**Télécom Paris** **PESTO IA**
**2019** **Ons Jelassi, Pavlo Mozharovskyi**

**TP n° 1 : Introduction à Python, Gradient Stochastique et Perceptron**

## Quelques remarques avant de démarrer

On va utiliser `Jupyter notebook` durant cette séance. Pour cela choisissez la version Anaconda sur les machines de l'école. Quelques points importants à retenir :

- Chargements divers -

```
import math # importe un package
import numpy as np # importe un package sous un nom particulier
from sklearn import linear_model # importe tout un module
from os import mkdir # importe une fonction
```

- Utilisation de l'aide -

```
help(mkdir) # pour obtenir de l'aide sur mkdir
linear_model.LinearRegression? # pour obtenir de l'aide sur LinearRegression
```

- Versions de package, localisation des fonctions -

```
print(np.__version__) # obtenir la version d'un package
from inspect import getsourcelines # obtenir le code source de fonctions
getsourcelines(linear_model.LinearRegression)
```

Rem: Pour tous les traitements numériques ont utilisera `numpy` (pour la gestion des matrices notamment) et `scipy`.

# 1 Introduction à Python, Numpy et Scipy

Il est bon de parcourir les documents suivants pour obtenir plus d'aide sur Python, pour ceux qui ne sont pas encore très familier avec ce langage :
https://docs.python.org/3/tutorial/introduction.html
https://docs.scipy.org/doc/numpy/user/quickstart.html?highlight=tutorial
https://docs.scipy.org/doc/scipy/reference/tutorial/index.html

1) Écrire une fonction `nextpower` qui calcule la première puissance de 2 supérieure ou égale à un nombre $n$ (on veillera à ce que le type de sortie soit un `int`, tester cela avec `type` par exemple).
2) En partant du mot contenant toutes les lettres de l'alphabet, générer par une opération de *slicing* la chaîne de caractère `cfilorux` et, de deux façons différentes, la chaîne de caractère `vxz`.
3) Afficher le nombre $\pi$ avec 9 décimales après la virgule.

4) Compter le nombre d'occurrences de chaque caractère dans la chaîne de caractères `s="HelLo WorLd!!!"`. On renverra un dictionnaire qui à chaque lettre associe son nombre d'occurrences.

5) Écrire une fonction de codage par inversion de lettres [1] : chaque lettre d'un mot est remplacée par une (et une seule) autre. On se servira de la fonction `shuffle` sur la chaîne de caractères contenant tout l'alphabet pour associer les lettres codées.

6) Calculer $2\prod_{k=1}^{\infty}\frac{4k^2}{4k^2-1}$ efficacement (approximativement, par exemple avec $k = 1, ..., 500$). On pourra utiliser `time` (ou `%timeit`) pour déterminer la rapidité de votre méthode. Proposer une version avec et une version sans boucle (utilisant `Numpy`).

7) Créer une fonction `quicksort` qui trie une liste, en remplissant les éléments manquants dans le code suivant. On testera que la fonction est correcte sur l'exemple `quicksort([-2, 3, 5, 1, 3])` :

```
def quicksort(ll):
    """ a sorting algorithm with a pivot value"""
    if len(ll) <= 1:
        return ll
    else:
        TODO # pivot = last element of the list ll.
        less = []
        greater = []
        for x in ll:
            if x <= pivot:
                TODO # append 'x' to 'less'
            else:
                TODO # append 'x' to 'greater'
        return TODO # concatenate quicksort(less), pivot and quicksort(greater)
```

Indices : la longueur d'une liste est donnée par `len(l)` ; deux listes peuvent être concaténées avec `l1 + l2` ; `l.pop()` retire le dernier élément d'une liste.

8) Sans utiliser de boucles `for` / `while` : créer une matrice $M \in \mathbb{R}^{5\times 6}$ aléatoire à coefficients uniformes dans $[-1, 1]$, puis remplacer une colonne sur deux par sa valeur moins le double de la colonne suivante. Remplacer enfin les valeurs négatives par 0 en utilisant un masque binaire.

