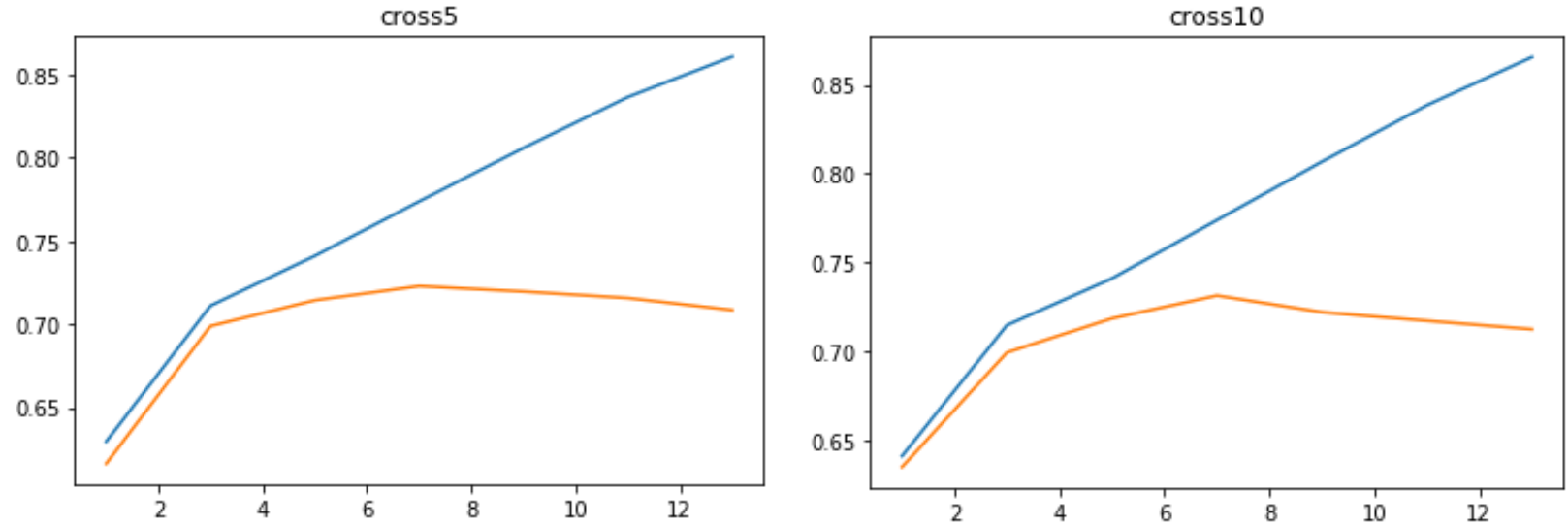
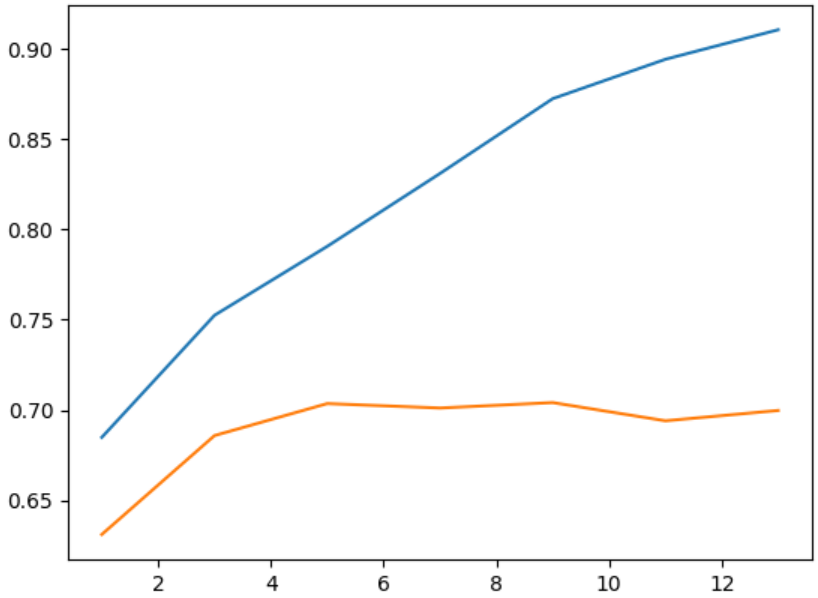
Compte Rendu de ARF
BEROUKHIM Keyvan

1) Arbre de décision, sélection de modèle



Performances en train (bleu) et test (orange) en fonction de la hauteur de l'arbre

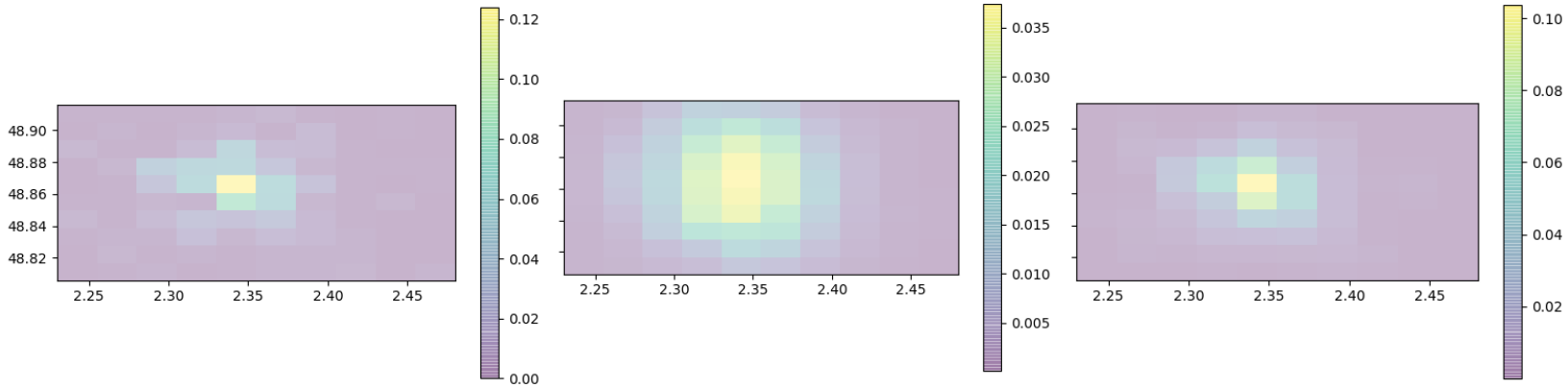
Les courbes ont été obtenues par **cross validation** sur 5 folds à gauche et 10 à droite pour plus de **précision**. On remarque qu'à partir d'une certaine profondeur, on commence à **sur-apprendre** les données d'apprentissage : les performances sur train augmentent mais les performances en test baissent. La **profondeur seuil varie** en fonction des instances et donc du choix de la partition en train et test.



