LELEC2870: Practical Sessions

October 6, 2020

Notations and vocabulary

- a scalar: A number represented with an upper-case letter when defining bounds on a series, or a lower-case one otherwise e.g. x in $1..X$
- a vector: A 1-dimensional array that will always be written with a **bold lower-case letter**
- a matrix: A 2-dimensional array that will always be written with a **bold upper-case letter**
- a tensor: A N-dimensional array that will always be written with a **bold upper-case letter**. While it uses the same notation as the matrix, it should always be clear which one is meant.
- an iteration: A time-measure for *iterative* algorithms. It denotes a pass over a part of the training dataset.
- an epoch: A time-measure for *iterative* algorithms. It denotes *one* pass over the complete training dataset. An epoch is composed of at least 1 iteration, but most of the time of multiple iterations especially when large datasets are involved.

1 Session 1

Objectives

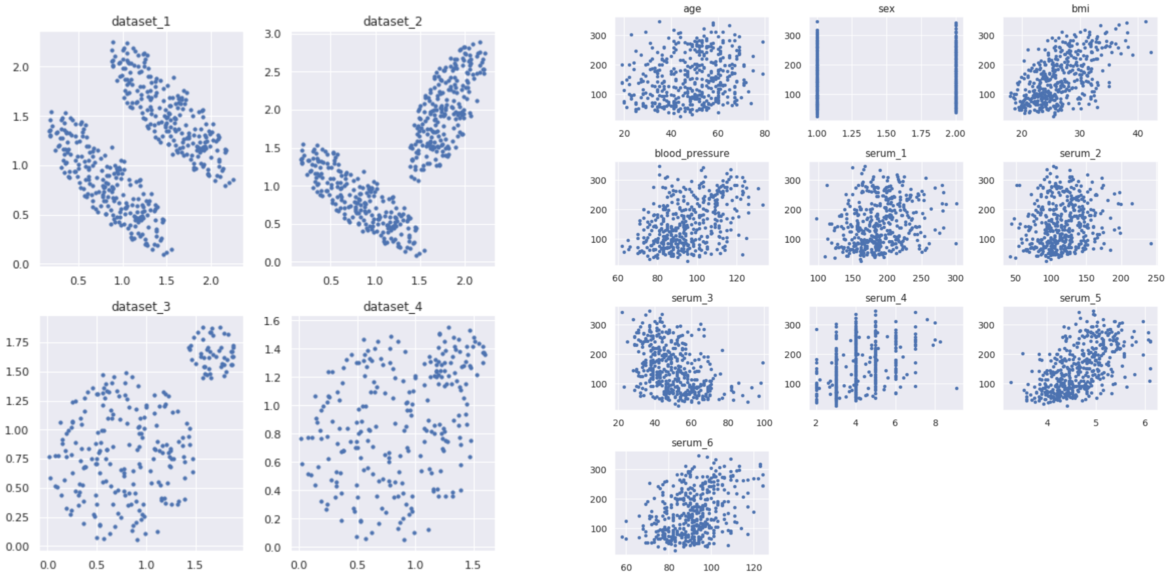
In this first exercise session, you will implement 2 variations on vector quantization techniques (K-means and frequency sensitive learning) as well as linear regression.
We provide you two datasets for the two topics covered in this session. Those datasets can be found on the Moodle page of this course.

BEFORE the session

Load the first two datasets and **plot** them.

1. For dataset 1 (**VQ**), you will see that groups of points are clearly visible, and your objective in this session will be to learn these groups automatically. There are 4 (synthetic) datasets with 2 features every time.

2. For dataset 2 (**LR**), your objective will be to learn the linear relation between the target and the inputs. The target variable ('t' in the dataframe) represent the blood sugar levels. The features ('X' in the dataframe) represent: the age (feature 1), the sex (feature 2), the body mass index (feature 3) and the blood pressure (feature 4) of the patient, as well as the result of several serum measurements (features 5 to 10)



(a) data1

(b) data2

Figure 1: (a) The scatterplots of the datasets for the clustering exercise (b) The blood sugar levels in function of the different features for the linear regression exercise

- **Vetor Quantization (VQ)**. It consists in reducing the size of a dataset while minimizing the loss of information, by summarising it with a set of centroids. An archive containing all necessary data and and a python script for visualization are on the Moodle website.
- **Linear Regression (LR)**. Linear methods often allow to obtain good results in regression, when the relationship between the features (or inputs) and the target (or output) is not too non-linear. You will implement linear regression for a set of features that you will select using simple linear methods.

