M1 Informatique AMU – TP1 ISD — 2023/24 — Rapport. **Date : 10 octobre 2023.**
Nom, prénom [+éventuel binôme], groupe TP# 1

Partie : Effets de normalisation

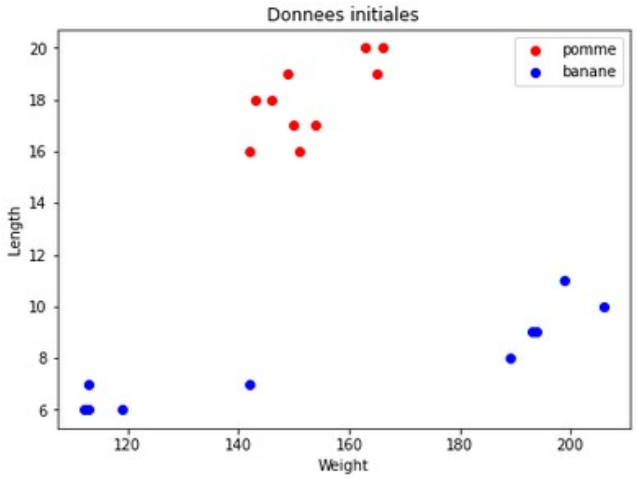
Tout d'abord, voici le code permettant de récupérer le jeu de données **fruits**, auquel nous avons ajouté par réflexe l'instruction « describe » pour obtenir les statistiques élémentaires. Via ces statistiques, nous constatons qu'il n'y a pas de données manquantes, que les échelles des attributs sont différentes, de même que les informations statistiques élémentaires. Ces échelles différentes doivent nous inciter à procéder à une normalisation du jeu de données.

```
import pandas as pd
poba = pd.read_csv('fruits.csv')
poba.describe()
```

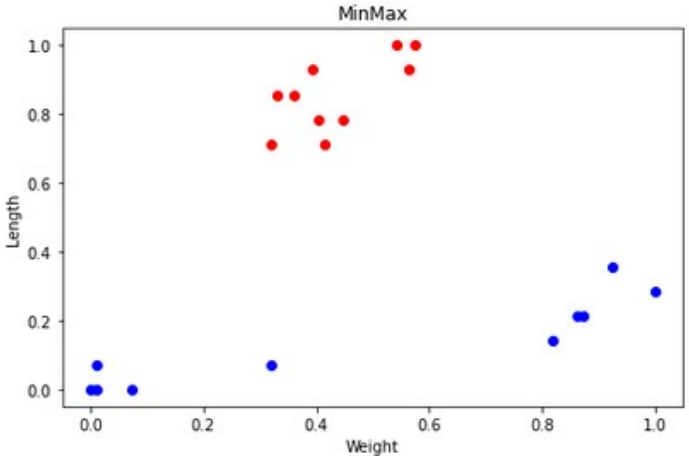
	Weight	Length	Fruit
count	20.000000	20.000000	20.000000
mean	155.450000	12.950000	0.500000
std	29.247537	5.423875	0.512989

Voici les nuages de points bruts (sans normalisation), et pour chaque procédé de normalisation, le nuage obtenu (code en annexe).

