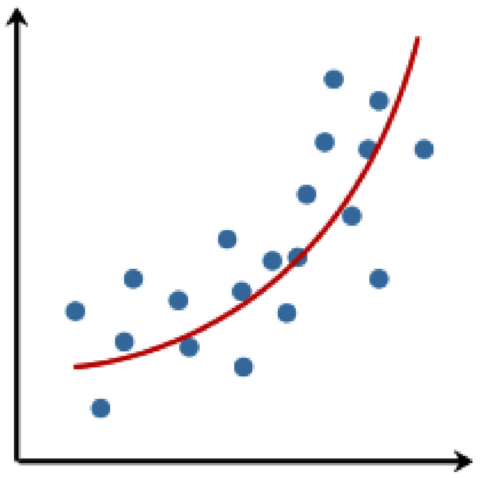


Stéphane GAZUT & Hanane SLIMANI

CEA, LIST, Laboratoire Instrumentation Intelligente, Distribuée et Embarquée (LIIDE)

CEA Saclay – Gif-sur-Yvette

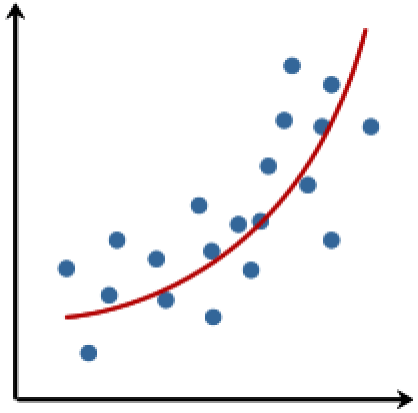
prenom.nom@cea.fr

TP N°2 - RÉGRESSION

MASTER SETI - INSTN

2023-2024

TP N°2 - RÉGRESSION



Objectifs du TP:

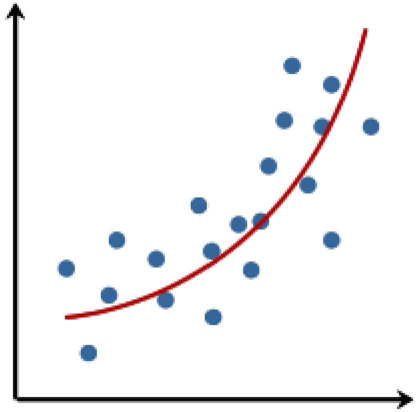
- On se propose de faire de la modélisation sur des problématiques de régression
- **Construire un modèle c'est très facile !!!**

CONSTRUIRE UN MODÈLE, EST-CE SI FACILE ?

- **Construire un modèle, c'est facile !!**
  - Choisir une famille de fonction: Régression Logistique, SVM, réseaux de neurones, régression linéaire…
  - Appeler la fonction *ad hoc* dans une librairie (sous R, python, Excel…)
  - Spécifier la matrice des entrées X, et le vecteur de sorties désirées Y
  - Appuyez sur « Entrée »
  - Récupérer les paramètres du modèle
  - …

