
x_train, x_test, y_train, y_test = utils.f_splitData(x.values,y,
                                                     percentage=0.333)
x_train, x_test                  = utils.f_normalizeData(x_train,x_test)
```

6.3.3 Results from the module statsmodels

The model is first trained with statsmodels as a baseline result in order to compare with the pytorch implementation. The output and code follow below.

```
Optimization terminated successfully.
         Current function value: 0.456115
         Iterations 7
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0763      0.056      1.371      0.170      -0.033       0.185
x1             0.4071      0.176      2.317      0.021       0.063       0.752
x2             0.2161      0.122      1.770      0.077      -0.023       0.455
x3            -1.7378      0.136    -12.822      0.000      -2.003      -1.472
x4             2.7230      0.209     13.058      0.000       2.314       3.132
x5             0.4238      0.064      6.614      0.000       0.298       0.549
x6             0.4112      0.063      6.493      0.000       0.287       0.535
==============================================================================
```

```
import statsmodels.api as stm
n_train       = x_train.shape[0]
```

```
X_train       = np.hstack([np.ones((n_train, 1)), x_train])
lgt           = stm.Logit(y_train, X_train)
fit_lgt_train = lgt.fit(maxiter=300)
lgtsumy       = fit_lgt_train.summary()

print(lgtsumy.tables[1])
```

6.3.4 Results from sklearn and numpy

The model is also fitted with sklearn in this paragraph. The standard deviations for the regression coefficients are computed with numpy arrays by implementing the expressions obtained from algebra at the beginning of the subsection, as follows:

```
import numpy as np
from sklearn import linear_model
logit     = linear_model.LogisticRegression(penalty='none',
                                    fit_intercept=False,max_iter=300)
resLogit  = logit.fit(X_train, y_train)
predProbs = resLogit.predict_proba(X_train)
Omega     = np.diagflat(np.product(predProbs, axis=1))
cov_theta = np.linalg.inv(np.dot(np.dot(X_train.T, Omega), X_train))

betahat_skl = resLogit.coef_.ravel()
print("betahat_skl: ", np.round(betahat_skl,4))
print("stdhat_skl: ", np.round(np.sqrt(np.diag(cov_theta)),3))
```

```
betahat_skl:  [ 0.0763  0.4071  0.2161 -1.7378  2.723   0.4238  0.4112]
stdhat_skl:  [0.056 0.176 0.122 0.136 0.209 0.064 0.063]
```

Thus a same result is obtained with the formula and with the python package for the variances of the regression coefficients. Note that the model here does not perform well with the variables which are correlated, hence adding a regularization could be required for a more useful model.

Note that currently the python variable "y" should be a one dimensional vector, "y.shape=(n_samples,)", thus obtained after the call "y=y.ravel()", which is the reason why this was added above at the definition of y when x and y were both created. Another shape of vector may not work well, hence checking the shape of y (and x) is a wise advice whenever pytorch or sklearn complains, with the current version of the modules.

6.3.5 First-order training with pytorch

With pytorch, the variance of the parameters is also retrieved after an iterative inference as follows.

```
import torch
#import glmlib.utils as utils
```

```
from torch.utils.data import TensorDataset, DataLoader
dataset_train     = TensorDataset( torch.Tensor(x_train),
                                   torch.Tensor(y_train) )
dataset_test      = TensorDataset( torch.Tensor(x_test),
                                   torch.Tensor(y_test) )
dl_train = DataLoader(dataset_train,shuffle=True,batch_size=100)
dl_test = DataLoader(dataset_test,shuffle=True,batch_size=100)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device
```

```
device(type='cuda', index=0)
```

```
import torch.nn as nn
import copy
px = x_train.shape[1]
nbmax_epoqs = 800
alpha_t     = 1e-2
debug_out   = 10

layers_regress = [ nn.Linear(px,1,bias=True) ]
model     =  utils.GNLMRegression("LogisticRegression",
                                  copy.deepcopy(layers_regress))
loss      = torch.nn.BCEWithLogitsLoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=alpha_t, momentum=0.0)
monitor   = utils.MyMonitorTest(model,loss,dl_train,dl_test,
                                  nbmax_epoqs,debug_out,device=device)

loss_train_s,tmax,monistopc  = utils.f_train_glmr(dl_train,model,
                                  optimizer,loss, monitor,device=device,
                                  transform_yb=utils.transform_yb,
                                  transform_yhatb=utils.transform_yhatb,
                                  printed=2)
```

```
loss= 1394.25159          t= 0 / 800      ( 0.0 %)
loss= 1282.63098          t= 800 / 800    ( 100.0 %)
```

For this training of the dataset, the test loss has converged under the train loss, this supposes a problem of underfitting according to the usual interpretation from the literature, hence the model is not enough flexible, not well trained, or informative variables are missing.

The purpose here is to illustrate the computation of the variance, without even tuning much the parameters. For this logistic regression model,

```
thetahat_torch = extract_weights_lin(model.cpu(),keybias="net.0.bias",
                                     keyweight="net.0.weight")
print( np.round(thetahat_torch,4) )
```

```
[ 2.0000e-03  3.7380e-01  1.9300e-01 -1.7576e+00  2.7142e+00  3.3390e-01
  4.8820e-01]
```

```
logLik_torch=0
with torch.no_grad():
    for b, (Xb,yb) in enumerate(dl_train):
        yhatb = model(Xb)
        logLik_torch -= loss(yhatb.reshape(yb.shape), yb)

float(logLik_torch.detach().numpy())
```

```
-1278.342041015625
```

This is near the solution from the dedicated python modules. A more cautious tuning of the parameters would lead to a nearer solution. For example sklearn leads to the solution.

```
print(np.round(resLogit.coef_.ravel(),4))
```

```
[ 0.0763  0.4071  0.2161 -1.7378  2.723   0.4238  0.4112]
```

```
phat = np.exp(X_train @ resLogit.coef_.ravel())
phat = phat/(1+phat)
np.sum(y_train * np.log(phat) + (1-y_train)*np.log(1-phat))
```

```
-1271.1912826172772
```

6.3.6 Hessian from second-order derivatives

The hessian is computed with minibatches because it is a sum of second-order derivatives over the whole sample. A loop cycles the dataset and sums the second-order derivatives from each minibatch as follows.

The number of weights is computed for the sizes before the calls to "hessian()" from pytorch.

```
theta_train_ = torch.Tensor(thetahat_torch)
```

```
p_model = 0
for p in model.parameters():
    if len(p.shape)>1:
        p_model += p.shape[1]
```

```
    else:
        p_model += 1
p_model
```

7

```
optimizer = torch.optim.SGD(model.parameters(),
                            lr=0.0, momentum=0.0)
optimizer.zero_grad()
I         = torch.zeros((p_model,p_model))
thessian  = torch.autograd.functional.hessian
theta_hat = theta_train_
t=0

for b,(Xb,yb) in enumerate(dl_train):
    print(".", end = '')
    Xb = torch.Tensor(np.hstack([np.ones((len(Xb), 1)), Xb]))

    def log_lik_b(theta):
        p_b = torch.exp(Xb@theta)
        p_b = p_b/(1+p_b)
        return torch.log(p_b.T)@ yb +  torch.log(1-p_b.T) @ (1-yb)

    I_b = -thessian(log_lik_b, theta_hat) / n_train
    I = I + I_b#.squeeze()
```

...

```
I = I.detach().numpy()
print()
print(np.round(I,3))
```

```
[[ 0.153  0.019  0.011  0.013 -0.002  0.011  0.013]
 [ 0.019  0.093  0.07   0.097  0.078  0.03   0.015]
 [ 0.011  0.07   0.088  0.079  0.066  0.027  0.011]
 [ 0.013  0.097  0.079  0.129  0.091  0.028  0.022]
 [-0.002  0.078  0.066  0.091  0.08   0.027  0.012]
 [ 0.011  0.03   0.027  0.028  0.027  0.162 -0.095]
 [ 0.013  0.015  0.011  0.022  0.012 -0.095  0.16 ]]
```

The resulting values of the standard-deviations are almost equal to the output from the previous estimations with a dedicated python module.

```
np.asarray(lgtsumy.tables[1].data)[:,2][1:]
```

```
array(['    0.056', '    0.176', '    0.122', '    0.136', '    0.209',
       '    0.064', '    0.063'], dtype='<U10')
```

```
np.round(np.sqrt(np.diag(np.linalg.inv(I))/n_train),3)
```

```
array([0.054, 0.171, 0.107, 0.134, 0.202, 0.064, 0.063], dtype=float32)
```

The difference comes from that the final solutions for the regression coefficients were only nearby as the training is for a nonlinear function with different initial values and different inferential procedures. The first solution with statsmodels comes from a second-order training procedure, see next chapter, while the second one with pytorch comes from a first-order one which asks for more careful settings. This is the price to pay in order to get a more scalable algorithm without requiring the full hessian at each step of the training, in order to avoid a non necessary costly numerical burden. An automatic or better setting of the learning rate with a stochastic approach for an optimal training would be required here. This is an almost mandatory option for convex objective functions because the global solution is reachable here.

6.3.7 Hessian from second-order derivatives (bis)

Let revisit the python code for more advanced models with hidden layers for instance. First, one has to remember that the parameters or weights of each layer are structured as follows in pytorch, the matrix for the linear transformation or "weight" and the vector for the "bias".

```
for p in model.parameters(): print(p.data, end= " ")
```

```
tensor([[ 0.3738,  0.1930, -1.7576,  2.7142,  0.3339,  0.4882]])
tensor([0.0020])
```

The weights are also available from the dictionary.

```
mn = model.net[0]
print(mn.state_dict()['bias'], mn.state_dict()['weight'])
```

```
tensor([0.0020]) tensor([[ 0.3738,  0.1930, -1.7576,  2.7142,  0.3339,
0.4882]])
```

There is a more direct access to the weights which allows to update their values too.

```
print(model.net[0].bias.data, model.net[0].weight.data)
```

```
tensor([0.0020]) tensor([[ 0.3738,  0.1930, -1.7576,  2.7142,  0.3339,
0.4882]])
```

```
print(theta_hat[0], theta_hat[1:])
```

```
tensor(0.0020) tensor([ 0.3738,  0.1930, -1.7576,  2.7142,  0.3339,  0.
↪4882])
```

The hessian is written as the derivative of the components of the gradient vector. It is adviced to keep the full list of parameters in an unique list, in order to avoid post-traitment, otherwise, a direct use may lead to an hessian per layer.

```
Imodel    = torch.zeros((p_model,p_model))
loss      = torch.nn.BCEWithLogitsLoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=0.0, momentum=0.0)
optimizer.zero_grad()
roll = True

for b,(Xb,yb) in enumerate(dl_train, n_train):
    print(".", end = '')

    yhatb       = model(Xb)
    loss_b      = loss(yhatb, yb.reshape(yhatb.shape))

    grad1rds_list = torch.autograd.grad(loss_b, model.parameters(), \
                                        create_graph=True, \
                                        retain_graph=True, \
                                        allow_unused=True)

    grad1rds_vec = torch.cat([g.view(-1) for g in grad1rds_list])
    if roll: grad1rds_vec = torch.roll(grad1rds_vec, 1)

    grad2rds_list = []
    for grad1rd in  grad1rds_vec:
        grad2rds = torch.autograd.grad(grad1rd, model.parameters(), \
                                        create_graph=True, \
                                        retain_graph=True, \
                                        allow_unused=True)

        grad2rds_list.append( np.hstack([ np.asarray(grad2rds[1].detach().
↪numpy()).squeeze(),  np.asarray(grad2rds[0].detach().numpy()).
↪squeeze()] ) )

    I_b    = np.asarray(grad2rds_list) / n_train
    Imodel = Imodel + I_b #.squeeze()
```

...

This solution here is the most direct [1], by iterating the components of the gradient, each row of

[1]Thanks to "https://discuss.pytorch.org/t/computing-hessian-for-loss-function/67216" , for an example of imple-

the hessian is computed numerically. The "intercept" or "bias" appears at the last row (and last column).

With q the number of parameters (bias and components of the weights) and $\theta = (\theta_1, \theta_2 \cdots, \theta_q)^T$, this corresponds to the definion of the hessian $H = \sum_i H_i$ where $H_i = \frac{\partial}{\partial \theta^T}[g_i]$ from the derivatives. The sum here on the whole train sample is performed with minibatches, $H = \sum_b^B \sum_{i \in s_b} H_i$ which is exactly equivalent to summing directly to the whole sample:

$$g_i = \frac{\partial}{\partial \theta}[\text{loss}(\hat{y}_i(\theta), y_i)] = \begin{pmatrix} \frac{\partial}{\partial \theta_1}[\text{loss}(\hat{y}_i(\theta), y_i)] \\ \frac{\partial}{\partial \theta_2}[\text{loss}(\hat{y}_i(\theta), y_i)] \\ \vdots \\ \frac{\partial}{\partial \theta_q}[\text{loss}(\hat{y}_i(\theta), y_i)] \end{pmatrix},$$

and,

$$H_i = \frac{\partial^2}{\partial \theta \partial \theta^T}[\text{loss}(\hat{y}_i(\theta), y_i)] = \begin{pmatrix} \frac{\partial}{\partial \theta_1}[g_1^T(\theta)] \\ \frac{\partial}{\partial \theta_2}[g_2^T(\theta)] \\ \vdots \\ \frac{\partial}{\partial \theta_q}[g_q^T(\theta)] \end{pmatrix}.$$

```
Imodel = Imodel.detach().numpy()
print()
print(np.round(Imodel,3))
```

```
[[ 0.153  0.019  0.011  0.013 -0.002  0.011  0.013]
 [ 0.019  0.093  0.07   0.097  0.078  0.03   0.015]
 [ 0.011  0.07   0.088  0.079  0.066  0.027  0.011]
 [ 0.013  0.097  0.079  0.129  0.091  0.028  0.022]
 [-0.002  0.078  0.066  0.091  0.08   0.027  0.012]
 [ 0.011  0.03   0.027  0.028  0.027  0.162 -0.095]
 [ 0.013  0.015  0.011  0.022  0.012 -0.095  0.16 ]]
```

There is actually a very small difference which may comes from regularization in the loss or just numerical differences.

```
print(np.mean(np.abs((I-Imodel)/I)))
```

```
3.234571e-07
```

For deep neural networks with hidden layers, the weights are likely to be in a matrix for each layer, thus each matrix parameter has to be vectorized by reshaping and by appending the rows

mentation which was altered for the wanted purpose.

or columns in order to make an unique vector from all these vectorized parameters. Note that recently a submodule for pytorch has been released in the version 1.12 for computing the hessian matrix, see "functorch", at "https://pytorch.org/functorch", associated to "vmap" for vectorization, "vjp" and "jvp", this is out of the scope herein. Note also that one may have to reduce the list of parameters to those which are marked for derivation, when all the parameters are not involved for the gradient. This happens for instance when performing the training of the classifier only, while leaving the first layers for the transformation of the images unchanged in "transfer learning" which used a pre-trained "convolution neural network" for images classification. An alternative approximation of the hessian is with the gradients for each data, as follows.

6.3.8 Hessian from first-order derivatives

An estimation of the hessian is available via the gradients only, let compare such solution with the previous one. In pytorch, the gradient is computed with a numerical procedure for minibatches for the training algorithm, thus, the same approach with minibatches of size one leads to the wanted alternative.

With q the number of parameters (bias and components of the weights) and $\theta = (\theta_1, \theta_2 \cdots, \theta_q)^T$, this corresponds to the definion $\tilde{H} = \sum_b^B \sum_{i \in s_b} \tilde{H}_i$ for the approximate hessian,

$$
\begin{aligned}
g_i &= \frac{\partial}{\partial \theta} [\text{loss}(\hat{y}_i(\theta), y_i)] \\
&= \begin{pmatrix} \frac{\partial}{\partial \theta_1} [\text{loss}(\hat{y}_i(\theta), y_i)] \\ \frac{\partial}{\partial \theta_2} [\text{loss}(\hat{y}_i(\theta), y_i)] \\ \vdots \\ \frac{\partial}{\partial \theta_q} [\text{loss}(\hat{y}_i(\theta), y_i)] \end{pmatrix},
\end{aligned}
$$

and,

$$
H_i \approx \tilde{H}_i = g_i g_i^T .
$$

The new estimation is first initialized to zero. The loop computes the gradients and accumulates the products. Here the gradients are required for each data sample vector, but the parameters are not updated because they were found just before.

```
def f_get_p_model(model):
    p_model = 0
    for p in model.parameters():
        if len(p.shape)>1:        # matrix          : array 2 dims
            p_model += p.shape[0] * p.shape[1]
        else:
            p_model += p.shape[0] # vector or scalar:  array 1 dim
    return p_model
```

For the case considered, a matrix of weights is not met because only the vector of regression coefficients is modeled. For deep networks, matrices are required for modeling the weights in order to compute the sum from a layer to another layer where the dimensions or numbers of nodes

are more than one. The loss is with a sum instead of a mean for each minibatch in order to retrieve directly a whole average for the considered sample.

```
def f_varianceMatrixFromGradients_ForParameters_modelNN(model, loss,␣
 ↪dataloader, n_train):
    p_model = f_get_p_model(model)
    Iapprox = torch.zeros((p_model,p_model))
    optimizer = torch.optim.SGD(model.parameters(), lr=0.0, momentum=0.0)
    optimizer.zero_grad()

    for b,(Xb,yb) in enumerate(dataloader):
        print(".", end = '')
        for i in range(Xb.shape[0]):  # to change into matrix diag
            Xb_i = Xb[i,:]
            yb_i = yb[i].ravel()
            yhatb_i = model(Xb_i).ravel()
            loss_b = loss(yhatb_i, yb_i)
            optimizer.zero_grad()
            loss_b.backward()
            gradient_vect = []
            with torch.no_grad():
                for p in model.parameters():
                    gradient_vect.append(p.grad.view(-1))
                gradient_vect = torch.cat(gradient_vect)
                gradient_vect = gradient_vect.reshape((p_model,1))
                Iapprox = Iapprox + \
                          gradient_vect @ gradient_vect.T /n_train
    return Iapprox, p_model
```

```
Iapprox, p_model = \
   f_varianceMatrixFromGradients_ForParameters_modelNN(model, loss,
                                                       dl_train, n_train)

Iapprox          = Iapprox.detach().numpy()
```

...

Let finally compare the resulting standard errors from the two estimators.

```
stdI1 = np.sqrt(np.diag(np.linalg.inv(I))/n_train)
stdI2 = np.sqrt(np.diag(np.linalg.inv(Iapprox))/n_train)
stdI2 = np.roll(stdI2,1)
print(np.round(stdI1,3))
print(np.round(stdI2,3))
```

```
[0.054 0.171 0.107 0.134 0.202 0.064 0.063]
[0.055 0.167 0.044 0.135 0.167 0.063 0.064]
```

