Reminder: the course material is at : `https://gitlab.inria.fr/flandes/fpml`

1 Regressions

1.1 Linear Regression

1.1.1 The simplest case: 1−dimensional linear regression

You should already know the framework of linear regression, in the case where $x \in \mathbb{R}$ (and the ground truth is also $t_n \in \mathbb{R}$). Typically, we want to minimize the *Mean Squared Error*, i.e. the cost function is:

$$J(\Theta, X, T) = \frac{1}{N}\sum_{n}^{N}(f_\Theta(x_n) - t_n)^2 \tag{1}$$

$$\text{with the model:} \qquad f_\Theta(x) = \theta_0 + \theta_1.x \tag{2}$$

There are only 2 real parameters : $\Theta = (\theta_0, \theta_1)$. The data are the tuples: $(X, T) = (x_n, t_n), n \in [1, N]$. We denote X the set of the x_n and T the set of the t_n.

1. Compute $\vec{\nabla}_\Theta J$. You may first compute $\frac{\partial J}{\partial \theta_0}$, then compute $\frac{\partial J}{\partial \theta_1}$.
2. Derive the update rule of the parameters, assuming we perform a Gradient Descent with learning rate η.
3. Write down on *a piece of paper*[1] (not code directly) the pseudo code of the learning algorithm (initialization, updates, stopping criterion, error computation).
4. Ok, now, you may start TP1.1-LinReg-1D.ipynb
5. What is the time complexity of this algorithm ? And space complexity ?

1.1.2 The "trick of the auxiliary ones" & Matrix formulation

We did ex. 1.1.1 in a rather manual, fastidious way.

But there is a very general trick which allows to re-cast the offset (the "b" in $ax+b$) into the other parameters, making equations –and code– much simpler. The idea is the following:

$$b + ax = \begin{pmatrix} b \\ a \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \end{pmatrix} \tag{3}$$

$$= \vec{\theta} \cdot \vec{x}' \tag{4}$$

Where $\vec{x}' = \begin{pmatrix} 1 \\ x \end{pmatrix}$ is the augmented x (with auxiliary ones), and you can guess that $\vec{\theta} = \begin{pmatrix} b \\ a \end{pmatrix}$. And more generally, for a vector $\vec{x} \in \mathbb{R}^d$,

$$b + \vec{a} \cdot \vec{x} = \begin{pmatrix} b \\ a_1 \\ a_2 \\ \dots \\ a_d \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \dots \\ x_d \end{pmatrix} \tag{5}$$

$$= \vec{\theta} \cdot \vec{x}' \tag{6}$$

Where $\vec{x}' = \begin{pmatrix} 1 \\ \vec{x} \end{pmatrix}$ is the augmented x (with auxiliary ones), and you can guess that $\vec{\theta} = \begin{pmatrix} b \\ \vec{a} \end{pmatrix}$.

Why do we say "ones" and not just "a one" ? Because the idea is to augment the entire raw data X with a "1" for each data point, and then use this "new data" as your effective data (and for this new data, you don't need an offset –the "b term– in your equations). For a data set X:

$$X \in \mathbb{R}^{N,d} = \begin{pmatrix} \vec{x}_1^T \\ \vec{x}_2^T \\ \dots \\ \vec{x}_N^T \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1d} \\ x_{21} & x_{22} & \dots & x_{2d} \\ \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & \dots & x_{Nd} \end{pmatrix} \tag{7}$$

The new data set X' with the auxiliary ones is:

$$X' \in \mathbb{R}^{N,d+1} = \begin{pmatrix} \vec{x}'_1 \\ \vec{x}'_2 \\ \dots \\ \vec{x}'_N \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & x_{12} & \dots & x_{1d} \\ 1 & x_{21} & x_{22} & \dots & x_{2d} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{N1} & x_{N2} & \dots & x_{Nd} \end{pmatrix} \tag{8}$$

[1] It doesn't have to be litterally paper, you can use your tablet and stylus if you prefer.