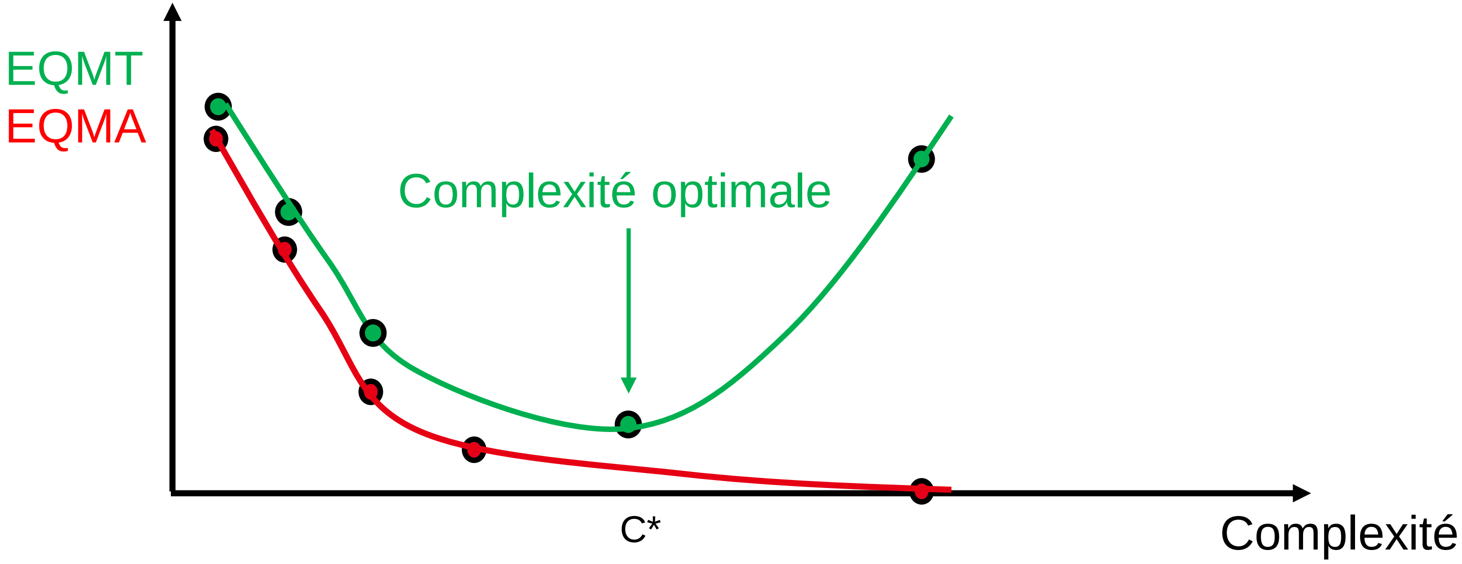
- **<u>La vraie question est</u>: comment construire un bon modèle et comment s'assurer que l'on construit un modèle optimal par rapport au jeu de données et à la famille de modèle utilisés …**

TP N°2 - RÉGRESSION



Objectifs du TP:

- On se propose de faire de la modélisation sur des problématiques de régression
- Objectif: Mettre en œuvre la méthodologie pour s'assurer que l'on construit un bon modèle avec deux familles de fonction:
  - Des modèles polynomiaux (modèles linéaires par rapport aux paramètres)
  - Des modèles de type réseaux de neurones MLP (non linéaires par rapport aux paramètres).



La complexité:

- Nombre de neurones en couche cachée pour un MLP
- Degré du polynôme pour un modèle polynomial

MODÈLES POLYNOMIAUX

- Les modèles polynomiaux, même s'ils peuvent décrire une relation non linéaire entre les entrées (x) et la sortie (y), **ils sont linéaires par rapport aux paramètres**.

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n$$

$$\hat{y} = \beta_0 h_0(x) + \beta_1 h_1(x) + \beta_2 h_2(x) + \cdots + \beta_n h_n(x) \quad \text{où} \;\; h_k(x) = x^k$$

$$\widehat{\boldsymbol{Y}} = \boldsymbol{H}\widehat{\boldsymbol{\beta}} \quad \text{où} \quad H \text{ est la matrice d'éléments } H_{ij} = h_j(x_i)$$

- On veut estimer les paramètres qui minimisent la fonction de coût des moindres carrés. La fonction de coût est:

$$\mathcal{L} = \left\|Y - \hat{Y}\right\|^2 = \left\|Y - H\hat{\beta}\right\|^2$$

- La fonction de coût est convexe et nous cherchons les paramètres tels que $\frac{\partial \mathcal{L}}{\partial \beta} = 0$

$$\frac{\partial \mathcal{L}}{\partial \beta} = 2H'\left(Y - H\hat{\beta}\right) = 0$$

$$H'Y - H'H\hat{\beta} = 0 \quad d'où \quad \widehat{\boldsymbol{\beta}} = (\boldsymbol{H}'\boldsymbol{H})^{-1}\boldsymbol{H}'\boldsymbol{Y}$$