Plus le **ratio** de données en train est **faible** (0.2 ici) **plus** le score obtenu en test est **fiable** mais **moins** le modèle appris est **performant**.

2.A) Estimation de densité

La **méthode des histogrammes** consiste à diviser l'espace de manière **statique** en un **quadrillage uniforme**. Les densités de probabilités sont alors estimées par la proportion de point dans chaque case. Le **facteur de discrétisation** détermine l'**expressivité du modèle** si on est en **sur/sous-apprentissage**. Le problème ne se pose pas ici car on est en 2D mais le **nombre de case** est **exponentiel** en le nombre de dimensions.
La **méthode à noyaux** est **dynamique** et pour un point donné, elle consiste à **moyenner les points** de l'ensemble d'apprentissage, **pondérés par leur similarité**. On peut utiliser différentes fonctions de similarité/distance, par exemple **Parzen** qui correspond à une similarité 1/0, ou une similarité **gaussienne**. Comme pour la méthode des histogrammes, on peut faire varier la distance seuil dans Parzen ou sigma dans gauss pour déterminer l'expressivité du modèle.



De gauche à droite : histogramme, Parzen et Gauss

Comme dans le TP précédent, on peut déterminer les paramètres optimums par cross-validation.

2.B) Classification

(non implémenté)
Si chaque point possède en plus un label binaire, on peut apprendre à classifier les points. On a vu deux modèles de régression (en prenant des étiquettes -1/1):
L'estimateur de **Watson-Nadaraya** (le dénominateur normalise les valeurs dans [-1, 1]) :

$$p(y_+|x) - p(y_-|x) = \frac{\sum_j y_j \phi(x - x_j)}{\sum_i \phi(x - x_i)}$$

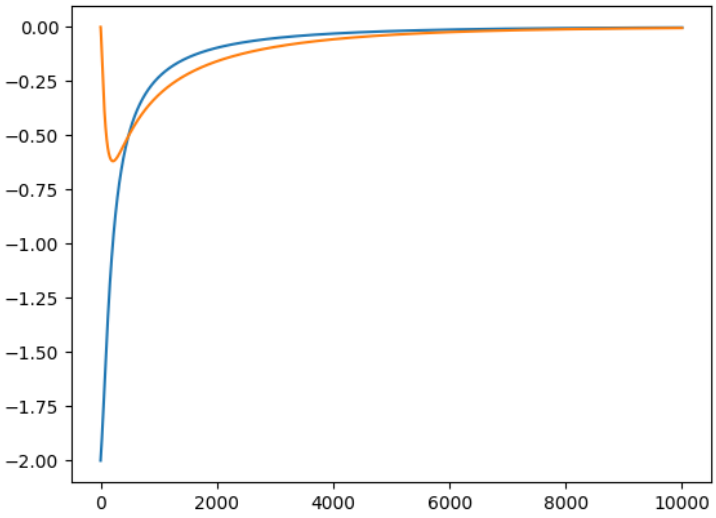
L'estimateur des **K Nearest Neighbours** :

$$\frac{1}{k} \sum_{j,\ x_j \in \{k\text{- plus proches}\}} y_j$$

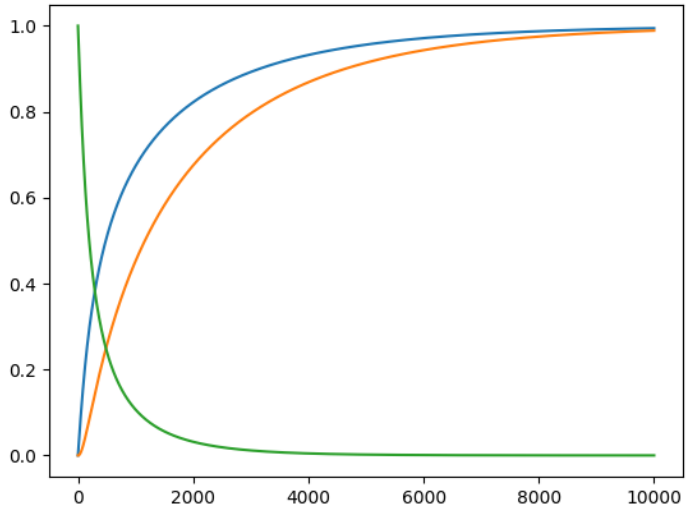
3.A) Descente de gradient

L'algorithme **Stochastic Gradient Descent** est un algorithme simple de descente de gradient, il consiste à chaque « epoch » à faire un pas dans la direction (ou l'opposé) du gradient multiplié par le coefficient de « learning rate ».

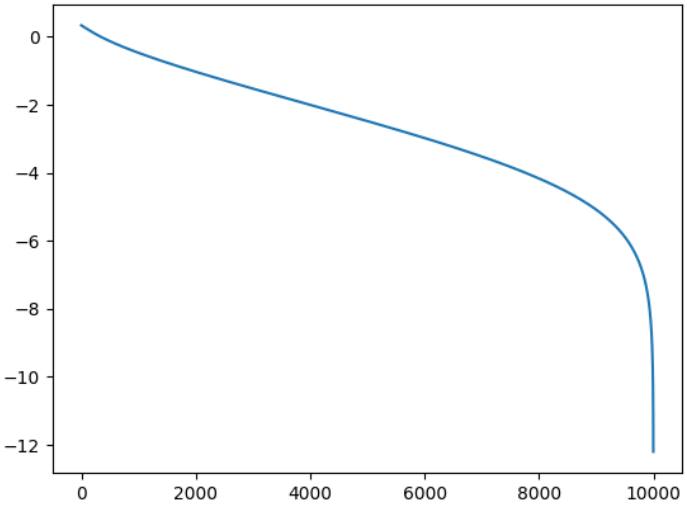
Afin de visualiser l'avancement de la convergence, les courbes suivantes sont toutes tracées en fonction de l'itération.