Where we omit some of the transpose operators ($.^T$) for readability.

Questions:

1. Re-write the solution of ex. 1.1.1 using this trick. In particular, re-write the gradient of J and the update of weights for the gradient descent, referring only to the new vectors $\vec{x}'$ instead of the original $\vec{x}$.

2. Use this elegant formulation to rewrite the update step without using any explicit sum (instead, making it a purely algebraic computation, i.e. with matrices and vectors only). To help you, here is an example: If I have the sum $S = \sum_n^N p_n q_n$, I can rewrite it by defining the appropriate vctors $\vec{p} \in \mathbb{R}^N, \vec{q} \in \mathbb{R}^N$, so that $S = \vec{p} \cdot \vec{q}$. If each $p_n \in \mathbb{R}$, but e.g. $q_n \in \mathbb{R}^k$, then I will go from $\vec{S} = \sum_n p_n \vec{q_n} \in \mathbb{R}^k$ to $\vec{S} = P \cdot Q \in \mathbb{R}^k$, where $P \in \mathbb{R}^N, Q \in \mathbb{R}^{N,k}$.

3. Ideally, you may even derive the gradient directly from the matrix notation (but you need to be sure of the rules to be applied.. they are not always trivial).

Note: in python, vector/matrix operations using numpy are about 100 to 200 times faster than simple python loops. So a purely "matricial" formulation will be way faster that one with a loop ! This is called *Vectorization*.

Be careful that in python, you may have to explicit the shape of vectors like $T \in \mathbb{R}^N$ as being matrices of size $T \in \mathbb{R}^{N,1}$, which is a different kind of object. In pen and paper exercises, we typically do not make any difference between vectors $\in \mathbb{R}^N$ and single-column matrices $\in \mathbb{R}^{N,1}$.

1.1.3 d-dimensional Linear Regression

This is sometimes called multivariable regression, but who is afraid of a vectorial input? (not to be mistaken with multi-variate regression, which is different, see ex. 1.3)

From the previous exercise, it is pretty straightforward to derive the algorithm for doing a linear regression on a d-dimensional input, i.e. $\vec{x}_n \in \mathbb{R}^d$.

0. What is the new model $f_\Theta(\vec{x})$?

How many parameters are there in this model ?

1. Compute $\vec{\nabla}_\Theta J$. Try to vectorize operations: do not explicit the dot products that involve sums of length d, instead, hide them as dot products (since python is good at vectorial operations).

2. Derive the update rule of the parameters, assuming we perform a Gradient Descent with learning rate η.

3. Write down on *a piece of paper* (not code directly) the pseudo code of the learning algorithm (initialization, updates, stopping criterion, error computation).

Advice: do not use many sums. Instead, use as many matrix operations (or dot product) as you can, since this will result in **faster python code** (by an factor of approximately $100 - 200$, not negligible!)

4. Ok, now, you may start TP1.2-LinReg-d-dimensional.ipynb

5. What is the time complexity of this algorithm ? And space complexity ?

1.2 Linear Regression with a weird Loss function

Here is an excerpt from the exam of 2020: (The algo "LinReg" is defined implicitly, see below 4 for details). The point is mostly to have you compute some slightly non trivial gradients.

If you do not know yet about *regularization*, you may just set $\lambda = 0$ (i.e. ignore the λ term in the equations).

1. For some reason, someone is interested in finding the solution to the optimization problem $J(\vec{w}, \{\vec{x}_n\}_n) = \frac{1}{4}\sum_n^N (\vec{w} \cdot \vec{x}_n - y_n)^4 + \frac{1}{4}\lambda ||\vec{w}||^4$. Derive the Gradient Descent algorithm's steps. This new algorithm may be called LinReg^2 (Lin. Reg., squared). (1 pt)

2. Describe the change(s) needed in the `fit` and `predict` functions (comparing LinReg and LinReg^2). (0.25 pt)

3. We may define a sort of "MQE", mean quartic error: $MQE = \frac{1}{N}\sum_{n=1}^N ([\text{error on point n}])^4$. After all, why not ? In your opinion, which algorithm will perform best in terms of the MQE, the usual LinReg with some regularization (similar to qestion 1), or LinReg^2 (similar to question 4a)? Explain why. Ideally, you may use maths notations. (0.75 pt)

1.3 Multi-variate Regression

Let's consider an extenstion of ex. 1.1.3.