1.1 Vector Quantization

Vector Quantization Notation

Consider a set of P vectors $\mathbf{X} = \{\mathbf{x}_p \mid p = 1...P\}$ in a D-dimensional space. A vector quantization consists of a set of Q D-dimensional vectors $\mathbf{Y} = \{\mathbf{y}_q \mid q = 1...Q\}$ minimizing an error criterion linked to the loss of information. Here, Q is typically chosen to be much smaller than P, since the goal of vector quantization is to summarise a large number of instances using only a few vectors. These vectors are called *centroids* and the set of the centroids forms the *codebook*.

We also define $\mathcal{S}_q$ to be the set of points assigned to a particular centroid $\mathbf{y}_q$ with $\mathcal{S}_q \subseteq \mathbf{X}$. This implies that $\forall \mathbf{x}_p \in \mathbf{X}, \exists! \mathbf{y}_q \in \mathbf{Y}, \forall r \in \{1..Q\} \setminus \{q\}$ s.t. :

$$\mathbf{x}_p \in \mathcal{S}_q \Leftrightarrow \|\mathbf{y}_q - \mathbf{x}_p\| \leq \|\mathbf{y}_r - \mathbf{x}_p\|.$$

In case of a point equidistant from 2 centroids, its cluster-assignment is random.

The error criterion is generally defined in terms of the distance between the data vectors and the centroids. Usually, it corresponds to the mean squared Euclidean distance between the data points and the centroids

$$E = \frac{1}{P} \sum_{q=1}^{Q} \sum_{\mathbf{x}_p \in \mathcal{S}_q} \|\mathbf{x}_p - \mathbf{y}_q\|^2 \tag{1}$$

During this exercise session, it may be interesting to **monitor** how this objective function changes over iterations and to compare quantization results in terms of this quantity.

1.1.1 Competitive Learning

Competitive learning is a vector quantization algorithm of the *winner-take-all* family. This means that at each presentation of a data vector $\mathbf{x}_p$, only the *winner* centroid is adapted. The centroid $\mathbf{y}_q$ is selected as a winner $\mathbf{y}_{q^*}$ if it is the closest to $\mathbf{x}_p$, with respect to the distance measure d. In other words,

$$\mathbf{y}_{q^*} \mid \forall r \in \{1..Q\} : \ d\left(\mathbf{x}_p, \mathbf{y}_{q^*}\right) \leq d\left(\mathbf{x}_p, \mathbf{y}_r\right). \tag{2}$$

The *adaptation rule* of the codebook is defined as

$$\mathbf{y}_{q^*} \leftarrow \mathbf{y}_{q^*} + \alpha_k \left(\mathbf{x}_p - \mathbf{y}_{q^*}\right) \tag{3}$$

where $\alpha_k > 0$ is a factor that controls the speed of the adaptation. In order to obtain convergence, the α_k factor has to decrease over time to end its run close to zero. Time is often described in number of *epochs* when using iterative algorithms. One *epoch* is one *pass* over every observation in the database.
A common method to adapt α after every epoch, is the hyperbolic decrease:

$$\alpha_{k+1} \leftarrow \frac{\alpha_k \beta}{\alpha_k + \beta} \tag{4}$$

where β is a constant. Generally, it is better to **randomize** the order of the observations in the database after each epoch.

