TP Notebook

May 5, 2021

1 Régression linéaire par moindres carrés

1.1 Implémentation de l'algorithme des moindres carrés

Les imports du notebook complet :

[1]:
```
# Install a pip package in the current Jupyter kernel
import sys
!{sys.executable} -m pip install matplotlib
!{sys.executable} -m pip install sklearn
%matplotlib notebook
```

```
Requirement already satisfied: matplotlib in c:\users\mini\anaconda3\lib\site-
packages (3.3.2)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.3 in
c:\users\mini\anaconda3\lib\site-packages (from matplotlib) (2.4.7)
Requirement already satisfied: python-dateutil>=2.1 in
c:\users\mini\anaconda3\lib\site-packages (from matplotlib) (2.8.1)
Requirement already satisfied: numpy>=1.15 in c:\users\mini\anaconda3\lib\site-
packages (from matplotlib) (1.19.2)
Requirement already satisfied: kiwisolver>=1.0.1 in
c:\users\mini\anaconda3\lib\site-packages (from matplotlib) (1.3.0)
Requirement already satisfied: cycler>=0.10 in c:\users\mini\anaconda3\lib\site-
packages (from matplotlib) (0.10.0)
Requirement already satisfied: pillow>=6.2.0 in
c:\users\mini\anaconda3\lib\site-packages (from matplotlib) (8.0.1)
Requirement already satisfied: certifi>=2020.06.20 in
c:\users\mini\anaconda3\lib\site-packages (from matplotlib) (2020.6.20)
Requirement already satisfied: six>=1.5 in c:\users\mini\anaconda3\lib\site-
packages (from python-dateutil>=2.1->matplotlib) (1.15.0)
Requirement already satisfied: sklearn in c:\users\mini\anaconda3\lib\site-
packages (0.0)
Requirement already satisfied: scikit-learn in c:\users\mini\anaconda3\lib\site-
packages (from sklearn) (0.23.2)
Requirement already satisfied: joblib>=0.11 in c:\users\mini\anaconda3\lib\site-
packages (from scikit-learn->sklearn) (0.17.0)
Requirement already satisfied: numpy>=1.13.3 in
c:\users\mini\anaconda3\lib\site-packages (from scikit-learn->sklearn) (1.19.2)
Requirement already satisfied: scipy>=0.19.1 in
```

```
c:\users\mini\anaconda3\lib\site-packages (from scikit-learn->sklearn) (1.5.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in
c:\users\mini\anaconda3\lib\site-packages (from scikit-learn->sklearn) (2.1.0)
```

[2]:
```
import random
from sklearn import neighbors
from sklearn.datasets import load_iris  # les donn´ees iris sont charg´ees
from sklearn.datasets import load_boston  # les donn´ees iris sont charg´ees
from sklearn.datasets import load_diabetes  # les donn´ees iris sont charg´ees
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
import pylab as pl  # permet de remplacer le nom "pylab" par "pl"
import numpy as np
import statistics as stat
```

Soit l'algorithme de régression linéaire par moindre carrés suivant : 1) Ajouter le vecteur 1 à la matrice $X \in \mathbb{R}^{nxd}$ contenant les données $x_{i=1}^{n}$ 2) Calculer $w = (X^T X)^{-1} X^T y$, avec $y = y_{i=1}^{n} \in \mathbb{R}^n$ 3) Retourner w On se propose de coder l'implémentation de la régression linéaire par moindre carrés :

[3]:
```
def reg(X, Y):
    ones = np.ones((len(X), 1))
    Xb = np.concatenate((X, ones), axis=1)

    Xt = np.transpose(Xb)
    w = np.dot(np.dot(np.linalg.inv(np.dot(Xt, Xb)), Xt), Y)

    print("w =", w)
    return w
```

Pour la réalisation de l'exercice nous utiliserons un jeu de données bi-dimensionnelle contenues dans un fichier. On ouvre le fichier et on stocke les valeurs dans 2 variables (x et y) que nous réutiliserons par la suite. La variable x est un vecteur en 2 dimensions, la variable y est en 1 dimension.

[4]:
```
# Lire le fichier
tt = np.loadtxt("dataRegLin2D.txt")
x = tt[:, :2]
y = tt[:, -1]
```

On représente les données sur un graphe en 3 dimensions :

[5]:
```
def graph3D(x,y):
    fig = pl.figure("Visualisation du jeu de données")
    ax = fig.add_subplot(111, projection='3d')
```

```
    ax.scatter(x[:,0],x[:,1],y, c=y)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_zlabel('y')
    pl.show()

# Graphe 3D
graph3D(x,y)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

A première vue il y a une dépendance linéaire entre les étiquettes x_i^1, x_i^2 et y_i selon l'angle de vue.

On va donc chercher si les valeurs sont bien dépendantes ou non. Pour cela on affiche les résultats de la régression linéaire des valeurs du fichier sur un graphe en 2D. En premier lieu on étudie les dépendances entre les étiquettes x_i^1 et y_i (en rouge) puis les étiquettes x_i^2 et y_i (en bleu).

[6]:
```
# Première étiquette
pl.figure("x^(1)")
pl.scatter(x[:, 0], y, c='red')
pl.show()

# Seconde étiquette
pl.figure("x^(2)")
pl.scatter(x[:, 1], y, c='blue')
pl.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

On peut voir que les données de la première dimension de x (i.e x_i^1) est "mal" répartie et produit une dispersion de points qui ne semblent pas avoir de cohérences entre eux. L'inverse se produit pour les données de la seconde dimension de x (i.e x_i^2) qui ont l'air d'être facilement approchées par une droite de régression.