MODÈLES POLYNOMIAUX

- La solution est unique et les paramètres du modèle polynomial sont obtenus par le calcul matriciel (1):

$$\widehat{\boldsymbol{\beta}} = (\boldsymbol{H'H})^{-1}\boldsymbol{H'Y}$$

- Supposons une application de R dans R, si nous souhaitons créer un modèle polynomial de degré 1 (une droite) à partir des données:

$$X = \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix}, \qquad H = \begin{pmatrix} 1 & x_1 \\ \dots & \dots \\ 1 & x_n \end{pmatrix}, \qquad Y = \begin{pmatrix} y_1 \\ \dots \\ y_n \end{pmatrix}$$

- Par le calcul (1), $\hat{\beta}$ sera un vecteur à deux composantes β_0 et β_1 qui définissent le modèle (la droite) d'équation $y = \beta_0 + \beta_1 x$

MODÈLES POLYNOMIAUX

- Même en changeant la complexité du modèle (degré du polynôme), les paramètres sont obtenus de la même manière:

$$\widehat{\boldsymbol{\beta}} = (\boldsymbol{H}'\boldsymbol{H})^{-1}\boldsymbol{H}'\boldsymbol{Y}$$

- C'est la matrice H qui change. Il faut ajouter des colonnes correspondant aux monômes de degrés supérieurs

$$X = \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix}, \qquad H = \begin{pmatrix} 1 & x_1 & x_1^2 \\ \dots & \dots & \dots \\ 1 & x_n & x_n^2 \end{pmatrix}, \qquad Y = \begin{pmatrix} y_1 \\ \dots \\ y_n \end{pmatrix}$$

- Dans ce cas, $\hat{\beta}$ sera un vecteur à trois composantes $\beta_0, \beta_1\ et\ \beta_2$ qui définissent le modèle de degré 2 d'équation $y = \beta_0 + \beta_1 x + \beta_2 x^2$