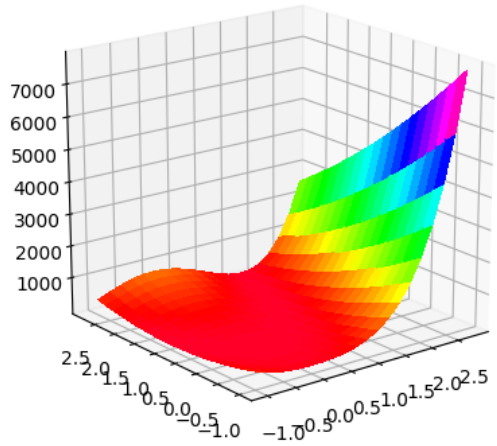
gradient selon x1 (bleu) et selon x2 (orange)
Le gradient tend vers 0 selon les deux axes.



valeur de x1 (bleu) x2 (orange) et f(x1,x2) en vert
La valeur de la fonction tend vers 0.
Le minimum est atteint en (1,1).



log de la distance entre x et x_{opt}

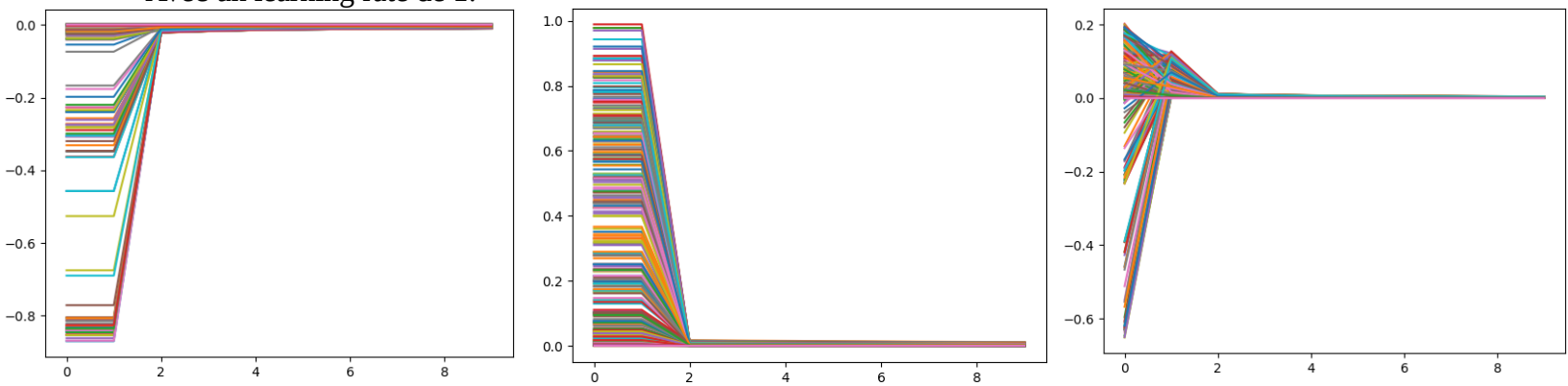


fonction de Rosenbrock

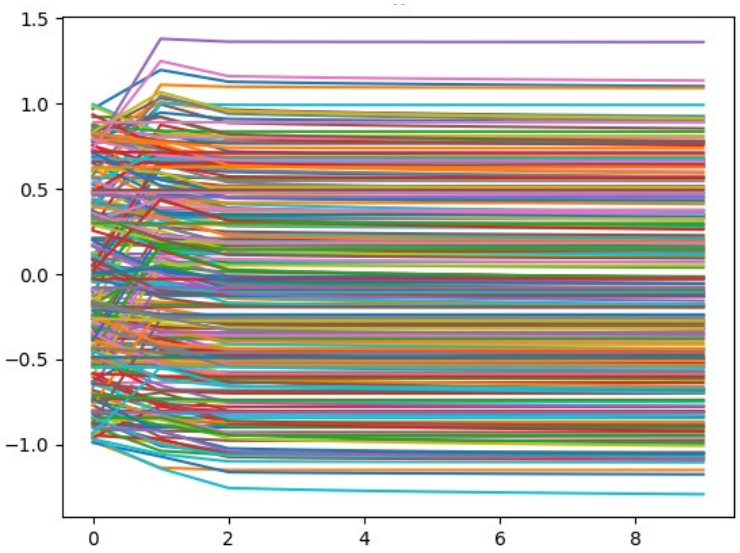
3.B) Régression logistique

Dans le TME 3 de MAPSI on classifiait par **maximum de vraisemblance**, on obtenait un score de **13.4 %** d'erreurs entre les classes '0' et '1'.
Avec la **régression logistique** on obtient **1 %** ! Ce meilleur score s'explique car on apprend ici à différencier les classes et non à représenter chacune d'entre elles séparément.

Avec un learning rate de 1:



Gradient selon chaque dimension en initialisant W à 0, 1 ou aléatoirement

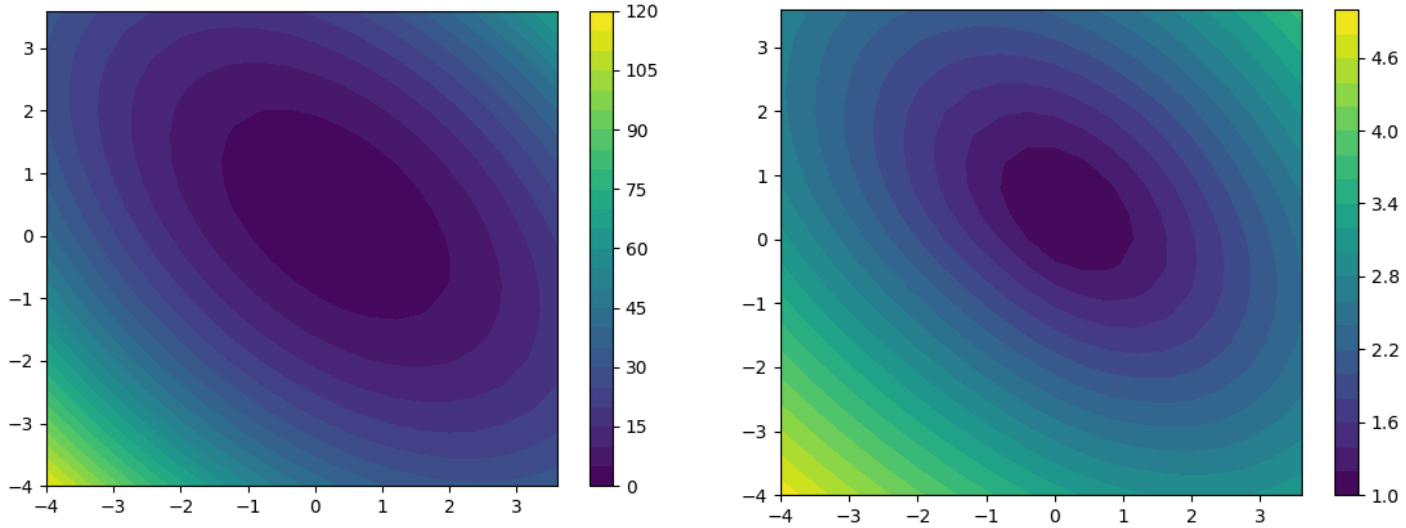


Valeurs de W correspondant à une initialisation aléatoire

L'algorithme converge très rapidement, quel que soit le learning rate ou le veteur de poids initial.