On cherche maintenant à calculer les droites de régression dans chacun des cas :

[7]:
```
# Paramètres de régression
x0p = reg(x[:, [0]], y)  # [ 0.44979209 -0.59832595]
x1p = reg(x[:, [1]], y)  # [ 1.43100678 -0.64475543]

def lin(x, xp):
    return xp[0] * x + xp[1]

# Graphiques 2D
```

```
pl.figure("Plotting des valeurs de x et leur droites de régression calculées")
pl.scatter(x[:, 0], y, c='red')
pl.scatter(x[:, 1], y, c='blue')
pl.plot(x, lin(x, x0p), color='red')
pl.plot(x, lin(x, x1p), color='blue')
pl.show()
```

```
w = [ 0.44979209 -0.59832595]
w = [ 1.43100678 -0.64475543]
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

On peut remarquer que les points rouge de x_i^1 et y_i sont dispersés et la droite de régression qui les représente passe grossièrement au milieu de tous les points, mais la cohérence des points entre eux est faible. En revanche les points bleus sont bien alignés et la droite de régression qui en résulte est proche des points.

On va calculer un vecteur de pondération w afin de réaliser une fonction qui à partir de valeurs x_{test} permet de prédire le label y_{test} :

[8]:
```
# Ensembles de test et d'entraînement
X_train, X_test, Y_train, Y_test = train_test_split(x, y, test_size=0.3,
                                                    random_state=random.
 ↪seed(123))# Calcul du vecteur de pondération w

w = reg(X_train, Y_train)
```

```
w = [ 0.24117648  1.36219868 -0.66643273]
```

On donne la formule de la fonction de la droite de régression, soit $x \in \mathbb{R}^2$, $f(x) = w_1.x_1 + w_2.x_2 + w_3$

[9]:
```
def pred_reg(x, w):
    return w[0]*x[0] + w[1]*x[1] +  w[2]
```

[10]:
```
# Calcul de l'erreur au carré entre les valeurs prédites et celles de␣
 ↪l'ensemble des données réelles
err_pred = []
for i in range(len(X_test)):
    # Calcul de la valeur prédite
    y_pred = pred_reg(X_test[i], w)
    # Différence entre le label prédit et le label réel
    err_pred.append(abs(Y_test[i] - y_pred)**2)
print(err_pred[:10], "...")
```

```
[0.0008432216758643346, 0.00048169106053614735, 0.04551381087952708,
2.1176088333966384e-05, 0.003854420579619422, 0.002787528053644163,
0.0011332030980929267, 0.0009188661813628292, 0.00581228908272243,
0.0022070561831069946] …
```

[11]:
```
# Moyenne d'erreur carrée
stat.mean(err_pred)
```

[11]:
```
0.010639537283340644
```

On trouve une prédiction proche à 2 décimales en moyenne des valeurs d'origines grâce à la fonction de prédiction entraînée par la méthode des moindres carrés.

1.1.1 Conclusion

On peut voir que les données en entrées qui semblaient, prises à part, être sans aucun lien, possède bien une corrélation qui permettent ensemble de déduire un modèle linéaire fiable avec une erreur à 10^{-2} environ par rapport aux données réelles.

2 Régression linéaire avec Scikit-learn

2.1 Sur les mêmes données simulées

Nous avons jusque là réalisé la méthode "à la main" en suivant l'algorithme de régression linéaire des moindres carrés. Maintenant nous allons réaliser implémenter et analyser de nouveaux algorithmes de régressions linéaires avec la librairie Scikit-learn.

Pour commencer on va recalculer avec les nouveaux outils les données de l'exercice précédent.

[12]:
```
model = LinearRegression().fit(X_train, Y_train)
r_sq = model.score(X_train, Y_train)
print('coefficient of determination:', r_sq)
print('intercept:', model.intercept_)
print('slope:', model.coef_)
```

```
coefficient of determination: 0.9932363948013285
intercept: -0.6664327324277761
slope: [0.24117648 1.36219868]
```

Pour rappel nous avions trouvé comme paramètres :

[13]:
```
w
```

[13]:
```
array([ 0.24117648,  1.36219868, -0.66643273])
```

[14]:
```
# Prédictions des labels y
y_pred = model.predict(X_test)
print(y_pred[:10], "...")
```