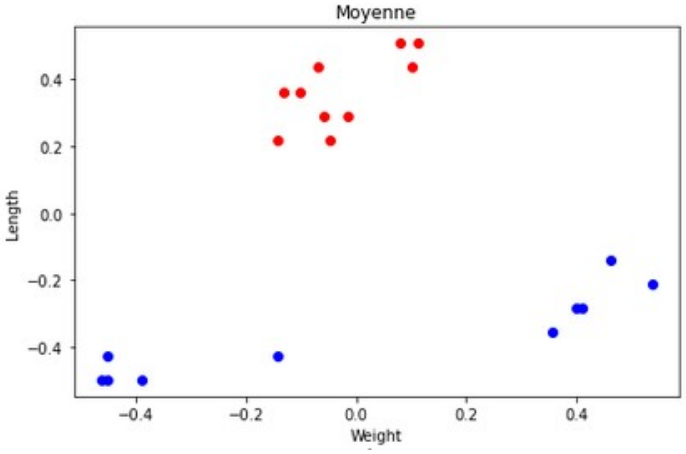
Nuage brut



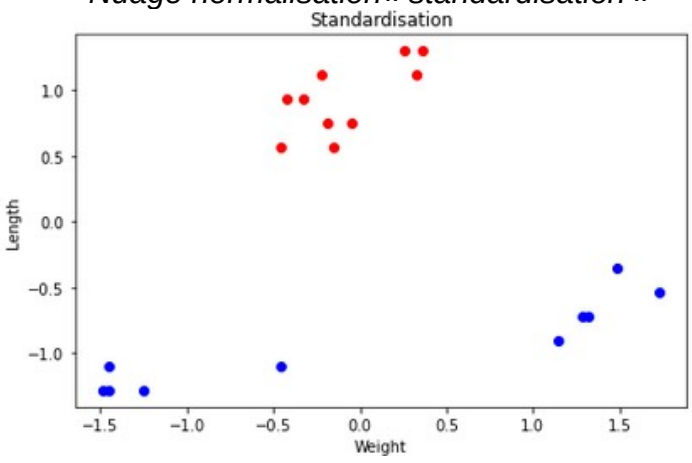
Nuage avec normalisation MinMax



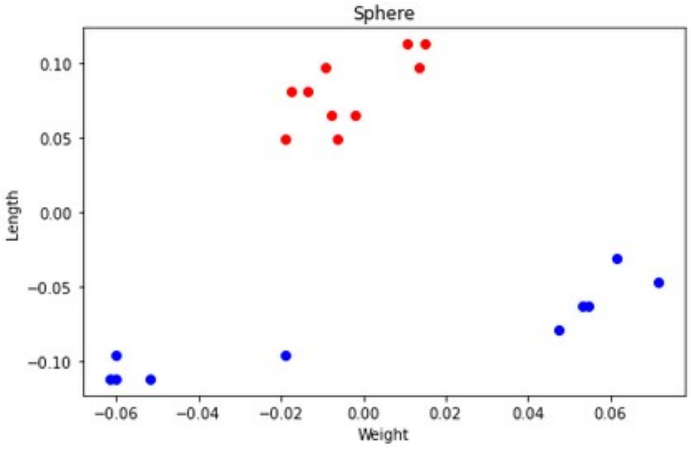
Nuage normalisation moyenne



Nuage normalisation« standardisation »



Nuage avec normalisation « ball » (sphere)



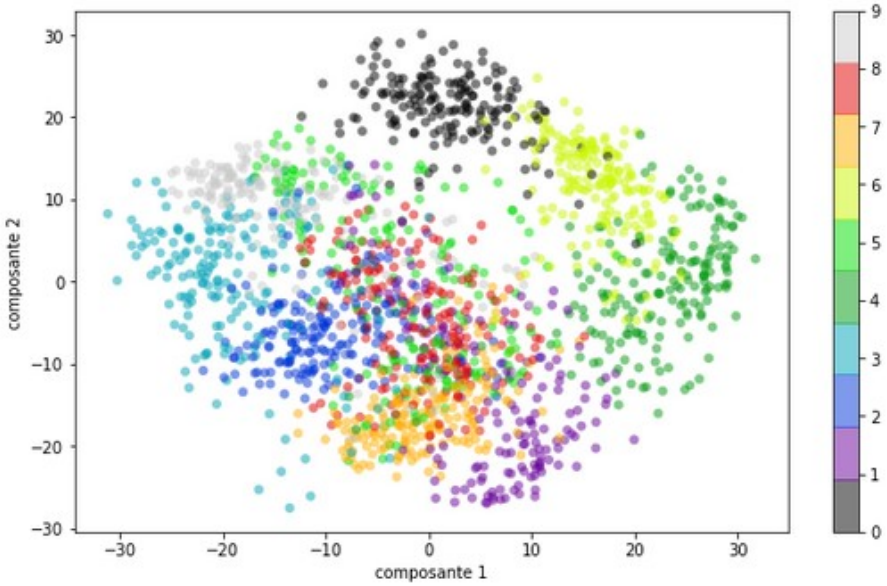
Commentaire

Les échelles des axes diffèrent selon la normalisation choisie, mais les variances et dispersions semblent respectées quel que soit le mode de normalisation. Donc finalement, grâce à la normalsation, on obtient la possibilité de comparer entre elles les deux colonnes (variables length et weight) qui deviennent du même ordre de grandeur, initialement avec ou sans moyennes et/ou écart-types identiques. Le choix de la bonne méthode de normalisation se fera au regard des autres colonnes, et de la nécessité de les comparer sur des échelles de grandeur comparables.

Codes de production des nuages de points, et de normalisation : fichier tp1_normalisation.py

Partie : Analyse en composantes principales (ACP)

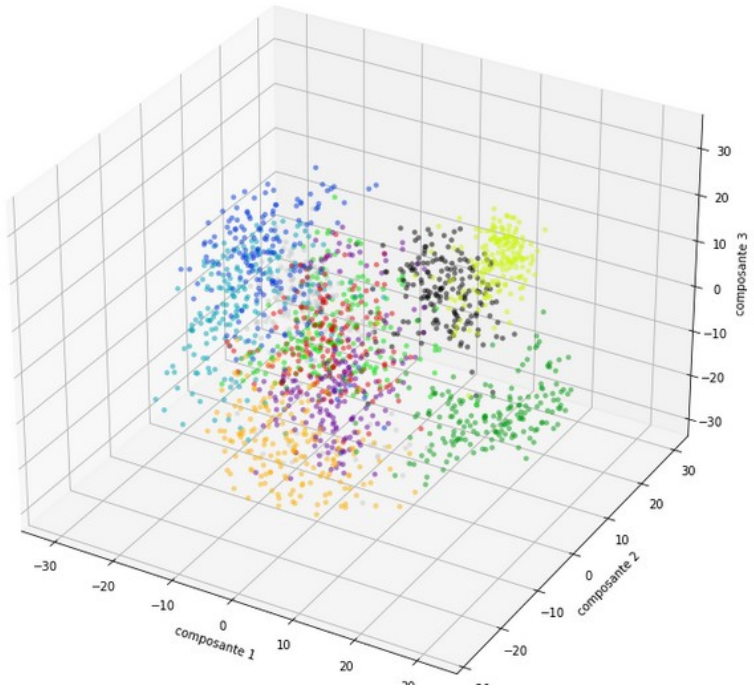
Voici le graphique obtenu suite à une ACP dont on a extrait les deux premières composantes principales.