We can write all this into the following pseudo-code or algorithmic format:

Algorithm 1 Competitive Learning

```
Y ← random_init()
for epoch in 1..n_epochs do
  prev_Y ← copy(Y)
  X ← shuffle(X)
  for xp in X do
    index ← argmin$\|xp - Y\|$
    yq ← Y[index]
    Y[index] ← yq + $\alpha \cdot$ (xp - yq)
  end for
  $\alpha \leftarrow \frac{\alpha\beta}{\alpha + \beta}$
  if no_change(prev_Y, Y) then
    break
  end if
end for
```

Implement competitive learning with 2 codebook initialisations:

- Select the centroids randomly (tip: remain inside the borders of the data)
- Sample the centroids randomly from the observations.

Visualize the vector quantizations using the *show_quantization* function provided in the archive. What is the problem with the first initialisation? Does it occur with the second one?

1.1.2 Frequency Sensitive Learning

Frequency sensitive learning (FSL) is also a *winner-take-all* algorithm. However, by introducing a penalisation on the centroids that often win, the problem of lost units is avoided. One centroid is *lost* when it is never chosen as the winner.

The selection rule for the winner centroid is replaced by

$$\mathbf{y}_{q^*} \mid \forall r \in \{1..Q\} : \; u_{q^*} d\left(\mathbf{x}_p, \mathbf{y}_{q^*}\right) \leq u_r d\left(\mathbf{x}_p, \mathbf{y}_r\right) \tag{5}$$

where u_q is related to the frequency of selection of $\mathbf{y}_q$. This factor u_q is initially equal to 1 and is incremented each time that the centroid $\mathbf{y}_q$ is selected as the winner. The adaptation rule remains

$$\mathbf{y}_{q^*} \leftarrow \mathbf{y}_{q^*} + \alpha_k \left(\mathbf{x}_p - \mathbf{y}_{q^*}\right) \tag{6}$$

Implement FSL. **Compare** the two types of codebook initialisation again. **Check** that the lost centroids problem is solved. Why is it so?

1.1.3 K-means

Finally we will address K-means, an algorithm with the same properties as Lloyd's (see lectures). It processes the P vectors all at once at *each* iteration, contrarely to the 2 previously seen quantization algorithms that iterate over the input vectors individually.

Each iteration goes as follows:

1. For each item in the dataset, find the nearest centroid (i.e. assign a label to each item) by using the Euclidean Distance.
2. Replace each centroid with the center of mass of the group labelled to it (the mean coordinates of all items assigned to this centroid).

Translating the centroid update into mathematical notation, we obtain:

$$\forall q \in 1..Q : \mathbf{y}_q \leftarrow \frac{1}{|\mathcal{S}_q|} \sum_{\mathbf{x}_p \in \mathcal{S}_q} \mathbf{x}_p \tag{7}$$

Implement K-means. What is the problem with the first initialisation (run your program several times to discover it)?
Another way of initializing K-means was proposed, K-means++. The initialization is described as follows:

1. Select a data-sample at random for the first centroid
2. Select a data-sample probabilistically based on its distance to its closest centroid.
3. Repeat step 2 until Q centroids are selected

Implement K-means++ algorithm! How does it address K-means' problem and why does it enhance the results?

1.1.4 Neural Gas (@Home)

Unlike the other algorithms, neural gas is a *winner-take-most* algorithm, which means that all the centroids are adapted at each presentation of a vector $\mathbf{x}_p$. The adaptation depends on the distance between the centroids and $\mathbf{x}_p$.

The first step of the algorithm consists in ranking the centroids in function of their distance (euclidean or other) to $\mathbf{x}_p$. Thus, one can define the neighbouring function h such that, if $\mathbf{y}_q$ is the c_{th} closest centroid to $\mathbf{x}_p$, then

$$h\left(\mathbf{x}_p, \mathbf{y}_q\right) = c - 1. \tag{8}$$

For example, this quantity becomes zero for the closest centroid ($c = 1$).

The second step consists in adapting each centroid using the adaptation rule

$$\forall q \in \{1..Q\} : \; \mathbf{y}_q \leftarrow \mathbf{y}_q + \alpha_k \exp\left(-\frac{h\left(\mathbf{x}_p, \mathbf{y}_q\right)}{\lambda}\right)\left(\mathbf{x}_p - \mathbf{y}_q\right) \tag{9}$$

where λ regulates the amount of neighbours which are affected by the update. It can decrease with the number of epochs.