9) Créer une matrice $M \in \mathbb{R}^{5\times 20}$ aléatoire à coefficients uniformes dans $[-1, 1]$. Tester que $G = M^\top M$ est symétrique et que ses valeurs propres sont positives (on parle de alors de matrice définie positive). Quel est le rang de $G$ ?

Aide : on utilisera par exemple `np.allclose, np.logical_not, np.all` pour les tests numériques.

# 2 Introduction à Pandas, Matplotlib, etc.

On pourra commencer par consulter le tutoriel :
http://pandas.pydata.org/pandas-docs/stable/tutorials.html

### - Charger des données -

On utilise la base de données [2] **Individual household electric power consumption Data Set**. Pour cela utiliser les commandes ci-dessous :

```
from os import path
import pandas as pd
import urllib
import zipfile
```

1. aussi connu sous le nom de code de César.
2. https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption

```
import sys

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00235/'
filename = 'household_power_consumption'
zipfilename = filename + '.zip'
Location = url + zipfilename
# testing existence of file:
if sys.version_info >= (3, 0):
    if not(path.isfile(zipfilename)):
        urllib.request.urlretrieve(Location, zipfilename)
else:
    if not(path.isfile(zipfilename)):
        urllib.urlretrieve(Location, zipfilename)
# unzip part
zip = zipfile.ZipFile(zipfilename)
zip.extractall()
na_values = ['?', '']
fields = ['Date', 'Time', 'Global_active_power']
df = pd.read_csv(filename + '.txt', sep=';', nrows=200000,
na_values=na_values, usecols=fields)
```

On ne s'intéresse dans un premier temps qu'à la grandeur `Global_active_power`.

1) Charger la base puis détecter et dénombrer le nombre de lignes ayant des valeurs manquantes.
2) Supprimer toutes les lignes avec des valeurs manquantes.
3) Utiliser `to_datetime` et `set_index` pour indexer le DataFrame (on prendra garde au format des dates internationales qui diffère du format français).
4) Afficher le graphique des moyennes journalières entre le 1er janvier et le 30 avril 2007. Proposer une cause expliquant la consommation fin février et début avril. On pourra utiliser en plus de `matplotlib` le package `seaborn` pour améliorer le rendu visuel.

On ajoute des informations de température pour cette étude : les données utiles étant disponibles dans le fichier `ECAD_2016-09-11.txt` [3]. Ici les températures relevées sont celles d'Orly (noter cependant qu'on ne connaît pas le lieux de relevé de la précédente base de données).

5) Charger les données avec `pandas`, et ne garder que les colonnes `DATE` et `TG`. Diviser par 10 la colonne `TG` pour obtenir des températures en degrés Celsius. Traiter les éléments de température aberrants comme des NaN.
6) Créer un DataFrame `pandas` des températures journalières entre le 1er janvier et le 30 avril 2007. Afficher sur un même graphique ces températures et la séries `Global_active_power`.

On considère maintenant le jeu de données `20080421_20160927-PA13_auto.csv`.

7) Proposer une visualisation de la pollution pour l'ozone sur la période d'étude.
8) Proposer une visualisation de la pollution par année pour l'ozone et pour le dioxyde d'azote. Commenter l'évolution temporelle.
9) Proposer une représentation par mois de la pollution à l'ozone et au dioxyde d'azote. Quel est le mois le plus pollué (pour chacun des polluants) ?

## Pour aller plus loin

Vous pouvez aussi consulter les pages suivantes :

— http://www.python.org
— http://scipy.org
— http://www.numpy.org
— http://scikit-learn.org/stable/index.html

3. on peut aussi trouver d'autres informations sur le site http://eca.knmi.nl/dailydata/predefinedseries.php

— http://www.loria.fr/~rougier/teaching/matplotlib/matplotlib.html

— http://jrjohansson.github.io/

Pour aller plus loin :

— http://blog.yhat.com/posts/aggregating-and-plotting-time-series-in-python.html

— http://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-tutor2-python-pandas.pdf

# 3 Introduction statistique

## Définitions et notations

On rappelle ici le cadre de la classification binaire supervisée, et l'on présente les notations que l'on utilisera :

— $\mathcal{Y}$ l'ensemble des étiquettes des données (*labels* en anglais). On traite le cas binaire : il y a donc deux classes. Il est pratique de raisonner avec $\mathcal{Y} = \{-1, 1\}$ pour représenter les étiquettes, car on va considérer des signes au cours de ce travail [4].