4.A) Régression linéaire : MSE (pour la classification)

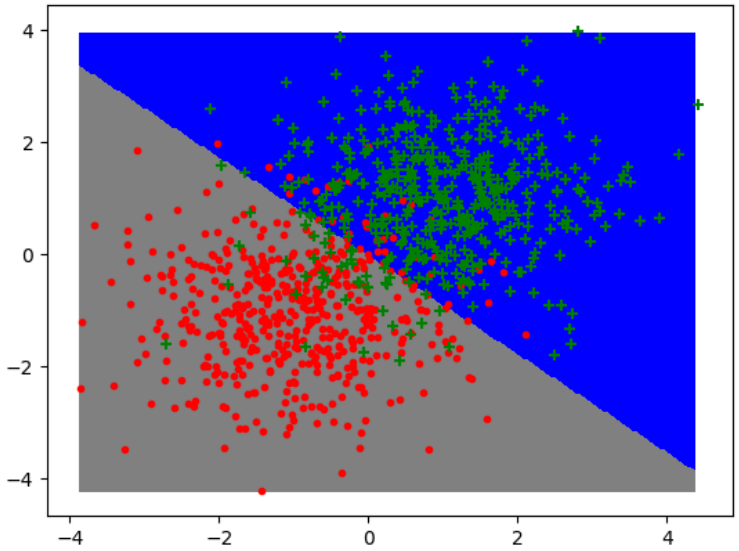
Comme prévu, la loss n'arrive pas à converger, elle tend vers l'infini même avec un learning rate très faible.



Les lignes de niveaux des deux fonctions correspondent,
mais MSE (à gauche) n'arrive pas à converger pour autant.

4.B) Régression linéaire : Perceptron

Sur la classification de chiffre (0 vs 1), le score obtenu est équivalent à celui de la régression logistique : **1 %** d'erreur.



Représentation graphique de la frontière de décision sur des données 2D.

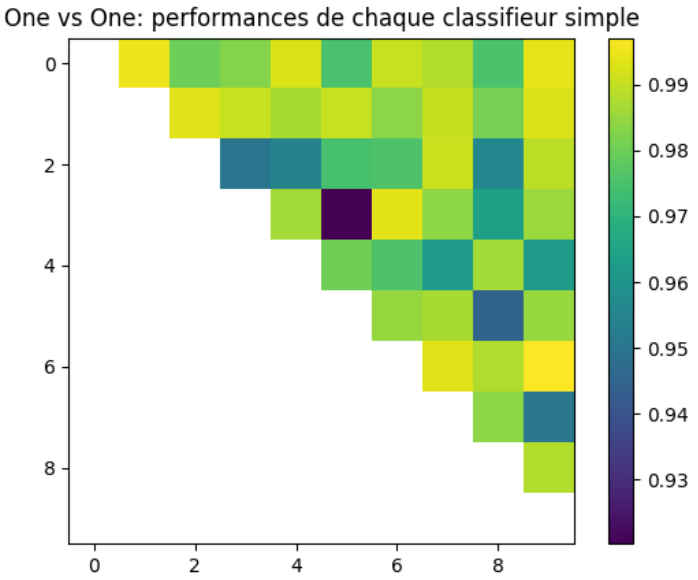
5.A) SVM, Grid Search par Cross Validation

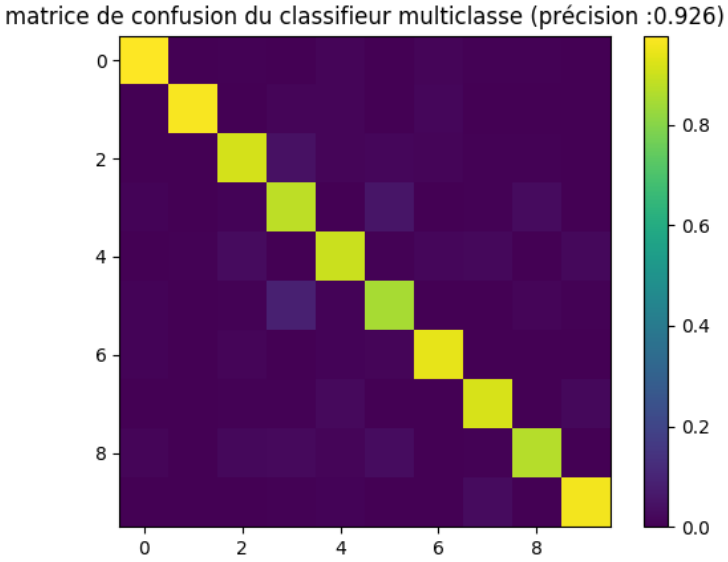
On fait varier plusieurs paramètres et on mesure les performances sur un **ensemble de validation** par cross-validation. Les meilleurs paramètres trouvés pour notre SVM ont ici été ‘kernel’: ‘linear’, ‘C’: 1, pour un score correspondant en **test** de 99.0 %.

5.B) SVM, Multiclasse

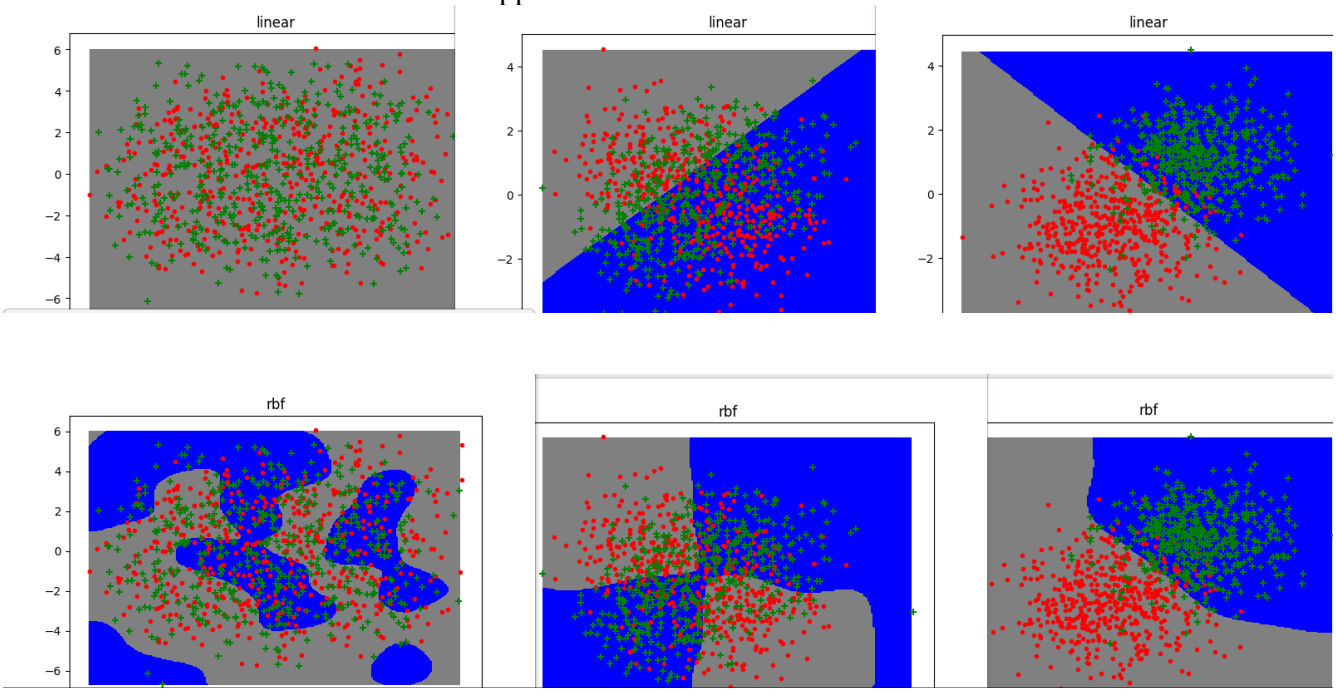
Pour la technique du **‘one vs one’** on entraîne n(n-1)/2 classifieurs binaires (un pour chaque couple). On les combine en renvoyant la classe majoritairement retournée par les classifieurs. On obtient une précision de **92 %** environ, c’est inférieur à de la classification binaire mais reste un bon score.

