TP_perceptron_SI221

April 2, 2024

0.1 Supervised Learning - Perceptron

1 Stephen Cohen

Look into the following pages to begin or find documentation on the python librairies used here: - http://www.python.org - http://scipy.org - http://www.numpy.org - http://scikit-learn.org/stable/index.html - http://www.loria.fr/~rougier/teaching/matplotlib/matplotlib.html

1.1 Introduction

In this lab session, we will work on a practical application of supervised learning, which is arguably the simplest one: **binary supervised classification**. We will generate artificial data from two different sources, and we will try to learn a classifier that will be able to separate the data, using the **perceptron**.

1.1.1 Definitions and notations

- $\mathcal{X}$ is a set of *examples/observations/samples* $\mathbf{x} = (x_1, x_2, ..., x_p) \in \mathbb{R}^p$. We call x_j the value taken by the j-th variable of $\mathbf{x}$, or its j-th *feature*.
- $\mathcal{Y}$ is the set of labels. We are in the **binary** case: there is only two possible labels. We choose to note them $\{-1, 1\}$, which will make things easy by allowing us to work with the `sign` function.
- $\mathcal{D}_n = \{(x_i, y_i), i = 1, .., n\}$ is a dataset containing n examples and their labels.
- There exists a probabilistic model governing the generation of our data given i.i.d (independent and identically distributed) random variables X and Y: $\forall i \in \{1, ..., n\}, (x_i, y_i) \sim (X, Y)$
- We would like to build, from $\mathcal{D}_n$, a function that we call a *classifier*,

$$\hat{f} : \mathcal{X} \rightarrow \{-1, 1\}$$

  which for a new data point $\mathbf{x}_{new}$ will give a label $\hat{f}(\mathbf{x}_{new})$.

[ ]:
```
"""From a lab created on Mon Sep 23 17:50:04 2013 by baskiotis, salmon, gramfort
Modified on Mon Nov 4 21:09:38 2019 by mozharovskyi
Modified on Wed Feb 17 by labeau"""

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import ListedColormap
from matplotlib import cm
```

```python
import seaborn as sns
from matplotlib import rc

################################################################
# Code for displaying labeled data - no understanding needed ! #
################################################################

symlist = ['o', 'p', '*', 's', '+', 'x', 'D', 'v', '-', '^']

rc('font', **{'family': 'sans-serif', 'sans-serif': ['Computer Modern Roman']})
params = {'axes.labelsize': 12,
          'font.size': 16,
          'legend.fontsize': 16,
          'text.usetex': False,
          'figure.figsize': (8, 6)}
plt.rcParams.update(params)

sns.set_context("poster")
sns.set_palette("colorblind")
sns.set_style("white")
sns.axes_style()
```

[ ]:
```
{'axes.facecolor': 'white',
 'axes.edgecolor': '.15',
 'axes.grid': False,
 'axes.axisbelow': True,
 'axes.labelcolor': '.15',
 'figure.facecolor': 'white',
 'grid.color': '.8',
 'grid.linestyle': '-',
 'text.color': '.15',
 'xtick.color': '.15',
 'ytick.color': '.15',
 'xtick.direction': 'out',
 'ytick.direction': 'out',
 'lines.solid_capstyle': <CapStyle.round: 'round'>,
 'patch.edgecolor': 'w',
 'patch.force_edgecolor': True,
 'image.cmap': 'rocket',
 'font.family': ['sans-serif'],
 'font.sans-serif': ['Arial',
  'DejaVu Sans',
  'Liberation Sans',
  'Bitstream Vera Sans',
  'sans-serif'],
 'xtick.bottom': False,
 'xtick.top': False,
```

```
'ytick.left': False,
'ytick.right': False,
'axes.spines.left': True,
'axes.spines.bottom': True,
'axes.spines.right': True,
'axes.spines.top': True}
```

1.2 Generating artificial data

For our first experiment, and in order to visualize what is happening, we will work with only two features (so, with $p = 2$) so that we can plot the data and the classifier.

1) Take a look at the function `rand_gauss(n, mu, sigma)`: this function returns n samples following the multi-dimensional normal distribution, with mean the vector $\mu =$ `mu`, and with the covariance matrix Σ being the diagonal matrix of the vector `sigmas` $= [\sigma_1, \sigma_2]$. Hence, the matrix

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}$$

Now, generate different datasets from the function `rand_bi_gauss` function. What does the second output correspond to ?

[ ]:
```python
def rand_gauss(n=100, mu=[1, 1], sigmas=[0.1, 0.1]):
    """Sample  points from a Gaussian variable.
    Parameters
    ----------
    n : number of samples
    mu : centered
    sigma : standard deviation
    """
    d = len(mu)
    res = np.random.randn(n, d)
    return np.array(mu + res * sigmas)




def rand_bi_gauss(n1=100, n2=100, mu1=[1, 1], mu2=[-1, -1], sigmas1=[0.1, 0.1],
                  sigmas2=[0.1, 0.1]):
    """Sample points from two Gaussian distributions.
    Parameters
    ----------
    n1 : number of sample from first distribution
    n2 : number of sample from second distribution
    mu1 : center for first distribution
    mu2 : center for second distribution
    sigma1: std deviation for first distribution
    sigma2: std deviation for second distribution
    """
```

```
    ex1 = rand_gauss(n1, mu1, sigmas1)
    ex2 = rand_gauss(n2, mu2, sigmas2)
    y = np.hstack([np.ones(n1), -1 * np.ones(n2)])
    X = np.vstack([ex1, ex2])
    ind = np.random.permutation(n1 + n2)
    return X[ind, :], y[ind]
```

The second output is a label corresponding to the samples.

- It assigns label 1 to samples from the first distribution (ex1).
- It assigns label -1 to samples from the second distribution (ex2).

Finally, there is a permutation of this values. This is the second output

    Code:</div>

[ ]:
```
# Generate data with rand_gauss
n = 10
mu = [1., 1.]
sigmas = [1., 1.]
data_gauss= rand_gauss(n, mu, sigmas)



print(data_gauss)
```

```
[[ 1.09273429  0.32860718]
 [ 2.26131814  0.74280446]
 [-0.50619192  0.03623683]
 [ 0.21637262  1.44941837]
 [ 2.03648032 -0.35803058]
 [-0.13032085  1.9615901 ]
 [ 0.93829445  2.87509939]
 [ 1.8764055   0.24387293]
 [-0.14224564  0.24306138]
 [ 0.34362511  0.48947179]]
```

[ ]:
```
# Generate data with rand_bi_gauss

n1 = 20
n2 = 20
mu1 = [1., 1.]
mu2 = [-1., -1.]
sigmas1 = [0.9, 0.9]
sigmas2 = [0.9, 0.9]
X1,Y1 = rand_bi_gauss(n1, n2, mu1, mu2, sigmas1, sigmas2)
X,Y = rand_bi_gauss(n1, n2, mu1, mu2, sigmas1, sigmas2)
```