— $\mathbf{x} = (x_1, \ldots, x_p)^\top \in \mathcal{X} \subset \mathbb{R}^p$ est une observation, un exemple, un point (ou un *sample* en anglais). La $j^{\text{e}}$ coordonnée de $\mathbf{x}$ est la valeur prise par la $j^{\text{e}}$ variable explicative (*feature* en anglais).

— $\mathcal{D}_n = \{(\mathbf{x}_i, y_i), i = 1, \ldots n\}$ un ensemble d'apprentissage contenant $n$ exemples et leurs étiquettes,

— Il existe un modèle probabiliste qui gouverne la génération des nos observations selon des variables aléatoires $X$ et $Y$ : $\forall i \in \{1, \ldots, n\}, (\mathbf{x}_i, y_i) \overset{i.i.d}{\sim} (X, Y)$.

— On cherche à construire à partir de l'ensemble d'apprentissage $\mathcal{D}_n$ une fonction appelée classifieur, $\hat{f} : \mathcal{X} \mapsto \{-1, 1\}$ qui pour un nouveau point $\mathbf{x}_{\text{new}} \in \mathcal{X}$ fournit une étiquette $\hat{f}(\mathbf{x}_{\text{new}})$.

## Génération artificielle de données

Dans un but d'expérimentation et de visualisation, il est plus aisé de travailler sur des données générées artificiellement. Ainsi on considère ici des variables explicatives (*features* en anglais) de dimension deux. Cela consiste à prendre $p = 2$ dans le formalisme ci-dessus.

1) Étudiez la fonction `rand_gauss(n, mu, sigmas)` qui engendre $n$ observations selon la loi normale multi-dimensionnelle de moyenne le vecteur `mu` et de matrice de covariance la matrice diagonale de diagonale `sigmas` $= [\sigma_1, \sigma_2]$, *i.e.,* la matrice de variance covariance est : $\begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}$. Générer ensuite plusieurs jeux de données à l'aide des fonctions `rand_bi_gauss`, `rand_clown` et `rand_checkers` pour différents paramètres d'entrée `n,mu,sigma`. Que renvoient ces fonctions ? À quoi correspond la seconde variable de sortie ?

2) Conservez quelques jeux de données afin de les utiliser dans la suite : pour chacun, il faudra sauver sous forme d'un tableau `numpy` à deux colonnes `X` les données, et dans un vecteur `Y` les labels correspondants à chaque exemple.

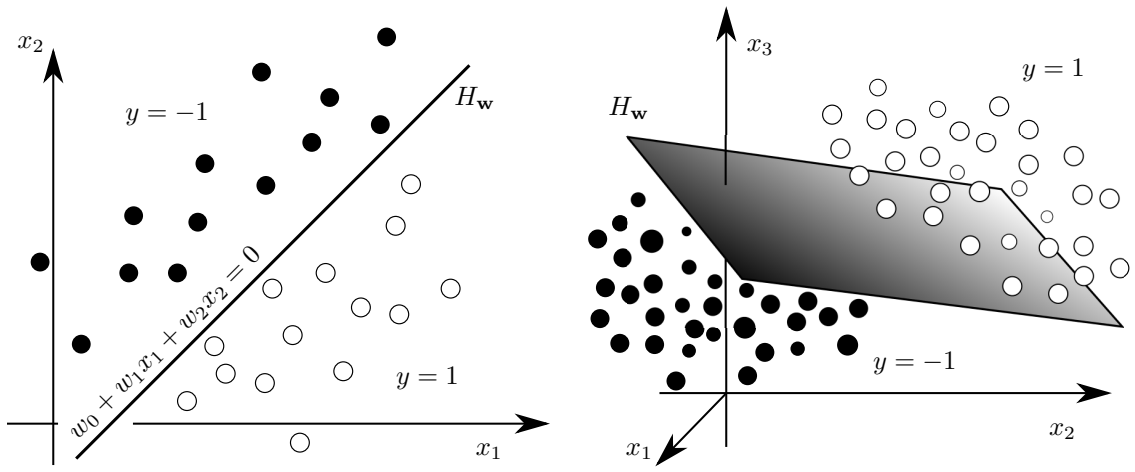
3) Utilisez la fonction `plot_2d` disponible dans `tp_perceptron_source.py` et qui permet de visualiser quelques jeux de données en fonction des étiquettes associées. Changer la couleur de la classe des $-1$.

