**'Data Analysis'**
**Computer Lab Session**
**Linear Regression**

——————————

**Master 1 MLDM / CPS$^2$ / COSI / 3DMT**
**Saint-Étienne, France**

Ievgen Redko

# 1 Outline

**Outcome**

The objective of this lab is to become familiar with Python functions for working with linear regression.

# 2 Introduction

Regression analysis is used for explaining or modeling the relationship between a single variable $Y$, called the *response*, *output* or *dependent* variable, and one or more *predictor*, *input*, *independent* or *explanatory* variables, $X_1, X_2, \ldots X_p$. The case of one explanatory variable ($p = 1$) is called simple linear regression. For more than one explanatory variable ($p > 1$), it is called multiple linear regression.

We use regression to estimate the unknown effect of changing one variable over another. When running a regression we are making two assumptions:

1. there is a linear relationship between two variables (i.e., $X$ and $Y$)

2. this relationship is additive (i.e., $Y = x_1 + x_2 + \cdots + x_N$).

A regression with two or more explanatory variables is called a multiple regression. Rather than modeling the mean response as a straight line, as in simple regression, it is now modeled as a function of several explanatory variables.

The module LinearRegression() from sklearn.linear_model library can be used to perform linear regression in Python as follows

```
model = LinearRegression ()
model . fit ( explanatory_vars , response )
```

Here the terms response and explanatory_vars in the function should be replaced by the names of the response and explanatory variables matrix, respectively, used in the analysis.

## Exercise 1: Linear Regression on Eucalyptus Dataset

Consider the following example: we are trying to estimate the height of eucalyptus trees based on their circumference (it is easier to measure the trunk of a tree than its height).

1. Import the eucalyptus dataset, print the summary of the dataset and plot the height (variable ht in meters) as a function of the circumference (variable circ in cm).

2. Compute then the linear regression by using the LinearRegression() function.

3. To examine the quality of the model and observations, plot the observations and the fitted line where the Y values are the predictions of the observation obtained using predict function.

4. Draw a confidence interval of 95% for the prediction. To do this, calculate the standard deviation of the errors as follows:
$$\text{stdev} = \sqrt{\frac{1}{n-2} * \text{sum\_errs}}$$
where $\text{sum\_errs} = \sum_{i=1}^{n}(y_i - y_{\text{pred}_i})^2$.

5. Fit a multiple linear regression model $\text{ht} = \beta_0 + \beta_1 \times \text{circ} + \beta_2 \times \sqrt{\text{circ}}$.

6. Compare the two methods by analyzing (a) the residuals of each model and (b) the $R^2$ coefficient of determination that can be calculated using the score() function of the model.

## Exercise 2: Multiple Linear Regressions on Ozone Dataset

Ozone is an inorganic compound with the chemical formula $O_3$. Ozone precursors are a group of pollutants, predominantly those emitted during the combustion of fossil fuels. Exposure to ozone and the pollutants that produce it is linked to premature death, asthma, bronchitis, heart attack, and other cardiopulmonary problems. Air quality guidelines such as those from the World Health Organization, the United States Environmental Protection Agency (EPA) and the European Union are based on detailed studies designed to identify the levels that can cause measurable ill health effects. According to scientists with the US EPA, susceptible people can be adversely affected by ozone levels as low as 40 nmol/mol. In the EU, the current target value for ozone concentrations is 120 $\mu$g/m$^3$ which is about 60 nmol/mol.

The file ozone.txt concerns the maximal concentration of ozone in the air recorded at Rennes (in Brittany, France) for each day during the summer 2001 ("maxO3") and the day before ("maxO3v"), and different measures of the weather: temperature (T), cloudiness (Ne for « nébulosité ») and wind speed (Vx) at 9:00 AM, 12:00 AM and 3:00 PM (9, 12, 15), maximal wind direction (wind), presence of rain or not (rain).

1. Import ozone dataset, plot the pairwise representation between the 11 first numerical variables and print the correlation coefficients between the 11 first numerical variables.

2. With which variable the maximal ozone concentration (maxO3) is the most correlated? Use Pandas' DataFrame.corr() function to find it out.

3. Plot maxO3 in function of this variable.

4. Compute the simple linear model of maxO3 in function of this variable, then trace the line obtained by this model on the graph.

5. Consider now all numerical variable (except maxO3) and compute the multiple linear regression of max03 as function of these variables.