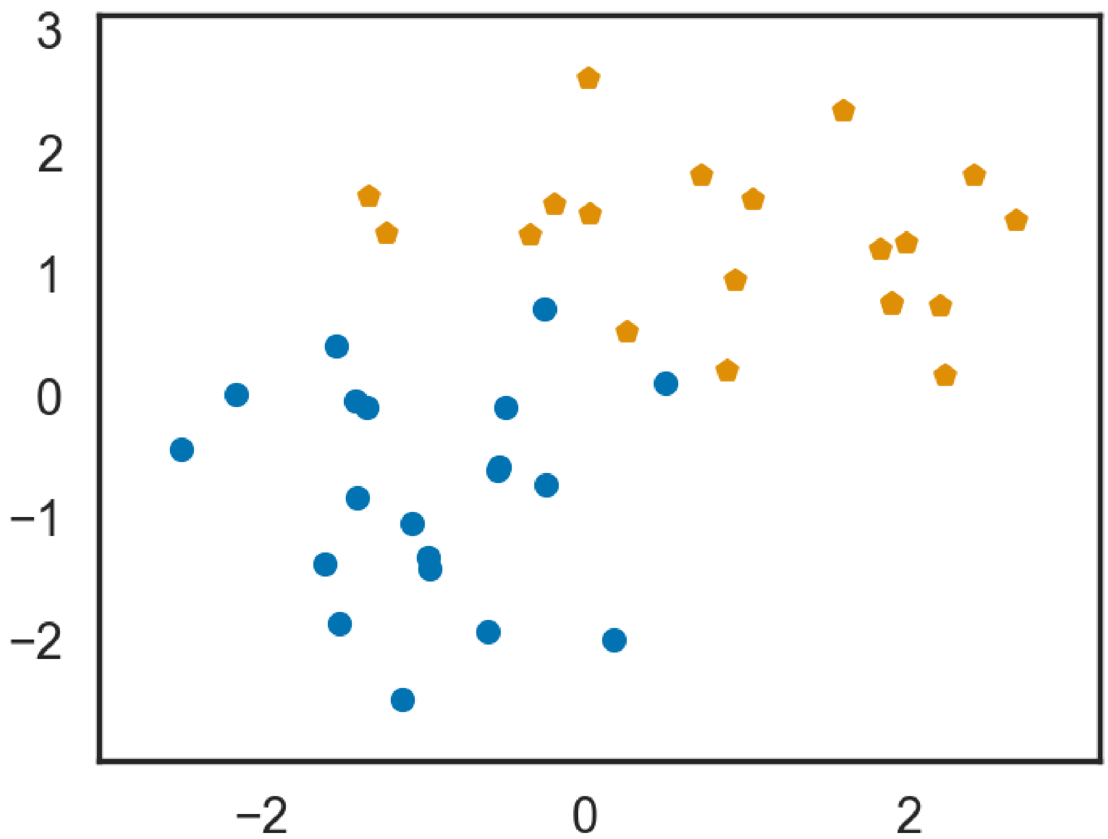
2) Keep some of these datasets to use them in the rest of the lab. For each of them, save them as a numpy array with two coloumns for X, and as a vector for Y. Use the function `plot_2d`

allowing you to visualize the data with their associated labels.

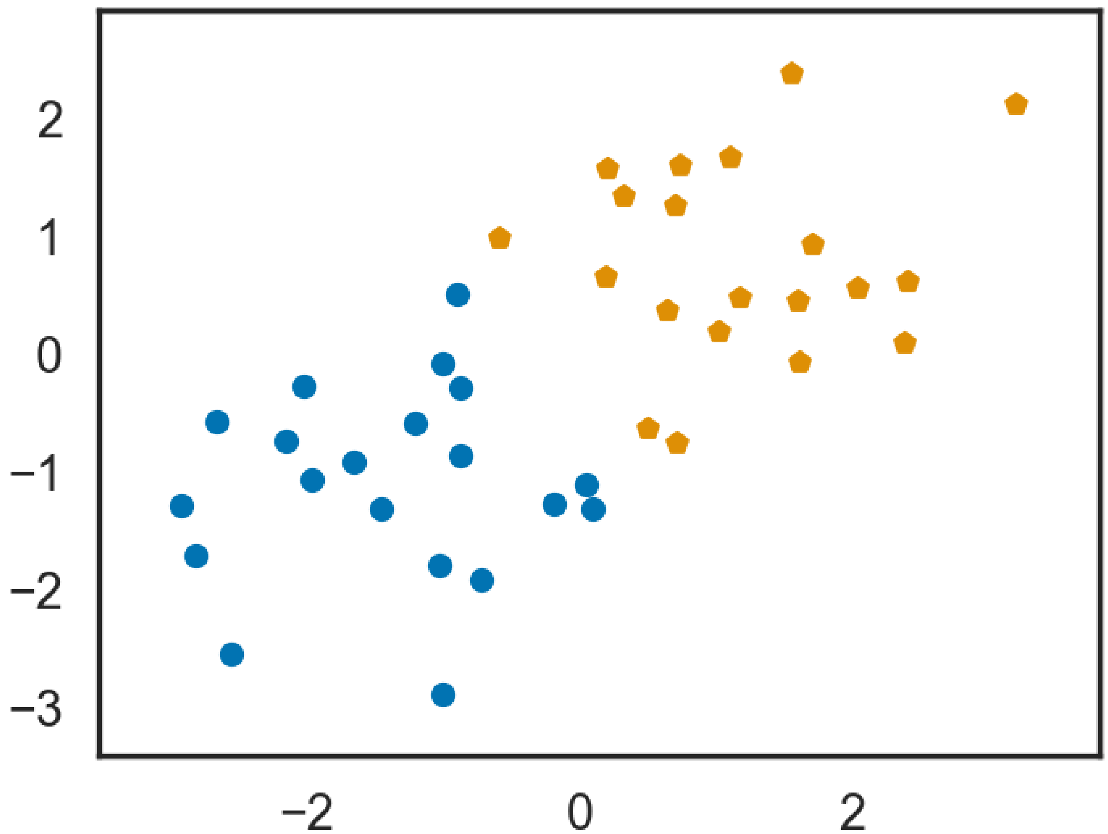
```
[ ]: def plot_2d(X, y, w=None, step=50, alpha_choice=1):
         """2D dataset data ploting according to labels.
         Parameters
         ----------
         X : data features
         y : label vector
         w :(optional) the separating hyperplan w
         alpha_choice : control alpha display parameter
         """
         min_tot0 = np.min(X[:, 0])
         min_tot1 = np.min(X[:, 1])

         max_tot0 = np.max(X[:, 0])
         max_tot1 = np.max(X[:, 1])
         delta0 = (max_tot0 - min_tot0)
         delta1 = (max_tot1 - min_tot1)
         labels = np.unique(y)
         k = np.unique(y).shape[0]
         color_blind_list = sns.color_palette("colorblind", k)
         sns.set_palette(color_blind_list)
         for i, label in enumerate(y):
             label_num = np.where(labels == label)[0][0]
             plt.scatter(X[i, 0], X[i, 1],
                         c=np.reshape(color_blind_list[label_num], (1, -1)),
                         s=80, marker=symlist[label_num])
         plt.xlim([min_tot0 - delta0 / 10., max_tot0 + delta0 / 10.])
         plt.ylim([min_tot1 - delta1 / 10., max_tot1 + delta1 / 10.])
         if w is not None:
             plt.plot([min_tot0, max_tot0],
                      [min_tot0 * -w[1] / w[2] - w[0] / w[2],
                       max_tot0 * -w[1] / w[2] - w[0] / w[2]],
                      "k", alpha=alpha_choice)
```

Code:</div>

```
[ ]: # Plot your generated data
     plot_2d(X,Y)
```



[ ]:
```
plot_2d(X1,Y1)
```



1.3 The perceptron

Linear classifiers (affine) A **linear classifier** is a classifier associating to each observation x a label in $\mathcal{Y}$ given its position related to an **affine hyperplane**. Each linear classifier is therefore linked to an affine hyperplan of $\mathbb{R}^p$, which we define by its **directing vector** (or normal vector) $\mathbf{w} = (w_0, w_1, ..., w_p) \in \mathbb{R}^{p+1}$. Note the supplementary dimension: it is used for what we call the *intercept* w_0 of the classifier (in French, *ordonnée à l'origine*). Geometrically, this allows the hyperplane to not go through the origin, and be shifted anywhere in the space. The hyperplane is then given by:

$$H_{\mathbf{w}} = \left\{ \mathbf{x} \in \mathbb{R}^p : \hat{f}_{\mathbf{w}}(\mathbf{x}) := w_0 + \sum_{j=1}^{p} w_j x_j = 0 \right\}.$$

In order to classify an observation $\mathbf{x}$ (i.e, affect a label 1 or -1, we use the function $sign(f(\mathbf{x}))$, where the *sign* function is defined as:

$$sign(x) = \begin{cases} 1, & \text{if } x \geq 0, \\ -1, & \text{if } x < 0 \end{cases}.$$

Hence, the function $\mathbf{x} \mapsto sign(\hat{f}_{\mathbf{w}}(\mathbf{x}))$ is the binary classifier of linear separation defined by $\mathbf{w}$. The objective of the perceptron is to find an hyperplane which separates in the best possible way the data, into two groups. We would like to have, on each side of the hyperplane, labels separated into homogenous groups. The vector **w** is called a **weight vector**.

For the next questions, you should reuse the datasets you obtained previously.