On peut repérer les classes correspondant aux chiffres (par exemple, orange = digit 7). Certaines sont homogènes en formant un cluster plutôt dense (le orange=7, violet=1, noir=0, etc.), et d'autres sont assez éclatés (par exemple, le vert 5, ou le bleu clair 3).

Certaines classes se confondent peu avec les autres (ex. le gris foncé/noir 0), alors qu'il existe des confusions entre certaines d'entre elles et beaucoup d'autres (ex. la classe rouge --8-- se mélange avec beaucoup d'autres, donc confusion entre ces classes).

Passons à une projection selon les trois principales composantes :



La plupart des classes (sauf celles au centre) se détachent mieux les unes des autres avec 3 variables de projection (composantes) plutôt qu'avec deux : il y a un vrai apport d'information au regard des confusions entre classes, et toujours la possibilité de visualiser le jeu de données. Certaines confusions ne semblent pas résolues, cependant (ex. rouge).

Codes relatifs à l'ACP 3 composantes : fichier tp1_acp.py

Partie : Données manquantes

Le code pour obtenir le nombre de valeurs manquantes global et par colonne est le suivant, où nos vérifions en passant la cohérence entre ces deux informations. Le résultat est cohérent (total de 1737 valeurs manquantes, réparties dans les colonnes).

```
# nb de valeurs manquantes sur
# toutes les colonnes du dataset D
def nb_val_manquantes(D):
    nbmv = 0
    for x in np.nditer(D):
       if np.isnan(x):
        nbmv += 1
    return nbmv

nbmv = nb_val_manquantes(mdd1)
```

```
# nb val manquantes par colonne (retour: tableau numpy)
def nbvmanquantes_col(X):
    nbcol = X.shape[1]
    nbmvcol = np.zeros(nbcol, dtype=int) #nb de val manquantes par colonne (variable)
    for col in range(nbcol):
        for x in np.nditer(X[:,col]):
            if np.isnan(x):
                nbmvcol[col] += 1
    return nbmvcol

nbmvcol = nbvmanquantes_col(mdd1)
print("Nombre de valeurs manquantes colonne par colonne = ", nbmvcol)
print("Nombre de valeurs manquantes au total = ", nbmvcol.sum(), " (must be equal to : ", nbmv, ')')
```

```
Nombre de valeurs manquantes colonne par colonne =  [37 32 37 23 33 40 32 26 24 29 27 27 21 32 34 34 35 26 26 22 30
23 28 32
 23 26 28 28 22 23 33 31 23 17 25 25 33 24 22 30 18 23 31 31 31 18 15 28
 24 26 20 26 27 27 23 24 33 30 26 28 31 19 31 24]
Nombre de valeurs manquantes au total =  1737  (must be equal to :  1737 )
```

Après avoir implanté les méthodes d'imputation – en annexe –, et les avoir testées via code ci-après (les tests sont possibles puisque nous savons quelles valeurs ont été supprimées, cf fonction score_imputation, donc nous pouvons comparer avec les valeurs imputées pour mesurer les écarts avec la « vérité terrain » – ground truth), nous obtenons :

Entrée [61]:
```
# MEILLEURE METHODE ?
def score_imputation(Xinit, Ximputed):
    nb_row = Xinit.shape[0]
    nb_col = Xinit.shape[1]
    error = 0.
    for i in range(nb_row):
        for j in range(nb_col):
            error = error + (Xinit[i,j] -  Ximputed[i,j]) ** 2
    return error
```

Entrée [62]:
```
# Calcul de l'erreur de chaque methode
data = load_digits().data
erreur_stationnaire = score_imputation(data, mdd1)
erreur_moy_cond = score_imputation(data, mdd2)
erreur_kppv = score_imputation(data, mdd3)
erreurs = [erreur_stationnaire, erreur_moy_condi, erreur_kppv]
print('Erreur stationnaire = ', erreur_stationnaire)
print('Erreur Moy/classe = ', erreur_moy_cond)
print('Erreur kppv = ', erreur_kppv)
```

```
Erreur stationnaire =  68365.0
Erreur Moy/classe =  18067.443026479214
Erreur kppv =  6661.551020408174
```

Conclusion. Dans le cas de ce jeu de données, et de ces valeurs manquantes à peu près équi-réparties dans les colonnes, il s'avère que l'erreur moindre est celle des *kppv*, et la pire est la stationnaire. Il ne faut pas en faire une généralité, cela dépend beaucoup des domaines de valeurs des variables (colonnes).

Les fonctions codant les méthodes d'imputation, et celles de comparaison ci-dessus, sont fournies en annexe dans le fichier tp1_md.py