

Learning theory for classification

g is the *classifier*.

$$\begin{aligned} g : \mathcal{X} &\rightarrow \mathcal{Y} \\ \mathbb{R}^d &\rightarrow \{0, 1\} \end{aligned}$$

To model the learning problem, we use the pair (X, Y) described by (μ, η) where μ is the probability measure:

$$\mu(A) = \mathbb{P}(X \in A)$$

And η is the regression of Y on X :

$$\eta(X) = \mathbb{P}(Y = 1|X = x) = \mathbb{E}[Y|X = x]$$

η is also called the *a posteriori probability*.

The Bayes classifier is:

$$\begin{cases} 1 & \text{if } \eta(x) > 1/2 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Or, if \mathcal{Y} is $\{-1, 1\}$, we write the classifier as such: $g(x) = 2\mathbb{1}\{\eta(x) > 1/2\} - 1$.

Theorem:

For any classifier $g: \mathbb{R}^d \rightarrow \{0, 1\}$,

$$\mathbb{P}(g^*(X) \neq Y) \leq \mathbb{P}(g(X) \neq Y)$$

In other words, the Bayes classifier is theoretically **the best classifier**.

Proof: express $\mathbb{P}(g(X) \neq Y) - \mathbb{P}(g^*(X) \neq Y)$ in terms of dummies (use complementaries) and show that it is superior to 0.

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 sigmoid function.

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

Rationale behind the use of the sigmoid function:

We look for the *a posteriori* probability $\mathbb{P}(y = 1|x) = \pi(x) = \hat{y}$.

The predicted variable \hat{y} is thus a probability.

The sigmoid function: $\sigma : z \rightarrow \frac{1}{1+e^{-z}}$ 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).

See *Neural Network* section for more details on optimization.

Linear discriminant analysis

We focus on the binary case, that is when $Y = +1$ or $Y = -1$, that is to sets of variables. These two conditional laws need to be gaussians with same covariance:

$X|Y = +1 \sim \mathcal{N}(\mu_+, \Sigma)$ with density f_+

$X|Y = -1 \sim \mathcal{N}(\mu_-, \Sigma)$ with density f_-

Let π_+, π_- be the simple probabilities $P(Y = +1), P(Y = -1)$

$$\begin{aligned} \mathbb{P}\{Y = +1|X = x\} &= \frac{\mathbb{P}\{Y=+1, X=x\}}{\mathbb{P}\{X=x\}} \\ \mathbb{P}\{Y = +1|X = x\} &= \frac{\mathbb{P}\{X=x|Y=+1\}\mathbb{P}\{Y=+1\}}{\mathbb{P}\{X=x\}} \\ \mathbb{P}\{Y = +1|X = x\} &= \frac{f_+ \pi_+}{\mathbb{P}\{X=x\}} \\ \mathbb{P}\{Y = +1|X = x\} &= \frac{f_+ \pi_+}{(\mathbb{P}\{X=x|Y=+1\}\mathbb{P}\{Y=+1\} + \mathbb{P}\{X=x|Y=-1\}\mathbb{P}\{Y=-1\})} \\ \mathbb{P}\{Y = +1|X = x\} &= \frac{f_+ \pi_+}{(f_+ \pi_+ + f_- \pi_-)} \\ \text{Similarly,} \\ \mathbb{P}\{Y = -1|X = x\} &= \frac{f_- (1-\pi_+)}{\mathbb{P}\{X=x\}} \\ \mathbb{P}\{Y = -1|X = x\} &= \frac{f_- (1-\pi_+)}{(f_+ \pi_+ + f_- \pi_-)} \end{aligned}$$

The result shows us that we can express the two conditionnal probabilities in terms of conditionnal densities and "simple" probabilities (π_+, π_-) .

Recall that multivariable gaussian density is: $f(x) = \frac{1}{\sqrt{2\pi|\Sigma|}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}$

In practice, μ_+, μ_-, π_+ and Σ are unknown. Thus we use empiric values:

$$\begin{aligned} \hat{\pi}_+ &= m/n \\ \hat{\mu}_+ &= \frac{1}{m} \Sigma \mathbb{1}_{\{y_i=+1\}} x_i \\ \hat{\mu}_- &= \frac{1}{n-m} \Sigma \mathbb{1}_{\{y_i=-1\}} x_i \\ \hat{\Sigma} &= \frac{1}{n-2} ((m-1)\hat{\Sigma}_+ + (n-m-1)\hat{\Sigma}_-) \\ \hat{\Sigma}_+ &= \frac{1}{m-1} \Sigma \mathbb{1}_{\{y_i=+1\}} (x_i - \hat{\mu}_+)(x_i - \hat{\mu}_+)^T \end{aligned}$$

$$\hat{\Sigma}_- = \frac{1}{n-m-1} \Sigma \mathbb{1}_{\{y_i=-1\}} (x_i - \hat{\mu}_-)(x_i - \hat{\mu}_-)^T$$