3) What does the linear separation given by the perceptron in dimension $p = 2$ correspond to? See if you can find (visually) a good separation for your datasets. When is $\hat{f}_{\mathbf{w}}(\mathbf{x})$ large? Negative? Positive? How can that function be interpreted geometrically? What does w_0 correspond to on the linear separation you found on your data?

    `Answer:</div>`

The linear separation when p=2 corresponds to a line.

Visually speaking, we can take the symetris axis beetween (1,1) and (-1,-1) which is define by y=-x ie w = [0,1,1]

The linear classifier is Negative when x is > 0 and Positive when x < 0.

It is large when abs(x) tends to + infinity.

w0 corresponds to the "ordonnée à l'origine".

[ ]:
```

```

4) Write the the function `predict(x,w)` that takes, as input, a vector $\mathbf{x} \in \mathbb{R}^p$ and a weight vector $\mathbf{w} \in \mathbb{R}^{p+1}$ and outputs the prediction $\hat{f}_{\mathbf{w}}(\mathbf{x})$. Then, write `predict_class(x,w)` that outputs the predicted label $sign\left(\hat{f}_{\mathbf{w}}(\mathbf{x})\right)$. Apply them to the following example, and display on the same plot your data, the hyperplane, and the two new points.

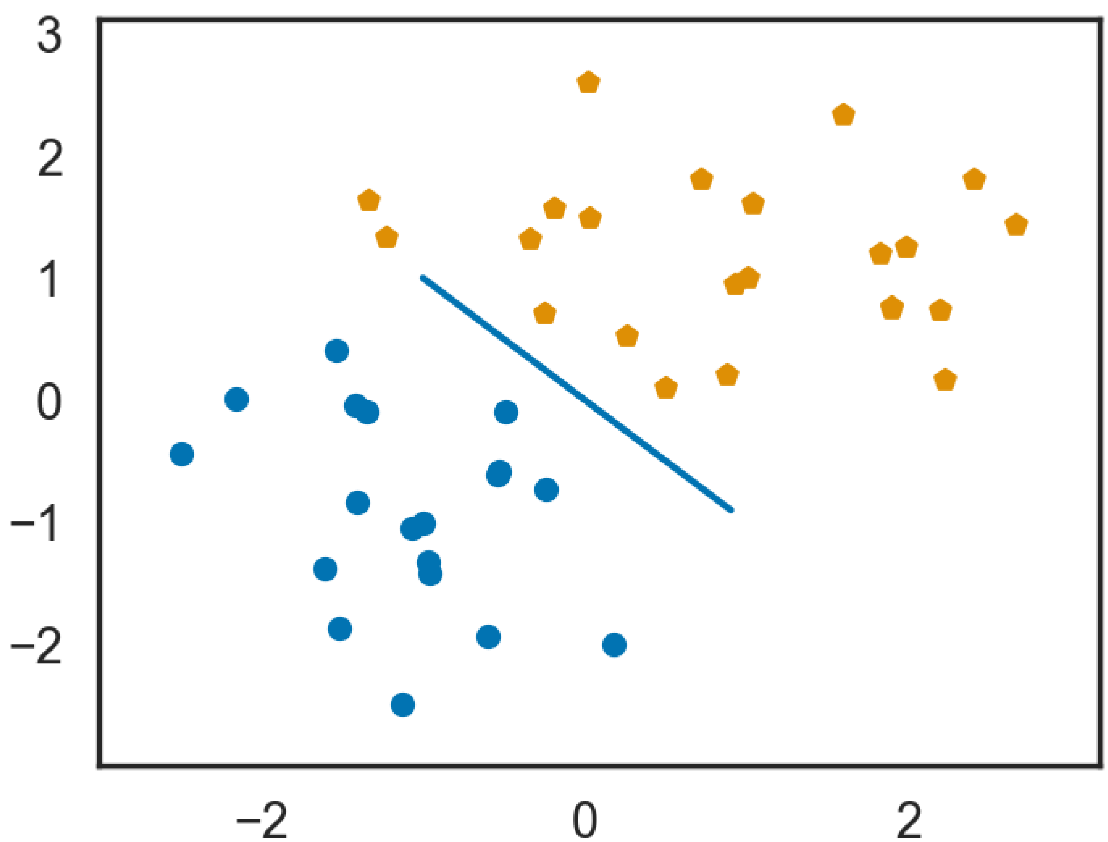
    `Code:</div>`

[ ]:
```
def predict(x, w):
    """Prediction from a normal vector."""
    s = w[0]
    for i in range(len(x)):
        s+=x[i]*w[i+1]
    return s



def predict_class(x, w):
    """Predict a class from at point x thanks to a normal vector."""
    s=predict(x,w)
    if s>0:
        return 1
    else :
        return -1
```

[ ]:
```
w1 = [0, 1, 1] # Visually, seems like it would make an okay normal vector for a␣
↪separating hyperplane
x_test_1 = [-1, -1]
x_test_2 = [1, 1]
```

```
# Make predictions for these two points and display them, with the data and␣
↪hyperplane.

Y_test_1 = predict_class(x_test_1,w1)
Y_test_2 = predict_class(x_test_2,w1)

Y2 = np.array([predict_class(x,w1) for x in X])


print("Pour x1 la prediction est : " +str(Y_test_1))
print("Pour x2 la prediction est : " +str(Y_test_2))
plt.plot(np.arange(-1, 1, 0.1),[-x for x in np.arange(-1, 1, 0.1)])
plot_2d(np.array([x_test_1,x_test_2]),np.array([Y_test_1,Y_test_2]))
plot_2d(X,Y2)
```

```
Pour x1 la prediction est : -1
Pour x2 la prediction est : 1
```



1.3.1 Perceptron learning rule

The **perceptron algorithm** consists in detecting when there is a mistake, meaning that there is a point that is misclassified, and moving $\mathbf{w}$ towards having this point on the 'right side' of the hyperplane. The change in $\mathbf{w}$ for a case when the point is misclassified is described by the following learning rule (Rosenblatt's):

$$\mathbf{w} \leftarrow \mathbf{w} + \epsilon \begin{pmatrix} 1 \\ \mathbf{x}_i \end{pmatrix} \cdot y_i$$

where ϵ is a *learning step*, which indicates how much we correct $\mathbf{w}$. The method is iterative: we will go through all examples we have in our data and update the points accordingly. Then, the algorithm is as follows:

- **Data**:
  - The observations and their labels $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : 1 \leq i \leq n\}$
  - The gradient step: ϵ
  - The maximal number of iterations: n_{iter}
- **Result**:
  - $\mathbf{w}$
- Randomly initialize $\mathbf{w}$; initialize $j = 0$
- While $j \leq n_{\text{iter}}$
  - $\mathbf{w} \leftarrow \mathbf{w}$
  - For $i = 1$ to n:
    - if $\hat{f}_{\mathbf{w}}(\mathbf{x}) \cdot y_i \leq 0$
      - $\mathbf{w} \leftarrow \mathbf{w} + \epsilon \begin{pmatrix} 1 \\ \mathbf{x}_i \end{pmatrix} \cdot y_i$
  - $j \leftarrow j + 1$

5. You have to complete the code for this procedure in the `perceptron` function:

   Code:</div>

[ ]:
```python
def perceptron(x, y, eps, niter, w_ini):
    """ Perceptron algorithm:
        - x : Data
        - y : label
        - eps : learning rate
        - niter : number of iterations
        - w_ini : initial weight
        """
    # Keep track of w at each iterations - the first one is w_ini
    w = np.zeros((niter, w_ini.size))
    w[0] = w_ini

    # Implement the learning loop
    j=0
    while 0<=j<niter-1:
        w[j+1]=w[j]
        for i in range(len(x)):
            if predict(x[i],w[j])*y[i]<= 0:
```

```
                w[j+1] = w[j+1] + eps*np.array([1,x[i][0],x[i][1]])*y[i]
        j=j+1

    return w
```

6) Test the perceptron algorithm on the following parameters and look at how **w** evolves during the iterations.