Pour la technique du **‘one vs rest’** on entraîne n classifieurs binaires. On les combine en renvoyant la classe la plus vraisemblable. On obtient une précision équivalente à celle obtenue par ‘one-vs-one’.





On observe les frontières de décision apprises sur différents dataset et avec un kernel linéaire ou rbf.



Code :
tme1

```
import  pickle
import  numpy as np
import matplotlib.pyplot as plt
from  decisiontree  import  DecisionTree
# data : tableau (films ,features), id2titles : dictionnaire  id -> titre ,
# fields : id  feature  -> nom
[data , id2titles , fields ]= pickle.load(open("imdb_extrait.pkl","rb"))
# la  derniere  colonne  est le vote
datax=data [: ,:32]
datay=np.array ([1 if x[33] >6.5  else  -1 for x in data])
tot = len(data)
profondeurs = range(1, 15, 2)
def scoreTrain():
    scores = []
    for depth in profondeurs:
        dt = DecisionTree(depth)
        dt.fit(datax ,datay)
        #dt.predict(datax [:5 ,:])
        scores.append(dt.score(datax ,datay))
        # dessine l'arbre  dans un  fichier  pdf   si pydot  est  installe.
        #dt.to_pdf("/tmp/test_tree.pdf",fields)
        # sinon  utiliser  http :// www.webgraphviz.com/
        #print(dt.to_dot(fields))
        #ou dans la  console
        #print(dt.print_tree(fields ))
    return scores
def scoreTrainTest(f:float):
    assert(f>0 and f<=1)
    l = int(tot*f)
    scoresTrain = []
    scoresTest = []
    for depth in profondeurs:
        dt = DecisionTree(depth)
        dt.fit(datax[:l] ,datay[:l])
        scoresTrain.append(dt.score(datax[:l] ,datay[:l]))
        scoresTest.append(dt.score(datax[l:] ,datay[l:]))
    return scoresTrain, scoresTest
def scoreCross(n=5):
    """fait la moyenne sur n tests
    taille test = tot/n"""
    assert(type(n) == int)
    scoresTrain = []
    scoresTest = []
    for depth in profondeurs:
        sTrain=0
        sTest=0
        for i in range(n):
            start = tot*i//n
            end = tot*(i+1)//n
            dt = DecisionTree(depth)
            xtrain = np.vstack((datax[:start],datax[end:]))
            ytrain = np.hstack((datay[:start],datay[end:]))
            dt.fit(xtrain ,ytrain)
            sTrain += dt.score(xtrain ,ytrain)
            sTest += dt.score(datax[start:end] ,datay[start:end])
        scoresTrain.append(sTrain/n)
        scoresTest.append(sTest/n)
    return scoresTrain, scoresTest
```

```
#plt.plot(r, scoreTrain())
for f in [.8, .5, .2]:
    train, test = scoreTrainTest(f)
    plt.plot(profondeurs, train)
    plt.plot(profondeurs, test)
    plt.title("trainTest"+str(f))
    plt.show()

for n in [2, 5, 10]:
    train, test = scoreCross(n)
    plt.plot(profondeurs, train)
    plt.plot(profondeurs, test)
    plt.title("cross"+str(n))
    plt.show()
```

tme2

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import pickle
parismap = mpimg.imread('data/paris-48.806-2.23--48.916-2.48.jpg')
## coordonnees GPS de la carte
xmin,xmax = 2.23,2.48   ## coord_x min et max
ymin,ymax = 48.806,48.916 ## coord_y min et max
def show_map():
    #extent pour controler l'echelle du plan
    plt.imshow(parismap,extent=[xmin,xmax,ymin,ymax],aspect=1.5)
poidata = pickle.load(open("data/poi-paris.pkl","rb"))
## liste des types de point of interest (poi)
print("Liste des types de POI" , ", ".join(poidata.keys()))
## Choix d'un poi
typepoi = "night_club"
## Creation de la matrice des coordonnees des POI
geo_mat = np.zeros((len(poidata[typepoi]),2))
for i,(k,v) in enumerate(poidata[typepoi].items()):
    geo_mat[i,:]=v[0]
## Affichage brut des poi
show_map()
## alpha permet de regler la transparence, s la taille
plt.scatter(geo_mat[:,1], geo_mat[:,0],alpha=0.8,s=3)
#########################################################
# discretisation pour l'affichage des modeles d'estimation de densite
steps = 10
xx,yy = np.meshgrid(np.linspace(xmin,xmax,steps),np.linspace(ymin,ymax,steps))
grid = np.c_[xx.ravel(),yy.ravel()]
# plt.show()
plt.close('all')
class ModelHisto:
    def fit(self, points, step):
        self.points = points
        self.step = step
        self.n = len(self.points)

    def predict(self, _grid):
        #ne se sert pas de la grille pour simplifier
        plt.figure("figureHistoTmp")
        h = plt.hist2d(self.points[:, 0], self.points[:, 1], bins=self.step)[0]
        h = h / self.n
        plt.close("figureHistoTmp")
        return h
def similarite_parzen(x1, x2, width):
    diff = x1[0] - x2[0], x1[1] - x2[1]
    norm = diff[0] ** 2 + diff[1] ** 2
    return 1/width**2 if norm < width**2 else 0
def similarite_gauss(x1, x2, sigma_2):
    diff = x1[0] - x2[0], x1[1] - x2[1]
    norm_2 = diff[0] ** 2 + diff[1] ** 2
    return 1/np.sqrt(2*np.pi*sigma_2) * np.exp(-norm_2/(2*sigma_2))
class ModelNoyau:
    def fit(self, points, distance):
        self.points = points
        self.n = len(self.points)
        self.similarite = distance