Classification

We predict class = 1 when $\mathbb{P}(Y = +1|X) > \mathbb{P}(Y = -1|X)$

$$\Rightarrow \frac{\mathbb{P}(Y=+1|X)}{\mathbb{P}(Y=-1|X)} > 1$$

$$\Rightarrow \log\left(\frac{\mathbb{P}(Y=+1|X)}{\mathbb{P}(Y=-1|X)}\right) > 0$$

\Rightarrow

Using previous conditional probability expressions, we end up with the following prediction rule:

$$\begin{cases} 1 & \text{if } x^T \hat{\Sigma}^{-1}(\hat{\mu}_+ - \hat{\mu}_-) > \frac{1}{2} \hat{\mu}_+^T \hat{\Sigma} \hat{\mu}_+ - \frac{1}{2} \hat{\mu}_-^T \hat{\Sigma} \hat{\mu}_- + \log(1 - m/n) - \log(m/n) \\ -1 & \text{otherwise} \end{cases} \quad (2)$$

$\hat{\mu}_+$, $\hat{\mu}_-$, $\hat{\pi}_+$ and $\hat{\Sigma}$ will be computed with *train data*.

x is the *test data*.

Note: $\hat{\Sigma}^{-1}(\hat{\mu}_+ - \hat{\mu}_-)$ is the **Fisher function** (Saporta).

Listing 1: LDA algorithm

```
class LDAClassifier():
```

```
    def fit(self, X, y):
```

```
        X_p = X[y == 1, :]
        X_m = X[y == -1, :]
```

```
        X_p_x1 = X_p[:, 0]
        X_p_x2 = X_p[:, 1]
        X_m_x1 = X_m[:, 0]
        X_m_x2 = X_m[:, 1]
```

```
        n = len(X)
        m = len(X_p)
```

```
        mean_p_x1 = np.mean(X_p_x1)
        mean_p_x2 = np.mean(X_p_x2)
        mean_p = np.array([mean_p_x1, mean_p_x2]) # mu_plus (estimated)
        cov_p = np.cov(np.transpose(X_p))
```

```
        mean_m_x1 = np.mean(X_m_x1)
        mean_m_x2 = np.mean(X_m_x2)
        mean_m = np.array([mean_m_x1, mean_m_x2]) # mu_minus (estimated)
        cov_m = np.cov(np.transpose(X_m))
```

```
        cov_est = (1/(n-2))*((m-1)* cov_p + (n-m-1)* cov_m) # sigma (estimated)
        inv_cov_est = np.linalg.inv(cov_est)
```

```
        a1 = np.dot(np.transpose(mean_p), inv_cov_est)
        a2 = np.dot(np.transpose(mean_m), inv_cov_est)
```

```
        # 2nd term in inequality
```

```
        self.alpha = 0.5*(np.dot(a1, mean_p) - 0.5*np.dot(a2, mean_m)) + np.log(1 - m/n) - np.log
```

```
        # 1st term in inequality
```

```

self.beta = np.dot(inv_cov_est, mean_p-mean_m)

return self

def predict(self, X):
    y_=[]

    for i in range(len(X)):
        X_pred = X[i]
        beta = np.dot(np.transpose(X_pred), self.beta)
        if (beta>self.alpha):
            Y_pred = 1
        else:
            Y_pred = -1
        y_.append(Y_pred)
    return np.array(y_)

```

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 (π_1, \dots, π_k) ($\sum \pi_i = 1$)

We can write:

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

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:

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

Estimation

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

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

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

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

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

We thus need to find $\operatorname{argmax}(l(\theta))$

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 $z = (z_1, \dots, z_k)$ be the vector of latent variables. We can express the density (*) as a joint function with respect to z :

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

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

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:

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

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

(see kmeans.pdf from OneDrive folders for more details)

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:

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

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

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

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

Algorithm 1 K-means

```
Input:  $x_1, \dots, x_n$ 
Output: clusters  $C_1, \dots, C_k$ 
Random values for  $\mu_1, \dots, \mu_k$ 
while no convergence do
    // Step 1: Update clusters
     $C_1, \dots, C_k \leftarrow \emptyset$ 
    for  $i = 1$  to  $n$  do
         $j \leftarrow \operatorname{argmin}_l \|x_i - \mu_l\|$ 
         $C_j \leftarrow C_j + \{i\}$  // 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
```

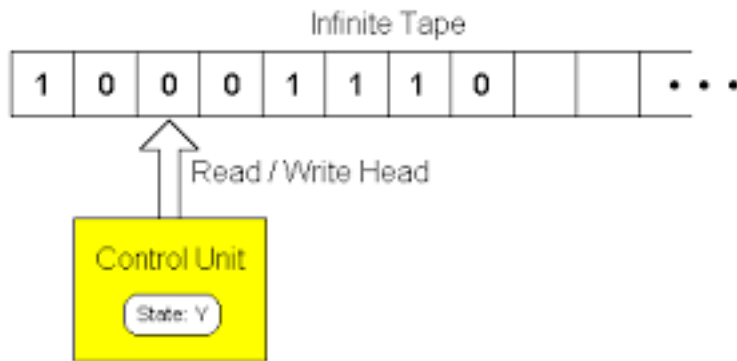
Listing 2: K-means++ initial centers selection

```
centers = []
centers.append(X[np.random.randint(X.shape[0])]) # initial 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...

Neural network

Logistic regression with a neural network mindset

Based on the example from coursera (*deeplearning.ai*)

Let us take the example of an image that we want to classify in a **binary** way: man/woman

The picture is vectorized as a vector of pixels : $\begin{pmatrix} x_1 \\ \dots \\ x_p \end{pmatrix}$

We use a regression to predict if it's a man/woman:

$$y = \omega^T x + b$$

Note: x are all the pixels of **one** image.

We want a probability in output (if it's ≥ 0.5 then we say it's a man).

We thus want the output to be $\hat{y} = \sigma(\omega^T x + b) = \mathbb{P}(y|x) \in [0, 1]$

(see regression part to get more details on the sigmoid)

Now since it's a binary classification, we want the y (real value) to be 0 or 1.

Thus, the loss function is:

$$\mathcal{L}(y, \hat{y}) = -y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

Proof: TODO from binary law (Bernoulli) to cross-entropy

The cost function is the empiric loss on all examples:

$$J(\omega, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

Forward propagation

$$x_1, x_2, \omega_1, \omega_p, b \rightarrow z = \omega_1 x_1 + \omega_2 x_2 + b \rightarrow \hat{y} = a = \sigma(z) \rightarrow \mathcal{L}(a, y)$$

- First arrow: regression

- Second arrow: probability
- Third arrow: error

Backward propagation

The idea is: with the error computed on the last step, we go backward in order to correct the parameters ω and b .

$$x_1, x_2, \omega_1, \omega_p, b \leftarrow z = \omega_1 x_1 + \omega_2 x_2 + b \leftarrow \hat{y} = a = \sigma(z) \leftarrow \mathcal{L}(a, y)$$

Example: we want to find ω_1 that minimizes the cost function:

$$\frac{d\mathcal{L}}{d\omega_1} = "d\omega_1" = \frac{d\mathcal{L}}{da} \frac{da}{dz} \frac{dz}{d\omega_1} = \dots = (a - y)x_1 = dzx_1$$

Steps:

We compute all the derivatives, then we apply the gradient descent

Listing 3: Gradient descent (logistic regression with a NN mindset)

```
for i in range(num_iterations):

    # Cost and gradient calculation
    grads, cost = propagate(w, b, X_train, Y_train) # propagation on ALL the training sample

    # Retrieve derivatives from grads
    dw = grads["dw"]
    db = grads["db"]

    # update parameters
    w = w - learning_rate * dw
    b = b - learning_rate * db

    # Record the costs
    costs.append(cost)
```

Listing 4: Propagation (logistic regression with a NN mindset)

```
def propagate(w, b, X, Y):

    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    A = sigmoid(np.dot(w.T, X) + b)
    cost = (- 1 / m) * np.sum(Y * np.log(A) + (1 - Y) * (np.log(1 - A)))

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw = (1/m)*np.dot(X, (A-Y).T)
    db = (1/m)*np.sum(A-Y)
```