```
[-1.68861172 -0.70011946 -1.57205034  1.51261825 -2.07580398 -0.39866995
  0.29926108 -0.17254681  0.04785537 -0.39687068] …
```

[15]:
```
# Formatage des données
pred = np.array(y_pred).reshape(-1, 1)

# Calcul de l'erreur moyenne carré pour chaque valeurs
```

```
print(mean_squared_error(Y_test, y_pred))
```

```
0.01063953728334065
```

La prédiction calculée par scikit-learn est identique à celle trouvée précédemment.

On peut appliquer cette méthode sur des jeux de données réels différents, par exemple le prix des maison à Boston ou sur les chiffres du diabète.

2.2 Sur des données réelles

2.2.1 Diabète

Chargement des données depuis scikit-learn.

[16]:
```
# Chargement des données de diabète
diabete = load_diabetes()
X_diabete = diabete.data
Y_diabete = diabete.target
```

Les attributs disponibles dans ces données sont : - age in years - sex - bmi body mass index - bp average blood pressure - s1 tc, T-Cells (a type of white blood cells) - s2 ldl, low-density lipoproteins - s3 hdl, high-density lipoproteins - s4 tch, thyroid stimulating hormone - s5 ltg, lamotrigine - s6 glu, blood sugar level

On réalise un jeu de données et on calcule la régression linéaire avec scikit-learn.

[17]:
```
# jeux de données de tests et d'entrainements créés à partir de l'ensemble des␣
 ↪données importées
X_train, X_test, Y_train, Y_test = train_test_split(X_diabete, Y_diabete,␣
 ↪test_size=0.3,
                                                     random_state=random.
 ↪seed(123))
#Calcul du modèle de régression linéaire
model = LinearRegression().fit(X_train, Y_train)
r_sq = model.score(X_test, Y_test)
print('coefficient of determination:', r_sq)
print('intercept:', model.intercept_)
print('slope:', model.coef_)

# Prédiction de données à partir des données de test
y_pred = model.predict(X_test)
```

```
coefficient of determination: 0.5239470898486587
intercept: 151.25873892911184
slope: [ -26.52214874 -180.86594371  536.54317624  376.10607069 -982.3076334
  586.52638684  206.51071041  290.9230085   772.1608697    48.72668569]
```

[18]:
```
# Moyenne de l'erreur au carré
print(mean_squared_error(Y_test, y_pred))
```

```
2516.2779455511345
```

L'erreur au carré est très importante, le modèle n'est pas bon.

2.2.2 Boston

Chargement des données depuis scikit-learn.

[19]:
```
# Chargement des données de Boston
boston = load_boston()
X_boston = boston.data
Y_boston = boston.target
```

Les attributs du jeu de données sont : - CRIM per capita crime rate by town - ZN proportion of residential land zoned for lots over 25,000 sq.ft. - INDUS proportion of non-retail business acres per town - CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise) - NOX nitric oxides concentration (parts per 10 million) - RM average number of rooms per dwelling - AGE proportion of owner-occupied units built prior to 1940 - DIS weighted distances to five Boston employment centres - RAD index of accessibility to radial highways - TAX full-value property-tax rate per $10,000 - PTRATIO pupil-teacher ratio by town - B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town - LSTAT % lower status of the population - MEDV Median value of owner-occupied homes in $1000's

On réalise un jeu de données et on calcule la régression linéaire avec scikit-learn.

[20]:
```
# jeux de données de tests et d'entrainements créés à partir de l'ensemble des␣
 ↪données importées
X_train, X_test, Y_train, Y_test = train_test_split(X_boston, Y_boston,␣
 ↪test_size=0.3,
                                                     random_state=random.
 ↪seed(123))

#Calcul du modèle de régression linéaire
model = LinearRegression().fit(X_train, Y_train)
r_sq = model.score(X_test, Y_test)
print('coefficient of determination:', r_sq)
print('intercept:', model.intercept_)
print('slope:', model.coef_)

# Prédiction de données à partir des données de test
y_pred = model.predict(X_test)
```

```
coefficient of determination: 0.7400562929533582
intercept: 30.689597208025916
slope: [-9.92357342e-02  4.46506915e-02  1.98266344e-02  2.59763417e+00
 -1.53036230e+01  4.24938338e+00 -7.76195611e-03 -1.49791688e+00
  3.12550873e-01 -1.29948360e-02 -8.88350844e-01  1.19546235e-02
 -4.98934576e-01]
```

[21]:
```
# Moyenne de l'erreur au carré
print(mean_squared_error(Y_test, y_pred))
```