# 4 Perceptron

## Les classifieurs linéaires (affines)

Un classifieur linéaire est un classifieur qui associe à chaque observation $\mathbf{x}$ une étiquette dans $\mathcal{Y}$ (ici $\mathcal{Y} = \{-1, 1\}$) selon sa position par rapport à un hyperplan affine. Chaque classifieur linéaire est donc lié à

4. Noter que pour d'autres méthodes, il peut être plus pratique de prendre $\mathcal{Y} = \{0, 1\}$.




(a) $p = 2$ (b) $p = 3$

FIGURE 1 – Exemple de séparation de classe (1 pour les points blanc, $-1$ pour les noirs) par des hyperplans affines, en dimension $p = 2$ et $p = 3$

un hyperplan (affines) de $\mathbb{R}^p$ que l'on définit pour un certain vecteur directeur (aussi dit *vecteur normal*) $\mathbf{w} = (w_0, w_1, \ldots, w_p)^\top \in \mathbb{R}^{p+1}$ par

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

Pour classer une observation $\mathbf{x}$ (*i.e.,* affecter une étiquette 1 ou $-1$) on utilise alors $\text{sign}\left(\hat{f}_{\mathbf{w}}(\mathbf{x})\right)$, où la fonction sign est définie par

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

Ainsi, la fonction $\mathbf{x} \mapsto \text{sign}(\hat{f}_{\mathbf{w}}(\mathbf{x}))$ est le classifieur binaire de frontière linéaire (affine) définie par $\mathbf{w}$.

L'objectif du perceptron est de trouver un hyperplan qui sépare le mieux possible les données en deux groupes. On aimerait donc que de chaque côté de l'hyperplan séparateur, les étiquettes soient le plus possible homogènes. Le vecteur $\mathbf{w}$ est appelé vecteur de poids. Le coefficient $w_0$ est l'ordonnée à l'origine (*intercept* en anglais). Pour plus de détails sur l'intérêt de cette méthode on pourra se référer à [HTF09].

Utilisez les données artificielles déjà construites pour illustrer les questions suivantes.

1) A quoi correspond la frontière de décision du perceptron en dimension $p = 2$ ? Trouvez (à la main) une bonne séparatrice sur les trois jeu de données simulées. Quand est-ce que $\hat{f}_{\mathbf{w}}(\mathbf{x})$ est grand ? négatif ? positif ? Quelle est la signification géométrique de cette fonction ? À quoi correspond $w_0$ ?
2) Vérifiez que la fonction `predict(x,w)` prend en entrée un vecteur $\mathbf{x} \in \mathbb{R}^p$ et un vecteur poids $\mathbf{w} \in \mathbb{R}^{p+1}$ et renvoie le vecteur de prédiction $\hat{f}_{\mathbf{w}}(\mathbf{x})$. Vérifiez ensuite que `predict_class(x,w)` renvoie bien l'étiquette prédite $\text{sign}\left(\hat{f}_{\mathbf{w}}(\mathbf{x})\right)$.

## Fonction de coût

Afin de mesurer l'erreur commise sur l'ensemble d'un jeu de données $\mathcal{D}_n$ il est nécessaire de se fixer une fonction de perte $\ell : \mathbb{R} \times \mathcal{Y} \mapsto \mathbb{R}^+$ qui mesure le coût d'une erreur lors de la prédiction d'un exemple. Le coût que l'on veut minimiser (en fonction de $\mathbf{w}$) est $\mathbb{E}\left[\ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y)\right]$, l'espérance de la fonction de perte sur l'ensemble des données. Trois fonctions de perte sont utilisées habituellement et définies ci-dessous :

— le pourcentage d'erreur : $\text{ZeroOneLoss}(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) = |y - \text{sign}(\hat{f}_{\mathbf{w}}(\mathbf{x}))|/2$,

— l'erreur quadratique : $\text{MSELoss}(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) = (y - \hat{f}_{\mathbf{w}}(\mathbf{x}))^2$,

— l'erreur *hinge* (*i.e.,* charnière en français) : $\text{HingeLoss}(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) = \max(0, 1 - y \cdot \hat{f}_{\mathbf{w}}(\mathbf{x}))$.