```
np.asarray(lgtsumy.tables[1].data)[:,2][1:]
```

```
array(['    0.056', '    0.176', '    0.122', '    0.136', '    0.209',
       '    0.064', '    0.063'], dtype='<U10')
```

This alternative estimator of the Fisher matrix information may be less accurate (with this sample size, and with smaller resulting standard-deviations, but also some runs of the algorithm with different initializations) for the estimation of the variance than the more usual one (which is exact for the logistic regression). It cannot suffer of numerical problems coming from the algorithmic evaluation of the second-order derivative. The matrix $\tilde{H}$ is also relevant for a second-order inferential procedure as studied next chapter.

6.4 Exercices

1. (stat+numpy) Write the function for the variance of the coefficient regression vector for the multinomial regression and compare the result with the output of statsmodels for the "iris dataset".

2. (stat+numpy) Compute the hessian matrix and the corresponding estimation of the variance matrix for the parameters when the dataset is from the example of chapter 4, from the data file "xy_2d_diskandnoise_reglogistic.txt". Does the estimation of the variance is relevant, explain why and propose an alternative method (with pytorch or eventually "scipy.optimize" by reimplementing the network model, like for the polynomial regression at chapter 1).

3. (stat+numpy) In order to better understand the issue with the dataset in the exercice just before, generate several datasets of same size with a circle within a cicle or a square within a square or any other convex regular shape, with an increasing surface for the outer geometrical objet (circle or square or other). Compute the hessian for the different datasets and different sizes of datasets, discuss the results.

4. (stat+numpy) Compute a candidate numerical value for the standard-deviation of the hidden latent layers with an update from the functions above (for enough large datasets in order to get positive diagonal terms after inversion of the hessian matrix). Find also a candidate numerical value for the variance for the predicted target variables $\hat{y}_i$ from a deep regression with one hidden layer.

5. (stat+numpy) a) After testing the lasso for the deep regression model from the introduction chapter which allows to fit the data from a polynomial regression, find the variance of the parameters with the hessian here. Compare with a resampling method such that boostrap. If the hessian is not well-defined explain why. b) Retry this by cancelling the zero terms and by removing them from the gradients and hessian. How this changes the results. c) Propose an experiment with artificial data for this same regression model in order to find which size of the dataset is required in order to get a well-defined and invertible hessian matrix, and also

if the distribution of the x_i or the range has an influence on this result. Does this result could change with a different polynomial function and how this could changes with the degree of the polynomial function.

6. (stat+numpy) a) Propose a method for the variance of the frontier with the example of classification from dataset "xy_2d_reglogistic.txt". Check if the true frontier is in the corresponding interval at 95% of confidence. b) Idem for the dataset "abalone".

7. (stat+numpy) Retrieve for the logistic regression the result about the alternative expression of the information matrix, by showing that $\mathbf{E}[(y_i - p_i)^2] = p_i(1 - p_i)$. Here, a clue is that $y_i^2 = y_i$ is a binary quantity, while $E[y_i] = p_i$. Check the result by a numerical simulation.

8. (stat) Retrieve the expression of the Fisher scoring for the logistic regression.

Chapter 7

Second-order training of (deep) glm

In this chapter, we are interested on the second-order estimation for nonlinear models such as (deep) generalized linear models where the distributions for modeling the dependence of the target variable y_i w.r.t the variables x_i belongs to the exponential family. For datasets of moderated size, the classical method for the optimization is the "Newton-Raphson procedure" (or NR) which uses the hessian matrix for rescaling the gradient vectors at each iteration. For larger datasets, the methods named "Quasi-Newton" (or NQ) prefer to approximate the hessian and the product with the gradient in order to avoid numerical issues coming from high dimensional spaces.

On the contrary to the first-order methods, the Newton-Raphson procedures are not suitable if the neural network is too large (too many parameters) with the current computer architecture because the hessian does not fit in memory or is too slow to compute. Another limit is when the model itself is too flexible and the dataset too small such that the variance is not reliable: the hessian matrix is ill defined, not invertible or gives negative values for the variance. It is supposed that all is going well here, otherwise a variable selection, a weights selection, a tuned regularization or if required a simpler model are often able to solve this issue.

In the first chapters, we have studied the linear models, their nonlinear extensions within the framework of the neural networks, and several iterative algorithms which update the parameters in order to minimize the loss functions (or maximize a loglikelihood). The chapter just before has discussed the second-order derivatives aggregated in the hessian matrix and the related variance for (linear) regression and classification when modeled with pytorch.

The hessian and its previous approximations are involved in the first part of the current chapter via an implementation with numpy for the second-order training of a regression model. A procedure from quasi-Newton methods concludes the chapter in a second part with pytorch via its available dedicated function.

7.1 Introduction to GLM

The family of models named generalized linear models (GLM), for regression and classification, is defined via the distribution named "exponential family" associated to a transformation.

7.1.1 Definitions

let denote θ for the natural parameter and ϕ for the dispersion parameter with positive values. The target variable y_i belongs to an exponential family with density of a very general form,

$$p(y|\theta,\phi) = \exp\left\{\frac{y\theta - b(\theta)}{a(\phi)} + c(y,\phi)\right\}.$$

In the generalized linear models the mean $\mu = \mathbf{E}(Y|x)$ is written via a function called link function which is a strictly increasing function, which is applied to the usual inner product of the parameter vector β with the vectors of explicative variables (plus bias one for the intercept) x_i. The link function $\eta(.)$ has an inverse $\eta^{-1}(.)$ and,

$$\begin{array}{rcl} g(\mu) & = & \eta(x^T\beta) \\ \mu & = & \eta^{-1}(\eta) \end{array}$$

For instance, these link functions are defined just below for two distributions in the regression problems met every days when dealing with datasets

- Gaussian: $\eta = \mu$ and 1 for the variance function.
- Poisson : $\eta = \log\mu$ and μ for the variance function.

Other link functions are available for the Inverse Gaussian, Gamma, Multinomial, Binomial and Poisson regressions among other ones.

7.1.2 Derivatives and variance

From any statistical textbook, the score or first-order derivative of the loglikelihood is obtained as follows:

$$\nabla\ell(\beta) \quad = \quad \sum_{i=1}^{n} \frac{(y_i - \mu_i)\mu_i'(\eta_i)}{\sigma_i^2} x_i \,.$$

The Hessian or second-order derivative of the loglikelihood is thus written:

$$\nabla^2\ell(\beta) \quad = \quad -\sum_{i=1}^{n}\left\{\frac{[\mu_i'(\eta_i)]^2}{\sigma_i^2} - \sum_{i=1}^{n}\frac{(y_i-\mu_i)\mu''(\eta_i)}{\sigma_i^2}\right\} x_i x_i^T \,.$$

The Fisher information matrix is the expectation of the hessian, with $E[y_i] = \mu_i$, while $\Omega = \mathrm{Diag}(\frac{[\mu_i'(\eta_i)]^2}{\sigma_i^2} : 1 \leq i \leq n)$, thus,

$$I_n(\beta) \quad = \quad \mathbf{E}[-\nabla^2\ell(\beta)] = \sum_{i=1}^{n}\frac{[\mu_i'(\eta_i)]^2}{\sigma_i^2} x_i x_i^T = X^T\Omega X \,.$$

The hessian or Fisher information matrix are the covariance matrix when inverted and divided by the sample size after, as explained in the previous chapter. This matrix is also suitable for decreasing the loss instead of just considering the first-order derivative, with the additional burden of its computation cost.

7.2 Algorithms for second-order training

A deep linear regression can be interpreted as a statistical nonlinear regression with a particular expression for the nonlinearities. Historically, the Gauss-Newton-method was introduced for inference of the parameters of such model, but with growing number of variables and large datasets, first-order algorithms are usually more practicable.

Thus the following algorithms are for a moderated size network, except in case of fast computers which are not available for every ones. The second-order approximation is presented just before the algorithms, this is the foundation.

7.2.1 Taylor serie at second-order

For the training procedure, a second-order approximation around a current solution denoted $\beta^{(t)}$ or here $\beta_{(t)}$ is written:

$$\ell(\beta) \approx \ell(\beta_{(t)}) + \nabla\ell(\beta_{(t)})^T(\beta - \beta_{(t)}) + \frac{1}{2}(\beta - \beta_{(t)})^T[\nabla^2\ell(\beta_{(t)})](\beta - \beta_{(t)})\,.$$

The optimization solves for the derivative of this approximation:

$$\nabla\ell(\beta_{(t)}) + [\nabla^2\ell(\beta_{(t)})](\beta - \beta_{(t)}) = 0_p\,.$$

Except for the Gaussian case with canonical link, the problem to solve is nonlinear and needs a gradient ascent.

The corresponding updates are thus, with a ridge term with parameter τ for insuring the inverse of the hessian matrix, as follows:

$$\beta_{(t+1)} = \beta_{(t)} - \alpha_t\,[H_t + \tau I_p]^{-1}\nabla\ell(\beta_{(t)})\,.$$

It is recognized exactly the same update formula than for the algorithm next paragraph.

7.2.2 Newton-Raphson procedure

Let remember from the previous chapters, we want to estimate regression or classification coefficients with a vector $\hat{\beta}$. The cost function or loss is minimized such that,

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}}\,\ell(\beta)\,.$$

We are still interested with minibatches but eventually also the full dataset if it is not too large, thus, with a number of B minibatches where eventually $B = 1$, the loss function is still written as,

$$\begin{array}{rcl} \ell(\beta) & = & \sum_{i=b}^{B}\ell_b(\beta) \\ & = & \sum_{b=1}^{B}\sum_{i\in s_b} -\log f(y_i; x_i, \theta) \\ & = & \sum_{b=1}^{B}\sum_{i\in s_b}\ell_i(\beta)\,. \end{array}$$

- The second-order update formula for $\beta_{(t)}$ is now written from the formula of the paragraph above,

$$\frac{d}{d\beta}\Big(\ell_b(\beta)\Big) = \left[\begin{array}{c} \frac{\partial}{\partial\beta_0}\Big(\ell_b(\beta)\Big) \\ \frac{\partial}{\partial\beta_1}\Big(\ell_b(\beta)\Big) \end{array}\right].$$

The new update formula of the parameters at each iteration remains with the gradient, but with the hessian which appears here as a multiplicative corrective factor such that if $\tau = 0$ the second-order update is:

$$\left[\begin{array}{c} \beta_0^{(t+1)} \\ \beta_1^{(t+1)} \end{array}\right] = \left[\begin{array}{c} \beta_0^{(t)} \\ \beta_1^{(t)} \end{array}\right] - \alpha_t \left[\begin{array}{cc} \frac{\partial^2}{\partial\beta_0\partial\beta_0^T}\Big(\ell_b(\beta)\Big) & \frac{\partial^2}{\partial\beta_0\partial\beta_1^T}\Big(\ell_b(\beta)\Big) \\ \frac{\partial^2}{\partial\beta_1\partial\beta_0^T}\Big(\ell_b(\beta)\Big) & \frac{\partial^2}{\partial\beta_1\partial\beta_1^T}\Big(\ell_b(\beta)\Big) \end{array}\right]^{-1} \frac{d}{d\beta}\Big(\ell_b(\beta)\Big).$$

A learning rate may be required because the second-order serie around the current solution may be not enough accurate, such as the update may be reduced in order to insure the convergence despite the gap with the real function.
- This is implemented next after with numpy for the Poisson regression (see next section for the expression of the loss and loglikelihood) where the target variable takes positive integer values or counts, such that the new distribution involved is changed. With $D_\mu = \text{Diag}(\mu_{(k)})$, the update is just for the NR algorithm:

$$\begin{aligned} \beta_{(k+1)} &= \beta_{(k)} - \alpha_t \left(\frac{d^2\ell(\beta_{(k)})}{d\beta d\beta^T} + \tau I_p\right)^{-1} \frac{d\ell(\beta_{(k)})}{d\beta} \\ &= \beta_{(k)} - \alpha_t \left(\sum_i \mu_i x_i x_i^T + \tau I_p\right)^{-1} \left[\sum_i (y_i - \mu_i) x_i\right] \\ &= \beta_{(k)} - \alpha_t \left(X^T D_\mu X + \tau I_p\right)^{-1} \left[X^T (y - \mu)\right] \end{aligned}$$

7.2.3 Natural gradient procedure

When denoting $D_{y-\mu} = \text{Diag}(y - \mu_{(k)})$ and with the alternative expression for the hessian matrix, this leads for the Poisson regression to the approximated NR algorithm:

$$\begin{aligned} \beta_{(k+1)} &= \beta_{(k)} - \alpha_t \left(\sum_i \frac{d\ell_i(\beta_{(k)})}{d\beta}^T \frac{d\ell_i(\beta_{(k)})}{d\beta} + \tau I_p\right)^{-1} \frac{d\ell(\beta_{(k)})}{d\beta} \\ &= \beta_{(k)} - \alpha_t \left(\sum_i (y_i - \mu_i)^2 x_i x_i^T + \tau I_p\right)^{-1} \left[\sum_i (y_i - \mu_i) x_i\right] \\ &= \beta_{(k)} - \alpha_t \left(X^T D_{y-\mu}^2 X + \tau I_p\right)^{-1} \left[X^T (y - \mu)\right] \end{aligned}$$

According to the experiments, this version has converged to a solution near the optimal one but more slowly: a correcting factor is required like a learning rate. Also the convergence may be not insured on the contrary to the usual NR algorithm. In the other hand, this allows a sequential approach with minibatches for consecutive computations of the inverse of the (approximated) hessian matrix as explained next.

7.2.4 First-order gradient update with minibatches

Following the first chapters, with s_b the data subset for the minibatch at iteration k from one epoch,

$$\beta_{(k+1)} = \beta_{(k)} - \alpha_t \frac{d\ell(\beta_{(k)})}{d\beta}.$$

This is written for the Poisson regression:

$$\beta_{(k+1)} = \beta_{(k)} + \alpha_k \sum_{i \in s_b} (y_i - \mu_i) x_i \,.$$

7.2.5 Natural gradient procedure with minibatches

When adding the gradients from each minibatch, the approximated hessian matrix is inverted sequentially thanks to linear algebra without explicit inversion but only with the Sherman-Morrison formula and also for faster processing the Woodbury identity. The mean is preferred to the sum, otherwise, the learning rate may need to change at each minibatch, as the sum is growing, while here it is kept constant.

Let denote $g_i = \frac{d\ell_i(\beta_{(k)})}{d\beta}$ the gradient vector for x_i. For simpler notation, the data are reordonned according to the shuffling if there is any. The size of a minibatch is m and the minibatch index is b, with next $b+1$. Note that it may happen that the last minibatch is smaller than m for Euclidean division reason (n divided by B), but this is not considered here.

- Adding one gradient vector one after the other for each minibatch (until m times)
  First, let consider the case when a data vector is added with, $m = m_b = |s_b|$ the size of the minibatches, at epoch k. Such as the current minibatch is number b_k with a total of B minibatches, supposed of equal sizes for lighter notation, thus, $b_k m = \sum_{b=1}^{b_k} |s_b|$, and, when adding a new gradient vector to the approximation of the hessian knowing that m should be added for a full minibatch,

$$\begin{aligned}
\tilde{H}_{b_k m+1} &= \frac{1}{\sum_{b=1}^{b_k} |s_b|+1} \left[b_k m \left(\frac{1}{\sum_{b=1}^{b_k} |s_b|} \sum_{b=1}^{b_k} \sum_{i \in s_b} g_i^T g_i \right) + g_\ell^T g_\ell \right] \\
&= \frac{1}{\sum_{b=1}^{b_k} m_b+1} \left[b_k m \left(\frac{1}{b_k m} \sum_{i=1}^{b_k m} g_i^T g_i \right) + g_\ell^T g_\ell \right] \\
&= \frac{1}{b_k m+1} \left(b_k m \tilde{H}_{b_k m} + g_\ell^T g_\ell \right) .
\end{aligned}$$

  From the Sherman-Morrison formula, this leads to repeat m times, with $\ell = \ell_k^s$ such that for $1 \leq s \leq m$, and for example $s = 1$,

$$\begin{aligned}
\tilde{H}_{b_k m+1}^{-1} &= (b_k m+1) \left(b_k m \tilde{H}_{b_k m} + g_\ell^T g_\ell \right)^{-1} \\
&= (b_k m+1) \left(\frac{1}{b_k m} \tilde{H}_{b_k m}^{-1} - \frac{1}{1+g_\ell^T \frac{1}{b_k m} \tilde{H}_{b_k m}^{-1} g_\ell^T} \frac{1}{b_k m} \tilde{H}_{b_k m}^{-1} g_\ell^T g_\ell \frac{1}{b_k m} \tilde{H}_{b_k m}^{-1} \right) \\
&= \frac{b_k m+1}{b_k m} \left(\tilde{H}_{b_k m}^{-1} - \frac{1}{b_k m} \frac{1}{1+g_\ell^T \frac{1}{b_k m} \tilde{H}_{b_k m}^{-1} g_\ell^T} \tilde{H}_{b_k m}^{-1} g_\ell^T g_\ell \tilde{H}_{b_k m}^{-1} \right) .
\end{aligned}$$

  This induces that the inversion is in closed-form from the previous step. When there are several data vectors coming from the new minibatches, this formula is involved m times, as the size of the minibatches. Let denote, $a_1 = \frac{mb}{m(b+1)}$ with $a_1 = 1$ at first minibatch of each epoch, and $a_2 = \frac{1}{m(b+1)}$. In the python code, the following update was tried and kept because the average is for a minibatch, and this seems to lead to a stable convergence here for different sizes of dataset, learning rate or minibatch sizes,

$$\tilde{H}_{bm+1}^{-1} = a_1 \tilde{H}_{bm}^{-1} - a_2 \frac{1}{1+a_2 g_\ell^T \frac{1}{bm} \tilde{H}_{bm}^{-1} g_\ell^T} \tilde{H}_{bm}^{-1} g_\ell^T g_\ell \tilde{H}_{bm}^{-1} \,.$$

For this dataset, the final result was relevant according to the numerical experiments with numpy (and also when changing the parameters for generating variants of the dataset for a poisson regression).

- Adding m gradient vectors for the whole minibatch (only 1 time)
The solution from the previous paragraph is slow for large m or B because a sum is involved, hence, a matricial version is preferred as follows with G_b aggregating the m vectors of gradients g_i from the minibatch b_k. This allows the Woodbury identity, by adding the whole minibatch of gradient vectors. It follows when denoting with just the index k from the minibatch b_k in the approximated hessian and $n_k = b_k m$, for a lighter notation:

$$\begin{aligned}
\tilde{H}_{k+1} &= \frac{1}{n_{k+1}}\left(n_k\tilde{H}_k + \sum_{\ell\in s_{k+1}} g_\ell^T g_\ell\right)\\
&= \frac{1}{n_{k+1}}\left(n_k\tilde{H}_k + G_{k+1}I_{k+1}G_{k+1}^T\right)\\
\tilde{H}_{k+1}^{-1} &= n_{k+1}\left\{\frac{1}{n_k}\tilde{H}_k^{-1} - \frac{1}{n_k}\tilde{H}_k^{-1}G_{k+1}\left(I_{k+1} + G_{k+1}^T\frac{1}{n_k}\tilde{H}_k^{-1}G_{k+1}\right)^{-1}G_{k+1}^T\frac{1}{n_k}\tilde{H}_k^{-1}\right\}\\
&= \frac{n_{k+1}}{n_k}\left\{\tilde{H}_k^{-1} - \frac{1}{n_k}\tilde{H}_k^{-1}G_{k+1}\left(I_{k+1} + G_{k+1}^T\frac{1}{n_k}\tilde{H}_k^{-1}G_{k+1}\right)^{-1}G_{k+1}^T\tilde{H}_k^{-1}\right\}\\
&\approx \tilde{H}_k^{-1} - \frac{1}{n_k}\tilde{H}_k^{-1}G_{k+1}\left(I_{k+1} + G_{k+1}^T\frac{1}{n_k}\tilde{H}_k^{-1}G_{k+1}\right)^{-1}G_{k+1}^T\tilde{H}_k^{-1}.
\end{aligned}$$

The simple expression just above seems slightly different from the proposed approaches from the literature. The expression is fully matricial and might be faster than repeating n times the Sherman-Morrison formula, at least for languages such as python with here numpy for processing the algebra. A perspective is to add some sketching with a random projection or pca (see next chapter about reductions), not discussed here.

Next, the different formula for updating the parameters and decreasing the loss function are compared with an artificial dataset for the Poisson regression.

7.3 Fitting a Poisson regression with statsmodels

The work directory is given from the function.

```
def towdir(s):
    return (str('./datasets_book/'+s))

import deepglmlib.utils as utils
import numpy as np
```

```
import importlib
importlib.reload(utils)
```

Let clean and check the computer memory state. There should be remaining some space otherwise the operating system risks to freeze.

```
import gc
gc.collect()
```

```
35
```

```
import psutil
memory = psutil.virtual_memory()
print(f" Memory used      : {memory.percent} %\n",
      f"Memory available : { round(memory.free / (1024.0 ** 3),2)} GB")
```

```
 Memory used      : 36.1 %
 Memory available : 5.14 GB
```

```
# import torch
# torch.cuda.empty_cache()
```

```
# !nvidia-smi
```

Dataset from the files

A data sample of 5000 observations from a Poisson regression with 6 independent variables is generated and stored in the computer files. The train and test subsamples are sampled from the whole sample. The sample is split into the train and test subsamples in order to check the model on new data. They are stored as files in the computer disk with also the true parameters. The datasets and the true parameters are loaded as follows.

```
n  = 5000
p1 = 7
p  = p1-1

print(n, p1, p)
```

```
5000 7 6
```

```
import numpy as np

X_train   = np.loadtxt(towdir("poisson_n5000_d7_Xtrain.txt"))
X_test    = np.loadtxt(towdir("poisson_n5000_d7_Xtest.txt"))
y_train   = \
   np.loadtxt(towdir("poisson_n5000_d7_ytrain.txt")).astype(np.int64)
y_test    = \
   np.loadtxt(towdir("poisson_n5000_d7_ytest.txt")).astype(np.int64)
beta0     = np.loadtxt(towdir("poisson_n5000_d7_beta0.txt"))
```

```
mu0       = np.loadtxt(towdir("poisson_n5000_d7_mu0.txt"))
idx_test  = \
   np.loadtxt(towdir("poisson_n5000_d7_idxtest.txt")).astype(np.int64)
idx_train = \
   np.loadtxt(towdir("poisson_n5000_d7_idxtrain.txt")).astype(np.int64)

n_train, p_train = X_train.shape
n_test, p_test   = X_test.shape
```

The true regression coefficients are supposed unknown during the training, this is the purpose of the model fitting to retrieve this vector. This induces that usually, the loglikelihood (and related bic aic or other ones) is mostly the only value (plus eventually the residuals and related) which is in stake in order to compare several models. For generated data, the true β and the true expectations $\mu_i = e^{\beta^T x_i}$ are known thus they can be compared with the estimated ones, $\hat{\beta}$ and $\hat{\mu}_i = e^{\beta^T x_i}$ after training.

Checking the target variable

The discrete distribution of the target variable y is visualized with a "barplot", after counting the number of observations per possible value of the variable. This is with the python class "Counter" from the module "collections" which asks for a list for the input sequence. The graphic is from the module "matplotlib" while "seaborn" allows more advanced outputs not used here.

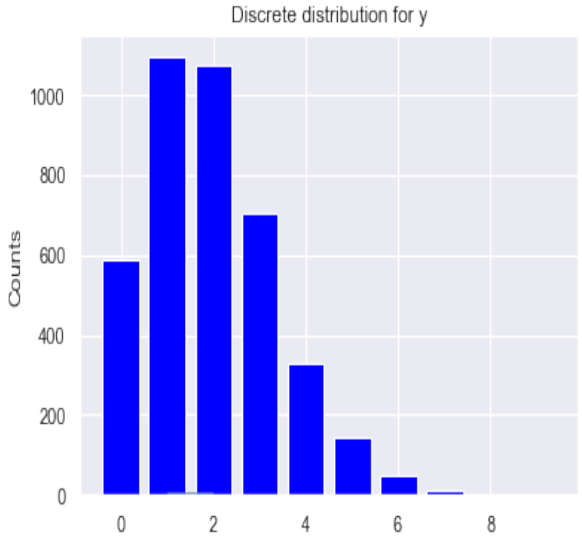

Figure 7.1: Barplot for the target variable from Poisson regression

```
from collections import Counter as counter
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot style

def f_barplot_poisson(y,printed_bars=True,printed_counts=True,labely="y"):
    dico_y = counter(y.squeeze().tolist())
    keys_y = np.sort([ int(k) for k in dico_y.keys()])
    if printed_counts:
        print([ (int(k),dico_y[k]) for k in keys_y])
    if printed_bars:
        plt.bar(dico_y.keys(),dico_y.values(),color="blue")
        plt.title(r'Discrete distribution for '+labely)
        plt.xlabel(r' ') #plt.xlabel(r'$rings$')
        plt.ylabel(r'Counts')
        plt.hlines(0,1,2)
        plt.show()
    return dico_y, keys_y

dico_y_train, keys_y_train = f_barplot_poisson(y_train,True,False)
```

This graphic with a barplot would suggest that there may be an excess of zeros but checking the mean and the variance confirms that the regular usual Poisson distribution is a good candidate:

```
print(f"mean(y_train) = {np.mean(y_train):2.4f}",
      f"var(y_train)  = {np.var(y_train):2.4f}")
print(f"mean(y_test) = {np.mean(y_test):2.4f}",
      f"var(y_test)  = {np.var(y_test):2.4f}")
```

```
mean(y_train) = 1.9475 var(y_train)  = 1.9832
mean(y_test) = 1.9110 var(y_test)  = 2.0011
```

The first component in X_i contains only the value one in order to include the intercept in the vector of regression coefficients β, otherwise this would be x_i for usual notation. This avoids to separate the intercept from the vectors, in order to get all the unknown quantities aggregated in an unique vector β. In the python modules such as sklearn, this is often possible to separate or include both, but for computation with numpy the algebra is lighter with an unique vector instead of a vector plus a scalar.

With the normalization of the columns, the true beta is not retrieved even approximately, but the parameters in the Poisson μ_i should be retrieved with the estimation $\hat{\mu}_i = e^{x_i^T \hat{\beta}}$ because the expectations remains identical.

Training of the regression model

This model is fitted via a second-order procedure with a python module.

```
import statsmodels.api as stm
ols= stm.Poisson(y_train, X_train)
fit_ols_train = ols.fit()
olssumy= fit_ols_train.summary()
```

```
Optimization terminated successfully.
         Current function value: 1.694970
         Iterations 4
```

The training was performed only in four iterations, which makes very appealing the second-order procedures when the sample is not too large. The regression coefficients are retrieved as follows:

```
beta_stm = fit_ols_train.params
beta_stm = beta_stm.reshape((len(beta_stm),1))
```

7.4 Fitting a Poisson regression with numpy

In this subsection, we are interested on inference procedures with the full dataset at each iteration (also called batch algorithms) and on incremental or sequential procedures with parts of the dataset at each iteration (also called minibatch algorithms). The four algorithms are in an unique python function in order to make easier the access to their contents and their call:

- "nr" for Newton-Raphson procedure on the whole data sample, with the exact hessian matrix from second-order derivatives.
- "ef" for natural gradient procedure on the whole data sample, with the approximated hessian from first-order derivatives.
- "gd" for gradient descent with minibatches, as in the first chapters, with the gradient vectors or first-order derivatives.
- "efseq" for natural gradient with minibatches with a sequential inverse of the matrix, with the approximated hessian.

Note that "ef" stands for "estimated fisher matrix", which is chosen in our case equal to the one previously called "approximated hessian", hence another naming could be "ng" instead for instance. The natural gradient procedure is like the newton-raphson procedure except that the hessian is the approximated one. The ridge correction is not discussed anymore because it has begun usual in any implementation. There exists precise differences between the numerous algorithms from the literature hence the reader might look for a review on the history of neural networks or statistical inference for more details about, this is out of the scope here.

7.4.1 Expressions of the gradient and hessian

With $\mu_i = \mu_i(\beta) = e^{\beta^T x_i}$, the Poisson regression is a member of the generalized linear model such that,

$$f(y_i;\beta) \quad = \quad \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}.$$

Thus, the loglikelihood is defined as the log of the n products from the mass functions above, one per sample observation, such that,

$$\begin{aligned}\ell(\beta) &= \log \mathcal{L}(\beta) \\ &= \sum_{i=1}^{n} \log f(y_i;\beta) \\ &= \sum_{i=1}^{n} \log\left(\frac{\mu_i^{y_i}}{y_i!}e^{-\mu_i}\right) \\ &= \sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y_i!\,.\end{aligned}$$

The maximum likelihood solution is:

$$\hat{\beta} = \underset{\beta}{argmax} \sum_{i=1}^{n} \left(y_i \beta^T x_i - e^{\beta^T x_i}\right).$$

For the optimization with $G_k = \frac{d\ell(\beta)}{d\beta}\Big|_{\beta_{(k)}}$ and $H_k = \frac{d^2\ell(\beta)}{d\beta d\beta^T}\Big|_{\beta_{(k)}}$, this leads to compute the derivatives as given in the previous section just above: this is a good exercice to retrieve these expressions.

7.4.2 Implementation of the four algorithms

In this subsection, the algorithms presented just before are tested with numpy by writing a loop and by updating the parameters at each iteration. On the contrary to the first-order procedures, the second-order procedures needs the hessian, hence this matrix is computed just after the gradient at each iteration. The python functions are defined as follows.