```
19.484249777113604
```

L'écart est moins important entre le modèle prédit et les données réelles, il est possible que le modèle soit bon.

2.3 Régression par Ridge et Lasso

La régression ridge et lasso sont des extensions de la régression linéaire par moindres carrés permettant d'éviter le risque de sur-apprentissage. Nous allons appliquer une régression Ridge et Lasso avec un coefficient $\alpha = 1.0$ sur les données de Boston et comparer les deux solutions.

[22]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X_boston, Y_boston,␣
 ↪test_size=0.3,
                                                     random_state=random.
 ↪seed(123))

# alp signifie que le coefficient alpha utilisé est celui par défaut, False␣
 ↪sinon pour calculer un coefficient optimisé
def ridge_boston(alp):
    alpha = 1.0 if alp else alphaRidge(X_train, Y_train)['alpha']
    print("alpha =", alpha)

    model = Ridge(alpha=alpha).fit(X_train, Y_train)
    r_sq = model.score(X_test, Y_test)
    print('coefficient of determination:', r_sq)
    print('intercept:', model.intercept_)
    print('slope:', model.coef_)

    # Prédiction
    y_pred = model.predict(X_test)
    print('Erreur moyenne carrée :', mean_squared_error(Y_test, y_pred))

# alp signifie que le coefficient alpha utilisé est celui par défaut, False␣
 ↪sinon pour calculer un coefficient optimisé
def lasso_boston(alp):
    alpha = 1.0 if alp else alphaLasso(X_train, Y_train)['alpha']
    print("alpha =", alpha)

    model = Lasso(alpha=alpha).fit(X_train, Y_train)
    r_sq = model.score(X_test, Y_test)
    print('coefficient of determination:', r_sq)
    print('intercept:', model.intercept_)
    print('slope:', model.coef_)

    # Prédiction
```

```
    y_pred = model.predict(X_test)
    print('Erreur moyenne carrée :', mean_squared_error(Y_test, y_pred))
```

[23]:
```
# Calcul du modèle de régression linéaire par Ridge
# True signifie que le coefficient alpha utilisé est celui par défaut
ridge_boston(True)
```

```
alpha = 1.0
coefficient of determination: 0.752857274529992
intercept: 35.824395703315346
slope: [-1.14331207e-01  5.30098122e-02 -1.89722212e-02  1.99399622e+00
 -1.03721387e+01  3.03304090e+00  1.77536280e-03 -1.37551380e+00
  3.17740819e-01 -1.33191169e-02 -8.68437210e-01  1.07023471e-02
 -5.51700057e-01]
Erreur moyenne carrée : 23.48649082764778
```

[24]:
```
# Calcul du modèle de régression linéaire par Lasso
# True signifie que le coefficient alpha utilisé est celui par défaut
lasso_boston(True)
```

```
alpha = 1.0
coefficient of determination: 0.6580166987838663
intercept: 43.5855614118965
slope: [-0.06713893  0.05927791 -0.          0.         -0.          0.31873318
  0.02314796 -0.66177396  0.28835817 -0.01602947 -0.67945978  0.00898695
 -0.74766289]
Erreur moyenne carrée : 32.49938937893664
```

Les deux méthodes donnent des résultats proches entre eux et par rapport à la méthode des moindres carrés. Nous allons maintenant chercher à calculer le meilleur coefficent α pour les deux nouvelles méthodes de régression linéaire. Pour cela nous utiliserons la méthode de cross-validation sur une grille de valeurs.

[25]:
```
## Méthodes pour déterminer le meilleure coefficient alpha les régressions par␣
 ↪Ridge et Lasso

def alphaRidge(X, y):
    alphas = np.logspace(-3, -1, 20)
    gscv = GridSearchCV(Ridge(), dict(alpha=alphas), cv=5).fit(X, y)
    return gscv.best_params_


def alphaLasso(X, y):
    alphas = np.logspace(-3, -1, 20)
    gscv = GridSearchCV(Lasso(), dict(alpha=alphas), cv=5).fit(X, y)
    return gscv.best_params_
```

[26]:
```
# Calcul du meilleur coefficient alpha pour la méthode du Ridge
alpha = alphaRidge(X_train, Y_train)['alpha']
alpha
```

[26]:
```
0.1
```

[27]:
```
# Calcul du meilleur coefficient alpha pour la méthode du Lasso
alpha = alphaLasso(X_train, Y_train)['alpha']
alpha
```

[27]:
```
0.00545559478116852
```

On peut donc utiliser le meilleur coefficient α pour les deux méthodes :

[28]:
```
# False signifie que le coefficient alpha utilisé est celui optimisé par le␣
 ↪calcul d'un coefficient en
# fonction des jeux de données
ridge_boston(False)
```

```
alpha = 0.1
coefficient of determination: 0.7524051380952133
intercept: 41.3219454747791
slope: [-1.18422917e-01  5.12381849e-02  1.23401927e-02  2.12166457e+00
 -1.81071246e+01  2.97614049e+00  8.60171700e-03 -1.50232365e+00
  3.37973873e-01 -1.27698855e-02 -9.60077997e-01  1.06767523e-02
 -5.41929361e-01]
Erreur moyenne carrée : 23.52945830001858
```