We have inputs $X = \{\vec{x}_n\}_n, \vec{x}_n \in \mathbb{R}^d$, and **now, in addition, also** $y \in \mathbb{R}^p, p > 1$. This means **we regress to several values** at once (for each data point). This is also called **multi-variate regression**, not to be mistaken with *multi-objective optimization* (which is much harder). This problem is reminiscent to that of

(sparse) **Dictionnary learning**, except here we do not impose any sparsity (but the framework is similar). See also **random projections** for some other ideas that are reminiscent of this formulation.

The fact that $d > 1$ makes it a *multivariable regression* problem (a rather *useless* vocabulary).

The fact that $p > 1$ makes it a **multi-variate regression** problem (a rather **useful** vocabulary).

Let's **assume that the input is properly standardized** or has been through some preoper pre-processing (this is very important !). We thus decide to use the Mean Squared Error as Loss function.

Preliminary question: Why is this useful to see this as a single model?

0. What is the new model $f_\Theta(\vec{x})$?

How many parameters are there in this model ?

1. Compute $\vec{\nabla}_\Theta J$. Try to vectorize operations.

2. Derive the update rule of the parameters, assuming we perform a Gradient Descent with learning rate η.

3. Write down on *a piece of paper* (not code directly) the pseudo code of the learning algorithm (initialization, updates, stopping criterion, error computation).

Use as many matrix operations (or dot product) as you can.

4. Ok, now, you may write down your own code.

5. What is the time complexity of this algorithm ? And space complexity ?

2 Non-Linear regression

2.1 Naive Polynomial regression

For simplicity, we may come back to the case of ex.1.1.1, where $x \in \mathbb{R}^1$. We now perform polynomial regression, i.e. the model is :

$$f_\Theta(x) = \sum_{p=0}^{P} \theta_p x^p \tag{9}$$

for some constant polynomial order P, e.g. $P = 3$.

How many parameters are there in this model ?

1. Compute $\vec{\nabla}_\Theta J$.

2. Derive the update rule of the parameters, assuming we perform a Gradient Descent with learning rate η.

3. Write down on *a piece of paper* (not code directly) the pseudo code of the learning algorithm (initialization, updates, stopping criterion, error computation).

4. Ok, now, you may look into .....ipynb Warning: because of powers of X, if $|X| > 1$, values can quickly overflow. Thus, it may be useful to either clip the gradient or otehrwise make sure to avoid overflow in the parameters updates.

5. What is the time complexity of this algorithm ? And space complexity ?

2.2 Feature maps

1. Re-cast the model of ex. 2.1 into a linear model, using a feature map, i.e. re-writing the inputs $x \in \mathbb{R}$ as extended features $\phi(x) \in \mathbb{R}^?$ (where you have to figure out the dimension of the feature map output).

2. Here the overflow occuring because we take powers of X acn be dealt with once and for all at the beggining. How ?

3. Assuming $x \in \mathbb{R}^2$, guess the correct feature map $\phi(\vec{x})$

4. Assuming $x \in \mathbb{R}^3$, guess the correct feature map $\phi(\vec{x})$

5. What's the limitation of this feature map ?

3 Linear regression – Algebra

We come back to the framework of ex. 1.1.1, or of ex. 1.1.3.