6. Compare the obtained results.

## Exercise 3: Gradient Descent and Closed-Form Solution

Let us consider the following least squares problem

$$\theta = \arg\min_{\theta} J(\theta) = \arg\min_{\theta} \|X\theta - y\|_2.$$

From the lecture, you know that the closed-form solution (a solution expressed analytically in terms of a finite number of certain "well-known" functions) is given by:

$$\theta = (X^T X)^{-1} X^T y.$$

While this solution can be very handy, in some cases one may need a more memory efficient way of finding $\theta$. For instance, imagine that $X$ is a very large but sparse matrix. e.g. $X$ might have 100,000 columns and 1,000,000 rows, but only 0.001% of the entries in $X$ are nonzero. There are specialized data structures for storing only the nonzero entries of such sparse matrices. Also imagine that we're unlucky, and $X^T X$ is a fairly dense matrix with a much higher percentage of nonzero entries. Storing a dense 100,000 by 100,000 element $X^T X$ matrix would then require $10^{10}$ floating point numbers (at 8 bytes per number, this comes to 80 gigabytes.) This would be impractical to store on anything but a supercomputer. Furthermore, the inverse of this matrix (or more commonly a Cholesky factor) would also tend to have mostly nonzero entries.

In such cases, one resorts to using the iterative methods such as, for instance, the gradient method presented below.

$$
\begin{aligned}
&k \leftarrow 0 \\
&\alpha \leftarrow \text{step size} \\
&sol_k \leftarrow \text{first guess of the solution} \\
&\textbf{while } k < n_{\text{iter}} \text{ do} \\
&\qquad sol_{k+1} \leftarrow sol_k - \alpha \nabla_\theta J(\theta) \\
&\qquad \textbf{if } |sol_{k+1} - sol_k| < eps: \\
&\qquad\qquad \textbf{break} \\
&\qquad k \leftarrow k + 1 \\
&\textbf{return } sol_{k+1}
\end{aligned}
$$

1. Let
$$J(\theta) = 1.2 * (\theta - 2)^2 + 3.2.$$

Calculate the derivative of $J(\theta)$ and deduce the analytic solution of $\arg\min_{\theta} J(\theta)$.

2. Implement the gradient descent function that solves $\arg\min_{\theta} J(\theta)$. Try different values of $\alpha$ and plot the convergence of the iterative procedure showing the path of the gradient descent.
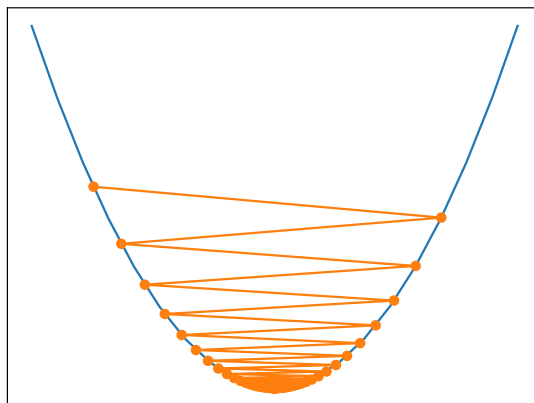
Fig. 1 – *Function $J(\theta) = 1.2 * (\theta - 2)^2 + 3.2.$ in blue and the gradient descent in orange when $\alpha = 0.8$.*

## Exercise 4: Newton's Method

Let's assume that we have a function $f(\theta)$. We aim at finding a value of $\theta$ such that $f(\theta) = 0$. In the Newton's method (seen in the lecture), one iteration updates $\theta^t$ as follows:

$$\theta^{t+1} = \theta^t - \frac{f(\theta^t)}{f'(\theta^t)}$$

When $\theta$ is a feature vector, we get:

$$\theta^{t+1} = \theta^t - H^{-1}\nabla_\theta \ell$$

where:

- $H$ is the Hessian matrix,

- $\nabla_\theta \ell$ is the gradient.

The idea of Newton's method is as follows: one starts with an initial guess which is reasonably close to the true root, then the function is approximated by its tangent line (which can be computed using the tools of calculus), and one computes the $x$-intercept of this tangent line (which is easily done with elementary algebra). This $x$-intercept will typically be a better approximation to the function's root than the original guess, and the method can be iterated.