Implement the neural gas algorithm. **Explain** the influence of the λ parameter. What happens when $\lambda = 0$? What happens when $\lambda \to +\infty$?

Compare a *winner-take-most* approach to a *winner-take-all* approach ? What are the advantages of the former?

1.2 Linear Regression

Linear Regression Notations

The output prediction of a linear regressor for an element p can then be written as follows:

$$t_p = w_p^0 + w_1.x_p^1 + w_2.x_p^2 + ... + w_D.x_p^D \tag{10}$$

with D the number of dimensions of input vector $\mathbf{x}$, $w_1...w_D$ the weights of each feature/dimension, and w_0 the independent term (=bias).

A Linear regressor can be equivalently described as a simple neural network with only one layer and a linear activation. The analytical expression of its output is

$$t_p = \sigma(\mathbf{x}_p\mathbf{w} + w_0). \tag{11}$$

where $\mathbf{w}$ is the weights column vector, w_0 is the bias, σ is the activation function and t_p is the output of the network. In order to simplify notations, the bias is generally included in the inputs and weights vectors, i.e. $\mathbf{x}_p$ becomes

$$\begin{pmatrix} 1 & \mathbf{x}_p \end{pmatrix}$$

and $\mathbf{w}$ becomes

$$\begin{pmatrix} w_0 \\ \mathbf{w} \end{pmatrix}.$$

This convention is used in the rest of this exercise session. Since the activation is linear, the output of one-layer neural networks becomes simply

$$t_p = \mathbf{x}_p\mathbf{w}. \tag{12}$$

Notice that now, $\mathbf{x}_p$ and $\mathbf{w}$ are both vectors of length $(D + 1)$.
We can extend this to vectorial outputs (of length T) in the following way

$$\mathbf{t} = \mathbf{X}\mathbf{w}, \tag{13}$$

with the shape of $\mathbf{X}$ being $T \times (D + 1)$

1.2.1 Feature Selection with Correlation

Since 10 features are available in the provided dataset, we ask you to select one (or more) of them to perform regression on in the following exercises. An example of a good selection criterion is the *correlation* between the feature itself and the target.

For each feature, **visualize** the relationship between the feature and the target `scatterplot` (`matplotlib`) and **compute** the correlation `corrcoef` (`numpy`). Can you use correlation to select the variables used by a linear model?

1.2.2 Univariate Linear Regression

Pseudo-Inverse method

We can present our dataset of P datapoints with $\mathbf{t}$ the vector of the targets of length N and $\mathbf{X}$ of shape $P \times (D+1)$ (the $+1$ is there to accommodate for the bias). We want to find the vector of weights $\mathbf{w}$ that minimizes the Mean Squared Error (MSE) criterion. Leading to the following equation

$$
\begin{aligned}
E &= \frac{1}{P}\|\mathbf{t}-\mathbf{X}\mathbf{w}\|^2 && (14)\\
\Leftrightarrow \quad \left(\frac{\partial E}{\partial \mathbf{w}}\right) &= \frac{2}{P}\left(\mathbf{t}-\mathbf{X}\mathbf{w}\right)\mathbf{X} = \mathbf{0} && (15)\\
\Leftrightarrow \quad \mathbf{0} &= \mathbf{X}^T\mathbf{t} - \mathbf{X}^T\mathbf{X}\mathbf{w} && (16)\\
\Leftrightarrow \quad \mathbf{X}^T\mathbf{X}\mathbf{w} &= \mathbf{X}^T\mathbf{t} && (17)\\
\Leftrightarrow \quad \mathbf{w} &= \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{t}. && (18)
\end{aligned}
$$

This is different from the course's definition due to the transposed way the matrices are defined.

For this first regression you'll have to **choose** one feature. Once you have selected it (based on your results from previous section), you can perform linear regression. **Implement** linear regression using the pseudo-inverse method `inv` (`numpy.linalg`). **Visualize** the result of your regression. In particular, compare your prediction with the actual target values. Are the results satisfactory ? How could you improve it?