[29]:
```
# False signifie que le coefficient alpha utilisé est celui optimisé par le␣
 ↪calcul d'un coefficient en
# fonction des jeux de données
lasso_boston(False)
```

```
alpha = 0.00545559478116852
coefficient of determination: 0.7520871811787887
intercept: 41.26703810909359
slope: [-1.18286352e-01  5.12867387e-02  1.17332135e-02  2.05005181e+00
 -1.79443522e+01  2.96731371e+00  8.64182009e-03 -1.49818479e+00
  3.37918892e-01 -1.28007207e-02 -9.58141369e-01  1.06868720e-02
 -5.43015721e-01]
Erreur moyenne carrée : 23.559674411728896
```

Avec la valeur du coefficient α optimisé, on se rend compte que les deux modèles de régression renvoient des résultats très similaires. Le coefficient α est bien ajusté par rapport au jeux de données et l'erreur est minimisé.

3 Algorithme du perceptron pour la classification binaire

[30]:
```
from pylab import rand
import pylab as pl
import numpy as np
```

En utilisant la fonction ci-dessous, on représente des données générées sur un graphique, les données d'apprentissages.

[31]:
```
def generateData(n):
    """
    generates a 2D linearly separable dataset with 2n samples.returns X an␣
 ↪array of 2D samples, and Y the samples label
    """
    xb = (rand(n) * 2 - 1) / 2 - 0.5
    yb = (rand(n) * 2 - 1) / 2 + 0.5
    xr = (rand(n) * 2 - 1) / 2 + 0.5
    yr = (rand(n) * 2 - 1) / 2 - 0.5
    inputs = []
    for i in range(n):
        inputs.append([xb[i], yb[i], -1])
        inputs.append([xr[i], yr[i], 1])
    data = np.array(inputs)
    X = data[:, 0:2]
    Y = data[:, -1]
    return X, Y
```

[32]:
```
# Génération des données
n = 100
x, y = generateData(n)
# Affichage des données d'apprentissage
pl.figure("Représentation des données d'apprentissage")
pl.scatter(x[:, 0], x[:, 1], c=y)
pl.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

On peut tout de suite voir qu'il existe deux ensembles catégorisés (violet et jaune) dans ce jeu de données et qu'ils sont linéairement séparables. L'outil qui permet de réaliser une séparation par un hyperplan entre 2 ensembles linéairement séparables est le perceptron binaire. On implémente donc l'algorithme du perceptron binaire.

[33]:
```
def perceptron(x, y):
    w = np.zeros((x.shape[1]))
    sorted = False
    while not sorted:
        sorted = True
```

```
        for xi, yi in zip(x, y):
            if yi * np.dot(w, xi) <= 0:
                sorted = False
                w += yi * xi
    print("w =", w)
    return w
```

On calcule le vecteur de pondération w qui résulte du perceptron avec nos données et on représente la droite générée par le vecteur de pondération sur un graphique. A noter que ce cas est fait à partir de données linéairement séparables par une droite passant par l'origine.

[34]:
```
# Calcul du vecteur de pondération par le perceptron
w = perceptron(x, y)

#Affichage sur le graphique de la droite
pl.figure("Affichage de la droite du classifieur")
pl.plot(x, (-w[0] / w[1]) * x)
pl.scatter(x[:, 0], x[:, 1], c=y)
pl.show()
```

```
w = [ 0.25297685 -0.59813372]
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Pour tester le vecteur de pondération w on va réaliser un ensemble de jeux de données de test et calculer l'erreur que la prédiction nous donne sur l'ensemble cet ensemble.

[35]:
```
# Génération des données de test
x_test, y_test = generateData(n)

# Calcul de l'erreur
e = 0
for xi, yi in zip(x_test, y_test):
    prod_vect_xw = np.dot(xi, w)
    # Teste la bonne appartenance à une catégorie des données
    if (prod_vect_xw > 0 > yi) or (prod_vect_xw < 0 < yi):
        e += 1
print(e)
```

```
0
```

Le vecteur w donne donc une prédiction parfaite, nous n'avons aucune erreur entre l'ensemble de test et l'ensemble prédit.

Enfin, nous avons vu le cas du perceptron sur des données séparables par un hyperplan passant par l'origine. Nous allons étudier le cas d'un hyperplan ne passant pas par l'origine. Voici le code qui génère les données pour ce cas :

[36]:
```
def generateData2(n):
    """
    generates a 2D linearly separable dataset with 2n samples.returns X an␣
 ↪array of 2D samples, and Y the samples label
    """
    xb = (rand(n) * 2 - 1) / 2 + 0.5
    yb = (rand(n) * 2 - 1) / 2
    xr = (rand(n) * 2 - 1) / 2 + 1.5
    yr = (rand(n) * 2 - 1) / 2 - 0.5
    inputs = []
    for i in range(n):
        inputs.append([xb[i], yb[i], -1])
        inputs.append([xr[i], yr[i], 1])
    data = np.array(inputs)
    X = data[:, 0:2]
    Y = data[:, -1]
    return X, Y
```

Nous représentons ces données sur un graphique :

[37]:
```
# Génération des données ne passant pas par l'origine
x, y = generateData2(n)
pl.figure("Représentation des données d'apprentissage")
pl.scatter(x[:, 0], x[:, 1], c=y)
pl.show()
```

On veut utiliser la technique qui permet de considérer nos données qui sont de dimensions d, dans $\mathbb{R}^d$, on va faire comme si elles étaient dans $\mathbb{R}^{d+1}$ et qu'elles avaient toutes 1 en dernière coordonnée. Du coup, on considère que w est aussi de dimension $d+1$ et la dernière coordonnée de w est b, le scalaire qui permet de générer un vecteur w qui ne passe pas forcément par l'origine.