Cette partie à pour but d'étudier ces différentes fonction de pertes. Ces trois fonctions sont codées dans le fichier source (ainsi que les gradients associés).

3) Supposons que $\mathbf{x} \in \mathbb{R}^p$ et $y \in \mathbb{R}$ soient fixes. Quelle est la nature des fonctions

$$\mathbb{R}^{p+1} \to \mathbb{R}^+$$
$$\mathbf{w} \mapsto \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y)$$

pour les trois pertes étudiées : constante, linéaire, quadratique, constante par morceaux, linéaire par morceaux, quadratique par morceaux, etc. ?

# 5 Algorithme de descente de gradient stochastique

Dans le cas général, il est bien sûr impossible de faire une recherche exhaustive de l'espace $\mathbb{R}^{p+1}$ où évolue $\mathbf{w}$ afin de trouver le coût minimum. De plus on ne peut pas observer $\mathbb{E}\left[\ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y)\right]$, on peut donc seulement tenter de minimiser sa contrepartie empirique : $\sum_{i=1}^{n} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$.

La méthode du perceptron consiste à utiliser une variante de l'algorithme de descente du gradient, une méthode usuelle et générale d'optimisation de fonction différentiable. La méthode de descente de gradient est itérative : à chaque étape, le poids courant est corrigé dans la direction du gradient, mais en sens opposé. L'algorithme converge dans le cas général vers un minimum local pour peu que le pas soit bien choisi. De plus, le minimum atteint est global pour les fonctions convexes.

La méthode du gradient stochastique est une variante qui propose de ne pas utiliser le gradient complet, qui requiert de calculer une somme sur les $n$ observations, mais plutôt de tirer (aléatoirement ... ou non) un couple $(\mathbf{x}_i, y_i)$ sur laquelle on calcul un gradient. On peut aussi montrer que cet algorithme converge sous certaines conditions *cf.* [SSBD14, page 157] ou [Bot98].

L'algorithme du perceptron est décrit de la manière suivante :

**Algorithme 1 :** Perceptron (version cyclique)

**Data :** les observations et leurs étiquettes $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : 1 \leq i \leq n\}$ ; le pas de gradient : $\epsilon$ ; le nombre maximal d'itérations : $n_{\text{iter}}$ ;
**Result :** $\mathbf{w}$
initialiser (aléatoirement) $\mathbf{w}$ ; initialiser $j = 0$
**while** $j \leq n_{\text{iter}}$ **do**
  $\mathbf{w} \leftarrow \mathbf{w}$
  **for** $i = 1$ *to* $n$ **do**
    $\mathbf{w} \leftarrow \mathbf{w} - \epsilon \nabla_{\mathbf{w}} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$
  $j \leftarrow j + 1$

Remarque La méthode du gradient stochastique est aussi disponible dans `sklearn` sous le nom `SGDClassifier` (SGD est l'abréviation *Stochastic Gradient Descent*). Une description est donnée sur la page : http://scikit-learn.org/stable/modules/sgd.html.