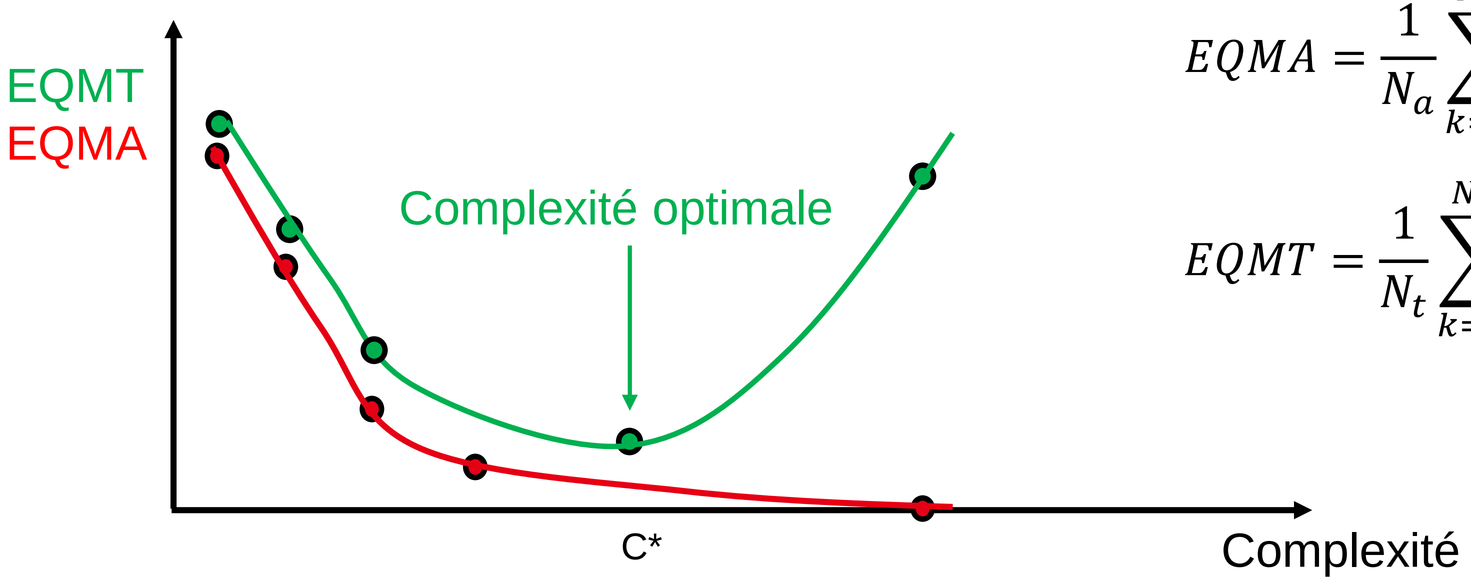
… et ainsi de suite …

MODÈLES POLYNOMIAUX

- **La question qui se pose est:**
  - Pour la famille de fonction des polynômes, quelle complexité doit-on choisir pour une base d'exemples donnée ?

- **C'est le but de cette première partie.**

- **Par rapport à une base de données fixée, trouver le modèle optimal en faisant varier la complexité.**



$$EQMA = \frac{1}{N_a}\sum_{k=1}^{N_a}(y_k - \hat{y}_k)^2$$

$$EQMT = \frac{1}{N_t}\sum_{k=1}^{N_t}(y_k - \hat{y}_k)^2$$

Mean Square Error calculée sur les données d'apprentissage et de test.

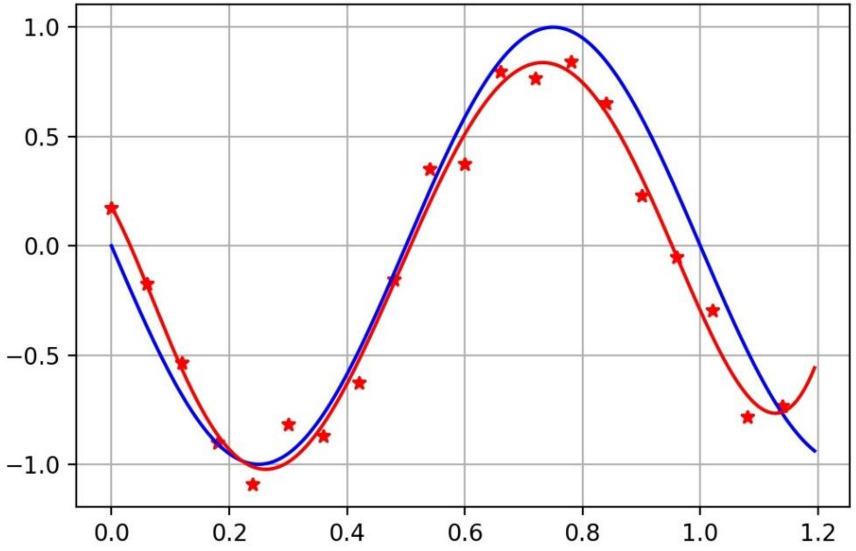
SCRIPT DU TP

```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import numpy as np


def genere_exemple_dim1(xmin, xmax, NbEx, sigma):
    x = np.arange(xmin, xmax, (xmax-xmin)/NbEx)
    y = np.sin(-np.pi + 2*x*np.pi) + np.random.normal(loc=0, scale=sigma, size=x.size)
    return x.reshape(-1,1), y


def getMSE(x, y, reg):
    return sum(pow(reg.predict(x)-y,2))/x.shape[0]


def plot_model(Xa, Ya, Xt, Yt, reg, nameFig):
    Ypred = reg.predict(Xt)
    plt.plot(Xa[:,1], Ya, '*r')
    plt.plot(Xt[:,1], Yt, '-b')
    plt.plot(Xt[:,1], Ypred, '-r')
    plt.grid()
    plt.savefig(nameFig+'.jpg', dpi=200)
    plt.close()


def plot_error_profile(L_error_app, L_error_test, nameFig):
    plt.plot(range(1, len(L_error_app)+1), L_error_app, '-r')
    plt.plot(range(1, len(L_error_test)+1), L_error_test, '-b')
    plt.grid()
    plt.savefig(nameFig+'.jpg', dpi=200)
    plt.close()


def plot_confusion(Xt, Yt, reg, nameFig):
    plt.plot(Yt, reg.predict(Xt), '.b')
    plt.plot(Yt, Yt, '-r')
    plt.savefig(nameFig+'.jpg', dpi=200)
    plt.close()
```