```
from scipy.special import factorial, gammaln

def f_gradient_poisson(X_,y_,mu_):
    return X_.T @ (y_ - mu_)     # gradient
```

```
class UpdaterMinibatchNaturalGradient:

    def __init__(self,p,alpha):
        self.p = p
        self.alpha = alpha
        self.Hinv = np.zeros((p,p))

    def resetHinv(self):
        self.Hinv=0*self.Hinv
        for k in range(self.p):
            self.Hinv[k,k] = 1/self.alpha

    def update(self,b,i_s,Gb):
        Ib = np.eye(Gb.shape[1])

        if b>1:
            a_b = i_s[b]/i_s[b-1]
```

```
            b_b = 1/i_s[b]

        if b<=1:
            a_b = 1
            b_b = 1

        IGHGinv    = np.linalg.inv(Ib+b_b*Gb.T@self.Hinv@Gb)
        self.Hinv  = \
          a_b*(self.Hinv-b_b*self.Hinv@Gb@IGHGinv@Gb.T@self.Hinv)

        gradient_b = - np.sum(Gb,axis=1).reshape((self.p,1))
        update_b   = self.Hinv @ gradient_b

        return update_b
```

The previous class is for the sequential inverse of the matrix in the natural gradient procedure with minibatches while the following function is for the fitting of the Poisson model with the fours approaches.

```
import time
```

```
def f_fit_poisson(X_,y_,n_epoqs=800, size_batch=50, alpha_t = 1e-4,
                  diff=1e-5, alpha=0.01, show=1, algo="nr",bound=2):

    start_time = time.time()
    n_, p_     = X_.shape
    y_         = y_.reshape(n_,1)

    beta_init  = 0*(np.random.uniform(size=p_,low=0.01,high=1.0)/p_ \
                 -0.5/p_).reshape(p_,1)
    beta       = beta_init.reshape(p_,1)
    mu         = np.exp(X_ @ beta)

    logL       = np.sum(y_ * np.log(mu) - mu - gammaln(y_+1))
    logLik_s   = []
    logLik_s.append(logL)

    if algo!="nr" and algo!="ef":      ## with minibatches
        H_old  = alpha * np.eye(len(beta))
        i_s = np.array( \
               np.mgrid[0:n_:complex(real=0,imag=size_batch)],dtype=int)
        if algo=="efseq1" or algo=="efseq1_diag": \
            Hinv=Hinv_old= (1/alpha) * np.eye(len(beta))
```

```
if algo=="nr" or algo=="ef":      ## with whole batch
    H_old      = 0.001 * np.eye(len(beta))

if algo=="efseq":
    updater_mnng = UpdaterMinibatchNaturalGradient(p_,alpha)

for epoch in range(0,n_epoqs):

    ##---  BEGIN FULL-BATCHES algoS ---
    if algo=="nr" or  algo=="nr_diag":
        f_hessian_poisson = f_hessian_poisson_newton

    if algo=="ef" or algo=="ef_diag":
        f_hessian_poisson = f_hessian_poisson_natural

    if algo=="nr" or algo=="ef" \
        or algo=="nr_diag" or algo=="ef_diag":
        mu       = np.exp(X_ @ beta)
        g        = f_gradient_poisson(X_,y_,mu)
        H        = f_hessian_poisson(X_,y_,mu,g,H_old)
        H        = H - alpha*np.eye(len(H))

        if algo=="nr_diag" or algo=="ef_diag":
            Hinv = np.zeros((p_,p_))
            for k in range(p_): Hinv[k,k] = 1/H[k,k]
            beta_new = beta - alpha_t*(Hinv @ g) #diagonal hessian
        else:
            Hinv = np.linalg.inv(H)
            beta_new = beta - (Hinv @ g) #eventually add alpha_t
    ##--- END FULL-BATCHES algoS ---

    ##--- BEGIN MINIBATCHES algoS ---
    if algo=="gd" or algo=="efseq1"\
            or algo=="efseq1_diag" or algo=="efseq":
        if algo=="efseq1" or algo=="efseq1_diag":
            Hinv=Hinv_old= (1/alpha) * np.eye(len(beta))
        for l in range(0,len(i_s)-1): #(no shuffling here)
            sb           = range(i_s[l]+1*(l>0),i_s[l+1])
            Xb           = X_[sb,:]
            yb           = y_[sb]
            mub          = np.exp(Xb @ beta)
            if algo=="gd":
                gradient_b = -Xb.T @ (yb - mub)
```

```
                gradient_b[gradient_b>bound]  = bound
                gradient_b[gradient_b<-bound] = -bound
                beta_new   = beta - alpha_t * gradient_b
            if algo=="efseq1" or algo=="efseq1_diag":
                gradient_b   = Xb.T @ np.diag(v=(yb-mub).ravel())
                gradient_b[gradient_b>bound]  = bound
                gradient_b[gradient_b<-bound] = -bound
                for i in range(gradient_b.shape[1]):
                    gb_i   = np.sqrt(mub[i])* \
                             gradient_b[:,i].reshape((p_,1))
                    if l>0: a = i_s[l]/i_s[l+1]
                    if l==0: a=1
                    b = 1/i_s[l+1]
                    Hinv   = a*Hinv_old - \
                      b/(1+b*gb_i.T @ Hinv @gb_i) * \
                      Hinv @ gb_i @ gb_i.T @ Hinv
                if algo=="efseq1_diag":
                    for k1 in range(p_):
                        for k2 in range(p_):
                            if k1!=k2:
                                Hinv[k1,k2] = 0.0
                gradient_b = - np.sum( \
                                    gradient_b,axis=1).reshape((p_,1))
                beta_new   = beta - alpha_t *Hinv @ gradient_b

            if algo=="efseq":
                if l==0: updater_mnng.resetHinv()
                gradient_b  = Xb.T @ np.diag(v=(yb-mub).ravel())
                gradient_b[gradient_b>bound]  = bound
                gradient_b[gradient_b<-bound] = -bound
                update_b = updater_mnng.update(l,i_s,gradient_b)
                beta_new   = beta - alpha_t * update_b
        ##--- END MINIBATCHES algoS ---

        normbb   = np.sqrt(np.sum((beta_new - beta)**2)/len(beta))
        beta     = beta_new
        mu       = np.exp(X_ @ beta)
        logL     = np.sum(y_ * np.log(mu) - mu - np.log(factorial(y_)))
        logLik_s.append(logL)

        if epoch>2 and logLik_s[-1]<logLik_s[-2]:
            alpha_t = alpha_t/2
        if show>1:
            print(f'Iter n°: {epoch:2d}',
```

```
                f'Log_lik = {logLik_s[-1]:2.4f}', \
                f'Dist_beta = {normbb:2.5f}',
                f'beta_hat[0] = {np.round(beta.flatten()[0],3)}')

        if normbb<diff and epoch>10:
            break
        else:
            if algo=="efseq1" or algo=="efseq1_diag": Hinv_old=Hinv
            if algo=="nr" or algo=="ef": H_old=H

    step_end = epoch
    beta_hat = beta
    logL_hat = logL
    mu_hat = np.exp(X_ @ beta_hat)
    H_hat  = -(X_.T @ (mu_hat * X_))
    std_hat = np.round(np.sqrt(np.diag(np.linalg.inv(-H_hat))),4)
    if show>=1:
        print(f'beta_hat = {np.round(beta_hat.flatten(),3)} logL_hat={np.
↪round(logL_hat,3)}')
    return {"beta":beta_hat, "std":std_hat,
            "logL":logL_hat, "mu":mu_hat,
            "step_end":step_end, "logL_s":logLik_s,
            "algo":algo, "time":(time.time() - start_time)}
```

The final loglikelihoods from each algorithm are compared at the end of the inference when the values of $\beta_{(k)}^{\text{algo}}$ for $k = 1, 2, \cdots$, become stable at the final $k = k_{\text{end}}^{\text{algo}}$. Indeed, each algorithm leads to its solution $\hat{\beta}^{\text{algo}}$ for the maximum likelihood hence a different value for the loglikelihood $\ell(\hat{\beta}^{\text{algo}})$. This does not ask for other informations than the available data sample, the usual situation in practice. As the unknown parameters are known here, it is also computed the mean squared error and the correlation coefficient from the true parameters μ_i and the estimations $\hat{\mu}_i^{\text{algo}}$ for each algorithm "algo", this allows a further analysis.

The results are checked with the correlation and the mse. The final result depends on the chosen test sample hence a cross-validation could to be preferred here. It is also better to run the fitting several times, with different random initializations and keep the best result.

```
from scipy.special import gammaln
def f_poisson_logLik(beta,X,y,name=None):
    beta = beta.reshape(len(beta),1)
    y = y.reshape((len(y),1)).astype(np.float64)
    mu_hat  = np.exp(X @ beta) #.ravel()
    logL    = np.sum(y * np.log(mu_hat) - mu_hat - gammaln(y+1))
    if name is not None: print(name+"=",np.round(logL,4))
    return logL
```

```
def f_mu_mse_cor_poisson(X,y,fit,mu0,isprint=None):
    beta = fit["beta"]
    algo = fit["algo"]
    mu_hat  = np.exp(X @ beta).ravel()
    mse_mu_hat = ( (mu_hat-mu0.ravel())**2 ).mean()
    cor_mu_hat = np.corrcoef(mu0.ravel(),mu_hat)[0,1]
    logLik = f_poisson_logLik(beta,X,y)
    if isprint is not None:
        print(str("mse_mu_"+algo+"="),np.round(mse_mu_hat,4),
              #str("cor(cor_mu_"+namethod+",mu)="),np.round(cor_mu_hat,4),
              str("logLik_"+algo+"="),np.round(logLik,4))
    return {"mu":mu_hat, "msemu":mse_mu_hat,
            "cormu":cor_mu_hat, "logL":logLik,
            "fit":fit}
```

The four algorithms are run one after the other as follows:

- For the batch methods, the Newton-Raphson updates with exact hessian.

```
def f_hessian_poisson_newton(X_,y_,mu_,g_new,H_old):
    return  -(X_.T @ (mu_ * X_)) # hessian
```

```
fit_nr = f_fit_poisson(X_train, y_train, show=-1, alpha=0.01,
                       diff=1e-5, algo="nr")
quali_nr_test = \
   f_mu_mse_cor_poisson(X_test, y_test, fit_nr, mu0[idx_test])
logL_nr_test = quali_nr_test["logL"]
print(f"logL_nr_test = {logL_nr_test:5.2f}")
```

```
logL_nr_test = -1694.39
```

The diagonal version "nr_diag" just inverses the diagonal matrix by keeping only the diagonal elements from the hessian matrix. A better way would be to compute this diagonal only, instead of zeroing the non diagonal cells.

- For the batch methods, the natural gradient updates with approximated hessian.

```
alpha_seq = 0.0001
def f_hessian_poisson_natural(X_,y_,mu_,g_new,H_old,size_chunk=500):
    g_new = X_.T @ np.diag(v=(y_-mu_).ravel())
    n_ = g_new.shape[1]
    p_ = g_new.shape[0]
    H = 0*H_old
    size_eye = size_chunk
    Ic = np.eye(size_eye)
    for idx_b in range(0, n_, size_chunk):
```

```
        idx_b2 = np.min( [idx_b+size_chunk,n_] )
        if idx_b2-idx_b != size_eye:
            size_eye = idx_b2-idx_b
            Ic = np.eye(size_eye)
        H += (g_new[:,idx_b:idx_b2] @ \
              Ic @ g_new[:,idx_b:idx_b2].T)/n_
    return -(1/alpha_seq)*(H + 0.001*np.eye(p_))
```

```
if n_train<=5000:
    fit_ef = f_fit_poisson(X_train,y_train, show=-1, alpha=0.1,
                           n_epoqs=200,diff=1e-5, algo="ef")
    quali_ef_test = \
      f_mu_mse_cor_poisson(X_test, y_test, fit_ef,mu0[idx_test])
    logL_ef_test = quali_ef_test["logL"]
    print(f"logL_ef_test = {logL_ef_test:5.2f}")
```

```
logL_ef_test = -1694.39
```

The diagonal version "ef_diag" inverses only the diagonal matrix by keeping the diagonal elements from the approximated hessian matrix.

- For the sequential methods, the updates with the gradient vectors from minibatches.

```
n_epoqs_max = 200
```

```
alpha_t=0.005
fit_gd = f_fit_poisson(X_train, y_train, show=-1, alpha_t=alpha_t,
                       n_epoqs=n_epoqs_max, size_batch=8, diff=1e-5,
                       algo="gd")
quali_gd_test = \
  f_mu_mse_cor_poisson(X_test, y_test, fit_gd,mu0[idx_test])
logL_gd_test = quali_gd_test["logL"]
print(f"logL_gd_test = {logL_gd_test:5.2f}")
```

```
logL_gd_test = -1697.75
```

- For the sequential methods, the updates "efseq" with the inverses of an approximated hessian from minibatches. Here, "efseq1" is for $B = n$ ($m = 1$) with the Sherman-Morrison formula only while "efseq" is for $B < n$ ($m > 1$) with the Woodbury identity.

```
alpha_t=0.1
fit_efseq = f_fit_poisson(X_train, y_train, show=-1, alpha_t=alpha_t,
                          alpha=0.12, n_epoqs=n_epoqs_max,
                          size_batch=8, diff=1e-5, algo="efseq")
quali_efseq_test = \
  f_mu_mse_cor_poisson(X_test, y_test, fit_efseq,mu0[idx_test])
```

```
logL_efseq_test = quali_efseq_test["logL"]
print(f"logL_efseq_test = {logL_efseq_test:5.2f}")
```

```
logL_efseq_test = -1700.37
```

The algorithms are compared visually via their loglikelihood in the next graphic.

```
import matplotlib.pyplot as plt
plt.plot(range(fit_nr["step_end"]+2),fit_nr["logL_s"], "r-",label='nr')
plt.plot(range(fit_ef["step_end"]+2),fit_ef["logL_s"], "b-",label="ef")
plt.plot(range(fit_gd["step_end"]+2),fit_gd["logL_s"], "g-.",label='gd')
plt.plot(range(fit_efseq["step_end"]+2),fit_efseq["logL_s"],
         "k--",label="efseq")
plt.xlabel("epoch or iteration")
plt.ylabel("loglikelihood")
plt.title("Training with several algorithms")
plt.legend()
plt.show()
```

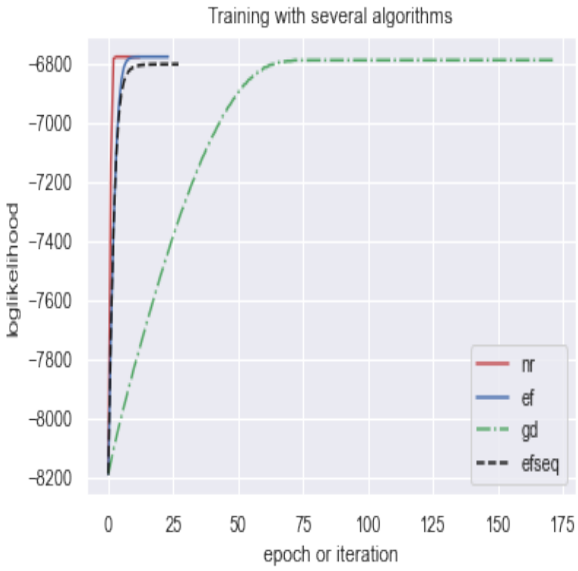

Figure 7.2: Losses per epoch for the first/second-order algorithms

For the sequential procedures with the numpy implementation above, there was required a "gradient clipping", with bounds component by component on the gradient during training: this has avoided a "divergence" when the consecutive values of the parameters do not stabilize with time. An alternative is a strict bound on the norm for all the components together which sometimes is required also in deep learning for some models. There the norm is kept constant instead of these truncated components.

Which algorithm behaves the best depends on the dataset and the settings hence, when possible, several algorithms should be tested and compared. Some improvement of the settings for the learning rate and the minibatches size is possible via "cross-validation" for instance. More advanced second-order sequential methods (for high dimensional models) exist in the literature and some are implemented on pytorch, one is tested at the next section.

Retrieving the exact counts is not in stake (because of the sampling generating the counts), thus the Poisson parameters are compared instead. Comparing the loglikelihood (or related bic and aic) is also more usual than the mse because the true parameters are often unknown. The numeral results are summarized in the following table.

```
import pandas as pd
```

```
algo_s = []
step_s = []
time_s = []

msemu_s_test = []
logL_s_test = []

for fit,quali in zip([fit_nr,fit_ef,fit_gd,fit_efseq],
                     [quali_nr_test,quali_ef_test,
                      quali_gd_test,quali_efseq_test]):
    algo_s.append( fit["algo"] )
    step_s.append( fit["step_end"] )
    msemu_s_test.append( quali["msemu"] )
    logL_s_test.append( quali["logL"] )
    time_s.append( fit["time"] )

n_train_s = [n_train,n_train,n_train,n_train]
n_test_s  = [n_test,n_test,n_test,n_test]
p_s       = [p1-1,p1-1,p1-1,p1-1]

results = [algo_s,logL_s_test, msemu_s_test,step_s,
           time_s,n_train_s,n_test_s, p_s]
results_pd = pd.DataFrame(results).transpose()
results_pd.columns = ["algo", "logL_test", "msemu_test",
                      "nb_steps", "time_train (s)", "n_train",
                      "n_test", "nb_vars"]

with pd.option_context('float_format', '{:.4f}'.format,
                       'display.expand_frame_repr', False):
    print(results_pd.to_string(index=False))#, header=False
```

```
 algo  logL_test msemu_test nb_steps time_train (s) n_train n_test nb_vars
   nr -1694.3877     0.0058       11      0.0928        4000   1000       6
   ef -1694.3853     0.0058       22      1.7282        4000   1000       6
   gd -1697.7522     0.0227      170      0.5884        4000   1000       6
efseq -1700.3708     0.0256       26      6.0062        4000   1000       6
```

Here, with $s = s_{\text{train}} \cup s_{\text{test}}$, and $n_{\text{test}} = |s_{\text{test}}|$, "msemu_test" is for the test sample:

$$\text{mse}(\hat{\mu}_{\text{test}}, \mu_{\text{test}}) = \frac{1}{n_{\text{test}}} \sum_{i \in s_{\text{test}}} (\hat{\mu}_i^{\text{test}} - \mu_i^{\text{test}})^2 .$$

With a factor of four, the batch procedures have lead to smaller mse than the minibatches procedures here despite a small difference for the loglikelihood, this may underline the superiority of such approaches for small datasets. Note that concerning these implementations with numpy, the python code with the minibatches have some issues here: the first-order procedure has not the best setting while the second-order could prefer (for some other datasets too) a full computation of the inverse of the hessian at each epoch. Testing such variants and improvements is an eventual exercice left to the reader in order to further practice these algorithms, it is adviced to separate each algorithm in a separated dedicated function before updating the python code. The next section presents a more robust and advanced approach from the "quasi-newton" methods, plus also a better setting for the gradient approach, both with the module pytorch and the gpu when available.

7.5 Fitting a Poisson regression with pytorch

The inference for the Poisson regression is implemented with pytorch, with the 5000 data vectors in order to use the numerical derivatives for deep neural networks and compare with the implementation with numpy for the linear case. First the usual first-order method is implemented for the new loss function, then a new optimizer class from pytorch with a more advanced approximation of the hessian is tested for a second-order method. Note that this part of the chapter is relevant for deep models while the part just before is for linear models, this would ask for implementing backpropagation with numpy, which is out of the scope herein and fully automatic with pytorch.

7.5.1 Example of training at first-order

Let define a more general python function for the training, with linear generalized models such as Gaussian, Bernoulli, Multinomial and Poisson. Note that other distributions are allowed by giving a loss function as input and choosing "MLP" for the architecture.

```
import deepglmlib.utils as utils
```

```
import torch.nn as nn
import torch
```

The loss function for the Poisson regression is already defined in the pytorch module, otherwise, it may be defined as:

```
# def poisson_cross_entropy(logtheta, inputs):
#         return torch.sum(- inputs * logtheta + torch.exp(logtheta),
#                          axis=0)
```

The new function presents a simplified access to the class for the neural networks, and returns a dictionary instead of separated variables at the end of the call.

```
import copy
# Only for linear and deep models glm and only sgd or lbfgs optimizer

def f_train_my_glm(dl_train, dl_test, layers, name_model,
                   nbmax_epoqs=10, debug_out=1, alpha_t=0.001,
                   device=None, momentum = 0.0, init_model = None,
                   transform_yb = None, transform_yhatb = None,
                   transform_xb = None, loss=None,
                   update_model=None, printed=0,
                   reduction="sum",loss_yy_model=None,
                   name_optimizer = "SGD", nbmax_iter_lbgs = 8,
                   K=None):   #, model=None):

    # select loss
    if loss==None:
        if (name_model== "LinearRegression" or \
            name_model== "MLP"):
                loss = torch.nn.MSELoss(reduction=reduction)

        if (name_model== "LogisticRegression" or \
            name_model== "LMLP"):
                loss = torch.nn.BCEWithLogitsLoss(reduction=reduction)

        if (name_model== "SoftmaxRegression" or \
            name_model== "MultinomialRegression" or \
            name_model== "SMLP" or \
            name_model== "MMLP"):
                loss = torch.nn.CrossEntropyLoss(reduction=reduction)

        if (name_model== "PoissonRegression" or \
            name_model== "PMLP"):
            #loss = poisson_cross_entropy
            loss = torch.nn.PoissonNLLLoss(reduction=reduction, \
                                           log_input=True, full=True, )

    if loss==None: loss = torch.nn.MSELoss(reduction=reduction)
```

```
    #check model names
    if (name_model!= "LinearRegression" and \
        name_model!= "LogisticRegression" and \
        name_model!= "SoftmaxRegression" and \
        name_model!= "MultinomialRegression" and \
        name_model!= "PoissonRegression" and \
        name_model!= "MLP" and \
        name_model!= "LMLP" and \
        name_model!= "SMLP" and \
        name_model!= "MMLP" and \
        name_model!= "PMLP"): name_model = "MLP"

    #if model is None:
    model     = utils.GNLMRegression(name_model,copy.deepcopy(layers))

    if (name_model== "SoftmaxRegression" \
            or name_model== "MultinomialRegression" \
            or name_model== "SMLP" \
            or name_model== "MMLP"):
                model.K=K ####if model.K is None:

    if name_optimizer!= "SGD" and name_optimizer!= "LBFGS": optimizer =␣
↪None

    if name_optimizer== "SGD":
        optimizer = torch.optim.SGD(model.parameters(), lr=alpha_t, \
                                    momentum=momentum)

    if name_optimizer== "LBFGS":
        optimizer = torch.optim.LBFGS(model.parameters(), lr=alpha_t, \
                                      max_iter=nbmax_iter_lbgs)

    monitor   = utils.MyMonitorTest(model,loss,dl_train,dl_test,
                                    nbmax_epoqs,debug_out,device,
                                    transform_yb = transform_yb,
                                    transform_xb = transform_xb)
    # train model
    #if device is not None: model=model.to(device)

    #model.train()
    loss_s,tmax,monistop = \
       utils.f_train_glmr(dl_train,model,optimizer,loss,monitor,
                          device=device,
                          transform_yb = transform_yb,
```

```
                            transform_yhatb = transform_yhatb,
                            transform_Xb = transform_xb,
                            update_model=update_model,
                            printed=printed,
                            loss_yy_model=loss_yy_model)

    return {"loss_train_s":loss_s, "tmax":tmax, "monistop":monistop,
            "model":model,"monitor":monitor, "loss":loss}
```

The transformations are as before, a standardization is possible for the columns of the design matrix X, and eventually an initialization of the model is required.