   Code:</div>

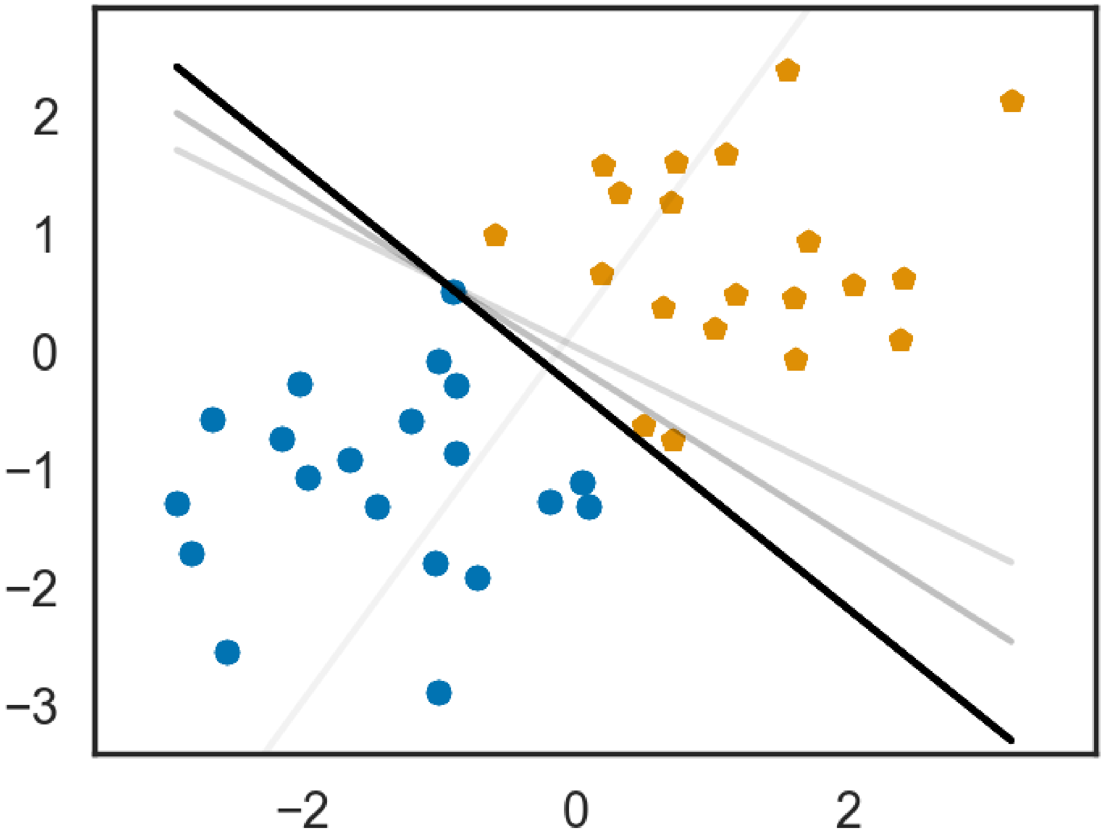
```
[ ]: # Choose an epsilon
epsilon = 0.2
# A number of iterations
niter = 10
# Initialize w_ini
std_ini = 1.
w_ini = std_ini * np.random.randn(X1.shape[1] + 1)

pep = perceptron(X1,Y1,epsilon,niter,w_ini)
print(pep)
```

```
[[ 0.10469747  0.84335041 -0.53262931]
 [-0.09530253  1.57716001  2.76097017]
 [ 0.30469747  1.8191027   2.48448195]
 [ 0.70469747  2.06104538  2.20799374]
 [ 0.70469747  2.06104538  2.20799374]
 [ 0.70469747  2.06104538  2.20799374]
 [ 0.70469747  2.06104538  2.20799374]
 [ 0.70469747  2.06104538  2.20799374]
 [ 0.70469747  2.06104538  2.20799374]
 [ 0.70469747  2.06104538  2.20799374]]
```

7) Display on the same figure the evolution of the boundaries according to the iterations. We can use the `plot_2d` function and its `alpha_choice` argument for that purpose.

   Code:</div>

```
[ ]: # Use a loop and the last argument of plot_2d to plot each hyperplane (at each
 ↪iteration)

for i in range(len(pep)):
    #def h(x,i):
    #   return (pep[i][0] + pep[i][1]*x)/(- pep[i][2])
    #plt.plot(np.arange(-4,4,0.2),[h(x,i) for x in np.arange(-4,4,0.2) ])
    plot_2d(X1,Y1,w=pep[i],alpha_choice=(i + 0.5)/niter )


#for i in range(len(pep)):
    #plot_2d(X1,Y1,pep[i])
```



1.3.2 General case: cost function

While the perceptron algorithm we saw works geometrically, in general, in order to measure the error associated to an entire dataset $\mathcal{D}_n$ it is necessary to set a loss function $\ell : \mathbb{R} \times \mathcal{Y} \mapsto \mathbb{R}^+$ which measures the cost h of an error i when predicting an example. The cost that we want to minimize (as a function of $\mathbf{w}$) is $\mathbb{E}\left[\ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y)\right]$, the expectation of the loss function on all the data. Three loss functions are commonly used and defined below : - the error percentage: $ZeroOneLoss(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) = |y - sign(\hat{f}_{\mathbf{w}}(\mathbf{x}))|/2$, - the quadratic error: $MSELoss(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) = (y - \hat{f}_{\mathbf{w}}(\mathbf{x}))^2$, - the hinge error (*charnière* in French): $HingeLoss(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) = \max(0, 1 - y \cdot \hat{f}_{\mathbf{w}}(\mathbf{x}))$. The purpose here is to study these different loss functions. These three functions are already implemented, as well as their associated gradient:

[ ]:
```
def zero_one_loss(x, y, w):
    """0-1 loss function."""
    return abs(y - np.sign(predict(x, w))) / 2.

def hinge_loss(x, y, w):
    """Hinge loss function."""
    return np.maximum(0., 1. - y * predict(x, w))
```

```python
def mse_loss(x, y, w):
    """Mean square error loss."""
    return (y - predict(x, w)) ** 2

def gr_hinge_loss(x, y, w):
    """Sub-gradient of the loss function hingeloss."""
    return np.dot(-y * (hinge_loss(x, y, w) > 0.),
                  np.concatenate(([1],x)))


def gr_mse_loss(x, y, w):
    """Gradient of the least squares lost function."""
    return -2. * np.dot(y - predict(x, w),
                        np.concatenate(([1],x)))
```

8) Suppose that $\mathbf{x} \in \mathbb{R}^p$ and $y \in \mathbb{R}$ are fixed. What is the nature of the functions $\mathbf{w} \mapsto \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y)$ for the three losses studied: constant, linear, quadratic, piecewise constant, linear by pieces, quadratic by pieces, etc. ?

   `Answer:</div>`

Pour la première fonction de loss (zero loss) : Il s'agit d'une fonction constante qui vaut |y-1|/2 et |y+1|/2

Pour la seconde fonction de loss (hinge_loss) : Il s'agit d'une fonction qui vaut 0 sur {w tq y * predict(x, w) < 1} puis est linéaire

Pour la troisième fonction de loss : Il s'agit d'une fonction quadratique en w

1.4 Stochastic gradient descent algorithm

In the general case, it is of course impossible to do an exhaustive search of the space $\mathbb{R}^{p+1}$ where evolves $\mathbf{w}$ to minimize the gradient of $\mathbf{w} \mapsto \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y)$ in order to find the minimum cost. Moreover, we cannot observe $\mathbb{E}\left[\ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y)\right]$, so we can only try to **minimize its empirical counterpart**:

$$\sum_{i=1}^{n} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$$

The **perceptron algorithm** actually consists in using a variant of the gradient descent algorithm, which is the usual method of optimizing a differentiable function. Here, at each step, the current weight is corrected in the same direction as the gradient, but in the opposite sense. The algorithm converges in the general case to a local minimum as long as ϵ, which we now call the **gradient step** is well chosen. Moreover, the minimum reached is global for convex functions.

The stochastic gradient method is a variant which proposes not to directly use the gradient, which requires calculating a sum on the n observations, but rather to draw (randomly … or not) a pair $(\mathbf{x}_i, y_i)$ on which a gradient is calculated. We can also show that this algorithm converges under certain conditions.