- **Voici:**
  - 1 fonction pour générer une base d'exemple bruitée en dimension 1
  - La fonction permettant de calculer l'erreur quadratique moyenne
  - Et 3 fonctions de visualisation

PLOT_MODEL

```
def plot_model(Xa, Ya, Xt, Yt, reg, nameFig):
    Ypred = reg.predict(Xt)
    plt.plot(Xa[:,1], Ya, '*r')
    plt.plot(Xt[:,1], Yt, '-b')
    plt.plot(Xt[:,1], Ypred, '-r')
    plt.grid()
    plt.savefig(nameFig+'.jpg', dpi=200)
    plt.close()
```



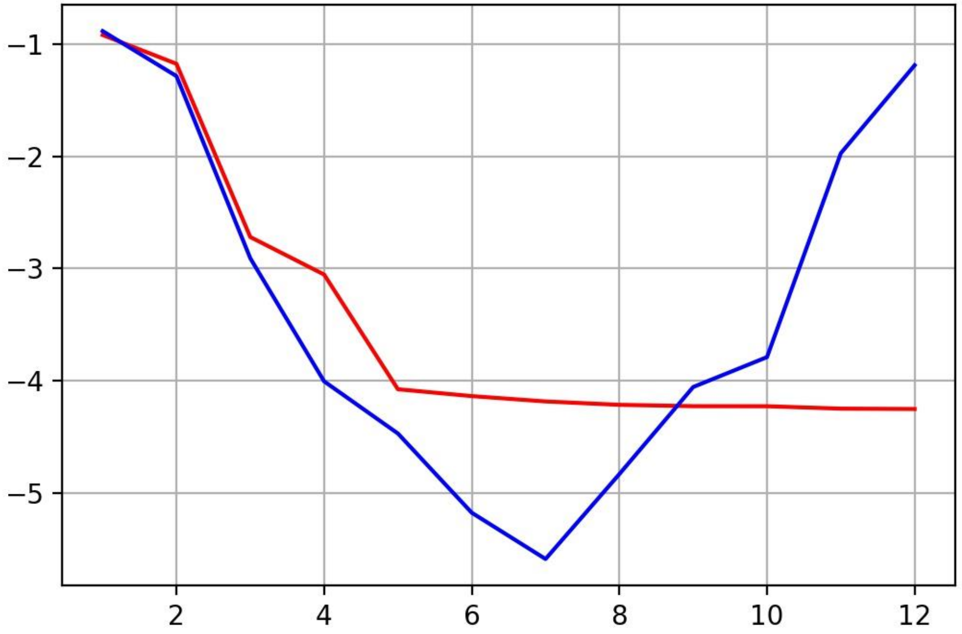
- **La fonction plot_model permettra d'avoir ce type de visualisation avec:**
  - Les points d'apprentissage de la base (étoiles rouges)
  - La vraie fonction recherchée (courbe bleue) portée par Xt (matrice X de test) et le Yt associé
  - La prédiction du modèle (courbe rouge), prédictions du modèle construit sur les données d'entrée Xt