On définit une nouvelle fonction *complete* qui ajoute cette dimension aux données.

[38]:
```
def complete(sample):
    return np.concatenate(
        (sample,
         np.ones((len(sample), 1))
         ),
        axis=1
    )
```

[39]:
```
# Ajout d'un attribut à 1 pour chaque données
x = complete(x)

# Calcul du perceptron avec biais
w = perceptron(x, y)

# Représentation graphique de l'hyperplan avec les données
pl.figure("Représentation de l'hyperplan du classifieur")
```

```
# Utilisation de l'équation de la droite à partir du vecteur de pondération
pl.plot(x, (-w[0] / w[1]) * x - w[2] / w[1])
pl.scatter(x[:, 0], x[:, 1], c=y)
pl.show()
```

```
w = [ 17.73549242  -0.7517577  -18.        ]
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

On peut calculer l'erreur avec le nouveau perceptron avec biais :

[40]:
```
# Test de l'erreur avec biais
# Génération des données de test
x_test, y_test = generateData2(n)
x_test = complete(x_test)

# Calcul de l'erreur
e = 0
for xi, yi in zip(x_test, y_test):
    if (np.dot(xi, w) > 0 > yi) or (np.dot(xi, w) < 0 < yi):
        e += 1
print(e)
```

```
1
```

Il peut arriver que les données ne soient pas toujours totalement linéairement séparables et on trouve donc quelques erreurs selon l'itération, mais elles restent minoritaires et n'impactent pas le résultat. L'hyperplan sépare bien les deux ensembles et ne passe pas par l'origine.

3.1 Perceptron à noyau

Pour représenter des données cette fois dans le cadre du perceptron à noyau, on implémente une nouvelle fonction :

[41]:
```
def generateData3(n):
    """generates a non linearly separable dataset with about 2n samples.The␣
 ↪third element of the sample is the label"""
    xb = (rand(n) * 2 - 1) / 2
    yb = (rand(n) * 2 - 1) / 2
    # (xb, yb) est dans le carré centré à l'origine de c^oté 1
    xr = 3 * (rand(4 * n) * 2 - 1) / 2
    yr = 3 * (rand(4 * n) * 2 - 1) / 2
    # (xb, yb) est dans le carré centré à l'origine de c^oté 3
    inputs = []
    for i in range(n):
        inputs.append(((xb[i], yb[i]), -1))
    for i in range(4 * n):
```

```
        # on ne conserve que les points extérieurs au carré centré à l'origine␣
↪de c^oté 2
        if abs(xr[i]) >= 1 or abs(yr[i]) >= 1:
            inputs.append(((xr[i], yr[i]), 1))
    return inputs
```

[42]:
```
# Génération de données
sample = generateData3(n)

# Formatage des données
x = [i[0] for i in sample]
x = [[i[0], i[1]] for i in x]
x = np.array(x)
y = [i[1] for i in sample]
y = np.array(y).reshape((-1, 1))

# Affichage sur un graphique des données
pl.figure("Affichage des données d'apprentissage")
pl.scatter(x[:, 0], x[:, 1], c=y)
pl.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Dans le cas du perceptron à noyau, les données ne sont par linéairements séparables, peu importe l'angle de vue voulu, c'est pour cela que l'on observe ce genre d'ensemble.

On cherche à plonger les données dans une dimension qui nous permet de les séparer. On considère le plongement $\phi : \mathbb{R}^2 \to \mathbb{R}^6$ défini par $\phi(x, y) = (1, x, y, x^2, x * y, y^2)$:

[43]:
```
#Fonction de plongement R2 à R6
def plongement(x):
    p = []
    for xi in x:
        pp = np.array([1, xi[0], xi[1], xi[0] ** 2, xi[0] * xi[1], xi[1] ** 2])
        p.append(pp)
    return np.array(p)
```

Pour représenter les données, on va former une séparation à partir de points calculés par le vecteur w. Pour réaliser cela on va calculer des points et les afficher s'ils sont proches de la séparation entre les deux classes.