The **perceptron algorithm** is now described as follows:

- **Data**:
  - The observations and their labels $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : 1 \leq i \leq n\}$
  - The gradient step: ϵ
  - The maximal number of iterations: n_{iter}
- **Result**:
  - $\mathbf{w}$
- Randomly initialize $\mathbf{w}$; initialize $j = 0$
- While $j \leq n_{\text{iter}}$
  - $\mathbf{w} \leftarrow \mathbf{w}$
  - For $i = 1$ to n:
    - $\mathbf{w} \leftarrow \mathbf{w} - \epsilon \nabla_{\mathbf{w}} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i, y_i))$
    - $j \leftarrow j + 1$

The code for this procedure is to complete in the `gradient` function:

*Note: The stochastic gradient method is also available in **`sklearn`** under the name `SGDClassify` (SGD is the abbreviation for Stochastic Gradient Descent). A description is given on the page: http://scikit-learn.org/stable/modules/sgd.html.*

9. Simple gradient descent is also called **batch**, and consists in calculating the true gradient $\frac{1}{n}\sum_{i=1}^{n} \nabla_{\mathbf{w}} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$. In the stochastic case, or SGD, we draw uniformly a random example instead. Implement both these ways of selecting examples in the following function.

    Code:</div>

```
[ ]: def gradient(x, y, eps, niter, w_ini, loss_fun, gr_loss_fun, stochastic=True):
         """ Algorithm for gradient descent:
             - x : Data
             - y : label
             - eps : learning rate
             - niter : number of iterations
             - w_ini : initial weight
             - loss_fun : cost function
             - gr_loss_fun : gradient of the cost function
             - stoch : True : implements SGD
             """
         w = np.zeros((niter, w_ini.size))
         w[0] = w_ini

         # We also keep track of the loss and initialize it
         loss = np.zeros(niter)
         loss[0] = np.array([loss_fun(x[i], y[i], w[0]) for i in range(len(x))]).
      ↪mean()
         indexes_shuffled = [i for i in range(len(x))]
         np.random.shuffle(np.array(indexes_shuffled))
         for i in range(1, niter):
             if stochastic: # Which indexes are we using in the SGD case ?
                 indexes = [indexes_shuffled[i]]
             else: # Which indexes are we using in the simple gradient case ?
```

```
            indexes = [i for i in range(len(x))]
# Update the weight and compute the new loss
        w[i] = w[i - 1]
        for index in indexes:
            w[i] -= eps * gr_loss_fun(x[index], y[index], w[i])
        loss[i] = np.array([loss_fun(x[k], y[k], w[i]) for k in range(len(x))]).
↪mean()
    return w, loss
```

[ ]:
```
print(X1[1])
```

```
[-1.96643056 -1.07615352]
```

[ ]:
```
print(X1[1,:])
```

```
[-1.96643056 -1.07615352]
```

10. The perceptron algorithm we first saw is equivalent to this more general one, with a specific loss function. Which one? In this case, interpret the following condition: $\hat{f}_{\mathbf{w}}(\mathbf{x}_i) \cdot y_i \leq 0$

    Answer:</div>

Dans le cas précédent, nous avions la fonction dont le gradient est : $(x_i, y_i, w) \mid -> -\epsilon \begin{pmatrix} 1 \\ \mathbf{x}_i \end{pmatrix} \cdot y_i$

Par conséquent, la fonction est $hinge_{Loss}$

Ainsi, la condition $\hat{f}_{\mathbf{w}}(\mathbf{x}_i) \cdot y_i \leq 0$ est équivalente au fait de se rapprocher du point d'équilibre. (Pour être dans la valeur du max)

11) We will graphically observe, with the function `plot_gradient`, the evolution of $\frac{1}{n}\sum_{i=1}^{n} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$ according to $\mathbf{w}$ following the steps of the algorithm. Why should the number of iterations be `niter * len(y)` instead of `niter` in the stochastic case ?

    Answer:</div>

Nous avons besoin de niter*len(y) dans le cas stochastique car dans le cas non stochastique nous avons niter*n itération. Ainsi pour le converger soit davantage assurée il vaut mieux réaliser plus d'itérations pour lisser les points abbérants

[ ]:
```
def plot_gradient(X, y, wh, cost_hist, loss_fun):
    """ display 4 figures on how  (stochastic) gradient descent behaves
    wh : solution history
    cost_hist : cost history
    loss_fun : loss function
    """
    best = np.argmin(cost_hist)
    plt.subplot(221)
    plt.title('Data and hyperplane estimated')
    plot_2d(X, y, wh[best, :])
    plt.subplot(222)
```

```python
    plt.title('Projection of level line and algorithm path')
    plot_cout(X, y, loss_fun, wh)
    plt.subplot(223)
    plt.title('Objective function vs iterations')
    plt.plot(range(cost_hist.shape[0]), cost_hist)
    plt.subplot(224, projection='3d')
    plt.title('Level line and algorithm path')
    plot_cout3d(X, y, loss_fun, wh)
```

[ ]:
```python
def frontiere(f, X, step=50, cmap_choice=cm.coolwarm):
    """Frontiere plotting for a decision function f."""
    min_tot0 = np.min(X[:, 0])
    max_tot0 = np.max(X[:, 0])
    min_tot1 = np.min(X[:, 1])
    max_tot1 = np.max(X[:, 1])
    delta0 = (max_tot0 - min_tot0)
    delta1 = (max_tot1 - min_tot1)
    xx, yy = np.meshgrid(np.arange(min_tot0, max_tot0, delta0 / step),
    np.arange(min_tot1, max_tot1, delta1 / step))
    z = np.array([f(vec) for vec in np.c_[xx.ravel(), yy.ravel()]])
    z = z.reshape(xx.shape)
    plt.imshow(z, origin='lower', interpolation="nearest", cmap=cmap_choice,
    extent=[min_tot0, max_tot0, min_tot1, max_tot1])
    plt.colorbar()
def frontiere_new(clf, X, y, w=None, step=50, alpha_choice=1, colorbar=True,
    samples=True, n_labels=3, n_neighbors=3):
    """Trace la frontiere pour la fonction de decision de clf."""
    min_tot0 = np.min(X[:, 0])
    min_tot1 = np.min(X[:, 1])

    max_tot0 = np.max(X[:, 0])
    max_tot1 = np.max(X[:, 1])

    delta0 = (max_tot0 - min_tot0)
    delta1 = (max_tot1 - min_tot1)
    xx, yy = np.meshgrid(np.arange(min_tot0, max_tot0, delta0 / step),
                         np.arange(min_tot1, max_tot1, delta1 / step))
    XX = np.c_[xx.ravel(), yy.ravel()]

    z = clf.predict(XX)
    z = z.reshape(xx.shape)
    labels = np.unique(z)
    color_blind_list = sns.color_palette("colorblind", labels.shape[0])
    my_cmap = ListedColormap(color_blind_list)
    plt.imshow(z, origin='lower', interpolation="mitchell", alpha=0.80,
            cmap=my_cmap, extent=[min_tot0, max_tot0, min_tot1, max_tot1])
```