  - Xa: Données d'entrée d'apprentissage
  - Ya: Sortie désirée des exemples contenus dans Xa
  - Xt: Données d'entrée de test
  - Yt: Sortie désirée des exemples contenus dans Xt
  - reg: le modèle de régression (obtenu par LinearRegression)

PLOT_ERROR_PROFILE

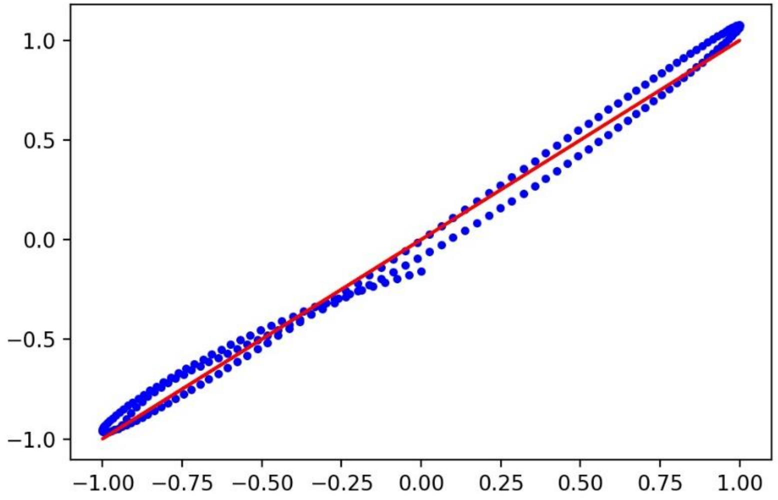
- **Va permettre de visualiser l’EQMA et l’EQMT.**

- **Vous pouvez utiliser une échelle log.**



PLOT_CONFUSION

```python
def plot_confusion(Xt, Yt, reg, nameFig):
    plt.plot(Yt, reg.predict(Xt), '.b')
    plt.plot(Yt, Yt, '-r')
    plt.savefig(nameFig+'.jpg', dpi=200)
    plt.close()
```



- Ce plot est intéressant, surtout lorsque l'on traite des problèmes en grande dimension qui ne permettent pas de visualiser la réponse du modèle.
- Il confronte le $Y_{\text{prédit}}$ avec le $Y_{\text{désiré}}$. Si le modèle est parfait alors $Y_{\text{prédit}} = Y_{\text{désiré}}$ et les couples de points $(Y_{\text{prédit}}, Y_{\text{désiré}})$ se trouvent sur la première bissectrice.
- Plus le modèle se dégrade et plus les points s'écartent autour de la première bissectrice.
- Ce type de graphe est obtenu quelle que soit la dimension du problème.

SCRIPT PRINCIPAL

Création de la base d'apprentissage (fonction avec ajout de bruit)

Création de la base de test (fonction sans bruit)

```
def main(degreMax=12, NbEx=20, sigma=0.2):
    xmin = 0
    xmax = 1.2

    xapp, yapp = genere_exemple_dim1(xmin, xmax, NbEx, sigma)
    xtest, ytest = genere_exemple_dim1(xmin, xmax, 200, 0)

    L_error_app = []
    L_error_test = []

    for i in range(1, degreMax+1):
        print("Degre = ", i)

        # Transformation des données d'entrée des bases d'app et de test
        [illegible]
        [illegible]
        [illegible]

        # Création du modèle linéaire
        reg = [illegible]

        # Estimation des erreurs d'apprentissage et de test
        L_error_app.append([illegible]
        L_error_test.append[illegible]

        # plot du model de degré i
        plot_model(Xa, yapp, Xt, ytest, reg, "Model_%02d" % i)


    # Déterminer le degré optimal
    best = np.argmin(L_error_test)+1
    print('Meilleur modele -> degre =', best)
    plot_error_profile(L_error_app, L_error_test, 'Profil_Err_App_Test')

    # Création du modèle final optimal
    [illegible]
    [illegible]
    [illegible]

    [illegible]


    plot_confusion(Xt, ytest, reg, 'Confusion')
```

Les modèles n'ont pas été gardés à chaque itération →
Création du modèle optimal pour le plot_confusion.
Non obligatoire si un plot_confusion est fait à chaque itération.