    def predict(self, grid):
        #x et y inversés
```

```
        res = np.array([self.predictPoint(y,x) for (x,y) in grid])
        return res / np.sum(res)

    def predictPoint(self, x, y):
        # parcourt tous les points de la base d'apprentissage
        s = sum(self.similarite((x, y), (px, py)) for (px, py) in self.points)
        return s / self.n
def _main():
    modelHisto = ModelHisto()
    modelHisto.fit(geo_mat, steps)

    modelParzen = ModelNoyau()
    modelParzen.fit(geo_mat, lambda x1, x2: similarite_parzen(x1, x2, .05))

    modelGauss = ModelNoyau()
    modelGauss.fit(geo_mat, lambda x1,x2: similarite_gauss(x1, x2, .0001))

    for model in [modelHisto, modelParzen, modelGauss]:
        res = model.predict(grid).reshape(steps,steps)
        plt.figure()
        plt.imshow(res, extent=[xmin,xmax,ymin,ymax], interpolation='none', alpha=0.3,
origin="lower")
        plt.colorbar()
    plt.show()
if __name__ == '__main__':
    _main()
```

tme3

```
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
def make_grid(xmin=-5, xmax=5, ymin=-5, ymax=5, step=20, data=None):
    """ Cree une grille sous forme de matrice 2d de la liste des points
    :return: une matrice 2d contenant les points de la grille, la liste x, la liste y"""
    if data is not None:
        xmax, xmin, ymax, ymin = np.max(data[:, 0]), np.min(data[:, 0]), \
                                 np.max(data[:, 1]), np.min(data[:, 1])
    x, y = np.meshgrid(np.arange(xmin, xmax, (xmax - xmin) * 1. / step),
                       np.arange(ymin, ymax, (ymax - ymin) * 1. / step))
    grid = np.c_[x.ravel(), y.ravel()]
    return grid, x, y
def draw_2D(f):
    grid, xx, yy = make_grid(-1, 3, -1, 3, 20)
    plt.figure()
    fgrid = np.array([f(x) for x in grid])
    plt.contourf(xx, yy, fgrid.reshape(xx.shape))

    fig = plt.figure()
    ax = fig.gca(projection='3d')
    surf = ax.plot_surface(xx, yy, fgrid.reshape(xx.shape), rstride=1, cstride=1,
                           cmap=cm.gist_rainbow, linewidth=0, antialiased=False)
    fig.colorbar(surf)
    plt.show()
def draw_1D(f, x_ini, n=100, eps=1e-4):
    x, f, grad = optimize(f[0], f[1], x_ini, eps, n)
    plt.figure("g")
    plt.plot(grad)
    plt.figure("xi & f(xi)")
    plt.plot(range(n), x, f)
    plt.figure("log(err)")
    plt.plot([np.log(np.linalg.norm(x[i] - x[-1])) for i in range(n)])
    plt.show()
def optimize(fonc, dfonc, xinit, eps, max_iter):
    """return (x_list, f_list, grad_list)"""
    xinit = np.array(xinit)
    x_list, f_list, grad_list = [], [], []
    x = xinit
    for _ in range(max_iter):
        x_list.append(x.copy())

        f_list.append(fonc(x))

        grad = np.array(dfonc(x))

        grad_list.append(grad)

        x -= eps * grad

    return np.array(x_list), np.array(f_list), np.array(grad_list)
def xcosx():
    """non convexe"""

    def f1d_1_val(x):
        return [x[0] * np.cos(x[0])]
```

```
    def f1d_1_grad(x):
        return [np.cos(x[0]) - np.sin(x[0]) * x[0]]

    return f1d_1_val, f1d_1_grad
def x2_minus_logx():
    """convexe"""

    def f1d_2_val(x):
        return [-np.log(x[0]) + x[0] ** 2]

    def f1d_2_grad(x):
        return [- 1 / x[0] + 2 * x[0]]

    return f1d_2_val, f1d_2_grad
def fRosenbrock():
    """Rosenbrock performance test problem for optimization algorithms"""

    def f2d_1_val(x):
        return 100 * (x[1] - x[0] ** 2) ** 2 + (1 - x[0]) ** 2

    def f2d_1_grad(x):
        return [100 * (-2 * x[0] * (2 * (x[1] - x[0] ** 2))) - 2 * (1 - x[0]),
                100 * 2 * (x[1] - x[0] ** 2)]

    return [f2d_1_val, f2d_1_grad]
def load_usps(filename):
    with open(filename, "r") as f:
        f.readline()
        data = [[float(x) for x in l.split()] for l in f if len(l.split()) > 2]
    tmp = np.array(data)
    return tmp[:, 1:], tmp[:, 0].astype(int)
class Logistic:
    def __init__(self):
        # w0 stocke b
        self.W = None

    @staticmethod
    def _loss(W, X, Y):
        W, b = W[1:], W[0]
        return sum(np.log(1 + np.exp(-(2 * Y[i] - 1) * (X[i].dot(W) + b)))
                   for i in range(len(Y))) / len(X)

    @staticmethod
    def _grad_loss(W, X, Y):
        W, b = W[1:], W[0]

        cache = np.array([-(2 * Y[j] - 1) / (1 + np.exp((2 * Y[j] - 1) * (X[j].dot(W) +
b))) for j in range(len(X))])

        grad_w = [np.sum(cache * X[:, i]) for i in range(len(W))]

        grad_b = np.sum(cache)
        grad = np.array([grad_b] + grad_w) / len(X)
        return grad

    def fit(self, datax, datay, eps, max_iter):
        w, list_f, grad = optimize(fonc=lambda x: self._loss(x, datax, datay),
                                   dfonc=lambda x: self._grad_loss(x, datax, datay),
                                   # zeros ou random normalement, tout marche ici
```