Suppose $f : [a, b] \longrightarrow \mathbf{R}$ is a differentiable function defined on the interval $[a, b]$ with values in the real numbers $\mathbf{R}$. The formula for converging on the root can be easily derived. Suppose we have some current approximation $x_n$. Then we can derive the formula for a better approximation. The equation of the tangent line to the curve $y = f(x)$ at the point $x = x_n$ is $y = f'(x_n)(x - x_n) + f(x_n)$, where $f'$ denotes the derivative of the function $f$.

The $x$-intercept of this line (the value of $x$ such that $y = 0$) is then used as the next approximation to the root, $x_{n+1}$. In other words, setting $y$ to zero and $x$ to $x_{n+1}$ gives $0 = f'(x_n)(x_{n+1} - x_n) + f(x_n)$.

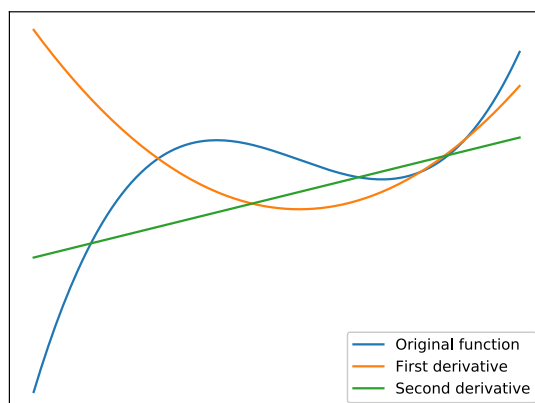Solving for $x_{n+1}$ gives

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

FIG. 2 – *Function* $f(x) = 5x^3 - 7x^2 - 40x + 100$ *and its derivatives.*

With Python, it is easy to calculate the derivatives for some special classes of functions. For instance, one can define the polynomial function $5x^3 - 7x^2 - 40x + 100$ using the function numpy.poly1d() and obtain the derivatives with the deriv() function.

After that, we can evaluate the values for these functions and plot them on the same graph (see Figure 2. We expect that when the initial function decreases, the derivative has negative values, and when the positive function increases, the derivative has positive values.

1. Implement the Newton's method as a function that takes as input:

   • symbolic representation of the objective function $f$

   **Remark 1** *Use* lambda *function to define a function symbolically. For instance, you can define a function* $f = \sqrt{x}$ *by writing* f = lambda x: np.sqrt(x) *and calling it using* $f(x)$ *for some variable* $x$.

   • its derivation $Df$

   • Initial point $x0$,

   • Precision $\epsilon$ and the maximum number of iterations $max_iter$

2. Run your code to find the solution of the following equations:

   • $x^2 - 4 = 0$

   • $5 * x^3 - 7 * x^2 - 40 * x + 100 = 0$

   • $\exp(-x)x = 0$

   • $x^{\frac{1}{3}} = 0$

3. Conclude based on the obtained results.

$\hookrightarrow$ *Need to do*

# Exercise 5: Regularized linear models

As explained in the related lecture, one may consider other candidate methods for solving a regression problem such as regularized Lasso and Ridge linear models. In Python, the implementation of these methods are available as part of the sklearn.linear_model modules.

1. Define a function $f(x) = \cos(1.5\pi x)$. Use this function to generate $n$ random observations. Add Gaussian noise $\mathcal{N}(0, 0.3)$ to these observations using np.random.randn() function.

2. Fit a linear regression model on this data set with polynomial features of orders 1 (usual linear model), 4 and 15. These latter can be defined using the sklearn.preprocessing.PolynomialFeatures function. Plot the obtained results. What particular phenomenon do you observe?

3. Change the used model to Lasso. Re-run the experiment and find the appropriate value of the regularization parameter to obtain the desired fit. Plot in the title the number of non-zero coefficients retained by the model. What are your conclusions?

4. Do the same with the Ridge model. Is the number of non-zero coefficients remains the same as in the previous case? If not, explain why.

5. Add a comparison with SVR model from sklearn.svm module. Use linear kernel and try different values of $C$ and $\epsilon$ parameters. Is this method prone to overfitting?

$$f = \min_{\beta} \|y - X\beta\|^2 + \lambda \|\beta\|^2$$

$$\frac{\partial f}{\partial \beta} = -2X^T(y - X\beta) + 2\lambda\beta = 0$$

$$\Rightarrow \lambda\beta = X^T y - X^T X \beta$$

$$\Rightarrow \beta(X^T X + \lambda I) = X^T y$$

$$\Rightarrow \beta = (X^T X + \lambda I)^{-1} X^T y$$