```
# ## STANDARDIZATION
# mean_x_train = torch.zeros((1,p_train))
# std_x_train = torch.zeros((1,p_train))
# # x_mean_train.shape
# # y_train = np.empty((0,1))
# for b, (xb, yb) in enumerate(dl_train):
#     mean_x_train += torch.mean(xb, axis=0).reshape((1,p_train)) /␣
 ↪n_train
#     std_x_train  += torch.mean(xb**2, axis=0).reshape((1,p_train)) /␣
 ↪n_train

# std_x_train = torch.sqrt(std_x_train - mean_x_train**2)

# mean_x_train, std_x_train

# def f_transform_x(xb):
#     return (xb-mean_x_train)/std_x_train
# #
# # def f_transform_x(xb):
# #     return xb

# mean_x_train = mean_x_train.to(device)
# std_x_train = std_x_train.to(device)

# f_transform_x = None
```

There is not an involved transformation for this dataset.

A main difference with before is that the loss is automatically chosen. Let define the dataloader for loading the dataset, and train the model.

```
# dico_y_train500, keys_y_train500 = f_barplot_poisson(y,True,False)
# dico_y_test500, keys_y_test500 = f_barplot_poisson(y,True,False)
```

```
from torch.utils.data import DataLoader, TensorDataset

dt_train = TensorDataset( torch.from_numpy(X_train[:,1:].astype(np.
 ↪float32)), torch.from_numpy(y_train.astype(np.float32)) )

dt_test  = TensorDataset( torch.from_numpy(X_test[:,1:].astype(np.
 ↪float32)), torch.from_numpy(y_test.astype(np.float32)) )

batch_size= 8
dl_train = DataLoader(dt_train, batch_size= batch_size,
                      shuffle=False,num_workers=1)
dl_test  = DataLoader(dt_test, batch_size= batch_size,
                      shuffle=False,num_workers=1)
n_train, p_train = dl_train.dataset.tensors[0].shape
n_test, p_test   = dl_test.dataset.tensors[0].shape

print(n_train, p_train, n_test, p_test)
```

```
4000 6 1000 6
```

Here it is implemented a gradient clipping which avoids the gradient to increase too much during the training. This seems mandatory for first-order algorithms, which seems not discussed in the literature much. The bounds for the clipping was chosen arbitrary after testing several values. Similarly, in some cases, the initialization seems to require a bounded value around 0 from an uniform distribution, otherwise the convergence was towards a mistaken solution. This may be related to some initializations for deep learning using gradient procedures. For the proposed generic implementation, it is required to declare a new function and add the function at the call while the initilization is by default for this version of pytorch.

```
def f_update_model(model,loss,optimizer,device,b=None,Xb=None,yb=None):
    alpha_t = next(iter(optimizer.param_groups))['lr'] #here constant!
    for p in iter(model.parameters()):
        p.grad[p.grad>2] = 2
        p.grad[p.grad<-2] = -2
        p.data = p.data - alpha_t * p.grad
```

The model is trained with the new function above for (deep) glm, this is more convenient because there is just need to define the structure of the network with the hidden layers and the name of the model.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

name_model = "PoissonRegression"
nbmax_epoqs = 80
debug_out   = 1
```

```
alpha_t     = 0.0001

layers = []
layers.append(nn.Linear(p_train,1, bias=True))

resus_gdth = f_train_my_glm(dl_train, dl_test, layers, name_model,
               nbmax_epoqs=nbmax_epoqs, debug_out=debug_out,
               alpha_t=alpha_t, transform_yb = utils.transform_yb,
               transform_yhatb = utils.transform_yhatb, device=device,
               update_model=f_update_model,printed=2,)
```

```
loss= 14113.87012        t= 0 / 80       ( 0.0 %)
loss= 6708.81836         t= 80 / 80      ( 100.0 %)
```

```
torch.save(resus_gdth["model"],
           "./datasets_book/poisson_regression_n5000_d7.pth")
torch.save(resus_gdth["model"].state_dict(),
           "./datasets_book/poisson_regression_w_n5000_d7.pth")
```

Let compute the statistics for measuring the quality of the final model according to the true parameters. Remember that this is not possible with real data except for the loglikelihood. Here the dataset is artificial and the additional indicators are informative for checking the implementation and further comparisons.

```
def fun_model2vector(model):
    beta = [p.detach().numpy().ravel()
               for p in model.parameters()]
    beta = [beta[(i + 1) % len(beta)]
               for i, x in enumerate(beta)]
    beta = np.concatenate( beta, axis=0 )
    return beta

beta_gdth = fun_model2vector(resus_gdth["model"].to(torch.device("cpu"))).
 ↪ravel()
beta_gdth = beta_gdth.reshape(len(beta_gdth),1)

fit_gdth         = {"beta":beta_gdth,"algo":"gdth"}
quali_gdth_test  = f_mu_mse_cor_poisson(X_test,y_test,
                                        fit_gdth,mu0[idx_test])
quali_gdth_train = f_mu_mse_cor_poisson(X_train,y_train,
                                        fit_gdth,mu0[idx_train])

logL_gdth_train = quali_gdth_train["logL"]
logL_gdth_test = quali_gdth_test["logL"]
```

```
print(f"logL_gdth_train= {logL_gdth_train:5.2f}")
print(f"logL_gdth_test = {logL_gdth_test:5.2f}")
```

```
logL_gdth_train= -6781.55
logL_gdth_test = -1696.55
```

The likelihoods and indicators are obtained above, but one may keep in mind that proceeding in "f_mu_mse_cor_poisson" with chunks would be better for larger datasets, this can be seen as an exercice left to the reader (otherwise, there is the risk of computer system freezing when python uses a large virtual memory).

In comparison, the module statsmodels leads to:

```
import statsmodels.api as stm
ols= stm.Poisson(y_train, X_train)
fit_ols_train = ols.fit()
beta_stm = fit_ols_train.params
beta_stm = beta_stm.reshape((len(beta_stm),1))

fit_stm        = {"beta":beta_stm,"algo":"stm"}
quali_stm_test = f_mu_mse_cor_poisson(X_test,y_test,
                                      fit_stm,mu0[idx_test])
quali_stm_train = f_mu_mse_cor_poisson(X_train,y_train,
                                       fit_stm,mu0[idx_train])

logL_stm_train = quali_stm_train["logL"]
logL_stm_test = quali_stm_test["logL"]
print(f"logL_stm_train = {logL_stm_train:5.2f}")
print(f"logL_stm_test  = {logL_stm_test:5.2f}")
```

```
Optimization terminated successfully.
         Current function value: 1.694970
         Iterations 4
logL_stm_train = -6779.88
logL_stm_test  = -1694.39
```

This is not very far to the solution above with pytorch. Actually some installations of pytorch may perform slower than numpy (which could be even faster with some gpu directive like from "numba"). But, the automatic derivatives are very appealing for (any) more complex models. The parameters are not so much tuned for the sequential methods with minibatches and the updates are not stochastic, which explain the lesser performance here.

Next, the second-order is implemented with pytorch for neural networks.

7.5.2 Example of L-BFGS for training at second-order

In this section, we are interested on quasi-newton methods. They aim at step k at approximating the usual update for the parameters, $\theta_{(k+1)} = \theta_{(k)} + \delta_k$. Here δ_k is unknown and solves for $B_k\delta_k + g_k = 0$ where $B_k \approx H_k$. The approximation B_k of the exact hessian H_k needs generally to be positive definite for insuring an inverse with good properties.

In particular the method called limited memory BFGS is an improved algorithm from the original research from the Broyden–Fletcher–Goldbarb–Shanno (BFGS) formula. This formula proposes an approximation of the hessian and its inverse from two consecutive values of the gradient, say g_k and g_{k+1} plus the two last consecutive data vectors, say x_k and x_{k+1}. The differences are involved, $s_k = x_{k+1} - x_k$ and $y_k = g_{k+1} - g_k$. By equating $B_{k+1}s_k = y_k$, and after advanced algebra, this leads to the update of the approximating hessian:

$$B_{k+1} = B_k - \frac{1}{s_k^T B_k s_k} B_k s_k s_k^T B_k + \rho_k y_k y_k^T .$$

Here, ρ_k denotes the factor $\frac{1}{y_k^T s_k}$. The inverse is in closed-form, via an algorithm with a truncated sum for the limited memory version.

The implementation is tricky and out of the scope, thus, the existing optimizer from pytorch is used. It is named "optim.LBFGS" and asks currently for a different implementation of the backpropagation (in the loop with the minibatches), exceptionnally, thus this must be rewritten in python by following the documentation. There exists many variants, but they are not implemented with pytorch, the available one is stable and performs well. Note also that the sequential process is managed by the algorithm, hence in this case, it is better to include all the dataset once in the dataloader. In some cases minibatches may induce a convergence but this is not sure at all and not wise to avoid the batch version (except for the recent variants called stochastics).

For this optimizer, it is added a test in order to check which optimizer is involved in the function "f_train_glmr()" from the module deepglmlib and file "utils.py". Hence, this is included in the function for training already in the companion file.

The call to the function is thus as previously by just changing the name of the optimizer and the dataloaders. For instance, some eventual regularization is added as follows.

```
lambda_l1 = 0.01
def loss_yy_model(lossb,model):
    lossb_b_rg = lossb
    lossb_b_l1 = (torch.abs(list(model.parameters())[0])+0.000001).sum()
    loss_b     = lossb_b_rg + lambda_l1 * lossb_b_l1
    return loss_b
```

```
dl_train = DataLoader(dt_train, batch_size= len(dt_train),
                      shuffle=False,num_workers=1)
dl_test  = DataLoader(dt_test, batch_size= len(dl_test),
                      shuffle=False,num_workers=1)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

name_model = "PoissonRegression"
nbmax_epoqs = 1
debug_out   = 1
alpha_t     = 0.001

layers_regress = []
layers_regress.append(nn.Linear(p_train,1, bias=True))

resus_lbfgsth = f_train_my_glm(dl_train, dl_test,
               layers_regress, name_model,
               nbmax_epoqs=nbmax_epoqs, debug_out=debug_out,
               alpha_t=alpha_t, transform_yb = utils.transform_yb,
               transform_yhatb = utils.transform_yhatb, device=device,
               update_model=f_update_model,printed=1,
               name_optimizer="LBFGS", nbmax_iter_lbgs=30,
               loss_yy_model=loss_yy_model)
```

```
loss= 6705.83057         t= 0 / 1        ( 0.0 %)
```

For some run, the algorithm may not converge, there are some hyperparameters to set. In particular "nbmax_iter_lbgs" for the "maximal number of iterations per optimization step", see the documentation of the optimizer for this one and the other ones.

The resulting metrics are as follows.

```
beta_lbfgsth = fun_model2vector(
                resus_lbfgsth["model"].to(torch.device("cpu"))).ravel()
beta_lbfgsth = beta_lbfgsth.reshape(len(beta_lbfgsth),1)

fit_lbfgsth         = {"beta":beta_lbfgsth,"algo":"lbfgsth"}
quali_lbfgsth_test  = f_mu_mse_cor_poisson(X_test,y_test,
                        fit_lbfgsth,mu0[idx_test])
quali_lbfgsth_train = f_mu_mse_cor_poisson(X_train,y_train,
                        fit_lbfgsth,mu0[idx_train])

logL_lbfgsth_train = quali_lbfgsth_train["logL"]
logL_lbfgsth_test = quali_lbfgsth_test["logL"]
print(f"logL_lbfgsth_train = {logL_lbfgsth_train:5.2f}")
print(f"logL_lbfgsth_test  = {logL_lbfgsth_test:5.2f}")
```

```
logL_lbfgsth_train = -6779.88
logL_lbfgsth_test  = -1694.39
```

The results are summarized in the following table for comparison purpose. The correlation is not

given as it is less informative than the mse. The second-order method is clearly performing well here as expected for a small dataset. The table is thus:

```
import pandas as pd

method_s = ["gd-mb-torch",
            "lbfgs-mb-torch",
            "stm (module)"]

logLik_s = [quali_gdth_test["logL"],
            quali_lbfgsth_test["logL"],
            quali_stm_test["logL"]]

mse_mu_s = [quali_gdth_test["msemu"],
            quali_lbfgsth_test["msemu"],
            quali_stm_test["msemu"]]

nbstep_s = [resus_gdth["tmax"],
            resus_lbfgsth["tmax"], 4]

n_train_s = [n_train,n_train,n_train]
n_test_s  = [n_test,n_test,n_test]
p_s       = [p_train,p_train,p_train]

results = [method_s, logLik_s, mse_mu_s, nbstep_s,
           n_train_s, n_test_s, p_s]

results_pd = pd.DataFrame(results).transpose()
results_pd.columns = ["algo", "logL_te",
                      "mse(mu_hat,mu)_te", "nb_steps_tr",
                      "n_train", "n_test", "nb_vars"]

with pd.option_context('float_format', '{:.4f}'.format,
                       'display.expand_frame_repr', False):
    print(results_pd.to_string(index=False))#, header=False
```

```
          algo    logL_te mse(mu_hat,mu)_te nb_steps_tr n_train n_test nb_vars
   gd-mb-torch -1696.5462           0.0076          80    4000   1000       6
lbfgs-mb-torch -1694.3859           0.0058           1    4000   1000       6
  stm (module) -1694.3877           0.0058           4    4000   1000       6
```

The implementation with pytorch leads to a same loglikelihood (and even slighly better) than the implementation from statsmodels, while the gradient approach is almost identical with a small difference. For the second-order training with pytorch, the hessian is approximated recently for pytorch with external modules with the diagonal hessian for instance in the case of very large

models: fully diagonal or by block per layer. External modules such as "backpack" are found in repositories, this allows the access to other approximation but sometimes available only for some architectures of neural networks. Second-order are more prone to minimum local, hence for non convex optimization they should not be considered without further checking. Some mixed method by combining with another optimizer at the beginning or at the end of the training looks like the best way to do here.

7.6 Exercices

1. (stat+numpy) Rewrite the function for the four second-order procedures into separated functions for each method. Test the algorithms for the linear regression and the logistic regression by changing the expression of the gradient vector and the hessian matrix, compare the output with statsmodels for the datasets from the previous chapters herein.
2. (stat+pytorch) Idem than just before, with pytorch instead of numpy. Compare with the result from the algorithm l-bfgs.
3. (numpy) Test a cross-validation of the newton-raphson algorithm and the approximated version with numpy and find out if five or ten folds are relevant for approaching the expected error.
4. (numpy) Test the deep Poisson regression with an artificial dataset with only one independent variable and a nonlinear function in μ. Compute the mean squared error for diverses architectures (varying number of hidden layers and number of neurons). See "https://link.springer.com/article/10.1007/s00521-009-0277-8" for instance for examples of nonlinear regression.
5. (stat+pytorch) Implement a zero inflated model for deep learning with count data. The dataloader with the id from each row may be required here if shuffling is true, in order to keep the ordering of the rows for the vector of mixing probabilities.
6. (stat+pytorch) In some not rare cases for deep neural networks, the hessian is expected to be ill-defined, with the inverse which leads to negative variances on the diagonal. Propose solutions in order to solve this issue. For instance test a regularization or test a Gan for tabular data in order to augment the number of observations from the dataset and test a variable selection method in order to keep only the useful variables in the input layer. Is there a difference between variable selection and weights removal from the neural network in order to improve the generalization. An example of network is from the artificial data "diskandnoise", see previous chapters.
7. (pytorch) Check the convergence by ploting locally the loss function as curves for each variable, and also deduce a numerical value for the gradient.
8. (pytorch) After a read at the documentation, test the module[1] "backpack" with more advanced second-order derivatives. Compare with l-bfgs for a small dataset. Compare with a first-order training for a large dataset.

[1] "https://pypi.org/project/backpack-for-pytorch/"

Chapter 8

Autoencoder compared to ipca and t-sne

In the previous chapters it has been studied the neural networks for regression and classification when the dataset is a data table: nonlinearities are added to the family of the generalized linear models. Thus, their definition, their training and their implementation have been presented for several datasets with several modules available with the computer language "python". These approaches are called "supervised" because they aim at predicting a variable y_i from a variable x_i. When only the variables x_i are available, the set of rows or the set of columns from the data table can be described by methods called "unsupervised". Typically, two main families of unsupervised methods exist: a) "clustering" (out of the scope) which proposes a descriptive "partitionning" without examples, hence according to a criterion such as minimum intra variance, and b) "reduction" (see this chapter) which proposes a new set of rows or columns according to a criterion such as maximum variance.

- The methods when they provide new low dimensional vectors $\hat{x}_i$ from the high dimensional ones x_i needs to preserve some information from the former space: the whole variance between data points, the distances between data points, their nearest neighbors or even the structure with the furthest ones. To be clear, the method named "pca" preserves mainly the variance while the one called "t-sne" preserves neighboors, thus they do not have exactly the same utility. Note that t-sne is the culminating method which came after several ones (Isomap, LLE, SNE, Laplacien Eigenmap and a few other ones) with lower performances.
- Several criteria from unsupervised methods are summarized in the table below.

Name	Type	Criterion	
K-means	Clustering	$(\{\hat{z}_{ik}\},\{\hat{\mu}_k\})$	$=\text{argmin}_z \sum_{i=1}^{n}\sum_{k=1}^{g} z_{ik}\|x_i-\mu_k\|^2$
GMM	Clustering	$(\{\hat{z}_{ik}\},\{\hat{\mu}_k\},\sigma)$	$=\text{argmin}_z \sum_{i,k} z_{ik}\log\mathcal{N}(x_i-\mu_k,\sigma^2)$
PCA	Reduction	$\hat{x}=x\hat{w}^T$, with $\hat{w}$	$=\text{argmax}_w \sum_i w^T\frac{1}{n}x_i^T x_i w$
ISOMAP	Reduction	$\hat{x}=\{\hat{x}_1,\cdots,\hat{x}_n\}$	$=\text{argmin}_{\hat{x}}\sqrt{\sum_{i,j} v_{i,j}\left[\delta_{i,j}-d_{i,j}\right]^2}$
t-SNE	Reduction	$\hat{x}=\{\hat{x}_1,\cdots,\hat{x}_n\}$	$=\text{argmin}_{\hat{x}}\sum_{i,j} p_{ij}\log\frac{p_{ij}}{q_{ij}}$

Here, a simplified version for principal component analysis (pca with the data columns supposed standardized and with only one latent component only, with also the constraint

$\hat{w}^T\hat{w} = 1$), and for gaussian mixtures models (gmm with here the non fuzzy version) are presented, see the literature and the documentation of sklearn for more information. Note also that $d_{i,j}$ is the Euclidean distance between x_i and x_j while $\delta_{i,j}$ is the Euclidean distance between two lower dimensional vectors $\hat{x}_i$ and $\hat{x}_j$, while $v_{i,j}$ is the graph of the nearest neighboors. For SNE and t-SNE, the probabilities p_{ij} and q_{ij} are respectively the transformations via an exponential function or t-distribution density function of $d_{i,j}$ and $\delta_{i,j}$ plus a normalization: this allows to improve Isomap by increasing the frontiers between the clusters from the data space or the projection space.

- With neural networks, it becomes possible to model the nonlinearities with hidden layers instead of the vicinity graph. Thus several architectures of neural networks has been invented for explaining the contents of a data table. For reducing the number of variables, among the existing methods of reduction, autoencoders are able to extract a linear or nonlinear subspace from the set of vectors x_i. A reduction of the dimension may be concerned by one of the following purposes: exploration, visualization, feature extraction (new variables before regression or classification), regularization and noise reduction, less computer memory and more speed, outlier detection. More generally this solves for the curse of dimensionality by retrieving the lower subspace where the data lies really instead of an high dimensional space with many useless variables. This comes from the numerous variables which share the same information and are correlated or can be explained by other variables. Removing the noisy and useless information allows to keep a summary which is informative for the human and the machine: for interpretation or computing.

- In an autoencoder, the reduced vectors denoted $\hat{x}_i$ come from an hidden layer of a neural network. It is defined very similarly than in the previous chapters, except that the output is not anymore a variable y_i but instead just x_i, while the input remains again x_i. Autoencoders are quite old neural networks which are modernized currently with the rise of computer modules such as pytorch, and they are able to leads to a better reduction than an usual linear method such as pca and even sometimes a better reduction than the current state of art methods such as t-sne. Autoencoder (ae) in the simplest linear case behaves like pca, but when nonlinearities are added and when combined with other neural architectures, ae is expected to perform better than other methods. But the training of an autoencoder is currently often difficult: because of the intensive search for the optimal hyperparameters and the unsupervised side. Thus, it is more often found autoencoders specialized in a family of datasets, for instance, medical images of a body part, clinical data for a given disease or biological data related to dna: each architecture of ae needs a careful training in order to be able to reduce the dimensionality of the available data and future nearly identical data.

The chapter is divided into three parts, first pca, ae and t-sne with an artificial dataset and the $60k$ digits dataset named "mnist" which is currently the main usual dataset for model comparisons.

8.1 Three pca methods and t-sne for dimension reduction

The method of pca is widely used and studied until today because in many situations, it is able to improve many models as a pre-processing for reducing the column space of the available data

matrix which aggregates the data vectors x_i as rows. It may be even included within a model in order to regularize an unknown latent matrix.

A data matrix in numerical algebra comes with an Euclidean space from the space of the columns, for the features or variables. The matrix may be not of full rank with some correlation between colums which can make the life difficult for some algorithms such as regression or even classification. These statistical duplicates induce a large number of columns, a numerical burden during the matricial computations or even an erroneous solution from the optimization. Hence, pca is very appealing because it aims at defining new variables which are a linear combination of the available variables and less numerous, while optimizing a cost function, the variance among these latent variables.

8.1.1 The pca method in brief

First, let recall the method for computing the new variables from principal component analysis and its projection, just before an analytical justification.

Algorithm

For correlation pca, one proceeds as follows,

a) The data matrix $X = (x_{ij})_{ij} \in \mathbb{R}^{n\times p}$ has its first columns standardized and centered, with zero means and unit variances. This results into the new matrix $Y = (y_{ij})_{ij} \in \mathbb{R}^{n\times p}$ with cells:

$$y_{ij} = \frac{x_{ij} - \bar{x}_j}{\sigma_j}\,.$$

b) The correlation matrix is found with this matrix as,

$$Cor_Y = \frac{1}{n-1} Y^T Y\,.$$

Note that some authors prefer the version with n instead of $n-1$.

c) Two different and equivalent approaches allow to find the solution for pca:
   - A singular value decomposition (svd, a method from linear algebra for any rectangular matrice) of this matrix Y of rank r, with:

$$\frac{1}{\sqrt{n-1}} Y = U_r \Delta_r V_r^T\,.$$

   - An eigen decomposition (eg, a method of linear algebra for any diagonalisable symmetric matrices), with:

$$\begin{array}{l} \frac{1}{n-1} Y^T Y = V_r \Delta_r^2 V_r^T \\ \frac{1}{n-1} Y Y^T = U_r \Delta_r^2 U_r^T\,. \end{array}$$

The two methods for a matricial decomposition lead to the same result, but numerically they have not the same numerical complexity or the same demand for the computer memory. If

the number of rows or columns is reduced, the square matrix is a good choice, but for a sparse matrix the rectangular matrix keeps the sparsity. Here in both cases, it is denoted Δ_r a diagonal matrix for the singular values λ_ℓ where $1 \leq \ell \leq r$, and the two orthogonal matrices U_r and V_r where:

$$\Delta = \text{Diag}\,(\lambda_1, \lambda_2, \cdots, \lambda_r)\,.$$
$$U_r^T U_r = \mathbb{I}_r \text{ and } V_r^T V_r = \mathbb{I}_r\,.$$

Note that the squares λ_ℓ^2 are called the eigenvalues, they come from the eigen decomposition of a symmetric matrix. For instance the covariance and correlation matrices are positive definite and symmetric.

d) This leads to the projection of Y for k (with $k \leq r$ and often $k \ll r$) principal components:

$$\begin{aligned} C_Y^{\text{pca}} &= YV_k \\ &= \sqrt{n-1}U_k\Delta_k^2\,. \end{aligned}$$

Here, Δ_k keeps only the singular values corresponding to the k largest ones in Δ_r while U_k and V_k are similarly truncated versions of U_r and V_r to the corresponding eigenvectors. This solution is also the principal components for X with: $C_X^{\text{pca}} = C_Y^{\text{pca}}$. This approach allows also an approximation of X instead of a projection, see next.

Some authors prefer to proceed directly with the covariance matrix, and other ones would add some additional scaling if an underlying metric is involved, this is discussed in the literature of exploratory data analysis. This explains why different implementations lead to different results because these options are not always fully documented. Some indicators from pca are not discussed here. Next, it is explained further the algebra for pca.