EXPLORATION

- Faites évoluer le nombre d’exemples
- Faites évoluer l’écart type du bruit sur les données

- Testez le modèle sur un ensemble de définition plus grand que celui des données d’apprentissage.

MODÉLISATION PAR RÉSEAUX DE NEURONES (MLP)

- Le principe est évidemment le même.
- Dans le cas d'un MLP, la solution n'est pas unique, il faut faire plusieurs initialisations pour chaque niveau de complexité.
- La décroissance de l'EQMT doit se faire sur la valeur moyenne des performances des modèles obtenus pour chaque niveau de complexité.

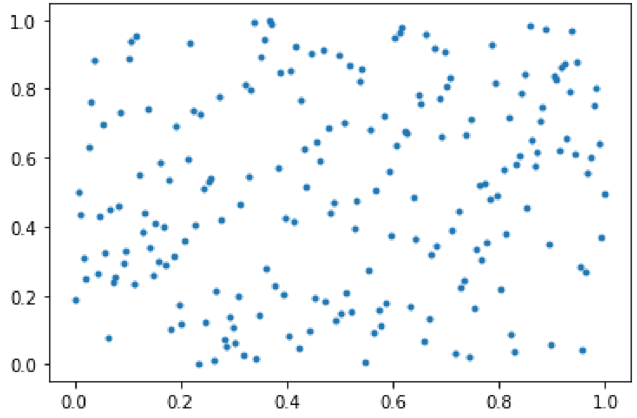
- *L'identification de la complexité optimale a été illustrée avec les modèles polynomiaux. Elle n'est pas indispensable pour cette deuxième partie du TP.*
- *Vous pourrez juste construire un modèle avec un nombre de neurones en couche cachée à déterminer qui permette d'obtenir un bon modèle même si ce n'est pas le meilleur.*

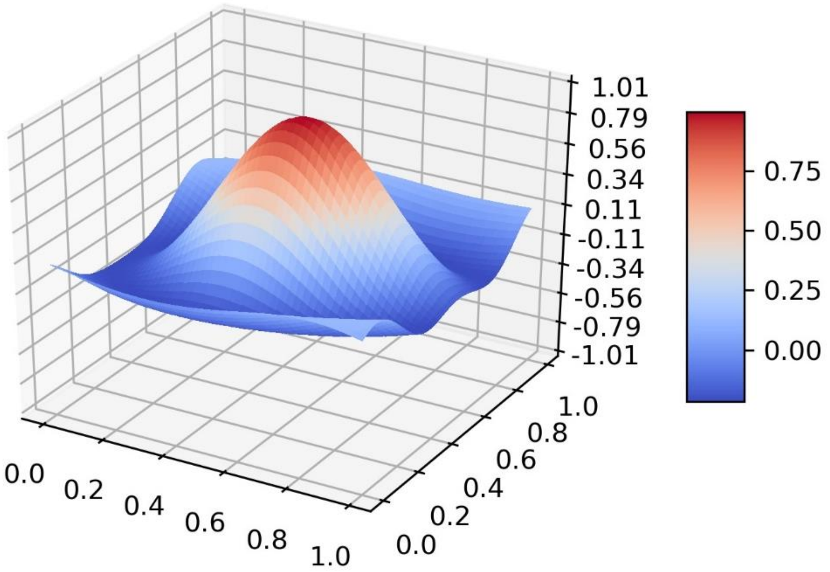
- Vous allez devoir construire un MLP en autonomie dans le cadre de la modélisation d'une surface de $R^2 \rightarrow R$.