[44]:
```
# Fonction pour réaliser le produit scalaire <w,x>
def f(w, x):
    w = w.reshape((-1, 1))
    x = plongement(x)
    return np.dot(x, w)
```

```
# Affichage de la courbe du classifieur par balayage
def plotRect(w):
    res = 500
    for x in range(res):
        for y in range(res):
            # la fonction f va effectuer le calcul du produit scalaire <w,x>␣
↪pour trouver la classe
            # On affiche les points qui sont à +- 0,01 de l'hyperplan d'après␣
↪cette fonction f pour afficher les "contours" de l'hyperplan
            if abs(f(w, [[-3 / 2 + 3 * x / res, -3 / 2 + 3 * y / res]])) < 0.01:
                pl.plot(-3 / 2 + 3 * x / res, -3 / 2 + 3 * y / res, 'xb')
    pl.show()
```

[45]:
```
# Plongement des données en R6 puis calcul du perceptron à noyau
px = plongement(x)
w = perceptron(px, y)

# Représentation graphique des données
pl.figure("Représentation des données d'apprentissage")
pl.scatter(x[:, 0], x[:, 1], c=y)
plotRect(w)
```

```
w = [-2.          0.6313598   0.28421828  2.5319108  -0.1787553   3.67908539]
```

Le plongement réalisé correspond à la fonction noyau k_1 définie par : $k_1((x_1, y_1), (x_2, y_2)) = 1 + x_1x_2 + y_1y_2 + x_1^2x_2^2 + x_1y_1x_2y_2 + y_1^2y_2^2$

[46]:
```
def k(a, b):
    x1 = a[0]
    x2 = b[0]
    y1 = a[1]
    y2 = b[1]
    return  1 + x1*x2 + y1*y2 + ((x1**2)*(x2**2)) + x1*y1*x2*y2 +␣
↪((y1**2)*(y2**2))
```

Etant donné un échantillon $S = ((x_1, y_1), ..., (x_n, y_n))$ où chaque $x_i \in \mathbb{R}^2$, une fonction noyau $k : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$, un ensemble de coefficients $c_1, ..., c_n \in \mathbb{N}$, l'ensemble VS des exemples de coefficients non nuls ($VS = \{(x_i, y_i) | c_i \neq 0\}$), on en déduit la fonction $f : \mathbb{R}^2 \to \mathbb{R}$ définie par : $f(x) = \sum_{(x_i,y_i)\in VS} c_i y_i k(x_i, x)$ En effet, cette fonction nous permettra de déterminer la valeur prédite par l'échantillon.

[47]:
```
def f_from_k(coeffs, support_set, k, x):
    t = 0
    for a, ss in zip(coeffs, support_set):
        xi = ss[0]
        yi = ss[1]
        t += a * yi * k(xi, x)
    return t
```

On définit la fonction du perceptron à noyau :

[48]:
```
def perceptron_noyau(X, Y, k):
    coef = []
    support_set = []
    isSorted = True
    maxTour = 20
    co = 0
    while isSorted and co < maxTour :
        isSorted = False
        for x, y in zip(X, Y):
            if y * f_from_k(coef, support_set, k, x) <= 0:
                isSorted = True
                support_set.append((x, y))  # (x,y)
                coef += [1]
        co += 1
    return coef, support_set
```

La fonction $f\_from\_k : x \rightarrow \sum_{i=1}^{n} \alpha_i y^i k(x, x^i)$ précédemment définie nous permet de réaliser la représentation graphique du classifieur On va calculer plusieurs valeurs autour du classifieur et afficher celles qui sont les plus proches (à 0.01 près) pour le rendre visible.

[49]:
```
coeff, support = perceptron_noyau(x, y, k)
print(coeff, support)
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [(array([-0.26050122,  0.16506138]),
array([-1])), (array([-0.71355824, -1.45621002]), array([1])), (array([
0.2225579 , -0.15252379]), array([-1])), (array([-0.33684388, -0.37017363]),
array([-1])), (array([0.2042793 , 1.18605645]), array([1])),
(array([-1.13174625,  0.30514257]), array([1])), (array([-0.26050122,
0.16506138]), array([-1])), (array([-0.33684388, -0.37017363]), array([-1])),
(array([ 1.07786906, -0.81143581]), array([1])), (array([-0.22238363,
-0.49791679]), array([-1]))]
```