Singular value decomposition (svd)

The definition is not always clear in the literature from one author to another one because the decomposition is written in three different ways. This paragraph allows a better understanding of this important part of the algorithm. The svd has a matricial notation for the following result for a any real matrix of rank k,

$$Y = \sum_{\ell=1}^{r} \lambda_\ell u_\ell v_\ell^T\,.$$

Here, $u_\ell \in \mathbb{R}^{n\times 1}$ and $v_\ell \in \mathbb{R}^{p\times 1}$ are basis vectors for respectively $\mathbb{R}^n$ and $\mathbb{R}^p$, hence,

$$u_\ell^T u_\ell = 1 \text{ and } u_{\ell_1}^T u_{\ell_2} = 0 \text{ if } \ell_1 \neq \ell_2$$
$$v_\ell^T v_\ell = 1 \text{ and } v_{\ell_1}^T v_{\ell_2} = 0 \text{ if } \ell_1 \neq \ell_2\,.$$

The quantities λ_ℓ are strictly positive (non equal to zero) and called the singular values. By keeping, only the first k largest singular values in the sum above, one get the best rank k approximation.

The three corresponding matricial notations are the following ones:

- Let denote U_r and V_r aggregating the vectors above, with $U_r = [u_1|u_2|\cdots|u_r] \in \mathbb{R}^{n\times r}$ and $V_r = [v_1|v_2|\cdots|v_r] \in \mathbb{R}^{p\times r}$ with $U_r^T U_r = \mathbb{I}_r$ and $V_r^T V_r = \mathbb{I}_r$ while $\Sigma_r = \text{Diag}(\lambda_1, \cdots, \lambda_r) \in$

$\mathbb{R}^{r\times r}$, one gets the matrial version of the svd in brief notation:

$$Y = U_r \Sigma_r V_r^T \, .$$

This first clear and concise notation is expanded into two notation because it is possible to add basis vectors to U_r and V_r when zeros are added to the matrix Σ. Remember that for this notation, the matrix is diagonal with not null diagonal elements, and thus, on the nondiagonal part of the matrix there are only null cells (equal to zeros).

- The second notation is a direct extension of the one above but the less natural, by computing $q = min(n, p)$, and by partially completing the basis for R^n and R^p, aggregated in U_*^{q-r} for U and V_*^{q-r} for V, thus $U_q = [U_r|U_*^{q-r}] \in \mathbb{R}^{n\times q}$ and $V_q = [V_r|V_*^{q-r}] \in \mathbb{R}^{p\times q}$ with $U_q^T U_q = \mathbb{I}_q$ and $V_q^T V_q = \mathbb{I}_q$, while Σ_q is the diagonal matrix Σ_r completed with $q-r$ zeros on the diagonal, $\Sigma_q \in \mathbb{R}^{q\times q}$, thus $U_q \Sigma_q V_q^T$ is read as:

$$Y = [U_r|U_*^{q-r}] \begin{bmatrix} \Sigma_r & 0_r^{q-r} \\ 0_{q-r}^r & 0_{q-r}^{q-r} \end{bmatrix} \begin{bmatrix} V_r^T \\ (V_*^{q-r})^T \end{bmatrix} .$$

- The third notation is not the more natural because Σ is not always diagonal. There are two complete basis, by fully completing the basis for R^n and R^p, aggregated in U_*^{n-r} for U and V_*^{p-r} for V, thus $U_n = [U_r|U_*^{n-r}] \in \mathbb{R}^{n\times n}$ and $V_p = [V_r|V_*^{p-r}] \in \mathbb{R}^{p\times p}$ with $U_n^T U_n = \mathbb{I}_n$ and $V_p^T V_p = \mathbb{I}_p$, while Σ_n^p is the diagonal matrix Σ_r completed with zeros, $\Sigma_n^p \in \mathbb{R}^{n\times p}$, thus $U_n \Sigma_n^p V_p^T$ is read as:

$$Y = [U_r|U_*^{n-r}] \begin{bmatrix} \Sigma_r & 0_r^{p-r} \\ 0_{n-r}^r & 0_{n-r}^{p-r} \end{bmatrix} \begin{bmatrix} V_r^T \\ (V_*^{p-r})^T \end{bmatrix} .$$

In the general case, $n \neq p$ such that Σ_n^p is not even square anymore.

Note that the algorithms for finding the eigenvalues and eigen vectors do not order the values of λ_ℓ by default, hence, one must reorder the singular values and eigen vectors because only the largest ones are kept for pca or low rank approximation. Ideally one looks for an implemention able to find only the largest or the smallest eigen components, for faster computation.

From the criterion to the algorithm

Let find an approximation $\hat{y}_i$ of y_i with a matrix $V_k \in \mathbb{R}^{p\times k}$ orthogonal (hence $V_k^T V_k = \mathbb{I}_k$) and

vectors c_i,

$$\hat{y}_i = \begin{pmatrix} \hat{y}_{i1} \\ \vdots \\ \hat{y}_{ip} \end{pmatrix} = \sum_{\ell=1}^{k} c_{i\ell} v_\ell = [v_1|v_2|\cdots|v_k] \begin{pmatrix} c_{i1} \\ \vdots \\ c_{ik} \end{pmatrix} = V_k c_i$$

$$\begin{aligned} (\hat{y}_1, \hat{y}_2, \cdots, \hat{y}_n) &= \text{argmin } S_n \\ &= \text{argmin} \sum_i \|y_i - \hat{y}_i\|^2 \\ &= \text{argmin} \sum_i \|y_i - V_k c_i\|^2 \end{aligned}$$

$$\begin{aligned} \frac{\partial}{\partial c_i} S_n &= \frac{\partial}{\partial c_i} \left[\sum_i \|y_i - V_k c_i\|^2 \right] \\ &= \frac{\partial}{\partial c_i} \left[(y_i - V_k c_i)^T (y_i - V_k c_i) \right] \\ &= \frac{\partial}{\partial c_i} \left[y_i^T y_i - 2 y_i^T V_k c_i + c_i^T V_k^T V_k c_i \right] \\ &= \left\{ -2 V_k^T y_i + 2 V_k^T V_k c_i \right\} \\ &= 0 \end{aligned}$$

Thus, both approximations c_i and $\hat{y}_i$ are linear projections for the data vector into two different spaces:

- The low dimensional one reduces the size of the vector or length from p to k,

$$c_i = V_k^T y_i \,.$$

- The low rank one reduces the size of the linear basis with only k orthogonal vectors,

$$\hat{y}_i = V_k V_k^T y_i \,.$$

When this solution c_i is inserted in the expression of the inertia, it is obtained:

$$S_n = Tr(Y - Y V_k V_k^T)(Y - Y V_k V_k^T)^T \,.$$

Hence,

$$\begin{aligned} & \text{argmin}_{V_k} \, S_n \\ = \; & \text{argmin}_{V_k} \, Tr\left\{ (Y - Y V_k V_k^T)(Y - Y V_k V_k^T)^T \right\} \\ = \; & \text{argmin}_{V_k} \, Tr\left\{ (Y - Y V_k V_k^T)(Y^T - V_k V_k^T Y^T) \right\} \\ = \; & \text{argmin}_{V_k} \, Tr\left\{ - Y V_k V_k^T Y^T + Y V_k V_k^T Y^T - Y V_k V_k^T V_k V_k^T Y^T \right\} + Tr(Y Y^T) \\ = \; & \text{argmin}_{V_k} \, - Tr(Y V_k V_k^T V_k V_k^T Y^T) \\ = \; & \text{argmax}_{V_k} \, Tr(V_k^T Y^T Y V_k) \,. \end{aligned}$$

Hence it is recognized for the columns of V_k the first eigenvectors of $Y^T Y$ by definition of an eigen decomposition of a symmetric matrix. This also induced that from these eigen vectors, one gets the low rank approximation of y_i with just $\hat{y}_i = V_k c_i = V_k V_k^T y_i$ which is sometimes called denoised. It is retrieved the approximation for x_i by inverting the standardization of the columns (mean and standard deviation), which concludes the method here. See a textbook on principal component analysis for the additional results for the method.

Note that "ipca" and "kpca" are variants, with ipca for "incremental pca" and kpca for "kernel pca". They come with alternative algorithms out of the scope here. The result from ipca should lead to nearly the same reduced space than pca because its aims at solving the same problem with chunks for scalability. As kpca is a nonlinear version of pca, it is able to improve the resulting reduction. Next, these three algorithms are compared to an autoencoder and t-sne with several indicators of quality of the visualization.

Illustration

The method pca is tested with a small dataset for better understanding. A toy dataset is generated, this is 80 points from a circle in a 3d space.

[1]:
```
from IPython.display import display, Latex
import numpy as np

def f_pca(x, nb_k=2):
    print("Mean computation")
    means_vct = np.mean(x,axis=0).reshape((3,1)).T
    print("Covariance computation")
    Cov_mat = np.cov(x.T)
    print("Eigen decomposition")
    Lambdas_vect, V_mat = np.linalg.eig(Cov_mat)
    print("Eigenvalues desc. sorting")
    ids = Lambdas_vect.argsort()[::-1]
    Lambdas_vect = Lambdas_vect[ids]
    V_mat = V_mat[:,ids]
    print("Principal components") # projection coordinates
    x_pca_numpy = (x - means_vct).dot(V_mat[:,0:nb_k])
    x_pca_numpy.shape
    return Lambdas_vect, V_mat, means_vct, Cov_mat, x_pca_numpy
```

It is compared the pca projections from sklearn and numpy.

[2]:
```
import math
x = []
ts = np.linspace(0, 2 * math.pi, 80)
for t in ts:
    x.append((5 * math.cos(t), 5 * math.sin(t)))
x    = np.asarray(x)[:,:]
P    = np.random.randn(6).reshape((2,3))
x    = x @ P
n, p = x.shape
```

[3]:
```
import deepglmlib.utils as utils
utils.f_plot_scatter3d(x,np.repeat(0,n),
```

```
    title="cicle with linear transformation",s=5)
```

[4]:
```
Lambdas_vect, V_mat, means_vct, Cov_mat, x_pca_numpy = f_pca(x)
```

```
Mean computation
Covariance computation
Eigen decomposition
Eigenvalues desc. sorting
Principal components
```

[5]:
```
print("Lambas=", Lambdas_vect)
print()
display(Latex(rf'Check the orthogonality and the norm: $V V^T = I_3$ ?'))
display(Latex(rf'$V V^T = $'))
print(V_mat @ V_mat.T)
print()
print("Check the diagonalization of the covariance matrix")
display(Latex(
    rf'For $k \in {1,2,3}$, $\Sigma v_k = \lambda_k v_k$ ?'))
for k in range(3):
    display(Latex(
       rf'$\Sigma v_{k+1}- \lambda_{k+1} v_{k+1} = $'))
    eigval_k = Lambdas_vect[k]
    eigvct_k = V_mat[:,k].reshape(1,3).T
    print(Cov_mat.dot(eigvct_k) - eigval_k*eigvct_k)
```

```
Lambas= [1.17714231e+01 4.11140711e+00 8.88178420e-16]
```

Check the orthogonality and the norm: $VV^T = I_3$?

$VV^T =$

```
[[ 1.00000000e+00  1.58431050e-17 -3.54272283e-17]
 [ 1.58431050e-17  1.00000000e+00 -1.09750710e-16]
 [-3.54272283e-17 -1.09750710e-16  1.00000000e+00]]
```

```
Check the diagonalization of the covariance matrix
```

For $k \in (1,2,3)$, $\Sigma v_k = \lambda_k v_k$?

$\Sigma v_1 - \lambda_1 v_1 =$

```
[[-4.44089210e-16]
 [-1.77635684e-15]
 [ 8.88178420e-16]]
```

$\Sigma v_2 - \lambda_2 v_2 =$

```
[[ 4.4408921e-16]
 [-8.8817842e-16]
 [ 4.4408921e-16]]
```

$\Sigma v_3 - \lambda_3 v_3 =$

```
[[2.10933408e-16]
 [8.49861456e-16]
 [3.38694653e-17]]
```

The pca from sklearn and numpy are compared.

[6]:
```
from sklearn.decomposition import PCA
pca_sklearn   = PCA(n_components=2)
x_pca_sklearn = pca_sklearn.fit_transform(x)
```

[7]:
```
x_pca_sklearn.shape, x_pca_numpy.shape
```

[7]:
```
((80, 2), (80, 2))
```

[8]:
```
print(np.mean(x_pca_numpy / x_pca_sklearn,axis=0),
      np.var(x_pca_numpy / x_pca_sklearn,axis=0))
```

```
[-1. -1.] [2.00081010e-30 7.16091876e-28]
```

The eigen vectors remain the same for the two implementations but their sign may change. This comes from the eigen decomposition which is not unique. The ellipsoidal shape of the circle is exactly retrieved. Because the points belong to a plane, the third pca component is zero.

[9]:
```
print(np.mean((x - means_vct).dot(V_mat[:,2])),
      np.var((x - means_vct).dot(V_mat[:,2])))
```

```
6.1758866250205545e-18 3.99994008434763e-31
```

[ ]:
```

```

Below, other expressions from the svd are also checked with three components.

[10]:
```
Lambdas_vect, V_mat, means_vct, Cov_mat, x_pca_numpy = f_pca(x,nb_k=3)
```

```
Mean computation
Covariance computation
Eigen decomposition
Eigenvalues desc. sorting
Principal components
```

[11]:
```
#Check the diagonalization of the covariance

V_mat.T @ Cov_mat @ V_mat
```

[11]:
```
array([[ 1.17714231e+01,  1.44920387e-16,  7.63455993e-16],
       [-3.97555307e-16,  4.11140711e+00,  1.86336172e-16],
       [ 6.54309454e-16,  3.75542305e-16,  5.32997603e-16]])
```

[12]:
```
#Check the inertia from pca with the sum of the eigenvalues

np.trace( V_mat.T @ Cov_mat @ V_mat ) - np.sum(Lambdas_vect)
```

[12]:
```
-3.552713678800501e-15
```

[13]:
```
#SVD of Y/sqrt(n-1) for alternative expressions

u,s,vh = np.linalg.svd((x-means_vct)/np.sqrt(n-1))
```

[14]:
```
u.shape, s.shape, vh.shape
```

[14]:
```
((80, 80), (3,), (3, 3))
```

[15]:
```
#Chek equality of the right eigenvectors from svd
# with the eigenvector from the eigen decomposition
# (and eventual flip for the signs)

V_mat / vh.T
```

[15]:
```
array([[ 1.,  1., -1.],
       [ 1.,  1., -1.],
       [ 1.,  1., -1.]])
```

[16]:
```
#Chek relation eigenvalues <-> singular values
```

[17]:
```
print(np.sum(np.abs(s**2 - Lambdas_vect)))
```

```
9.769962616701378e-15
```

[18]:
```
#Check covariance expression with matrix Y

Y = x-means_vct

print( ( 1/(n-1) * Y.T @ Y ) / Cov_mat )
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

[19]:
```
#Chek expressions with the trace from svd

print( np.trace( vh.T @  ( 1/(n-1) * Y.T @ Y ) @ vh ),
       np.trace( u.T @  ( 1/(n-1) * Y @ Y.T ) @ u ) )
```

```
15.882830218784203 15.882830218784193
```

[20]:
```
#Check alternative expression for pca with right eigenvectors from svd
# Third coordinates are not informative because from rounded zeros

v = vh.T
print( np.mean(x_pca_numpy  / (Y @ v) , axis=0),
  "\n", np.var(x_pca_numpy  - (Y @ v) , axis=0) )
```

```
[ 1.          1.         -1.88684972]
 [9.10441310e-31 4.49914722e-30 6.19437172e-31]
```

[21]:
```
#Check alternative expression for pca with left eigenvectors from svd
# Third coordinates are not informative because from rounded zeros

print(np.mean(((u[:,0:3] @ np.diag(v=s)))*np.sqrt(n-1) / x_pca_numpy ,
      axis=0),"\n",
      np.var(((u[:,0:3] @ np.diag(v=s)))*np.sqrt(n-1) / x_pca_numpy ,
      axis=0))
```

```
[ 1.          1.         -0.12721302]
 [1.82462449e-28 1.39194799e-27 5.65938553e+00]
```

8.1.2 Implementations of pca with sklearn

We consider an illustration with a dataset from Gaussian classes in 10 dimensions. The data can be considered generated from a mixture model, a probabilistic model for clustering which supposes g classes or clusters with centers μ_k, here mostly well separated with clear frontiers between them. Each data vectors x_i belongs to one of the g groups which can be seen as spheres or ellipsoids in the space of the columns. For a two-dimensional reduced space, this induces that the data from the group k should appear visually in a circle or in an ellipse around the projection of μ_k for this artificial example. This is like looking the data by displaying only two components of the vectors x_i but instead with new variables or components which are able to better show the data.

```
def towdir(s):
    return (str('./datasets_book/'+s))
```

```
import deepglmlib.utils as utils
import numpy as np
```

```
import importlib
importlib.reload(utils)
```

```
<module 'deepglmlib.utils' from
'/home/rodolphe/Documents/ARTICLEs/BOOK/deepglmlib/utils.py'>
```

Example of a dataset with 9 clusters and a 3d view

```
import numpy as np
xy = np.load(towdir('x_y_10d_ae.npy'))
x = xy["x"]
y = xy["y"]
x.shape, y.shape
```

```
((3000, 10), (3000, 1))
```

In a three dimensional view, the data are as follows when showing the three first components.

```
cols = ["blue","red","green","orange","purple",
        "brown","olive","magenta","cyan", "black"]
```

```
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import numpy as np

def f_plot_scatter3d(z,y,title=" ",xlabel="x",ylabel="y"):
    fig = plt.figure(figsize=(8, 6), dpi=80)
    ax = plt.axes(projection='3d')
    for k in iter(np.unique(y)):
        ax.scatter3D(z[:,0], z[:,1], z[:,2], c="black",s=1.5)
    plt.gca().set(#aspect='equal',
                  title=title,
                  xlabel=xlabel, ylabel=ylabel)
    plt.show()
```

```
f_plot_scatter3d(x[:,0:3],y,title="9 classes in 3 dimensions")
```

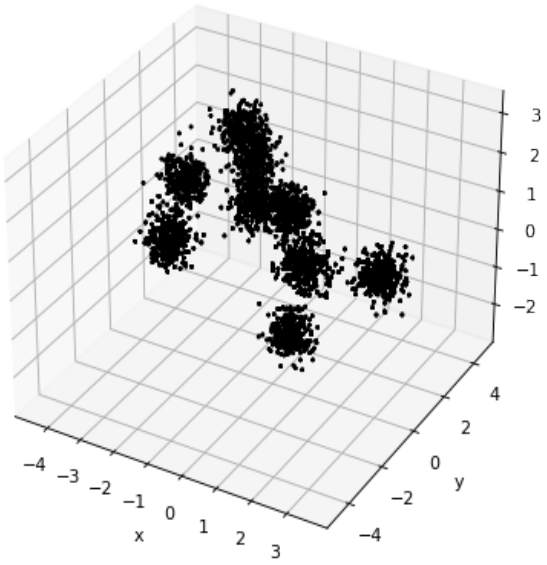

Figure 8.1: Visualization of 3 variables among 10 for 9 classes.

The classes are not fully apparent when a different color or a different marker is not added to the points for each class. This information is generally unknown, except if some clustering was performed before in order to help the data analysis and the visualization.

With real data, the variables y_i should be found in an unsupervised way (clustering), otherwise if they are known a supervised method for visualization may be better actually in order to use the additional information (see "fda", "discriminant analysis" in the literature), this is out of the scope herein.

Next, the projections are from the high dimensional space, 10 here, into the bidimensional plane, hence 2 instead of 3 among the usual choices. The labels y_i are unknown during the computations, the corresponding colors are just added to each points from the visualization after processing the projection. This allows to better compare visually the differences between the methods.

PCA with sklearn

The algorithm for pca is already implemented in sklearn. The call to the function available with the module is written as follows.

```
import numpy as np
from sklearn.decomposition import PCA
UPCA = PCA(n_components=2)
z_pca = UPCA.fit_transform(x)
```

Thus, the pca projection leads for the first principal plane in 2d.

```
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse

def f_plot_scatter(z,y,title="",xlabel="",ylabel="",isellipse=False):
    y = y.astype(np.int8).ravel()
    if isellipse is False:
        fig = plt.figure(figsize=(8, 6), dpi=80)
        fig.set_figwidth(8)
        fig.set_figheight(6)
        plt.title(title)
        for k in iter(np.unique(y)):
            plt.scatter(z[y==k,0], z[y==k,1], c=cols[k], s=1.5)
        plt.gca().set(#aspect='equal',
                      title=title,
                      xlabel=xlabel, ylabel=xlabel)
    if isellipse is True:
        fig, ax = plt.subplots(figsize=(8, 6), dpi=80)
        fig.set_figwidth(8)
        fig.set_figheight(6)
        plt.title(title)
        for k in iter(np.unique(y)):
            x1_s   = z[y == k, 0]
            x2_s   = z[y == k, 1]
            y_s    = y[y == k]
            cov = np.cov(x1_s, x2_s)
            vals, vecs = np.linalg.eigh(cov)
            order = vals.argsort()[::-1]
            vals = vals[order]
            vecs = vecs[:,order]
            theta = np.degrees(np.arctan2(*vecs[:,0][::-1]))
            w, h = 3.5 * np.sqrt(vals)
            ell = Ellipse(xy=(np.mean(x1_s), np.mean(x2_s)),
              width=w, height=h, angle=theta, color='black', alpha=0.1)
            ell.set_facecolor(cols[k])
            ax.add_artist(ell)
            ell2 = Ellipse(xy=(np.mean(x1_s), np.mean(x2_s)),
               width=w, height=h, angle=theta, color='black', alpha=1)
            ell2.set_facecolor('None')
            ax.add_artist(ell2)
            ax.scatter(x1_s, x2_s, label='.', c=cols[k],
            lw = 0, alpha=1, s=1.5)
        ax.set_xlabel(xlabel, fontsize=20)
        ax.set_ylabel(ylabel, fontsize=20)
    plt.show()
```

```
title = title="Projection of 10d dataset with pca"
f_plot_scatter(z_pca, y, title=title, isellipse=True)
```

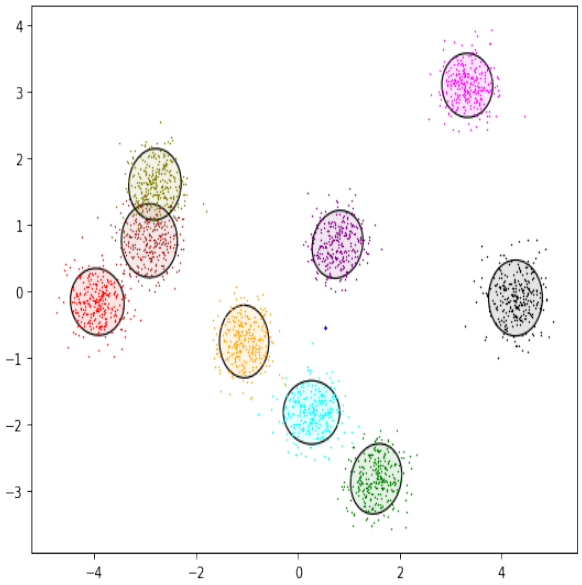

Figure 8.2: Visualization of two first pca components for 9 small classes

For each group an ellipse is added on top of its scatterplot for helping the visualization. Some data points may appear less near to the cluster as this is allowed by the Gaussian distribution by definition.

There is a small overlapping because some clusters are near in the data space. For this example, we will check how behave an autoencoder.

Incremental PCA with sklearn

For larger datasets, it is preferred an incremental algorithm which operates with chunks from a data file. With sklearn it is possible to enter the minibatches manually one after another as for our loops with pytorch, via the function "partial_fit()". Here, this is sklearn which cycles the dataset with minibatches until convergence.