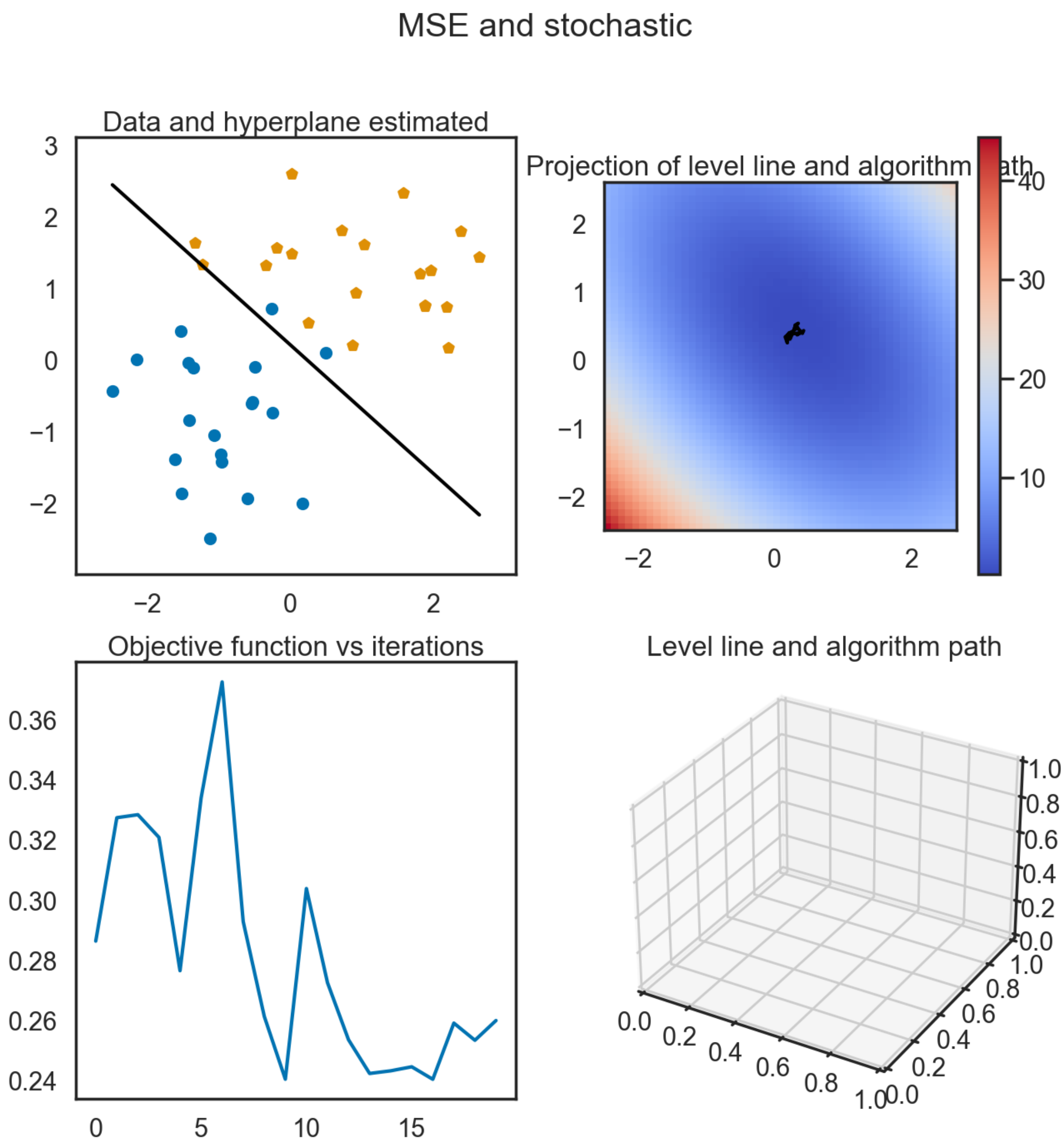
```
    if colorbar is True:
        ax = plt.gca()
        cbar = plt.colorbar(ticks=labels)
        cbar.ax.set_yticklabels(labels)
# color_blind_list = sns.color_palette("colorblind", labels.shape[0])
# sns.set_palette(color_blind_list)
        ax = plt.gca()
    if samples is True:
        for i, label in enumerate(y):
            label_num = np.where(labels == label)[0][0]
            plt.scatter(X[i, 0], X[i, 1], c=[color_blind_list[label_num]],
                        s=80, marker=symlist[label_num])
    plt.xlim([min_tot0, max_tot0])
    plt.ylim([min_tot1, max_tot1])
    ax.get_yaxis().set_ticks([])
    ax.get_xaxis().set_ticks([])
    if w is not None:
        plt.plot([min_tot0, max_tot0],
                 [min_tot0 * -w[1] / w[2] - w[0] / w[2],
                 max_tot0 * -w[1] / w[2] - w[0] / w[2]],
                 "k", alpha=alpha_choice)
    plt.title("L=" + str(n_labels) + ",k=" +str(n_neighbors))

def frontiere_3d(f, data, step=20):
    #ax = plt.gca(projection='3d')
    fig = plt.figure()
    ax = fig.add_subplot(projection="3d")
    xmin, xmax = data[:, 0].min() - 1., data[:, 0].max() + 1.
    ymin, ymax = data[:, 1].min() - 1., data[:, 1].max() + 1.
    xx, yy = np.meshgrid(np.arange(xmin, xmax, (xmax - xmin) * 1. / step),
    np.arange(ymin, ymax, (ymax - ymin) * 1. / step))
    z = np.array([f(vec) for vec in np.c_[xx.ravel(), yy.ravel()]])
    z = z.reshape(xx.shape)
    ax.plot_surface(xx, yy, z, rstride=1, cstride=1,linewidth=0.,␣
↪antialiased=False, cmap=plt.cm.coolwarm)
def plot_cout(X, y, loss_fun, w=None):
    """Plot the cost function encoded by loss_fun,
      Parameters,----------
      X : data features
      y : labels
      loss_fun : loss function
      w : (optional) can be used to give a historic path of the weights """
    def _inter(wn):
        ww = np.zeros(3)
        ww[1:] = wn
        return np.array([loss_fun(X[k], y[k], ww) for k in range(len(X))]).
↪mean()
```

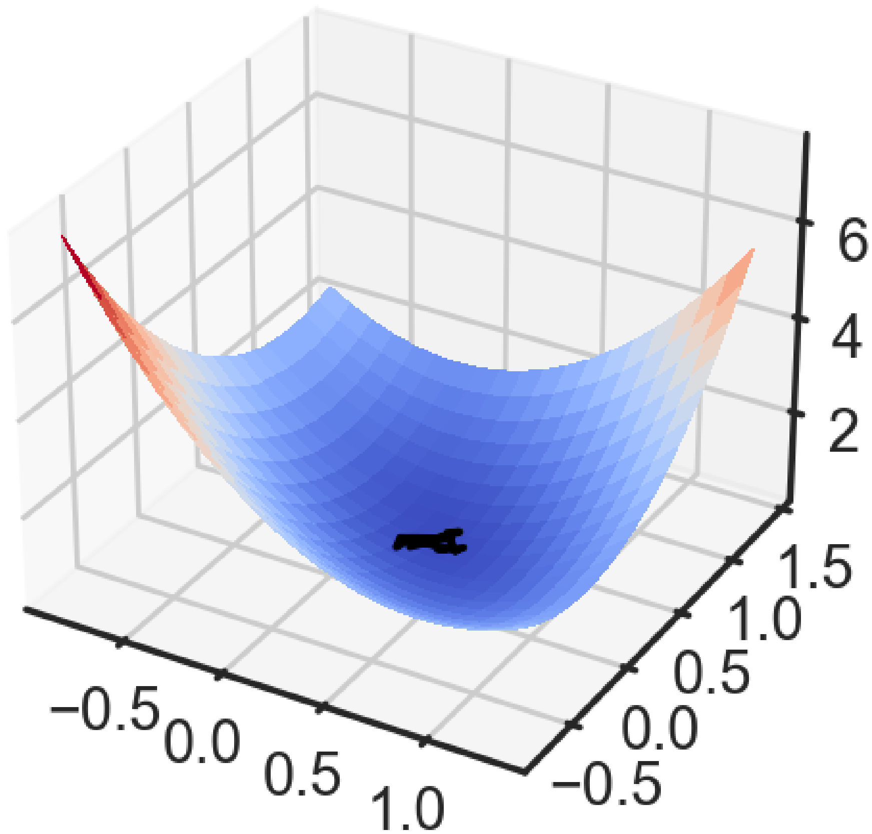
```
    datarange = np.array([[np.min(X[:, 0]), np.min(X[:, 1])],[np.max(X[:, 0]),␣
↪np.max(X[:, 1])]])
    frontiere(_inter, np.array(datarange))
    if w is not None:
        plt.plot(w[:, 1], w[:, 2], 'k')
        plt.xlim([np.min(X[:, 0]), np.max(X[:, 0])])
        plt.ylim([np.min(X[:, 1]), np.max(X[:, 1])])
def plot_cout3d(x, y, loss_fun, w):
    """ trace le cout de la fonction cout loss_fun passee en parametre, en x,y,
en faisant varier les coordonnees du poids w.
W peut etre utilise pour passer un historique de poids"""
    def _inter(wn):
        ww = np.zeros(3)
        ww[1:] = wn
        return np.array([loss_fun(x[k], y[k], ww) for k in range(len(x))]).
↪mean()
    datarange = np.array([[w[:, 1].min(), w[:, 2].min()],[w[:, 1].max(), w[:,␣
↪2].max()]])
    frontiere_3d(_inter, np.array(datarange))
    plt.plot(w[:, 1], w[:, 2], np.array([_inter(w[i, 1:]) for i in range(w.
↪shape[0])]), 'k-', linewidth=3)
```

12) Experiment on different datasets: use either the cost functions provided here with `gradient`, either the `sklearn` function. Study performance according to the following points: the number of iterations, the cost function, the difficulty of the problem (whether the classes are easily separable or not, which you can act on by choosing different gaussian to generate from). Do you at any point observe strange behaviour ? If so, what is the reason?