1) Décrire un pseudo algorithme "Perceptron version aléatoire", une variante qui visite les observations en effectuant un tirage aléatoire uniforme (faire le cas avec remise et sans remise). On ne demande pas de re-coder cette fonction ici. C'est cette version qui est généralement appelée méthode de descente de gradient stochastique. Remarquez que c'est celle qui est proposée par défaut dans la fonction `gradient`.
2) On va observer graphiquement avec `plot_gradient` l'évolution de $\frac{1}{n}\sum_{i=1}^{n} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$ selon $\mathbf{w}$ suivant les étapes de l'algorithme. La descente de gradient simple est aussi appelée *batch* (*i.e.,* lot ou stock en français), et consiste à calculer le vrai gradient $\frac{1}{n}\sum_{i=1}^{n} \nabla_{\mathbf{w}} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$. Lancer l'exemple avec et sans l'option stochastique. Pourquoi le nombre d'itérations est `niter *len(y1)` pour le cas stochastique ?
3) Expérimentez sur différents jeux de données : utiliser soit les fonctions fournies dans le fichier source, soit la fonction de `sklearn`. Étudiez les performances selon les points suivants : le nombre d'itérations, la fonction de coût, la difficulté du problème (si les classes sont facilement séparables ou non par un hyperplan). Observez-vous des comportements étranges, si oui quelle en est la raison ?

4) Étudiez comme précédemment le comportement de l'algorithme de descente de gradient, avec l'option `stoch=True` désactivée.

5) Question optionnelle : étudiez numériquement la vitesse de convergence dans le cas suivant : les $\mathbf{x}_i$ sont des points uniformément repartis sur les segments $\{0\} \times [0, M]$ (alors les $y_i$ valent 1) ou bien sur le segment $\{\delta\} \times [0, M]$ (alors les $y_i$ valent $-1$). De plus la proportion de 1 est égale à 1/2. On regardera l'impact de $\delta, M$ et $n$ sur le temps de convergence du perceptron.

6) Proposez/codez des variantes de conditions d'arrêt pour l'algorithme.

7) Afficher sur une même figure l'évolution des frontières en fonctions des itérations. On pourra utiliser la fonction `plot_2d` et son argument `alpha_choice` pour cela.

8) Question optionnelle : trouver un fonction de perte telle que l'algorithme du perceptron soit équivalent à la version donnée par l'Algorithme 2 (qui est la version initiale de l'algorithme). Interprétez la condition suivante : $\hat{f}_{\mathbf{w}}(\mathbf{x}_i) \cdot y_i \leq 0$.

**Algorithme 2 :** Perceptron "classique" (version cyclique)

**Data :** les observations et leurs étiquettes $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : 1 \leq i \leq n\}$ ; le pas de gradient : $\epsilon$ ; le nombre maximal d'itérations : $n_{\text{iter}}$ ;
**Result : w**
initialiser (aléatoirement) $\mathbf{w}$ ; initialiser $j = 0$
**while** $j \leq n_{\text{iter}}$ **do**
  $\mathbf{w} \leftarrow \mathbf{w}$
  **for** $i = 1, \ldots, n$ **do**
    **if** $\hat{f}_{\mathbf{w}}(\mathbf{x}_i) \cdot y_i \leq 0$ **then**
      $\mathbf{w} \leftarrow \mathbf{w} + \epsilon \begin{pmatrix} 1 \\ \mathbf{x}_i \end{pmatrix} \cdot y_i$
  $j \leftarrow j + 1$

Pour aller plus loin on pourra consulter `hagan.okstate.edu/4_Perceptron.pdf`, et [Sha11, SSBD14] pour un point de vue plus moderne sur la technique du gradient stochastique.

### Perceptron : linéaire ... seulement ?

9) Quelle est la formule analytique d'une ellipse, d'une hyperbole et d'une parabole en 2D ?

10) Proposez une méthode pour réussir à classifier le jeu de données `clown` en créant des interactions d'ordre deux. En pratique, peut-on généraliser au delà de l'ordre deux facilement ? On pourra utiliser la fonction `poly2` du fichier source, qui plonge les données bi-dimensionnelles dans l'espace des fonctions polynomiales de degré 2 en les données.

11) Sur le jeu de données `clown`, faites quelques expériences en transformant vos données et tracez les frontières de décision.

## Références


[Bot98] L. Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9) :142, 1998. 6

[HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer Series in Statistics. Springer, New York, second edition, 2009. `http://www-stat.stanford.edu/~tibs/ElemStatLearn/`. 5

[Sha11] S. Shalev-Shwartz. Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4(2) :107–194, 2011. `http://www.cs.huji.ac.il/~shais/papers/OLsurvey.pdf`. 7

[SSBD14] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning : From theory to algorithms*. Cambridge University Press, 2014. 6, 7