```
from sklearn.decomposition import IncrementalPCA
IPCA   = IncrementalPCA(n_components=2, batch_size=10)
z_ipca = IPCA.fit_transform(x)
title  = "Projection of 10d dataset with ipca"
f_plot_scatter(np.vstack([-z_ipca[:,0], -z_ipca[:,1]]).T,
               y, title=title, isellipse=True)
```

```
/home/rodolphe/TensorFlow/myenv/lib/python3.8/site-
packages/sklearn/decomposition/_incremental_pca.py:338: RuntimeWarning:␣
 ↪invalid
value encountered in true_divide
  explained_variance_ratio = S**2 / np.sum(col_var * n_total_samples)
```

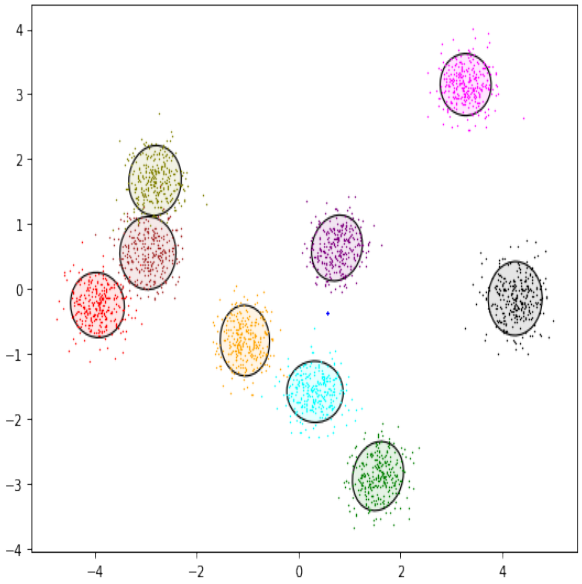

Figure 8.3: Visualization of two first ipca components for 9 classes

The representation is not exactly the same because the sign from the eigen vectors may be inverted and the iterative algorithm depends on hyperparameters such as the size of the minibatches, but this is nearly the same visualization than pca. Note that a pca algorithm is also implemented in pytorch natively, see "torch.pca_lowrank()". For large matrices, approximations of pca via randomization or random projection are appealing alternatives to the exact solution.

Kernel PCA with sklearn

In the nonlinear choice, it is considered a kernel pca which proposes a trick (related to "svm", "support vector machines") in order to change the inner products into nonlinear functions. Some options are:

- kernel is with the choices ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘cosine’, with by default=‘linear’ with is for the usual pca.
- gamma is a float for kernel coefficient in rbf, poly and sigmoid kernels. By default it is equal to 1/n.
- degree is by default=3 and only for poly kernels.

```
from sklearn.decomposition import KernelPCA
KPCA   = KernelPCA(n_components=2, kernel='sigmoid')
z_kpca = KPCA.fit_transform(x)
title  = "Projection of 10d dataset with kpca"
f_plot_scatter(z_kpca, y , title=title, isellipse=True)
```

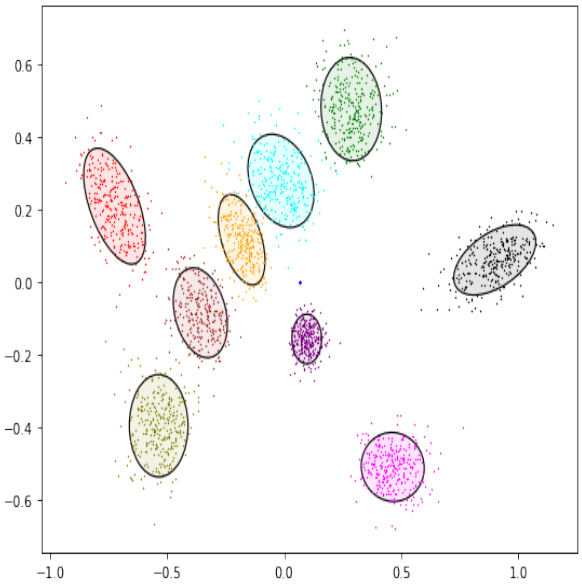

Figure 8.4: Visualization of two first kpca components for 9 classes

This is not working so well here with clusters very near, but some clusters are more separated or smaller than with a linear kernel.

8.1.3 Implementation of t-sne with sklearn

Let try also t-sne, a recent nonlinear method for reduction and visualization.

```
from sklearn.manifold import TSNE
z_tsne = TSNE(n_components=2, init='pca').fit_transform(x)
title  = "Projection of 10d dataset with tsne"
f_plot_scatter(z_tsne, y ,title=title, isellipse=True)
```

```
/home/rodolphe/TensorFlow/myenv/lib/python3.8/site-
packages/sklearn/manifold/_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
/home/rodolphe/TensorFlow/myenv/lib/python3.8/site-
packages/sklearn/manifold/_t_sne.py:991: FutureWarning: The PCA␣
 ↪initialization
```

```
in TSNE will change to have the standard deviation of PC1 equal to 1e-4 in␣
 ↪1.2.
This will ensure better convergence.
  warnings.warn(
```

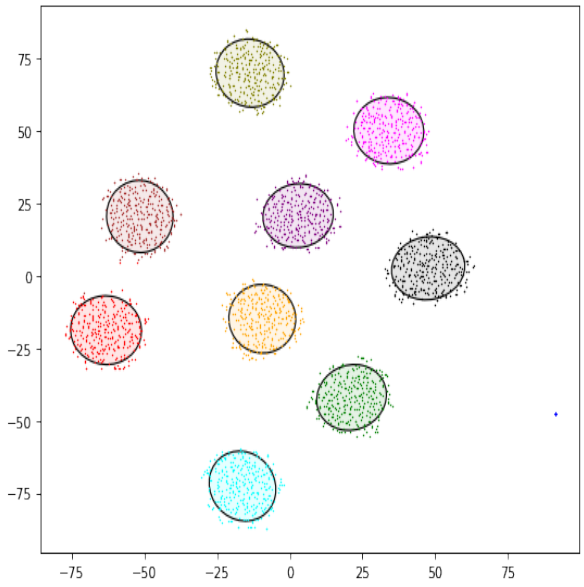

Figure 8.5: Visualization from t-sne after pca for 9 classes

Here for most samples tested from the distribution considered, as for this draw, the clusters are visually well separated because the overlap is not so high. Otherwise small or large overlapping may appear with a cluster behind another one on the visualization map like in the orthogonal projection from pca above. This method is today widely used in theory and practice for visualization, despite that it is not yet perfect: the output from this nonlinear method often replaces the previous maps from pca in many research domains (machine learning, biology, physics, ...).

8.1.4 Quality indicators

Numerical indicators for judging the quality of a reduction or a visualization are available from the literature in python modules. Each indicator depends on the property that one wants to underline. For instance, the "Davies-Bouldin" and the "Silhouette" focus on the separation of the clusters projected from existing label $\{y_i\}$ which are not used by the unsupervised reduction method.

The Davies-Bouldin index is the average similarity between each class and its most similar one. This indicator is preferred minimal because it decreases if the classes are more separated. The average of the Silhouettes is the mean value from average dissimilarities. This indicator is belongs to the interval $[-1,+1]$ and is preferred maximal because it increases for more compact classes.

In python, these numerical indicators are implemented in sklearn, they are obtained from the call to the following function.

```
import numpy as np
from sklearn.metrics import davies_bouldin_score
from sklearn.metrics import silhouette_score
def f_score_projection(z,y,name="",show=False):
    y = y.ravel()
    db = davies_bouldin_score(z, y)
    sl = silhouette_score(z, y)
    if show is True:
        print("Davies_Bouldin_score of",name, "=",np.round(db,3),
              "\nSilhouette_score     of",name, "=",np.round(sl,3))
    return db, sl
```

```
db_pca, sl_pca = f_score_projection(z_pca,y,"pca",False)
db_ipca, sl_ipca = f_score_projection(z_ipca,y,"ipca",False)
db_kpca, sl_kpca = f_score_projection(z_kpca,y,"kpca",False)
db_tsne, sl_tsne = f_score_projection(z_tsne,y,"tsne",False)
```

As expected, t-sne perfoms better than the pca methods, while pca and ipca are equivalent. The resulting values are given below in a table after adding autoencoders to the list.

8.2 Autoencoders with pytorch

In this section, autoencoders are trained with pytorch for two datasets, the small dataset from above and a large dataset from handwritten digit images. We are mainly interested on the definition of the autoencoders, their training for a linear or nonlinear model with hidden layers and how to combine them with glm in order to eventually improve the resulting regression or classification.

8.2.1 Definition and link with pca

The neural networks for autoencoder are defined with the same output and input, the multidimensional vector x_i. They are unsupervised because the label variables y_i are not involved in the definition and in the training of the model. They have been called "auto-associative" neural networks by the past, a larger family which includes other networks. There are two cases, the linear reduction and the nonlinear reduction.

Linear case

One popular interpretation in the literature of the autoencoder is its close relation to pca in the linear case. The authors of a recent communication [1] have recalled their result from 1988: writing the linear autoencoder formally as with the matrix H which aggregates the values h_i of the hidden

[1] See "https://doi.org/10.1007/s00422-022-00937-6, Biological Cybernetics", 2022) for a discussion on autoencoders and pca plus some related recent architectures of neural networks.

layer for the input x_i,

$$\begin{aligned} H &= W_1 X \quad \text{encoder (reduces the input, dimension } p \text{ to } k) \\ \hat{X} &= W_2 H \quad \text{decoder (retrieves the input, dimension } k \text{ to } p)\,, \end{aligned}$$

then the solution for the optimal matrices $\hat{W}_2$ and $\hat{H}$ needs to solve for the minimum of:

$$\|X - \hat{X}\|^2 = \|X - W_2 H\|^2\,.$$

The solution comes from standard linear algebra[2] with the "best rank k approximation of X", such that with the svd $X = U_n \Sigma_n V_n^T$, the solutions are respectively:

$$\begin{aligned} \hat{W}_2 &= U_k S^{-1} \\ \hat{H} &= S \Sigma_k V_k^T \\ \hat{W}_1 &= \hat{W}_2^T\,. \end{aligned}$$

Here, U_n, Σ_n V_n are the respective truncated matrices from U_n, Σ_n V_n by keeping the part corresponding to the number of nodes k of the unique hidden layer while S is an arbitrary non-singular matrix. The author had concluded that an autoencoder with an hidden layer and a linear output activation function learns the "weights that span the same subspace as the one spanned by the principal component vectors" when the loss function is the MSE.

Without the orthogonalization, linear autoencoders are expected to be less effective than pca for linear projection. Actually, orthogonality is not an issue and may be introduced via some kind of explicit penalization for more advanced linear settings according to recent architectures of the literature. With nonlinearities they become able to improve dramatically the reduction. They are also flexible with the possibility to be associated with other architectures of neural networks.

General case

The idea is to define after the input layer a set of hidden layers with a decreasing number of nodes until a dimension chosen for the reduction. This layer is the "bottleneck layer" where the reduction is obtained. After this layer, the hidden layers have a increasing number of nodes until the size of x_i. The first set of layers are the encoder and the second set of layers is the decoder. The autoencoder is able to encode a vector x_i into a reduced vector $\hat{h}_i$ and then to decode the reduction into an approximation $\hat{x}_i$ of the input vector with eventually some small error:

$$\hat{x}_i \approx x_i\,,$$

where,

$$\begin{aligned} \hat{h}_i &= \text{encoder}(x_i) \quad \text{is the reduction} \\ \hat{x}_i &= \text{decoder}(\hat{h}_i) \quad \text{is the reconstruction}\,. \end{aligned}$$

There are several results here after training:

[2] See Introduction to matrix computation, Stewart G (1973)

- the reductions $\{\hat{h}_i\}$ which allows a visualization and any post-processing.
- the decoder() which allows to get new vectors from the continuous latent reduced space.
- the reconstructions $\{\hat{x}_i\}$ which is eventually an improved version of the input vector.
- the encoder() for contructing the reduction for new input vectors if required.

Note that the dimension for the latent space may be two or three for visualization, but this may be more because an additional method may be used for visualizing the latent space.

8.2.2 Visualization of artificial data

Roughly speaking y_i is replaced by x_i in the models of neural network from the previous chapters. We focus here on unsupervised encoders in pytorch: we define a class for an autoencoder and train this new network with the numerical derivatives.

Model as a python class

A general class with hidden layers is defined as previously but with two type of hidden layers.

```
import torch.nn as nn

class AutoEncoder(nn.Module):
    def __init__(self, name, layers_encoder, layers_decoder,
                 init_layers = None):
        super().__init__()
        self.name = name
        self.layers_encoder = layers_encoder
        self.layers_decoder = layers_decoder
        self.net_encoder = nn.Sequential(*layers_encoder)
        self.net_decoder = nn.Sequential(*layers_decoder)
        self.init_layers = init_layers
        if self.init_layers is not None:
            for k in self.init_layers:
                torch.nn.init.xavier_uniform_(self.net_encoder[k].weight)
                torch.nn.init.xavier_uniform_(self.net_decoder[k].weight)

    def forward(self, x):
        encoded = self.net_encoder(x)
        decoded = self.net_decoder(encoded)
        return decoded

    def encoder(self,x):
        encoded = self.net_encoder(x)
        return encoded

    def decoder(self,z):
```

```
        decoded = self.net_decoder(z)
        return decoded
```

The class allows to define the part of the network for the encoder and the part for the decoder. The forward pass needs to encode to the reduced space and then to decode to the data space. This suggests also to define two new functions for each of these two parts.

Initialization of the network

The initialization is by default or the "Xavier" one in our implementation. For nonlinearities, it has to be careful otherwise the activation functions may saturate such as the gradient cancels out too early for some nodes, or on the contrary the gradient may increase too much. This is the role of the initialization to avoid or limit these events for the activation functions having this issue. There are diverse approaches implemented in pytorch, the fully random one and the careful random one, with diverse versions in the module[3] "torch.nn.init" as summarized below for a matrix of weights, say a pytorch tensor.

Initialization	Values drawn from	Parameters
"uniform_"	Uniform distribution $\mathcal{U}(a,b)$	with $a=0$ and $b=1$ by default
"normal_"	Normal distribution $\mathcal{N}(\text{mean},\text{std}^2)$	with mean $=0$ and std $=1$ by default
"xavier_uniform_"	Uniform distribution $\mathcal{U}(-a,a)$	$a=\sqrt{\frac{6\times\text{gain}^2}{\text{fan\_in}+\text{fan\_out}}}$
"xavier_normal_"	Normal distribution $\mathcal{N}(0,\text{std}^2)$	$\text{std}=\sqrt{\frac{2\times\text{gain}^2}{\text{fan\_in}+\text{fan\_out}}}$
"kaiming_uniform_"	Uniform distribution $\mathcal{U}(-\text{a},\text{a})$	$\text{a}=\sqrt{\frac{3\times\text{gain}^2}{\text{fan\_mode}}}$
"kaiming_normal_"	Normal distribution $\mathcal{N}(0,\text{std}^2)$	$\text{std}=\sqrt{\frac{\text{gain}^2}{\text{fan\_mode}}}$

The two intermediate rows and the two last rows are respectively for the Glorot and He estimation. The values for gain, fan_in, fan_out or fan_mode are the input size, output size and for the mode one of them. This is different to the previous bounds on the gradient in the previous chapter: by shriking the initial weights the inialization remains random and small. This helps a better convergence by avoiding an early vanishing of the gradient when the weight value is near the saturation of the activation function because of the "S shape" of most of them.

Dataloader for input and output

The data is loaded as pytorch tensors. Note that the class labels are optional and not used during training, they could be dummy variables here.

```
import torch
from torch.utils.data import TensorDataset, DataLoader
import deepglmlib.utils as utils
dataset = TensorDataset( torch.from_numpy(x).float(),
```

[3] "https://pytorch.org/docs/stable/nn.init.html"

```
                            torch.from_numpy(y).int() )
dl_train, dl_test, n, n_train, n_test = utils.f_splitDataset(dataset)
```

Structure of the layers

An example of autoencoder is as follows for a non orthogonal projection related to pca.

```
import torch.nn as nn
import copy

layers_encoder = []
layers_encoder.append(nn.Linear(x.shape[1],7, bias=True))
layers_encoder.append(nn.Linear(7, 4))
layers_encoder.append(nn.Linear(4, 2))

layers_decoder = []
layers_decoder.append(nn.Linear(2, 4))
layers_decoder.append(nn.Linear(4, 7))
layers_decoder.append(nn.Linear(7,x.shape[1]))
```

Training function

The parameters are optimized with the following function for an Euclidean loss. For the python code, the only difference with previously is the output (x_i instead of y_i) of the network and the optimizer Adam.

```
def f_train_autoencoder(dl_train,autoencoder,nbmax_epoqs,
                        lr,device=None,epoch_print=5):

    optimizer = torch.optim.Adam(autoencoder.parameters(), lr=lr)
    loss = nn.MSELoss(reduction='sum')
    loss_s = np.zeros(nbmax_epoqs)
    if device is not None: autoencoder=autoencoder.to(device)
    autoencoder.train()

    t=0
    for epoch in range(nbmax_epoqs):
        loss_t = 0
        for step, tuple_b in enumerate(dl_train):
            xb = tuple_b[0]
            yb = tuple_b[1]
            if device is not None:
                xb=xb.to(device)
                yb=yb.to(device)
```

```
            xb_hat = autoencoder(xb)
            lossb = loss(xb_hat, xb)
            optimizer.zero_grad()
            lossb.backward()
            optimizer.step()
            loss_t += lossb
        loss_s[t] = loss_t
        if epoch % epoch_print == 0 \
           or (epoch == nbmax_epoqs-1 and epoch_print<=nbmax_epoqs):
            print("t=",t,"\tloss=",
                  np.round(loss_t.detach().cpu().numpy(),5))
        t+=1

    autoencoder.eval()
    tmax = t

    return loss_s, tmax
```

Optimizers for deep networks

The optimizer Adam is an alternative to the optimizer SGD presented in the previous chapters, which performs well in practice when the network has many hidden layers in an high dimensional space. This is currently the state of art for deep networks.

Several optimizers are available with pytorch natively (see also the available modules from repositories), in the module[4] "torch.optim", with for instance:

Optimizer	Description
"SGD"	a stochastic gradient descent with minibatches and an optional momentum, extends the Robbins–Monro method (1951)
"Adam"	the Adam algorithm (2014) is a recent popular stochastic gradient descent based on "adaptive estimates of lower-order moments", it combines AdaGrad and RMSProp
"LBFGS"	the L-BFGS algorithm (1989), see the previous chapter for an illustration
"RPROP"	the "resilient backpropagation algorithm" (1992), it is a pioneer approach for adaptive estimates and can perform well (improved by RMSProp)

Results of the training

An exemple of training with the Adam optimizer for this autoencoder was stored in a file, it is loaded as follows.

[4] "https://pytorch.org/docs/stable/optim.html"

```
autoencoder =  AutoEncoder("AE-3-2",copy.deepcopy(layers_encoder),
                           copy.deepcopy(layers_decoder))
```

```
###loss_train_s, tmax = f_train_autoencoder(dl_train,autoencoder,300,0.
 ↪001,epoch_print=50)
# loss_train_s, tmax = f_train_autoencoder(dl_train,autoencoder,1000,0.
 ↪0001,epoch_print=100)

# torch.save(autoencoder,"./datasets_book/autoencoder_10d_ae.pth")
# torch.save(autoencoder.state_dict(),"./datasets_book/
 ↪autoencoderw_10d_ae.pth")
```

```
autoencoder.load_state_dict(torch.load(towdir("autoencoderw_10d_ae.pth")))
```

```
<All keys matched successfully>
```

The reduction $\hat{h}_i$ in the latent space from the bottleneck layer is retrieved for each data vector x_i of the dataset with "encoder()", the function implemented for this purpose:

```
z_ae = autoencoder.encoder(torch.from_numpy(x).float()).detach().numpy()
```

```
print(z_ae.shape)
```

```
(3000, 2)
```

The latent space is visualized as for pca.

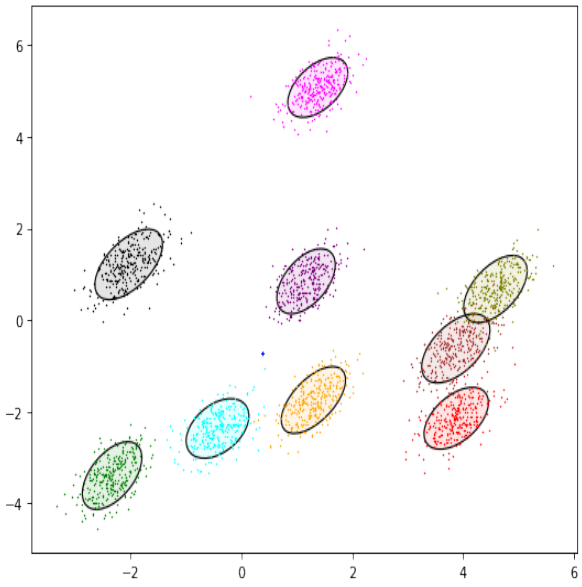

Figure 8.6: Visualization from linear autoencoder for 9 classes

The output just before comes from the following python program.

```
title = "Projection of 10d dataset with ae"
f_plot_scatter(z_ae, y, title=title, isellipse=True)
```

The resulting projection is for this example very similar to pca, but there is one dimension of pca which is compressed. A similar subspace than from pca is retrieved but with some linear transformation. Note that the choice of structure for the network is not exactly corresponding to pca also because there is an intermediate hidden layer before and after the bottleneck layer.

This is a known observation from the literature, that the autoencoder is able to retrieve pca despite that it does not include the hypothesis of orthogonality. The numerical indicators allow to retrieve these similarities if they are identical.

```
db_ae, sl_ae = f_score_projection(z_ae,y,"ae",False)
```

Nonlinear autoencoder and training

In this paragraph, we are interested on a non linear projection, hence let test another structure of autoencoder with nonlinear activation functions. The resulting latent space should perform better for the separation between the clusters than a linear model such that pca.

```
layers_encoder = []
layers_encoder.append(nn.Linear(x.shape[1],7, bias=True))
layers_encoder.append(nn.Tanh())
layers_encoder.append(nn.Linear(7, 4))
layers_encoder.append(nn.Tanh())
layers_encoder.append(nn.Linear(4, 2))

layers_decoder = []
layers_decoder.append(nn.Linear(2, 4))
layers_decoder.append(nn.Tanh())
layers_decoder.append(nn.Linear(4, 7))
layers_decoder.append(nn.Tanh())
layers_decoder.append(nn.Linear(7,x.shape[1]))

autoencoder_nl =  AutoEncoder("AE-",copy.deepcopy(layers_encoder),copy.
 ↪deepcopy(layers_decoder), [0,2])
```

An exemple of training for this autoencoder (with nbmax_epoqs=1500 and lr=0.001) was stored in a file, it is loaded.

```
autoencoder_nl.load_state_dict(torch.load(towdir("autoencoder_nlw_10d_ae.
 ↪pth")))
```

```
<All keys matched successfully>
```

```
z_ae_nl = autoencoder_nl.encoder(torch.from_numpy(x).float()).detach().
  ↪numpy()
```

```
title = "Projection of 10d dataset with nonlinear ae"
f_plot_scatter(z_ae_nl, y, title=title, isellipse=True)
```

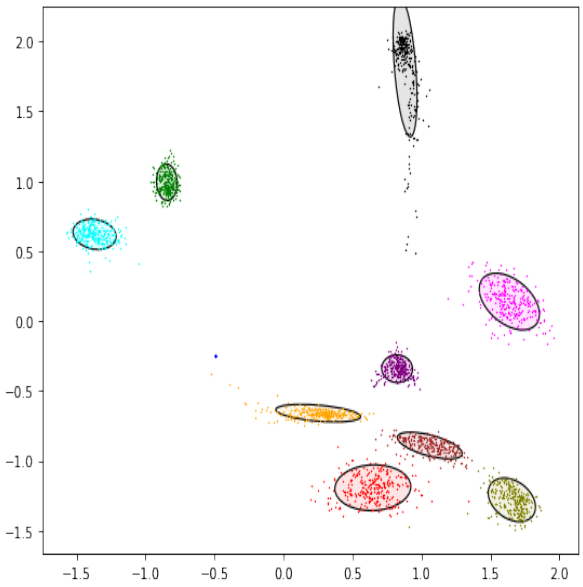

Figure 8.7: Visualization from nonlinear autoencoder for 9 classes

```
db_ae_nl, sl_ae_nl = f_score_projection(z_ae_nl,y,"ae_nl",False)
```

The resulting model is able to separate better the projection points from some classes but there is also some distortion among the clusters with ellipsoidal shapes and diverse orientations for their main axis. With high dimensional data, the autoencoder is also able to perform better than pca when associated with a post-processing via t-sne for biological data for instance.

Comparison of different visualizations with two indicators

The numerical results for the artificial dataset are summarized just below with the indicators Davis-Bouldin and Silhouettes. Alternative quality indicators related to the nearest neighboors instead are also of first interest but not presented herein. For the two indicators considered, the best visualization separates the best the true classes on its graphic and gets the best indicator values.

