Logistic regression

Logistic regression is used for binary classification.
It is quite similar to a simple linear regression in the sense that the objective is to find optimal weights ω to predict a variable. However, in the logistic regression we use a sigmoïd function.

Rem: "logistic" because the logistic law has a sigmoïd function as a repartition function.

Rationale behind the use of the sigmoïd function:
We look for the *à posteriori* probability $\mathbb{P}(y = 1|x) = \pi(x) = \hat{y}$.
The predicted variable $\hat{y}$ is thus a probability.

The sigmoïd function: $\sigma : z \to \frac{1}{1+e^{-z}}$ is well adapted because of two reasons:
1) We want an output variable that is included in $[0, 1]$
2) $\frac{\pi(z)}{1-\pi(z)}$ represents the relationship between a distribution and its complementary (good in binary case), and it is just a transformation of $\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1-e^z}$

Thus, we have:
$\hat{y} = \mathbb{P}(y = 1|x) = \sigma(\omega^T x + b) = \frac{1}{1-e^{-(\omega^T x+b)}}$

Estimation
Estimation is done using maximum likelihood. Maximum likelihood is finding the parameter that maximizes the probability to have a specific event (x_i, y_i) but in our case, it is a *conditional* maximum likelihood since we want to maximize the *à posteriori* probability that depends on x.

$L(\omega, b) = \prod_{i=1}^{n} \pi(x_i)^{y_i}(1 - \pi(x_i))^{1-y_i}$

This equation has no analytic solution. We use a numeric method to find the optimal parameters (see optimizaton algorithms).

Expectation-Maximization (EM) in the case of GMM (Gaussian Mixture Model)

(for more details, see document *gmm.pdf* in Cloud folder)

A GMM sample is composed of j Gaussian variables (*clusters*) distributed with proportions $(\pi_1, ..., \pi_k)$ $(\Sigma\pi_i = 1)$
We can write:

$$X \sim \mathcal{N}(\mu_Z, \Sigma_Z) \qquad with \;\; Z \sim \pi$$

π is not really a law but more the proportions of each Gaussian categories.
Thus, X has a density which is a weighted-average of all Gaussian densities:

$$p_\theta(x) = \Sigma_{j=1}^{k} \pi_j f_j(x) \qquad (*)$$

Estimation
We want to estimate $\theta = (\pi, \mu, \Sigma)$ where:

$\pi = (\pi_1, ..., \pi_k)$, $\mu = (\mu_1, ..., \mu_k)$, $\Sigma = (\Sigma_1, ..., \Sigma_k)$

To do so, we use the maximum likelihood method (product of densities across all samples):

$$p_\theta(x) = \Pi_{i=1}^n p_\theta(x_i)$$

$$l(\theta) = log(\Pi_{i=1}^n p_\theta(x_i)) = \Sigma_{i=1}^n log(p_\theta(x_i))$$

We thus need to find $argmax(l(\theta))$
Problem: the likelihood function is not convex!

The expectation-maximization problem is used when we have *latent variables* (= variables for which we don't know their associated distribution).
Let $z = (z_1, ..., z_k)$ be the vector of latent variables. We can express the density $(*)$ as a joint function with respect to z:

$$p_\theta(x, z) = p_\theta(z)p_\theta(x|z)$$

$$l(\theta, z) = ... = \Sigma(log\pi_{z_i}) + \Sigma(logf_{z_i}(x_i))$$

A classic optimization (in case of Gaussians) give us empirical values as solutions e.g. $\hat{\pi_j} = \frac{n_j}{n}$
Problem: we don't know j!

We will thus use the *expected* log-likelihood method.
Let us find another expression of the likelihood:

$$p_\theta(x, z) = p_\theta(x)p_\theta(z|x)$$

As seen previously: $p_\theta(x, z) = \Pi\pi_{z_i} f_{z_i}(x_i)$
$p_\theta(z|x) = \Pi p_\theta(z_i|x_i) = \frac{\Pi\pi_{z_i} f z_i(x_i)}{p_\theta(x_i)} \propto \Pi\pi_{z_i} f z_i(x_i)$

Given an initial parameter θ_0, the *expected* log-likelihood is written as such:

$$\mathbb{E}_{\theta_0}[l(\theta; z)] = \Sigma p_{\theta_0}(z|x) l(\theta; z)$$
$$\mathbb{E}_{\theta_0}[l(\theta; z)] = \Sigma_j \Sigma_i p_{ij}(log\pi_j + log f_j(x_i))$$

We now have an expression that doesn't depend on z but only on p_{ij} and we know that $n_j = \Sigma_i p_{ij}$

K-means

Objective: group data into k clusters so that samples in the same cluster are close to each other w.r.t. the Euclidean distance.