MODÉLISATION PAR RÉSEAUX DE NEURONES (MLP)

- Comme pour un micro-projet:
  - Je vous mets à disposition des points dans R² (200 points) qui constitueront les inputs.
  - Une fonction permettant de calculer un sinus cardinal (sin(x)/x) dans R² → vous aurez ainsi la valeur y désirée à partir des entrées x. On souhaite donc modéliser la surface sinus cardinal en entrainant un MLP sur la base d'apprentissage de 200 points.
  - Cette fonction vous permettra également de générer une base de test. Par exemple en faisant un maillage de l'espace des entrées et en calculant le y désiré pour chaque point.
  - Vous devrez identifier les fonctions dans scikit-learn qui vous permettent de construire un MLP.
  - Vous pourrez visualiser le résultat du modèle en faisant une figure de la surface modélisée et le graphe de confusion.

```
In [5]: plt.plot(X[:,0], X[:,1], '.')
Out[5]: [&lt;matplotlib.lines.Line2D at 0x203d6119710&gt;]
```





QUELQUES ÉLÉMENTS

```python
with open(fileData, 'r') as f:
    f.readline()  # skip the header
    X = np.loadtxt(f, delimiter = ';')
```

Lecture des inputs X à partir de fileData (fichier csv mis à disposition).

```python
def sinus_cardinal(x):
    A = np.array([[1, 1], [-2, 1]])
    b = np.array([0.2, -0.3])

    x = -np.pi + 2*x*np.pi
    z = A.dot(x + b)
    h = np.sqrt(np.transpose(z).dot(z))
    if np.abs(h) < 0.001:
        y = 1
    else:
        y = np.sin(h)/h
    return y
```

Fonction permettant de calculer le Sinus Cardinal. Attention, x est ici un vecteur.

QUELQUES ÉLÉMENTS

```python
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np



def plot_surf(figName, regr=None):
    step_v = 0.005

    x1v = np.arange(0,1,step_v)
    x2v = np.arange(0,1,step_v)
    Xv, Yv = np.meshgrid(x1v, x2v)

    R = np.zeros(Xv.shape)
    for i,x1 in enumerate(x1v):
        for j,x2 in enumerate(x2v):
            if not regr:
                R[i,j] = sinus_cardinal(np.array([x1, x2]))
            else:
                R[i,j] = regr.predict(np.array([[x1, x2]]))[0]

    fig = plt.figure()
    ax = fig.gca(projection='3d')

    # Plot the surface.
    surf = ax.plot_surface(Xv, Yv, R, cmap=cm.coolwarm,
                           linewidth=0, antialiased=False)

    ax.set_zlim(-1.01, 1.01)
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

    fig.colorbar(surf, shrink=0.5, aspect=5)
    plt.savefig(figName, dpi=300)
    plt.close()
```

- La fonction plot_surf

STRUCTURE DE VOTRE SCRIPT PRINCIPAL

- Charger les inputs X → Xapp
- Calculer les Y associés → Yapp

- Créer Xtest (par exemple un maillage sur R^2). Avec un maillage (40 x 40) vous aurez 1600 points de test, ce qui est suffisant.
- Calculer les Y associés → Ytest

- Créer un modèle MLP en spécifiant le nombre de neurones en couche cachée en utilisant Xapp et Yapp.
- Calculer les prédictions du modèles sur les inputs Xtest → Ypred

- Visualiser les surfaces [Xtest, Ytest] et [Xtest, Ypred]
- Faire la visualisation (plot_confusion) de Ytest vs Ypred

Commissariat à l'énergie atomique et aux énergies alternatives
Institut List | CEA SACLAY NANO-INNOV | BAT. 861 – PC142
91191 Gif-sur-Yvette Cedex - FRANCE
www-list.cea.fr