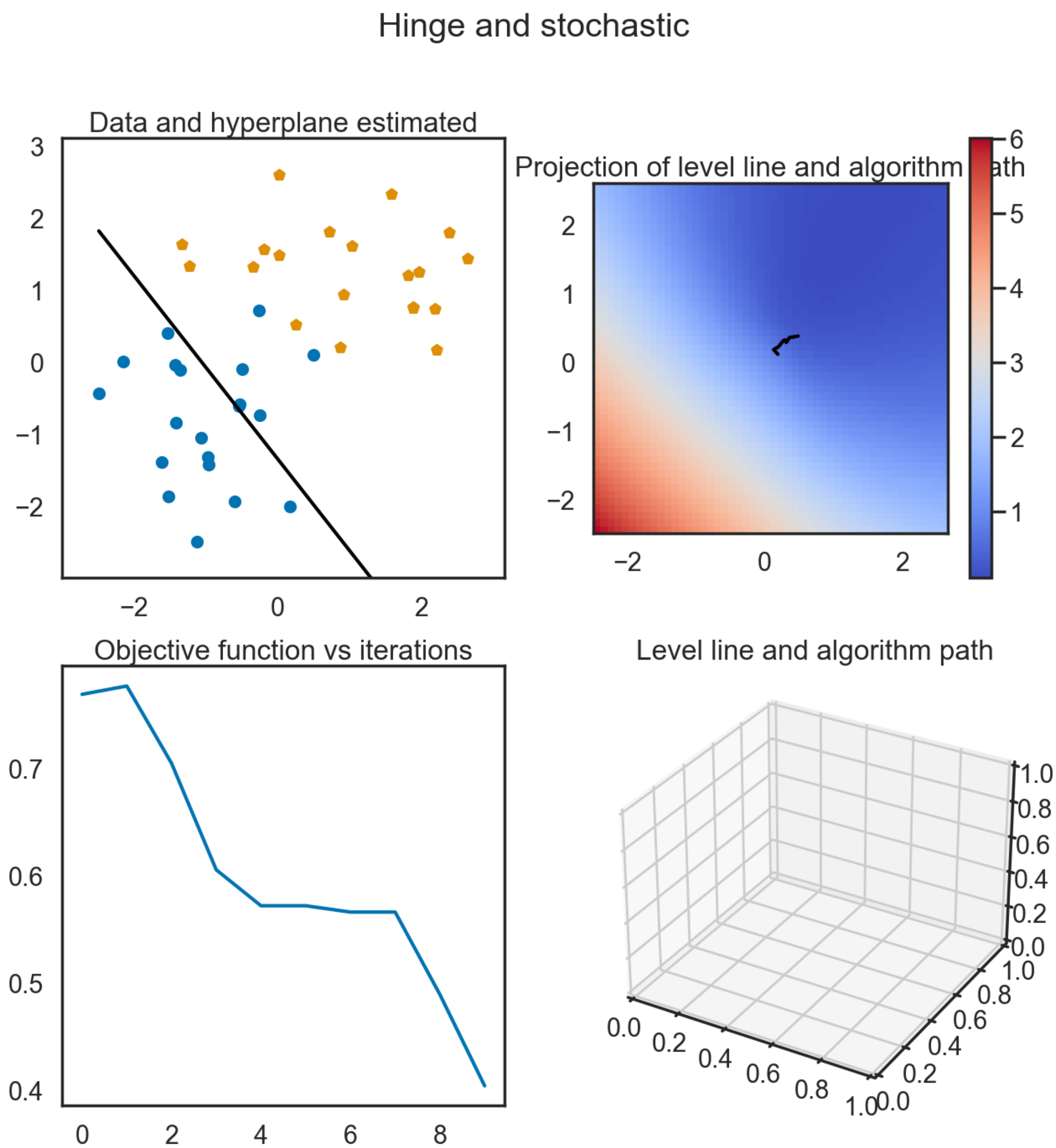
    Code:</div>

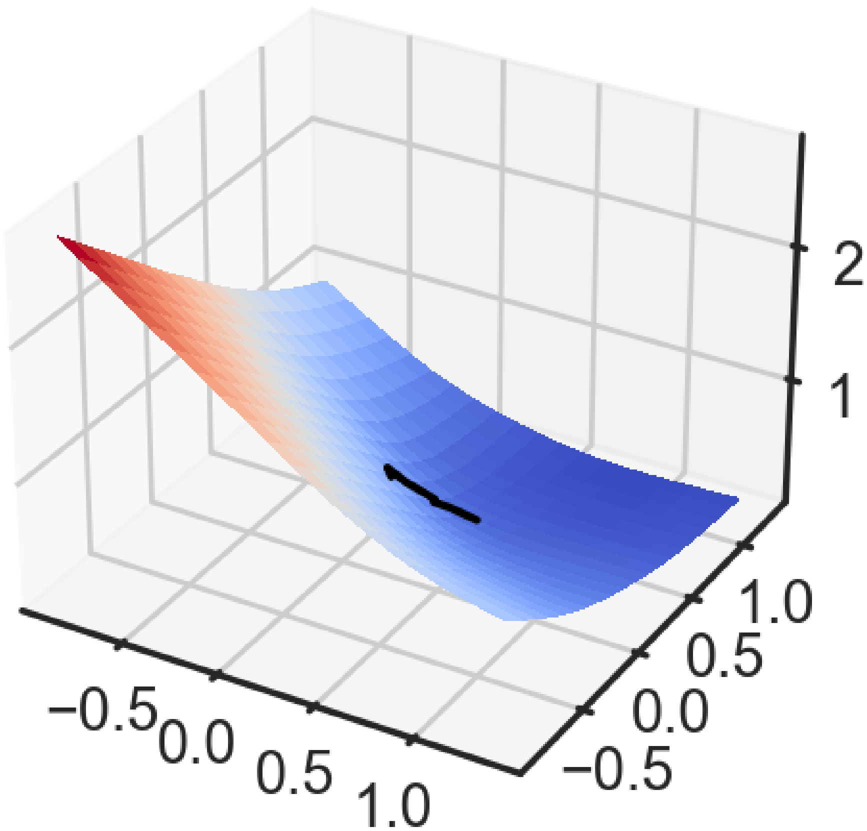
[ ]:
```
epsilon = 0.05
niter = 20
plt.figure(8, figsize=(15, 15))
plt.suptitle('MSE and stochastic')
std_ini = 1.
w_ini = std_ini * np.random.randn(X1.shape[1] + 1)
23
wh, cost_hist = gradient(X, Y, epsilon, niter, w_ini, mse_loss,␣
↪gr_mse_loss,stochastic=True)
plot_gradient(X, Y, wh, cost_hist, mse_loss)
plt.show()
```


MSE and stochastic
Data and hyperplane estimated
Projection of level line and algorithm path
Objective function vs iterations
Level line and algorithm path



[ ]:
```
epsilon = 0.05
niter = 10
std_ini = 1.
w_ini = std_ini * np.random.rand(X1.shape[1] + 1)

plt.figure(8, figsize=(15, 15))
plt.suptitle('Hinge and stochastic')

w_hinge, loss_hinge = gradient(X, Y, epsilon, niter, w_ini, hinge_loss,␣
 ↪gr_hinge_loss, stochastic=True)
plot_gradient(X,Y,w_hinge,loss_hinge, hinge_loss)

plt.show()
```