It is quite rare, almost unique to linear regression, but for that model, direct computation of the minimum of the cost function is actually possible! This is not commonplace in Machine Learning, so we propose this exercise because it's a classic result, but don't expect to find these kind of exact solutions in ML very often. Usually, we solve optimization problems numerically (and approximately, without unicity guarantees).

1. Recall J and its gradient. Write the gradient as a pure product of matrices, letting all sums disappear.

2. The extremum condition is $\vec{\nabla}_\Theta J = \vec{0}$. Use it to compute directly Θ^* by assuming some big matrix has an inverse.

3. Estimate the computational cost of this inverse.

4. Check that the extremum of J you found is a minimum. Explain why it is unique.

5. Adapt your calculations to regularized (Ridge) regression. (Reminder: Ridge regularization amounts to adding a term $+\lambda||\theta||_2^2$ to the cost function J.

4 Exam 2020 - Linear Regression - exercise very closely related to the Lecture's material (8 points)

1. Describe the linear regression algorithm, when it's performed without regularization, and the solution is found using Gradient Descent (not the exact analytical solution that can sometimes be achieved).
   You will start by describing the kind of task that linear regression is made for, then define the cost function to be minimized (use least squares), the optimization algorithm that is used, and then write a very short pseudo-code of what the `fit` function may look like. You will also provide the corresponding `predict` function. (2 pts)

2. Let's compute an update of Linear Regression. Let's suppose the data is one dimensional: $X = (1, 1, 2, 2)$, $y = (1, 2, 2, 3)$. The initial weight vector is $w = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. Advice: write X in a matrix form. Use the learning rate $\eta = 1$. Hint: if this seems long to you, keep this computation for later (it's ok to refer to a point later in your paper). (1.5 pt)

3. If the relationship between input $\vec{x}$ and output y is not linear but has some dependence on products of the input features $x_1, x_2, \ldots$, what can we do? (0.5 pt)

4. First variation on Linear Regression. (forget the data of question 2, we go back to the general case).

   (a) For some reason, someone is interested in finding the solution to the optimization problem $J(\vec{w}, \{\vec{x}_n\}_n) = \frac{1}{4}\sum_n^N (\vec{w} \cdot \vec{x}_n - y_n)^4 + \frac{1}{4}\lambda||\vec{w}||^4$. Derive the Gradient Descent algorithm's steps. This new algorithm may be called LinReg^2 (Lin. Reg., squared). (1 pt)

   (b) Describe the change(s) needed in the `fit` and `predict` functions (comparing LinReg and LinReg^2). (0.25 pt)

   (c) We may define a sort of "MQE", mean quartic error: $MQE = \frac{1}{N}\sum_{n=1}^N([\text{error on point n}])^4$. After all, why not ? In your opinion, which algorithm will perform best in terms of the MQE, the usual LinReg with some regularization (similar to qestion 1), or LinReg^2 (similar to question 4a)? Explain why. Ideally, you may use maths notations. (0.75 pt)

5. Second variation: mini-batches. Let's forget about question 4. We recall the idea of the mini-batches. The idea is that instead of optimizing the error for all examples (training data samples) at once, we can use only a subset of them (chosen at random, or cycling through the data, for instance first using examples 1-100, then 101-201, etc).

   (a) Describe what this means, for a `fit` function, in terms of maths and pseudo-code. You may assume that the shape of $\vec{\nabla}_\Theta J$ is known and so you may use the term $\vec{\nabla}_\Theta J$ generically. (1 pt)

   (b) In practice, for complex models (e.g. for Deep Networks), it is often observed that mini-batch GD leads to better generalization than full batch GD. Can you explain intuitively why ? Hint: imagine that your training data is of good quality, but that the input space is so big that there is still "randomness" in the data, in that your training set is not very similar to the test set, simply because of the "sampling noise" (the kind of noise that makes it so that when you flip a coin 10 times, typically it does not give 5 heads and 5 tails). (0.75 pt)

   (c) Cite another potential advantage of using mini-batches instead of the "full batch" GD. (0.25 pt)