```
import pandas as pd

method_s = ["pca (3000)", "ipca (3000)", "kpca (3000)", "t-sne (3000)",
```

```
            "linear ae (3000)", "non linear ae (3000)"]

db_s = [db_pca, db_ipca, db_kpca, db_tsne, db_ae, db_ae_nl]
sl_s = [sl_pca, sl_ipca, sl_kpca, sl_tsne, sl_ae, sl_ae_nl]

results            = [method_s, db_s, sl_s]
results_pd         = pd.DataFrame(results).transpose()
results_pd.columns = ["method (sample size)",
                      "davis-bouldin",
                      "silhouettes"]
with pd.option_context('float_format', '{:.4f}'.format,
                       'display.expand_frame_repr', False):
    print(results_pd)
```

```
  method (sample size) davis-bouldin silhouettes
0            pca (3000)        0.4781      0.6542
1           ipca (3000)        0.4481      0.6624
2           kpca (3000)        0.4755      0.6472
3          t-sne (3000)        0.3839      0.7342
4      linear ae (3000)        0.4439      0.6606
5  non linear ae (3000)        0.3315      0.7663
```

In summary, the autoencoders are able to outperform pca and even t-sne as expected, for some datasets and some settings of the algorithms. Visually, if the indicators are better for this trained autoencoder, it appears clearly on the graphics that some relative distances between the classes are very different between the approaches. Hence some improvement is still required here for the autoencoder (choice of number of layers, nodes and the activation functions plus the training). This trained nonlinear autoencoder does not always keep the shape of the distribution of the classes from the data space when its performs the projection with the encoder. On the contrary, pca is linear thus the Gaussian shape of the classes remains with less varied distortions but also less separation for the frontiers. A cross-validation may lead to a better training but is costly, see sklearn and its module for mlp. Note that autoencoders and pca allow to project easily new data such as a test sample. For t-sne this is less natural as one needs to freeze the former projection points or build an external model such as a neural network for learning the former projection.

Next a real dataset is processed with the presented python class for autoencoders, this is a classical dataset with grey images from handwritten digits (between 0 to 9) which is worth to consider as it appears the most often in the deep learning literature for the comparison of algorithms.

8.2.3 Autoencoder for 60000 images

In this section, a large dataset of images which is currently often presented as an example is projected on a nonlinear latent space with an autoencoder. It is defined with three hidden layers where the middle one is the latent space for the projection. Before an incremental pca is implemented with minibatches because the dataset is kept on the computer disk. The format of the data file is

the binary hdf5 compression which is still near the state of art until today for tabular data, at least for a sequential access to its local contents.

Dataset (from the image files to one binary hdf5 file)

```
import torch; torch.manual_seed(0)
import torch.nn as nn
import torch.nn.functional as F
import torch.utils
import torch.distributions
import torchvision
import numpy as np
import matplotlib.pyplot as plt#; plt.rcParams['figure.dpi'] = 200
```

```
# device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Let download the dataset[5] with pytorch in a tensor format in order to save each image into a row of a compressed data file. The first call to "MNIST()" takes the files from a remote server into the chosen directory of the computer disk. All the next calls can access directly the local files until they are eventually removed by the user.

```
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST

dir_mnist = towdir("")

dl_train = DataLoader( MNIST(dir_mnist, train=True, download=True,
                             transform=torchvision.transforms.ToTensor()),
                       batch_size=200, shuffle=True)

dl_test  = DataLoader( MNIST(dir_mnist, train=False, download=True,
                             transform=torchvision.transforms.ToTensor()),
                       batch_size=200, shuffle=True)
```

We then save the images in a file hdf5.

```
import numpy as np
import tables

def f_save_dataloader_to_h5py(dataloader,filename,d):
    file = tables.open_file(filename, mode='w')
```

[5] "http://yann.lecun.com/exdb/mnist/"

```
    x_atom = tables.Float64Atom()
    y_atom = tables.Int16Atom()
    x_ds = file.create_earray(file.root, 'x', x_atom,(0,d))
    y_ds = file.create_earray(file.root, 'y', y_atom,(0,1))

    for step, (xb, yb) in enumerate(dataloader):
        xb = xb.detach().numpy()
        xb = xb.reshape((len(xb),d))
        yb = yb.detach().numpy()
        yb = yb.reshape((len(yb),1))
        x_ds.append(xb)
        y_ds.append(yb)

    file.close()
```

```
f_save_dataloader_to_h5py(dl_train,towdir("mnist60000.h5"),28*28)
f_save_dataloader_to_h5py(dl_test,towdir("mnist10000_test.h5"),28*28)
```

Incremental PCA with sklearn and an hdf5 file

An incremental pca is obtained by cycling the file, and then calling a function from sklearn as follows.

```
import h5py
from sklearn.decomposition import IncrementalPCA
def f_ipca_from_h5py(filename,n_components=2,
                     batch_size=10,chunk_size=100):
    file = h5py.File( filename , 'r' )
    data_y= file[ 'y' ]
    data_x= file[ 'x' ]
    n    = data_x.shape[ 0 ]  #; print(n)
    d    = data_x.shape[ 1 ]  #; print(d)
    ipca = IncrementalPCA(n_components= n_components,
                          batch_size= batch_size )

    for i in range ( 0 , n//chunk_size):
        ipca.partial_fit(data_x[i*chunk_size : (i+ 1 )*chunk_size,:])
        y_i = data_y[i*chunk_size : (i+ 1 )*chunk_size]
        if i==0:
            y = y_i.reshape((len(y_i),1))
        else:
            y = np.append(y,y_i, axis=0)
```

```
    if len(y)<n:
        next_i = (n//chunk_size)*chunk_size
        y_i = data_y[next_i:n]
        y = np.append(y,y_i, axis=0)

    file.close()
    return ipca, y, n, d
```

```
n_components = 50

ipca_mnist, y_mnist, n_mnist, d_mnist = \
   f_ipca_from_h5py(towdir('mnist60000.h5'),
                    n_components=n_components,\
                    batch_size=10, chunk_size=100)
```

```
print(type(ipca_mnist))
print(y_mnist.shape)
```

```
<class 'sklearn.decomposition._incremental_pca.IncrementalPCA'>
(60000, 1)
```

We get a python object for the projection, hence we just project our dataset now.

```
def f_projection_from_ipca_from_h5py(filename,ipca,z_ipca,chunk_size=100):
    file = h5py.File( filename , 'r' )
    data_x = file[ 'x' ]
    n    = data_x.shape[ 0 ]  #; print(n)
    d    = data_x.shape[ 1 ]  #; print(d)
    data_y = file[ 'y' ]
    y = np.zeros((n,1))

    for i in range ( 0 , n//chunk_size):
        z_ipca_i = \
              ipca.transform(data_x[i*chunk_size : (i+ 1 )*chunk_size,:])
        y_i = data_y[i*chunk_size : (i+ 1 )*chunk_size]

        y[i*chunk_size : (i+ 1 )*chunk_size] = y_i.reshape((len(y_i),1))
        z_ipca[i*chunk_size : (i+ 1 )*chunk_size,:] = z_ipca_i

    return z_ipca, y
```

Here the size of the chunks divides with an integer the number of rows of the dataset, otherwise the remaining rows must be projected too. The projection is directly saved in a file on the disk, with a virtual array created from "memmap()" with numpy (see next after for more about this binary file format).

```
filename_z_ipca_mnist  = towdir('z_ipca_mnist60000.memmap')

z_ipca_mist            = np.memmap(filename_z_ipca_mnist,
                                   dtype='float32', mode='w+',
                                   shape=(60000,n_components))
z_ipca_mist.shape
```

```
(60000, 50)
```

```
z_ipca_mist, y_mnist = \
     f_projection_from_ipca_from_h5py(towdir('mnist60000.h5'),
                                      ipca_mnist,z_ipca_mist,250)
```

```
print(z_ipca_mist.shape)
print(y_mnist.shape)
```

```
(60000, 50)
(60000, 1)
```

The resulting projection is below, from the two first principal components.

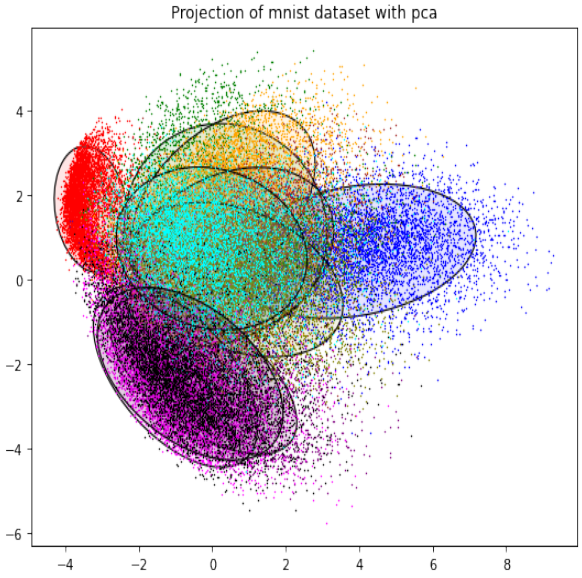

Figure 8.8: Visualization of two first ipca components for 60000 images

This graphic is obtained from the following python program.

```
title = "Projection of mnist dataset with pca"
f_plot_scatter(z_ipca_mist[:,0:2], y_mnist, title=title, isellipse=True)
```

The overlap of the ten classes of digits on the projection is not informative and even prone to errors of interpretation for the human who wants to understand the structure of the data cloud. This is expected from images of digits which are often very similar. For instance, the digits 3 and 9 may be both handwritten with open or close curves, and with only small differences from a writer to another. Too many similar classes are not well separated with a linear projection. This is a limit of pca which is just linear and not able to process well complex data clouds, the method needs numerous latent variables instead of just a few ones.

The obtained projection points are all shown without filtering for the first ipca plane. Note that in pca, there exists additional indicators called "contributions", "cosinus" and "qualities" which allow to remove points from the projection plane. The pca plane is the scatter plot from two new variables from pca, one by axis. When pca becomes a statistical exploratory method instead of just an algorithm, some data are known to be not well represented on the plane. When too far from the projection plane and with the worse indicator values, these points could be better shown by other new variables from pca, the ones corresponding to smaller eigenvalues than the two leading ones. The limit of this old approach is that for data with numerous variables an unknown number of diverses reduced dimensions is required for each unknown class in order to retrieve the underlying whole classification.

Current machine learning approaches prefer generally a nonlinear transformation of the variables, a non Euclidean space or a neighboors graph. Pca is able to improve these methods by helping the removal of the useless dimensions from the data table and the noises from its rows and its columns which correspond to the smallest eigenvalues. Autoencoders are able to provide such improved nonlinear view of the data in a flexible way. Its latent space may be able to separate better the unkown classes (or "clusters") even in an unsupervised approach without the knowledge of the true classification.

Nonlinear autoencoder with pytorch for a dataset with 60k rows

An autoencoder is trained for this dataset as follows. We propose to handle the dataloader with the hdf5 file with extension ".h5" which is suitable for large datasets (or small computers). This is performed by defining a new class extending "Dataset" from pytorch. The images are rows in the matrix 'x' and the labels are components in the vector 'y' from the hdf5 file.

```
import h5py
import torch
class DatasetH5(torch.utils.data.Dataset):
    def __init__(self, file_path, xname='x', yname='y'):
        super(DatasetH5, self).__init__()
        h5_file = h5py.File(file_path , 'r')
        self.x = h5_file[xname]
        self.y = h5_file[yname]
    def __getitem__(self, index):
        return (np.float32(self.x[index,:]),
                np.float32(self.y[index,:]),
```

```
                np.int32(index))
    def __len__(self):
        return self.x.shape[0]
```

The dataloader is created as usually, but with this new object dataset for pointing to the rows in the file.

```
dataset_mnist = DatasetH5(towdir('mnist60000.h5'),'x','y')
print(dataset_mnist.x.shape, dataset_mnist.y.shape)
```

```
(60000, 784) (60000, 1)
```

We consider the function for splitting the sample into two sets, the one for training and the one for testing.

```
# import deepglmlib.utils as utils
dl_train, dl_test, n, n_train, n_test = \
         utils.f_splitDataset(dataset_mnist,0.8,100)
```

```
print(n,n_train,n_test)
```

```
60000 48000 12000
```

With the data available via a dataloader, let define an example of autoencoder and train it. The encoder reduces the original size of the data vectors into a latent space which is then re-transformed into the original data space with the decoder. This supposes the choice of the hyperparameters: number of hidden layers, number of nodes for each layer and activation functions. For large datasets and high dimensions, this supposes a powerful computer in order to train several neural networks and keep the best one.

For instance for mnist, from 784 dimensions (for an image with heigth 28 pixels and with width 28 pixels), the first layer induces a new reduced dimension equal to 512 before another layer for a reduction to only 2 dimensions for visualization. This is the encoding phase, which is followed by the decoding phase which is exactly the inverse. If the dimensions of the bottleneck are just two or three the autoencoder is suitable for a direct visualization. With more than three nodes, the purpose is a reduction hence another method for post-processing the latent space is required.

Here the autoencoder is implemented for a direct visualization as follows. We define the encoding layers and the decoding layers with two python lists.

```
size_z = 2

layers_encoder = []
layers_encoder.append(nn.Linear(784,512, bias=True))
layers_encoder.append(nn.ReLU())
layers_encoder.append(nn.Linear(512,256, bias=True))
layers_encoder.append(nn.ReLU())
```

```
layers_encoder.append(nn.Linear(256,128, bias=True))
layers_encoder.append(nn.ReLU())
layers_encoder.append(nn.Linear(128,64, bias=True))
layers_encoder.append(nn.ReLU())
layers_encoder.append(nn.Linear(64,32, bias=True))
layers_encoder.append(nn.ReLU())
layers_encoder.append(nn.Linear(32,size_z, bias=True))

layers_decoder = []
layers_decoder.append(nn.Linear(size_z, 32, bias=True))
layers_decoder.append(nn.ReLU())
layers_decoder.append(nn.Linear(32, 64, bias=True))
layers_decoder.append(nn.ReLU())
layers_decoder.append(nn.Linear(64, 128, bias=True))
layers_decoder.append(nn.ReLU())
layers_decoder.append(nn.Linear(128, 256, bias=True))
layers_decoder.append(nn.ReLU())
layers_decoder.append(nn.Linear(256, 512, bias=True))
layers_decoder.append(nn.ReLU())
layers_decoder.append(nn.Linear(512, 784, bias=True))
layers_decoder.append(nn.Sigmoid())
```

The last activation function is sigmoidal hence the output of the neural network belongs to $[0;1]$. The components of the data vectors should belong to the same range.

This architecture looks too complex for this dataset with only 50000 data vectors for training so many weights but has lead to a relevant low dimensional projection as a first test. Note that the number of parameters is:

```
model_tmp =  AutoEncoder("AE_tmp",
                copy.deepcopy(layers_encoder),
                copy.deepcopy(layers_decoder))

print("Number of weights in the autoencoder: ",
      utils.f_get_p_model(model_tmp))

del model_tmp
```

```
Number of weights in the autoencoder:  1153874
```

The training is with the same minibatches than for training the neural networks for nonlinear generalized linear models, except that the variable y_i is not used here. No monitor class is implemented here and no loss for the test sample neither. The two lists of layers for encoding and decoding are processed by the forward function in the python class modeling the autoencoder. The main difference for this new function "f_train_fromh5_autoencoder()" with the previous function tested with

the small artificial dataset, "f_train_autoencoder()", is the penalization for the loss function. The id of the row is also eventually loaded because used in some cases like debugging: when shuffling is true in the dataloader the order of the rows is lost. For adding other optional computations, see for instance the function "f_train_glmr()" and the class "Monitor" as examples.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
def f_train_fromh5_autoencoder(dl_train,autoencoder,nbmax_epoqs,
                               lr,  epoch_print=3,  device=None,
                               transform_x=None,loss_yy_model=None):
    autoencoder = autoencoder.to(device)
    autoencoder.train()
    optimizer = torch.optim.Adam(autoencoder.parameters(), lr=lr)
    #optimizer = torch.optim.SGD(autoencoder.parameters(), lr=lr)
    loss_func = nn.MSELoss(reduction='sum')

    loss_s = np.zeros(nbmax_epoqs)
    for epoch in range(nbmax_epoqs):
        loss_b = 0
        for step, tuple_b in enumerate(dl_train):
            ib = None
            xb = tuple_b[0]
            yb = tuple_b[1]
            if len(tuple_b)==3: ib = tuple_b[2]
            if device is not None: xb = xb.to(device)
            if transform_x is not None: xb=transform_x(xb)
            xb_hat = autoencoder(xb)
            loss = loss_func(xb_hat, xb)
            if loss_yy_model is not None:
                loss = loss_yy_model(loss,ib,xb,yb,xb_hat,autoencoder)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            loss_b += loss
        loss_s[epoch] = loss_b.detach().cpu().numpy()
        if epoch%epoch_print==0 \
           or (epoch==nbmax_epoqs-1 and epoch_print<=nbmax_epoqs):
            print("epoch=",epoch,"\tloss=",np.round(loss_s[epoch],5))
        autoencoder.eval()
    return loss_s, epoch+1
```

The optimizer here is Adam because it performs very well for large models and was able to reduce the loss function automatically without additional advanced function for the learning rate. As seen previously, this is not required for linear models which just need an optimizer SGD because of the convexity of their loss function. The documentation of pytorch discusses this more advanced

optimizer and a large part of the literature on images processing where the recent optimizer Adam has become one of the best optimizers currently. This illustrates that a clever combination of different approaches, here two optimizers with "Adam ≈ AdaGrad+RMSProp", can achieve great results.

The autoencoder is trained and its weights are stored in a computer file.

```
autoencoder =  AutoEncoder("AE",copy.deepcopy(layers_encoder),
                           copy.deepcopy(layers_decoder))
```

```
# loss_train_s, tmax_ae_mnist = \
#   f_train_fromh5_autoencoder(dl_train,autoencoder,20,0.0003,
#                              device=device,transform_x=f_transform_x)

### torch.save(autoencoder,"./datasets_book/autoencoder_mnist60000.pth")
### torch.save(autoencoder.state_dict(),
               "./datasets_book/autoencoderw_mnist60000.pth")
```

The trained model is loaded by retrieving the weights from the computer file and the weights of the python class are filled with these values.

```
#autoencoder = autoencoder.cpu()
autoencoder.load_state_dict( \
    torch.load(towdir("autoencoderw_mnist60000.pth")))
```

```
<All keys matched successfully>
```

```
# utils.f_draw(range(0,tmax_ae_mnist),loss_train_s,
               "b-","t","loss (minibatch)"," ")
```

For a large dataset, one must be careful when computing the coordinates from the latent space by cycling the dataset with the dataloader. This is the purpose of the following function.

```
def f_get_latent_space_autoencoder(autoencoder,dl_train,
                                   device=None,transform_x=None,):
    autoencoder = autoencoder.to(device)
    autoencoder.eval()
    with torch.no_grad():
        i_ae = []
        t=0
        for step, tuple_b in enumerate(dl_train): #xb,yb,ib
            ib = None
            xb = tuple_b[0]
            yb = tuple_b[1]
            if len(tuple_b)==3: ib = tuple_b[2]
```

```
            #
            if device is not None: xb = xb.to(device)
            if transform_x is not None: xb=transform_x(xb)
            zb_hat = autoencoder.encoder(xb).detach().cpu().numpy()
            if t == 0:
                z_ae = zb_hat
                y_ae = yb.detach().numpy()
                if ib is not None: i_ae = ib
            else:
                z_ae = np.append(z_ae, zb_hat, axis=0)
                y_ae = np.append(y_ae,yb.detach().numpy())
                if ib is not None: i_ae = np.append(i_ae,ib)
            t+=1
    return z_ae, y_ae, i_ae
```

The full dataset is projected here from the trained model with the train sample, otherwise, the dataloader from the train sample or the test sample can be preferred eventually for validation purpose.

The resulting projection is as follows, after training the nonlinear autoencoder.

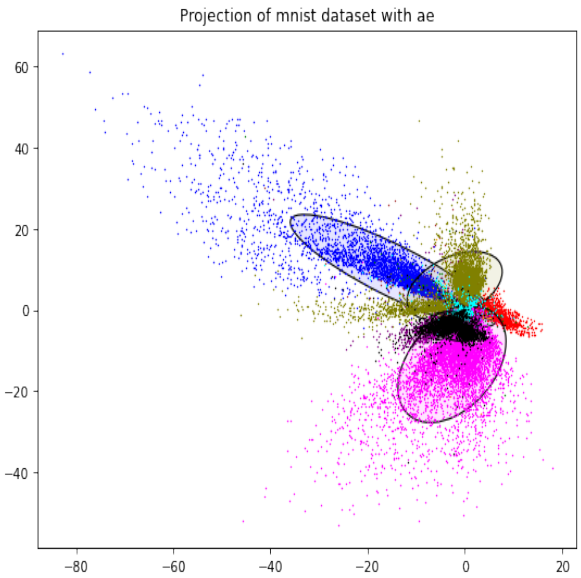

Figure 8.9: Visualization from nonlinear autoencoder for 60000 images

The python programs for this output is just below.

```
from torch.utils.data import TensorDataset, DataLoader

dataset_mnist = DatasetH5(towdir('mnist60000.h5'),'x','y')
dl_all = DataLoader(dataset_mnist,shuffle=False,batch_size=250)
```

```
z_ae_nl_mnist, y_ae_nl_mnist, i_ae_mnist = \
    f_get_latent_space_autoencoder(autoencoder.cpu(), dl_all,
                                   transform_x=f_transform_x,
                                   device=device)
```

```
title = "Projection of mnist dataset with ae"
f_plot_scatter(z_ae_nl_mnist, y_ae_nl_mnist, title=title, isellipse=True)
```

Then let compare the quality of the projection for the full sample after training.

A sample of the 60000 rows is drawn in order to get faster computation of the numerical indicators.