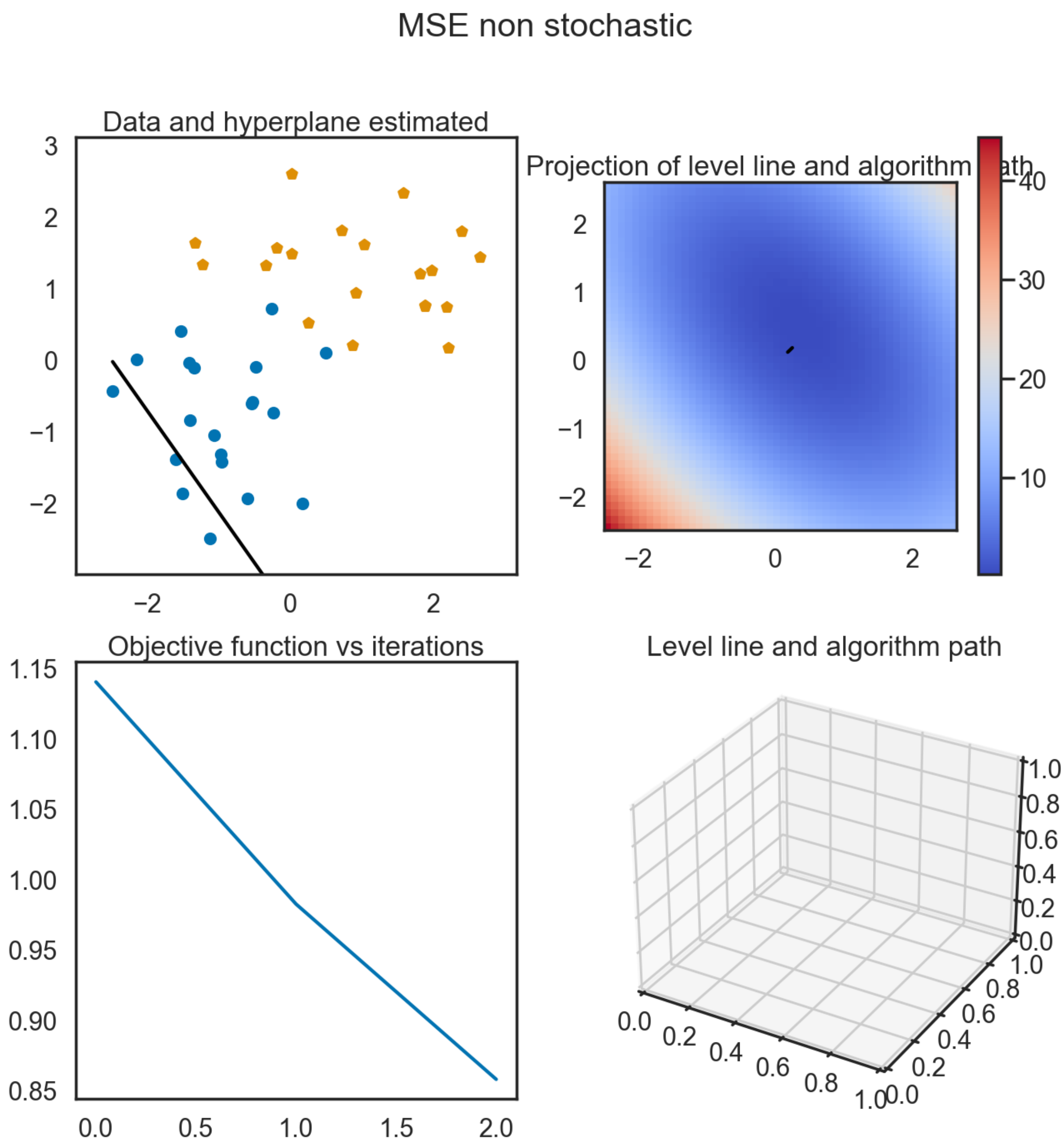

Hinge and stochastic
Data and hyperplane estimated
Projection of level line and algorithm path
Objective function vs iterations
Level line and algorithm path



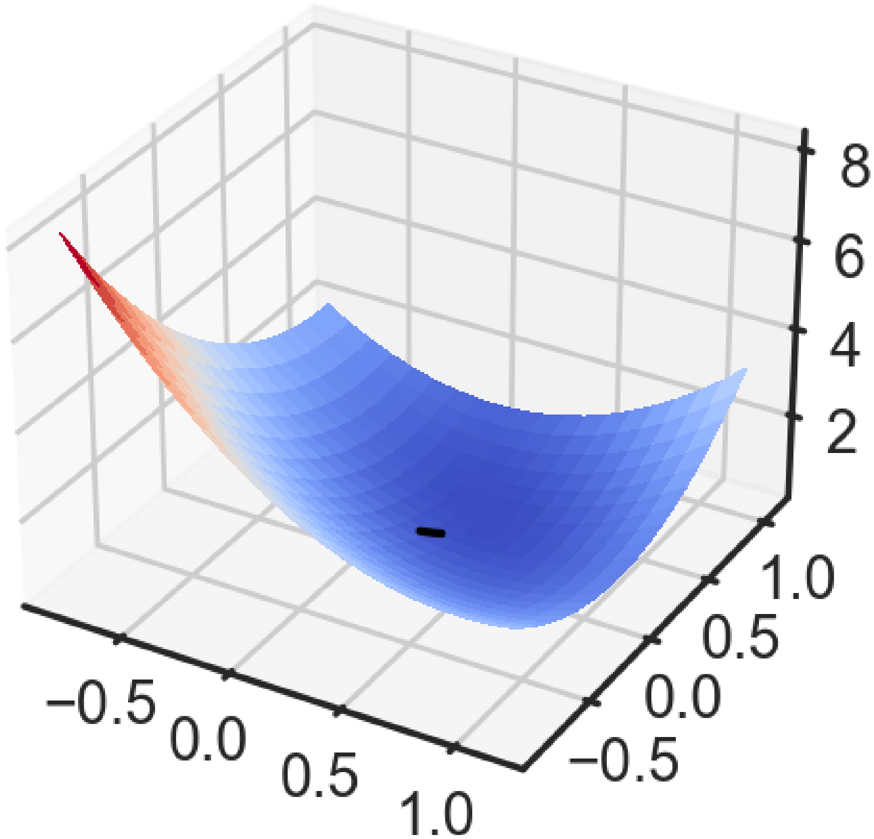
13) Study as before the behavior of the gradient descent algorithm, but with the option `stoch=True` disabled.

Code:</div>

```
[ ]: epsilon = 0.001
niter = 3
plt.figure(8, figsize=(15, 15))
plt.suptitle('MSE non stochastic')

wh, cost_hist = gradient(X, Y, epsilon, niter, w_ini, mse_loss,
 ↪gr_mse_loss,stochastic=False)
plot_gradient(X, Y, wh, cost_hist, mse_loss)
plt.show()
```



MSE non stochastic
Data and hyperplane estimated
Projection of level line and algorithm path
Objective function vs iterations
Level line and algorithm path



Bonus question: Implement the prediction function of the probabilistic perceptron seen in class, using the given `sigmoid` function. Then, implement the NLL-loss and its gradient as seen in class, and experiment with them similarly as before.

Code:</div>

[ ]:
```
# Careful, this is a naïve implementation that is not numerically stable
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def predict_probability(x, w):
    """Predict a probability to be in class 1 from at point x thanks to a␣
 ↪normal vector."""
    return # Complete here
```

[ ]:
```
def nll_loss(x, y, w):
    """Maximum likelihood estimation loss"""
    y = (y+1)/2 # We need 0/1 classes instead of -1/1 classes
    return # Complete here
```

```python
def gr_nll_loss(x, y, w):
    """Sub-gradient of the loss function hingeloss."""
    y = (y+1)/2
    return # Complete here
```

[ ]:
```python
epsilon = 0.05
niter = 3
plt.figure(8, figsize=(15, 15))
plt.suptitle('MLE and stochastic')

'''
Complete here
'''
plt.show()
```

```
<Figure size 1500x1500 with 0 Axes>
```

Adapted from Chloé Clavel and Matthieu Labeau's lab, all questions and suggestions should be sent to maria.boritchev@telecom-paris.fr .