[50]:
```
def plotRectK(coef,supp, fk):
    res = 150
    for x in range(res):
        for y in range(res):
            if abs(f_from_k(coef, supp, fk, [-3/2+3*x/res,-3/2+3*y/res]))<0.01:
                pl.scatter(-3/2 + 3*x/res, -3/2 + 3*y/res, color='b')
    pl.show()

pl.figure("Représentation de l'hyperplan par le perceptron noyau")
pl.scatter(x[:, 0], x[:, 1], c=y)
plotRectK(coeff, support, k)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

On obtient le même résultat que dans le perceptron précédent. Cela peut vouloir dire que la fonction noyau est bien adapté avec notre problème.

Nous allons maintenant tester le perceptron avec une autre fonction noyau, celle du noyau gaussien. La fonction prend un paramètre σ que nous allons modifier pour analyser les résultats, on prendra $\sigma = \{20, 10, 1, 0.5, 0.2\}$.

[51]:
```
import math
def kg20(x,y):
    sigma=20
    return np.exp(-((x[0]-y[0])**2+(x[1]-y[1])**2)/sigma**2)
def kg10(x,y):
    sigma=10
    return np.exp(-((x[0]-y[0])**2+(x[1]-y[1])**2)/sigma**2)
def kg1(x,y):
    sigma=1
    return np.exp(-((x[0]-y[0])**2+(x[1]-y[1])**2)/sigma**2)
def kg05(x,y):
    sigma=0.5
    return np.exp(-((x[0]-y[0])**2+(x[1]-y[1])**2)/sigma**2)
def kg02(x,y):
    sigma=0.2
    return np.exp(-((x[0]-y[0])**2+(x[1]-y[1])**2)/sigma**2)
```

$\sigma = 20$

[52]:
```
coeff, support = perceptron_noyau(x, y, kg20)
pl.figure("Sigma = 20")
pl.scatter(x[:, 0], x[:, 1], c=y)
plotRectK(coeff, support, kg20)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

$\sigma = 10$

[53]:
```
coeff, support = perceptron_noyau(x, y, kg10)
pl.figure("Sigma = 10")
pl.scatter(x[:, 0], x[:, 1], c=y)
plotRectK(coeff, support, kg10)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

$\sigma = 1$

[54]:
```
coeff, support = perceptron_noyau(x, y, kg1)
pl.figure("Sigma = 1")
pl.scatter(x[:, 0], x[:, 1], c=y)
plotRectK(coeff, support, kg1)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

$\sigma = 0.5$

[55]:
```
coeff, support = perceptron_noyau(x, y, kg05)
pl.figure("Sigma = 0.5")
pl.scatter(x[:, 0], x[:, 1], c=y)
plotRectK(coeff, support, kg05)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

$\sigma = 0.2$

[56]:
```
coeff, support = perceptron_noyau(x, y, kg02)
pl.figure("Sigma = 0.2")
pl.scatter(x[:, 0], x[:, 1], c=y)
plotRectK(coeff, support, kg02)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

La conclusion sur l'utilisation du noyau gaussien,c'est que plus le coefficient sigma est élevé, moins la précision est grande. Donc si le coefficient est trop élevé, la classification sera trop précise et empêche de visualiser les résultats, alors que dans le cas inverse, si le coefficient est trop faible il va absolument mal classifier et la limite entre deux classes sera floue. Ici nous pouvons garder un coefficient σ de 1.

4 Exercice

On se propose de réaliser un perceptron avec de nouvelles valeurs.

[57]:
```
# Récupération des valeurs dans le fichier de données
f = open("./learn.data","r")
training_set = []
x = f.readline()
while x:
    training_set.append(eval(x))
    x = f.readline()
f.close()
```

[58]:
```
# Formatage des données
X_file = []
Y_file = []
for v in training_set:
    t = v[0]
    X_file += [[t[0],t[1]]]
```

```
    Y_file += [v[1]]
```

[59]:
```
# Création de jeux de données
X_train, X_test, Y_train, Y_test = train_test_split(X_file, Y_file, test_size=0.
↪3,
                                                    random_state=random.
↪seed(123))
```

[60]:
```
# Entrainement d'un nouveau perceptron sur le jeu de train
coeff, support = perceptron_noyau(X_train, Y_train, kg1)
```

[61]:
```
# Représentation des données de train
pl.figure("Représentation des données d'apprentissage")
pl.scatter(np.array(X_train)[:,0], np.array(X_train)[:,1], c=Y_train)
pl.show()
```

[62]:
```
# Représentation des données d'apprentissage et du classifieur
pl.figure("Représentation des données d'apprentissage et du classifieur␣
↪calculé")
pl.scatter(np.array(X_train)[:,0], np.array(X_train)[:,1], c=Y_train)

res = 100
for i in range(res):
    for j in range(res):
        if abs(f_from_k(coeff, support, kg1, [20*i/res,-1+2*j/res]))<0.01:
            pl.scatter(20*i/res,-1+2*j/res, color='b')
pl.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Finalement nous pouvons écrire une fonction qui va reprendre tous ces principes et calculer un score de prédiction sur un ensemble S de données. Ici nous prendrons (x_test, y_test) comme ensemble S.

[63]:
```
def score(S, coeffs, support_set, k):
    x_test, y_test = S
    error = 0
    for x_val, y_val in zip(x_test, y_test):
        if np.sign(f_from_k(coeffs, support_set, k, x_val)) != np.sign(y_val):
            error += 1
    print(f"Trouvé {error} erreurs pour {len(x_test)} valeurs")
    return 1 - error/len(x_test)

score((X_test, Y_test), coeff, support, kg1)
```

```
Trouvé 1 erreurs pour 28 valeurs
```

[63]: 0.9642857142857143