```
i_s = np.random.binomial(size=z_ipca_mist.shape[0],p=0.05,n=2)
```

```
db_ipca_mnist, sl_ipca_mnist = \
   f_score_projection(z_ipca_mist[i_s==1,:],
                      y_mnist[i_s==1].ravel(),"ipca-mnist",False)
```

```
db_ae_nl_mnist, sl_ae_nl_mnist = \
   f_score_projection(z_ae_nl_mnist[i_s==1,:],
                      y_mnist[i_s==1].ravel(),"ae-mnist",False)
```

Visually, this trained autoencoder seems not performing better than pca for this dataset but also the setting is not optimal. According to the literature, a more usual way to do is to choose a higher dimension for the latent space and perform after a nonlinear reduction such as t-sne. Note also that better architectures of autoencoder exist but often ask for far more intensive computations. Convolution autoencoders which model the neighboors between the pixels are a first step towards a better reduction, this is out of the scope of tabular data.

8.3 Low dimensional reduction via t-sne

The criterion optimized by t-sne involves the Euclidean distances between the data vectors, thus, a linear reduction is often better for these vectors in order to remove the noises and to reduce the computer time for computing the distances. If pca is the usual method for the pre-processing, random projection is a possible alternative method or an interesting complement.

8.3.1 Reduction with a random projection

For a large number of columns, a reduction is performed with a random projection.

The random projection method in brief

Random projection (rp) is a linear reduction method different from pca. It allows to approximate the Euclidean distance between two vectors. Each data vector $x_i \in \mathbb{R}^p$ is reduced to $\hat{x}_i \in \mathbb{R}^q$ with a matrix $R \in \mathbb{R}^{q \times p}$ where $q < p$. The new vector is written as follows:

$$\hat{x}_i = R x_i \,.$$

The reducing matrix may be sparse or dense. The cells of this matrix are random values generated from a Gaussian distribution (or eventually another one). With the right choice for the dimension of the reduction q, and a small ε depending on the value q, it is written that with high probability,

$$(1-\varepsilon)\|\hat{x}_i - \hat{x}_j\| \leq \|x_i - x_j\|^2 \leq (1+\varepsilon)\|\hat{x}_i - \hat{x}_j\| \,.$$

It is recognized the same inequality than in the Johnson-Lindenstrauss lemma. A smaller ε asks for a large dimension q, but for a large p one gets easily $q \ll p$ with an enough small ε, such as the gain is relevant. When the matrix R is sparse, the reductions are fast to compute for dense vectors x_i such as this version is said to be "database friendly", underlying that it is relevant for huge datasets. On the contrary to pca, rp does not ask for any matrix decomposition, just to generate a matrix for the reduction.

Implementation with memmap

The random projection is implemented with sklearn which generates the matrix R for a chosen reduced dimension q. Also a warning appears if the corresponding approximated distances are inaccurate in the lower dimension. For datasets which can't fit in the computer memory, the transformation is applied with chunks from a data file, in this paragraph.

The dataset which comes from a dataloader is stored in the computer disk in a data file, a binary compressed file which can be read as a numpy array. This is a virtual array from the computer disk instead of the computer memory, see "memmap()" from numpy. This format is such that there is no need for loops and chunks to process the rows of the array if the resulting arrays fit the computer memory: it allows linear algebra with other python objects of type numpy arrays and with the same operators. But this may be slower than hdf5 and loops for some computations. Other formats of compressed file exists for python, even with distributed architecture or better parallel processing, but not always compatible with numpy.

In order to save the result from the transformation directly on the computer disk, a loop is implemented. The linear algebra from numpy remains available, but applied to only subsets of rows until completing the full transformation of the data matrix. The optional shuffling is set to false in the dataloader for a sequential processing.

```
# import numpy as np

# # save dataloader into memmap file (numpy)
# # save into memmap format from numpy a dataloader (2 arrays)
# # x,y from dataloader et nb size of a minibatch
# def f_save_dl_xy_to_2memmap(dataloader,
```

```
#                                                   filename_x,
#                                                   filename_y,
#                                                   n=None,p=None,
#                                                   is_index=False):
#     if n==None or p==None:
#         n, p  = dataloader.dataset.tensors[0].shape

#     x_map = np.memmap(filename_x, \
#                           dtype='float32', mode='w+', shape=(n,p))

#     y_map = np.memmap(filename_y, \
#                           dtype='float32', mode='w+', shape=(n,1))
#     i_b=0
#     if is_index:
#         for b,(xb,yb,ib) in enumerate(dataloader):
#             x_map[i_b:(i_b+len(yb)),:] = xb
#             y_map[i_b:(i_b+len(yb)),:] = yb.reshape((len(yb),1))
#             i_b = i_b + len(yb)
#     else:
#         for b,(xb,yb) in enumerate(dataloader):
#             x_map[i_b:(i_b+len(yb)),:] = xb
#             y_map[i_b:(i_b+len(yb)),:] = yb.reshape((len(yb),1))
#             i_b = i_b + len(yb)

#     np.savetxt(str(filename_x+".shape.txt"),[n,p])
#     np.savetxt(str(filename_y+".shape.txt"),[n,1])
#     del x_map, y_map
```

The dependent and independent variables are stored in two separated files.

```
filename_x = towdir('x_mnist60000.memmap')
filename_y = towdir('y_mnist60000.memmap')
```

```
n = dataset_mnist.x.shape[0]
p = dataset_mnist.x.shape[1]

utils.f_save_dl_xy_to_2memmap(dl_all, filename_x, filename_y,
                              n=n, p=p, is_index=True)
```

From the data file, a virtual numpy array is created via a function as below.

```
import numpy as np
```

```
# del x_map, y_map
```

```
def f_read_memmap(filename_x,n,p):
    x_map = np.memmap(filename_x, dtype='float32',
                      mode='r', shape=(n,p))
    return x_map

def f_write_memmap(filename_x,n,p):
    x_map = np.memmap(filename_x, dtype='float32',
                      mode='w+', shape=(n,p))
    return x_map
```

```
n, p = (60000, 784)

x_map = f_read_memmap(filename_x,n,p)
y_map = f_read_memmap(filename_y,n,1)
```

```
x_map.shape, y_map.shape
```

```
((60000, 784), (60000, 1))
```

Let project with a random projection the space of the columns in a smaller dimension, $120 > 50$, where 50 is the usual value for pca before t-sne. A random projection is able to keep accurate the distances between rows after the linear transformation when the reduced space is enough large.

```
from sklearn import random_projection
GaussianRP = random_projection.GaussianRandomProjection

p_random = 120
# mapper      = GaussianRP(n_components=p_random)

# x_map_mnist_reduced = mapper.fit_transform(x_map_mnist[0:100,:])

# RR = mapper.components_

# print(RR.shape)
```

```
# np.savetxt("./datasets_book/RR_120_784_mnist60000.txt",RR)
```

```
RR = np.loadtxt(towdir("RR_120_784_mnist60000.txt"))
```

The random projection allows to keep the distances while reducing the size of the vectors. Thus, the dataset has it number of columns reduced (from 784 to 120) by the product to the right with this matrix (after transpose). This is done with chunks/minibatches in order to proceed directly with the data on the computer disk here.

```
import numpy as np

def f_save_to_reduction_to_memmap_files(filename_xin,
                                        filename_xout,
                                        R,n= None,
                                        size_minibatch = 50):

    p_in, p_out = R.shape

    xin_map  = np.memmap(filename_xin, dtype='float32',
                         mode='r', shape=(n,p_in))

    xout_map = np.memmap(filename_xout, dtype='float32',
                         mode='w+', shape=(n,p_out))

    for idx_b in range(0, n, size_minibatch):
        idx_b2 = np.min( [idx_b+size_minibatch,n] )
        xb = xin_map[idx_b:idx_b2,:]
        xout_map[idx_b:idx_b2,:] = xb @ R

    np.savetxt(str(filename_xin+".shape.txt"),[n,p_in])
    np.savetxt(str(filename_xout+".shape.txt"),[n,p_out])
    del xin_map, xout_map
```

The reduced matrix is stored in a new data memmap file with 120 variables instead of the previous 784 ones.

```
filename_xin  = towdir('x_mnist60000.memmap')
filename_xout = towdir("x_rp120_mnist60000.memmap")

f_save_to_reduction_to_memmap_files(filename_xin,
                                    filename_xout,
                                    R=RR.transpose(),
                                    n = x_map.shape[0],
                                    size_minibatch = 250)
```

The new reduced matrix is accessed via a python object, z_rp_mnist, without loading its whole contents from the disk while the python object y_mnist remains available for the labels.

```
filename_z = towdir("x_rp120_mnist60000.memmap")

n = 60000
p = 120
```

```
z_rp_mnist = f_read_memmap(filename_z,n,p)
y_mnist    = y_map

z_rp_mnist.shape, y_mnist.shape
```

```
((60000, 120), (60000, 1))
```

Note that with this format, we keep the size of the matrix in a separated textual file when the data is stored, in order to provide the information when the data file is read later.

This reduced matrix is involved next, just after the projection with the 50 components from the incremental pca, in order to compare the results.

8.3.2 Projection t-sne after pca

In this subsection, the dataset mnist is visualized with t-sne thanks to a fast implementation, "opentsne". Other implementations are available with python. This one includes "barnes-hut t-sne" for instance which is able to approximate t-sne in order to make t-sne scalable for large datasets. Remember that the method is based on a sparse nearest neighboors graph which is mandatory for a large dataset and a limited computer memory. By following the documentation, and accessing the data directly from the computer disk thanks to the memmap file, one gets the following result:

```
import openTSNE
def f_projection_from_openTSNE(x_map,x_map_init,perplexity=30,n_jobs=1,
                               random_state=0,verbose=False):

    aff = openTSNE.affinity.PerplexityBasedNN(
        x_map,
        perplexity=perplexity,
        n_jobs=n_jobs,
        random_state=random_state,
    )

    z_init_openTSNE = openTSNE.initialization.rescale(x_map_init)

    #%%time
    z_tsne = openTSNE.TSNE(
        n_jobs=n_jobs,
        verbose=verbose,
    ).fit(affinities=aff, initialization=z_init_openTSNE)

    return z_tsne, z_init_openTSNE, aff
```

The output from t-sne for mnist after the initialization from the 50 first ipca components is as follows.

```
# import openTSNE

# x_map_init = copy.deepcopy( z_ipca_mist[:,0:2] )

# z_mnist_tsne, z_init_mnist_tsne, aff_mnist = \
#     f_projection_from_openTSNE(x_map = z_ipca_mist,
#                                x_map_init = x_map_init,
#                                perplexity=30,n_jobs=3,
#                                random_state=0,verbose=True)
```

```
### np.savetxt("./datasets_book/z_mnist_tsne.txt",z_mnist_tsne)
### np.savetxt("./datasets_book/y_mnist.txt",y_mnist)
```

```
z_mnist_tsne = np.loadtxt(towdir("z_mnist_tsne.txt"))
y_mnist      = np.loadtxt(towdir("y_mnist.txt"))
```

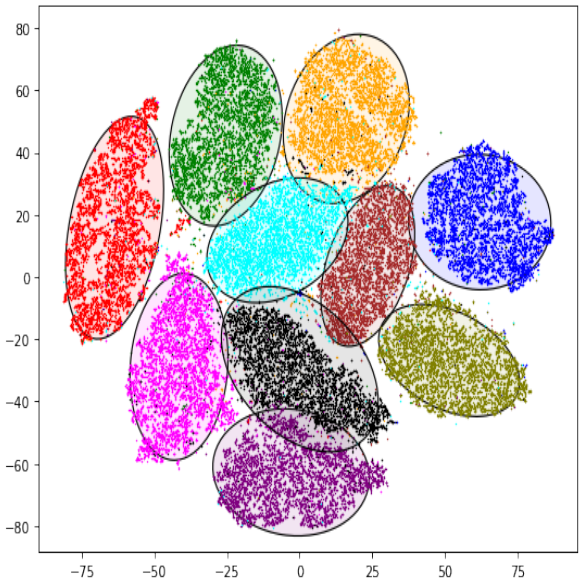

Figure 8.10: Visualization from t-sne after ipca for 60000 images

The python code for the graphic just before is below, it is recognized the ten classes almost perfectly separated. The method t-sne allows an unsupervised visualization of the natural classes on the contrary to pca for instance.

```
title=" "
f_plot_scatter(z_mnist_tsne, y_mnist, title=title, isellipse=True)
db_tsne_mnist, sl_tsne_mnist = f_score_projection(z_mnist_tsne[i_s==1,:],
                                                  y_mnist[i_s==1].ravel(),
                                                "tsne-mnist-after-pca",False)
```

8.3.3 Projection t-sne after rp+pca

Here rp is for random projection. The reduction before t-sne is a random projection (120 reduced dimensions) followed by an incremental pca (50 reduced dimensions). Hence, rp is just a preprocessing step before ipca, while ipca is followed by t-sne at the end.

It is better to reduce and standardize the columns before the partial fit because the whole dataset is not available at the beginning. The random projection allows to accelerate pca computations but with a small difference for the quality of the projection because of the approximate distances.

```
mn = np.sum(z_rp_mnist,axis=0)
sd = np.sqrt(np.var(z_rp_mnist,axis=0))

filename_zout = towdir("x_rp120_mnist60000_standardized.memmap")

z_rp_mnist_strd = np.memmap(filename_zout,
                            dtype='float32', mode='w+',
                            shape=(n,120))
size_chunks = 100

for idx_b in range(0, n, size_chunks):
    idx_b2 = np.min( [idx_b+size_chunks,n] )
    zb                    = z_rp_mnist[idx_b:idx_b2,:]
    z_rp_mnist_strd[idx_b:idx_b2,:] = (zb - mn)/sd

del z_rp_mnist_strd
```

```
size_chunks  = 100
n_components = 50
z_rp_mnist_strd = \
  f_read_memmap( towdir("x_rp120_mnist60000_standardized.memmap"),
                 n,120 )
z_ipca_150rp    = \
   f_write_memmap( towdir("z_ipca50_150rp_mnist60000.memmap"),
                   n,n_components )

from sklearn.decomposition import IncrementalPCA
ipca = IncrementalPCA( n_components= n_components,
                       batch_size= size_chunks )
print("compute transformation pca")
for idx_b in range(0, n, size_chunks):
    idx_b2 = np.min( [idx_b+size_chunks,n] )
    ipca.partial_fit(z_rp_mnist_strd[idx_b:idx_b2,:])
print("compute reduced coordinates")
for idx_b in range(0, n, size_chunks):
```

```
    idx_b2 = np.min( [idx_b+size_chunks,n] )
    z_ipca_150rp[idx_b:idx_b2,:] = \
      ipca.transform(z_rp_mnist_strd[idx_b:idx_b2,:])
del z_rp_mnist_strd, z_ipca_150rp
```

```
compute transformation pca
compute reduced coordinates
```

```
z_ipca_150rp    = \
   f_read_memmap( towdir("z_ipca50_150rp_mnist60000.memmap"),
                  60000,50 )
```

```
# import openTSNE

# x_map_init = copy.deepcopy( z_ipca_150rp[:,0:2] )

# x_map = z_ipca_150rp

# z_mnist_tsne_2, z_init_mnist_tsne_2, aff_mnist_2 = \
#     f_projection_from_openTSNE(x_map = x_map,
#                                x_map_init=x_map_init,
#                                perplexity=30,n_jobs=3,
#                                random_state=0,verbose=True)
```

```
### np.savetxt("./datasets_book/z_mnist_tsne_2.txt",z_mnist_tsne_2)
### np.savetxt("./datasets_book/y_mnist_2.txt",y_mnist)
```

```
z_mnist_tsne_2 = np.loadtxt(towdir("z_mnist_tsne_2.txt"))
y_mnist_2      = np.loadtxt(towdir("y_mnist_2.txt"))
```

```
title=" "
f_plot_scatter(z_mnist_tsne_2, y_mnist, title=title, isellipse=True)

db_tsne_mnist_2, sl_tsne_mnist_2 = \
      f_score_projection(z_mnist_tsne_2[i_s==1,:],
                         y_mnist[i_s==1].ravel(),
                         "tsne-mnist-after-rand-proj",False)
```

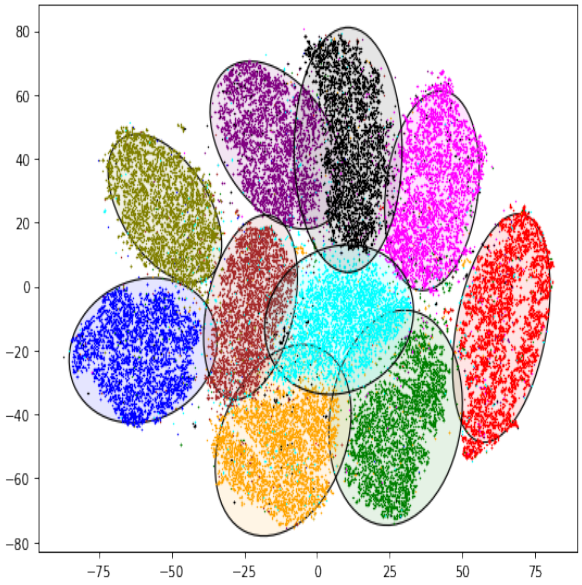

Figure 8.11: Visualization from t-sne after rp+ipca for 60000 images

The random projection is by definition and as its names says: "random". Hence several random projections may be tried in order to get the best final result. Some noise was added during the reduction, resulting to some data points between the clusters. Several clusters have changed their relative positions but the ten classes of digits are retrieved. This is informative because if two points keep the same vicinity for two different random projections, that means that they belong to the same cluster. Next the two t-sne projections are compared numerically with the previous ones.

8.3.4 Comparison of the visualizations

In summary, the four methods have performed as follows according to the numerical indicators.

```
import pandas as pd

method_s = ["ipca (60000)",
            "non linear ae (60000)",
            "t-sne-after-pca (60000)",
            "t-sne-after-randproj+pca (60000)",]

db_s = [db_ipca_mnist, db_ae_nl_mnist, db_tsne_mnist, db_tsne_mnist_2]
sl_s = [sl_ipca_mnist, sl_ae_nl_mnist, sl_tsne_mnist, sl_tsne_mnist_2]

results    = [method_s, db_s, sl_s]
results_pd = pd.DataFrame(results).transpose()
results_pd.columns = ["method (sample size)",
                      "davis-bouldin", "silhouettes"]
```

```
with pd.option_context('float_format', '{:.4f}'.format,
                       'display.expand_frame_repr', False):
    print(results_pd)
```

```
                method (sample size) davis-bouldin silhouettes
0                        ipca (60000)        3.4704      0.0621
1              non linear ae (60000)        4.0593      0.0220
2            t-sne-after-pca (60000)        0.8676      0.3763
3  t-sne-after-randproj+pca (60000)        1.0620      0.3321
```

Here, t-sne as a dedicated method for reduction and visualization outperforms this trained autoencoder but uses more information with the vicinity graph. Note that another recent method could be used here instead of t-sne for visualization, such as u-map. Tuning further the hyperparameters or the training should lead to better results but is time consuming for the computer. More advanced autoencoder architectures may do better than t-sne, see the literature. Currently, autoencoders are combinated with other neural networks with different architectures and have also been defined for other kinds of data such as time series, text, graph, or rgb images. The number of parameters is high or even huge which makes them prone to overfitting, a problem mostly solved with very large datasets and big workstations with powerful gpus.

8.4 Autoencoder associated with a glm

In this section, we are interested on autoencoder in the generalized regression models via alternative models. When the space of the variables is large it is usual to select the best subset via aic, bic or lasso. In machine learning data are often complex and numerous, hence neural networks are able to lead to new approaches with better results for these data.

Examples of extensions for regression/classification with reduction

There exists alternative approaches to a glm within a neural network which remains nevertheless the more usual way to do currently. In all cases, a penalization may be added, it is worth to try as explained in the chapter on the lasso regression, otherwise other regularizations are available.

Approach	Description	Criterion
Direct	Neural network embedding the glm with no regularization and reduction	$\text{argmin}_w F(x;w)$
	Neural network embedding the glm with a penalty term for regularization	$\text{argmin}_w F(x;w) + \alpha R(w)$
Plugin	Autoencoder/reduction followed by glm by consecutive minimization of the losses	$\text{argmin}_w F(\text{argmin}_h G(x;h);w)$
Combined	Autoencoder/reduction associated to glm by minimization of a combination of losses	$\text{argmin}_{w,h} G(x;h) + \lambda\ F(h;w)$

Example with one hidden layer in the autoencoder for linear regression

The model for the basic autoencoder with one hidden layer is as follows:

$$f_\theta(x_i) = g_x(W_x^T g_h(W_h x_i, b_h), b_x)\,.$$

This is with two stages:

$$h_i = h(x_i) = g_h(W_h x_i, b_h)$$

$$\hat{x}_i = \hat{x}(h_i) = g_x(W_x h_i, b_x)\,.$$

This leads to the three architectures from autoencoder to regression, with same expression for the predictions, $\hat{y}_i = \hat{\beta}^T \hat{h}_i + \hat{b}_y = \hat{\beta}^T g_h(\hat{W}_h x_i, \hat{b}_h) + \hat{b}_y$, but three different loss functions to optimize.

Approach	Criterion
Direct	$(\hat{\beta}, \hat{b}_y, \hat{W}_h, \hat{b}_h) = \underset{\beta, b_y, W_h, b_h}{\operatorname{argmin}} \sum_i \|y_i - \beta^T h(x_i) - b_y\|^2$
Plugin	$(\hat{W}_x, \hat{W}_h, \hat{b}_h, \hat{b}_x) = \underset{W_x, W_h, b_h, b_x}{\operatorname{argmin}} \sum_i \|x_i - \hat{x}(h(x_i))\|^2$ $\hat{\beta}, \hat{b}_y = \underset{\beta, b_y}{\operatorname{argmin}} \sum_i \|y_i - \hat{\beta}^T \hat{h}(x_i) - b_y\|^2$
Combined	$(\hat{W}_x, \hat{W}_h, \hat{b}_h, \hat{b}_x, \hat{\beta}, \hat{b}_y) = \underset{W_x, W_h, b_h, b_x, \beta, b_y}{\operatorname{argmin}} \sum_i \|x_i - \hat{x}(h(x_i))\|^2 + \lambda \sum_i \|y_i - \beta^T h(x_i) - b_y\|^2$

These models are not implemented here, another bias (or intercept) term may be added in some cases for x_i. The plugin and combined approaches count nearly each one two times the number of parameters of the direct approach because of the decoding step, hence they may ask for more data to train. They seem to be tested in the literature with more advanced autoencoders such as for image processing for medical images and clinical data for instance. This is because, one may have even additional data for the combination or plugin approach, by extending the vector x_i but with a part not reduced by the autoencoder. For instance, for images with additional descriptive variables or even text, another network may process this other information and the result may be merged with the reductions for the images. This idea is very convenient but the resulting loss is not obvious to minimize because several networks are in stake. They ask for different optimizations -with different learning rates and other hyperparameters- while being linked. See next section for related exercices, not solved herein.

8.5 Exercices

1. (sklearn) Implement the autoencoder with sklearn for the small dataset and improve the model by cross-validation.

2. (sklearn) For dataset mnist of the chapter, look for a better setting for the autoencoders (number of hidden layers, number of neurons for each hidden layer, activation functions and

even type of hidden layers like linear or dropout). See the documentation of sklearn (or torch.nn) for the available layers and activation functions.

3. (pytorch) Test further a sparse autoencoder and test some other architectures of autoencoder available in the literature and code repositories. Check the generalization with two samples via the two loss functions, and via the quality of the reduction for the test sample.

4. (pytorch+other) Compare the running time for alternative ways to cycle the dataset with the minibatches from mnist, consider several ones among the following examples.

   - Direct access from the image files in directories.
   - Access to the images stored in one unique hdf5 file.
   - Access to other file formats (numpy memmap, numpy npz, dask, TFRecord, pandas csv, spark).
   - Access without dataloader or with minibatches stored in separated files.

   Check: if the access can be made multi-core in parallel, and validate by looking at the the cpu cores stats during the running (command "top" with ubuntu).
   Check: eventually by profiling the code and measuring the running time for loading the minibatch from the computer hard disk to the cpu (and the gpu), with a profiler for python or pytorch.

5. (opentsne) Propose a way to improve the projection from t-sne for mnist with the random projection plus pca while keeping the data on the computer disk. For instance, several parameters need to be set, such that the dimension of the random projection.

6. (pytorch) Test the visualization of other datasets of tabular or images data, such that:

Many natively in pytorch	"https://pytorch.org/vision/0.8/datasets.html"
102 Category Flower	torchvision.datasets.Flowers102
CIFAR	torchvision.datasets.CIFAR10
CelebFaces	torchvision.datasets.CelebA
Traffic Sign Recognition	torchvision.datasets.GTSRB
MedMNIST v2 - Biomedical	"https://medmnist.com/"
MNIST-like fashion product	"https://github.com/zalandoresearch/fashion-mnist"
Stanford Dogs Dataset	"http://vision.stanford.edu/aditya86/ImageNetDogs/"

   Because pca is doing well at reduction, the purpose is here to achieve at least an equivalent result as pca at pre-processing for a k-dimensional reduction with $k = 2$ (or eventually more) for the autoencoder. Images from some datasets need to be standardized in order to compare accurately the pixels and their colors, hence testing a classification allows to check this.

7. (pytorch) Test a "Fisher" approach as in parametric likelihood models, by reducing the gradient vectors with an incremental pca at each epoch. Check the change of the first eigen vectors and the first eigenvalues during training and for different choices of the hyperparameters.

8. (pytorch) Test the combination of an autoencoder with a glm for a dataset of images completed with descriptive variables. Examples of datasets are images of objects (cars, houses, etc) plus a few descriptive variables with the price to be predicted at "kaggle.com". Another