```
                            xinit=np.random.random(datax.shape[1] + 1) * 2 - 1, # rajoute
un nombre pour représenter b
                            eps=eps, max_iter=max_iter)
        plt.figure(); plt.title("w"); plt.plot(w[2:])
        plt.figure(); plt.title("f"); plt.plot(list_f[2:])
        plt.figure(); plt.title("g"); plt.plot(grad[2:])
        plt.show()
        self.W = w[-1]
        return list_f

    def predict(self, datax):
        """maximum de vraisemblance, compare
        P(y==1|x) = 1/1+e-wx+b    à   1 - // """
        W, b = self.W[1:], self.W[0]
        pred = 1 / (1 + np.exp(-(datax.dot(W) + b)))
        return np.where(pred < .5, 0, 1)

    def score(self, datax, datay):
        """return nb de bonne classifications"""
        return np.sum(self.predict(datax) == datay)
def main():
    # draw_1D(x2_minus_logx(), x_ini=[10.], eps=1e-1, n=100)
    # draw_1D(fRosenbrock(), x_ini=[0.,0.], eps=1e-3, n=10000)
    # draw_2D(fRosenbrock()[0])

    c0, c1 = 0, 1
    trainx, trainy = load_usps("USPS_train.txt")
    ind_kept = np.where((trainy==c0) | (trainy==c1))[0]
    trainx = trainx[ind_kept]
    trainy = trainy[ind_kept]

    l = Logistic()
    print("fitting")
    losses = l.fit(trainx, trainy, eps=1e0, max_iter=1000)
    print("fitted")
    print("losses:", losses) # log(1 + e(x)) donne des inf au lieu de x
    plt.show()

    testx, testy = load_usps("USPS_test.txt")
    ind_kept = np.where((testy==c0) | (testy==c1))[0]
    testx = testx[ind_kept]
    testy = testy[ind_kept]
    score = l.score(testx, testy)
    print("score", score, "/", len(testy), "=", score/len(testy))
if __name__ == '__main__':
    main()
```

tme4

```
import matplotlib.pyplot as plt
import numpy as np
from tME3 import tme3
import sys
sys.path.append('./utils/')
from utils import arftools
def mse(datax, datay, w):
    """ retourne la moyenne de l'erreur aux moindres carres """
    return np.mean(np.square(datax.dot(w.T) - datay))
def mse_g(datax, datay, w, biais=0.0):
    """ retourne le gradient moyen de l'erreur aux moindres carres """
    return - 2 * np.mean(datax.T.dot(datay - datax.dot(w.T) + biais), axis=0)
def hinge(datax, datay, w):
    """ retourne la moyenne de l'erreur hinge """
    return np.mean(np.maximum(0, datay - datax.dot(w) + 1))
def hinge_g(datax, datay, w, biais=0.0):
    """ retourne le gradient moyen de l'erreur hinge """
    x = datax.dot(w.T) + biais
    x = np.multiply(datay, x)
    return np.mean(np.where(x < 1, - np.multiply(datay, datax), 0), axis=0)
class Lineaire(object):
    def __init__(self, loss=hinge, loss_g=hinge_g, use_biais=False, max_iter=1000, eps=1e-
3):
        """ :loss: fonction de cout
            :loss_g: gradient de la fonction de cout
            :max_iter: nombre d'iterations
            :eps: pas de gradient
        """
        self.max_iter = max_iter
        self.eps = eps
        self.loss, self.loss_g = loss, loss_g
        self.w = None
        self.use_biais = use_biais
        self.b = None if use_biais else 0.0

    def fit(self, datax, datay, log=None, testx=None, testy=None):
        """ :datax: donnees de train
            :datay: label de train
            :testx: donnees de test
            :testy: label de test
        """
        # on transforme datay en vecteur colonne
        datay = datay.reshape(-1, 1)
        N = len(datay)
        datax = datax.reshape(N, -1)
        D = datax.shape[1]

        if testx is not None:
            testy = testy.reshape(-1, 1)
            testx = testx.reshape(len(testy), D)

        self.w = np.random.random((1, D))

        if self.use_biais:
            self.b = np.random.random()

        list_score_train, list_score_test = None, None
        if log:
            list_score_train = []
```

```
            if testx is not None:
                list_score_test = []

        #todo dérivée du biais sans l'intégrer dans w ?

        for epoch in range(self.max_iter):
            grad = self.loss_g(datax, datay, self.w, biais=self.b)
            self.w -= self.eps * grad

            if log and epoch in log:
                if list_score_train is not None:
                    list_score_train.append(self.score(datax, datay))
                if list_score_test is not None:
                    list_score_test.append(self.score(testx, testy))

            #if self.b:
            #    self.b -= grad[1]

        return list_score_train, list_score_test

    def predict(self, datax):
        if len(datax.shape) == 1:
            datax = datax.reshape(1, -1)
        return np.sign(datax.dot(self.w.T)) # dot renvoie shape (1,1)


    def score(self, datax, datay):
        return sum(self.predict(x)[0][0] == y for (x, y) in zip(datax, datay.reshape(-
1))) / len(datay)
def show_usps(data):
    plt.imshow(data.reshape((16, 16)), interpolation="nearest", cmap="gray")
    plt.colorbar()
    plt.show()
def plot_error(datax, datay, f):
    grid, x1list, x2list = arftools.make_grid(xmin=-4, xmax=4, ymin=-4, ymax=4)
    plt.contourf(x1list, x2list, np.array([f(datax, datay, w) for w in
grid]).reshape(x1list.shape), 25)
    plt.colorbar()
    plt.show()
def _main_presque_sep():
    trainx, trainy = arftools.gen_arti(nbex=1000, data_type=0, epsilon=1)
    testx, testy = arftools.gen_arti(nbex=1000, data_type=0, epsilon=1)
    print("MSE (pas fait pour la classification)")
    plt.figure()
    plot_error(trainx, trainy, mse)

    epochs = 1000
    plotted = range(0, epochs, epochs//10)
    carres = Lineaire(mse, mse_g, max_iter=epochs, eps=0.1)
    err_train, err_test = carres.fit(trainx, trainy, plotted, testx, testy)
    # print(err_train)
    # print(err_test)
    print("w (diverge) : ", carres.w)
    print("Score : train %f, test %f" % (carres.score(trainx, trainy), carres.score(testx,
testy)))
    plt.plot(plotted, err_train)
    plt.plot(plotted, err_test)
    plt.close('all')
```

```
    print("Hinge")
    plt.figure()
    plot_error(trainx, trainy, hinge)

    epochs = 1000
    plotted = range(0, epochs, epochs//10)
    perceptron = Lineaire(hinge, hinge_g, max_iter=epochs, eps=0.1)
    err_train, err_test = perceptron.fit(trainx, trainy, plotted, testx, testy)
    print("w:", perceptron.w)
    print("Score : train %f, test %f" % (perceptron.score(trainx, trainy),
perceptron.score(testx, testy)))
    plt.plot(plotted, err_train)
    plt.plot(plotted, err_test)

    plt.figure()
    arftools.plot_frontiere(trainx, perceptron.predict, step=200)
    arftools.plot_data(trainx, trainy)
    plt.show()
def _main_usps():
    print("\nUSPS (hinge) 0 vs 1")
    train = tme3.load_usps("../tME3/USPS_train.txt")
    test  = tme3.load_usps("../tME3/USPS_test.txt")