The cost function minimization is written as such:

$$\underset{C_1,...,C_k;\mu_1,...,\mu_k}{\operatorname{argmin}} \Sigma_{j=1}^k \Sigma_{i \in C_j} ||x_i - \mu_j||^2$$

Where μ_j is the mean, also called gravity center or cluster center:

$$\mu_j = \frac{1}{|C_j|}\Sigma_{i \in C_j} x_i$$

Algorithm 1 K-means

Input: $x_1, ..., x_n$
Output: clusters $C_1, ..., C_k$ Random values for $\mu_1, ..., \mu_k$
while no convergence **do**
  // Step 1: Update clusters
  $C_1, ..., C_k \leftarrow \emptyset$
  for $i = 1$ to n **do**
    $j \leftarrow argmin_l ||x_i - \mu_l||$
    $C_j \leftarrow C_j + \{i\}$ // We add observation i to the cluster C_j
  end for
  // Step 2: Update cluster centers
  for $j = 1$ to k **do**
    $\mu_j \leftarrow 0$
    $n_j \leftarrow 0$
    for i in C_j **do** // We loop on all observations of each cluster
      $\mu_j \leftarrow \mu_j + x_i$
      $n_j \leftarrow n_j + 1$
    end for
    $\mu_j \leftarrow \mu_j / n_j$
  end for
end while

The quality of the clustering strongly depends on the initial center values. This is why the algorithm is generally run multiple times for different initial values. The best clustering (i.e., that of minimum cost) is returned.

K-means++

To improve the quality of the clustering, we choose the initial cluster centers far from each other:
- select the first cluster center uniformly at random among the n data samples
- select the following cluster centers at random, with a probability proportional to the square distance to the closest current cluster center

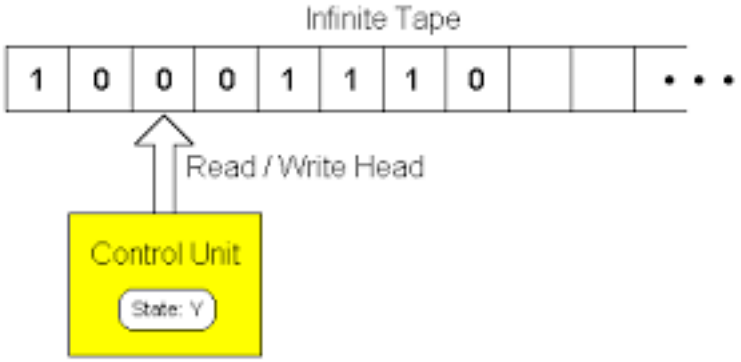
Listing 1: K-means++ initial centers selection

```
centers = []
centers.append(X[np.random.randint(X.shape[0])]) # inital center = one random sample
distance = np.full(X.shape[0], np.inf) # a vector (n,1) with only infinity terms
for j in range(1, self.n_clusters):
    distance = np.minimum(np.linalg.norm(X - centers[-1], axis=1), distance) # size (n,1);
# distance = the smallest distance associated with
# the last added center
    p = np.square(distance) / np.sum(np.square(distance)) # probability vector [p1,...,pn]
# the highest probability in p is associated
# with the biggest distance w.r.t the last added center
    sample = np.random.choice(X.shape[0], p = p) # one sample is
                                                 # selected according to probabilities
```

```
centers.append(X[sample])
```

Note: this problem is called *NP-hard problem*. It means that its complexity is at least equal to the complexity of an NP-problem
NP-problem: a problem is NP if it can be determined by a non-deterministic Turing machine in polynomial time. Intuitively, a problem is NP if we can quickly verify if one candidate is a solution of the problem. E.g. "travelling salesman problem" = let d be a distance and n be a number of cities. Is there an itinerary with distance $\geq d$ stopping by every city? -> easy to check...

Turing machine (1936)



"non-deterministic turing machine": itinerary can be represented by a tree...