    # extrait seulement deux classes
    id_c0, id_c1 = 0, 1
    kept_train = np.where((train[1] == id_c0) | (train[1] == id_c1))[0]
    kept_test  = np.where((test [1] == id_c0) | (test [1] == id_c1))[0]
    train = [train[0][kept_train], train[1][kept_train]]
    test  = [test [0][kept_test ], test [1][kept_test ]]
    # met les étiquettes -1/1
    train[1] = np.where(train[1] == id_c0, -1, 1)
    test [1] = np.where(test [1] == id_c0, -1, 1)

    epochs = 100
    plotted = range(0, epochs, epochs//10)
    perceptron = Lineaire(hinge, hinge_g, max_iter=epochs, eps=1e0)
    err_train, err_test = perceptron.fit(train[0], train[1], log=plotted)
    # print("err_train:", err_train)
    plt.plot(plotted, err_train)
    plt.show()
    print("Score : train %f, test %f" % (perceptron.score(train[0], train[1]),
perceptron.score(test[0], test[1])))

def _main():
    _main_presque_sep()

    # _main_usps()

if __name__ == "__main__":
    _main()
```

tme5

```
import numpy as np
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
from sklearn import model_selection
from sklearn import metrics
import matplotlib.pyplot as plt
import tme3
import sys
sys.path.append('./utils/')
from utils import arftools
all_train = tme3.load_usps("../tME3/USPS_train.txt")
all_test = tme3.load_usps("../tME3/USPS_test.txt")
def _main_biclass():
    print("SVM (grid search - cross val)")
    # extrait seulement deux classes
    id_c0, id_c1 = 1, 7
    kept_train = np.where((all_train[1] == id_c0) | (all_train[1] == id_c1))[0]
    kept_test = np.where((all_test[1] == id_c0) | (all_test[1] == id_c1))[0]
    train = [all_train[0][kept_train], all_train[1][kept_train]]
    test = [all_test[0][kept_test], all_test[1][kept_test]]
    # met les étiquettes -1/1
    train[1] = np.where(train[1] == id_c0, -1, 1)
    test[1] = np.where(test[1] == id_c0, -1, 1)
    svc = SVC()

    # x_train, x_test, y_train, y_test = model_selection.train_test_split(x,y,test_size=.2)
    # svc.fit(x_train,y_train)
    # score = svc.score(x_test, y_test)
    #
    # skf = model_selection.StratifiedKFold(n_splits=3)
    # for train_ind, test_ind in skf.split(x,y):
    #     x_train, x_test = x[train_ind], y[test_ind]
    #     y_train, y_test = y[train_ind], y[test_ind]
    #
    # scores = model_selection.cross_val_score(svc, x, y, cv=3)

    parameters = [
        {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
        {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']}]
    clf = model_selection.GridSearchCV(svc, parameters, cv=3)
    clf.fit(train[0], train[1])
    print("best params", clf.best_params_, "score validation", clf.best_score_)
    score_test = clf.score(test[0], test[1])
    print("score test", score_test)
class PredicteurOneVsOne:
    def __init__(self, predicteurs):
        self.predicteurs = predicteurs

    def predict(self, X):
        res = []
        for x in X:
            mat_scores = np.array([f.predict(x.reshape((1,-1))) if f is not None else 0
                                   for f in self.predicteurs.reshape(-
1)]).reshape((len(self.predicteurs), -1))
            list_class_score = []
            for i in range(len(self.predicteurs)):
                # la matrice est triangulaire supérieure sans diagonale
                # compte le nombre de vote de notre classe contre les autres
                class_score = - np.sum(mat_scores[i,i+1:])
```

```
                # compte le nombre de vote des autres classes contre la notre
                class_score += np.sum(mat_scores[:i,i])

                list_class_score.append(class_score)
            res.append(np.argmax(list_class_score))
        return np.array(res)


def _main_one_vs_one_perso():
    print("one vs one perso")
    predicteurs_simples = np.full((10, 10), None, dtype=object)
    predicteurs_simples_score = np.full((10, 10), np.nan)
    for id_c0 in range(10):
        for id_c1 in range(id_c0 + 1, 10):
            kept_train = np.where((all_train[1] == id_c0) | (all_train[1] == id_c1))[0]
            kept_test = np.where((all_test[1] == id_c0) | (all_test[1] == id_c1))[0]
            train = [all_train[0][kept_train], all_train[1][kept_train]]
            test = [all_test[0][kept_test], all_test[1][kept_test]]
            # met les étiquettes -1/1
            train[1] = np.where(train[1] == id_c0, -1, 1)
            test[1] = np.where(test[1] == id_c0, -1, 1)

            svc = LinearSVC(C=1, max_iter=int(1e5))
            svc.fit(train[0], train[1])
            predicteurs_simples[id_c0][id_c1] = svc
            predicteurs_simples_score[id_c0][id_c1] = svc.score(test[0], test[1])

    plt.imshow(predicteurs_simples_score)
    plt.colorbar()
    plt.title("One vs One: performances de chaque classifieur simple")

    predicteur = PredicteurOneVsOne(predicteurs_simples)
    predicted = predicteur.predict(all_test[0])
    conf = metrics.confusion_matrix(all_test[1], predicted)
    precision = np.sum(np.diagonal(conf))/len(all_test[0])
    conf = conf / np.sum(conf, axis=1)
    print("    linearSVC précision", precision)

    plt.figure()
    plt.imshow(conf)
    plt.colorbar()
    plt.title("matrice de confusion du classifieur multiclasse
(précision :"+str(round(precision, 3))+")")
    plt.show()
def _main_scikit_learn():
    print("Scikit Learn")
    print("one vs one")

    svc = SVC(kernel="linear", gamma="auto").fit(all_train[0], all_train[1])
    print("    SVC linear kernel précision", svc.score(all_test[0], all_test[1]))

    svc = SVC(gamma="auto").fit(all_train[0], all_train[1])
    print("    SVC rbf kernel    précision", svc.score(all_test[0], all_test[1]))

    print("one vs rest")

    svc = LinearSVC(max_iter=int(1e6), multi_class="ovr").fit(all_train[0], all_train[1])
    print("    LinearSVC précision", svc.score(all_test[0], all_test[1]))
def _main_contourf():
    for data_type in [0,1,2]:
```

```
        trainx, trainy = arftools.gen_arti(nbex=1000, data_type=data_type, epsilon=1)
        # testx, testy = arftools.gen_arti(nbex=1000, data_type=data_type, epsilon=1)
        param_list = [{'kernel':"linear", 'gamma':"auto"}, {'kernel':"rbf",
'gamma':"auto"}]
        for param in param_list:
            svm = SVC(**param)
            svm.fit(trainx, trainy)
            plt.figure()
            arftools.plot_frontiere(trainx, svm.predict, step=200)
            arftools.plot_data(trainx, trainy)
            plt.title(param["kernel"])
    plt.show()

if __name__ == '__main__':
    print("SVM BI CLASSE")
    # _main_biclass()
    print("\nSVM MULTI CLASSE")
    # _main_one_vs_one_perso()
    # _main_scikit_learn()
    _main_contourf()
```