Apprentissage automatique non-supervisé

Introduction

À la différence de l'apprentissage supervisé, l'apprentissage non supervisé est celui où l'algorithme doit opérer à partir d'exemples non annotés. En effet, dans ce cas de figure, l'apprentissage par la machine se fait de manière entièrement indépendante. Des données sont alors renseignées à la machine sans qu'on lui fournisse des exemples de résultats.

Ainsi, dans cette situation d'apprentissage, les réponses que l'on veut trouver ne sont pas présentes dans les données fournies : l'algorithme utilise des données non étiquetées. On attend donc de la machine qu'elle crée elle-même les réponses grâce à différentes analyses et au classement des données.

Dans ce cadre, l'ensemble des données collectées est traité comme des variables aléatoires. En effet et contrairement à l'apprentissage automatique qui se doit de trouver un modèle à partir de données étiquetées : f(X) Y, il utilise seulement des données non étiquetées : il n'y a pas de variable Y à prédire.

L'utilisation de l'apprentissage non supervisé peut être réunie en problèmes de *clustering* et *d'association*.

Clustering

Un problème de clustering est un problème pour lequel on attend de la machine qu'elle **rassemble sous forme de cluster** des objets présents dans des groupes de données, et ce de la manière la plus juste et efficace possible. Cette technique, bien que parfois difficile à comprendre par l'homme, est très utilisée dans le domaine du marketing pour placer dans des groupes les différents clients par exemple. Un exemple d'algorithme très souvent utilisé dans le clustering est le K-means.

Association

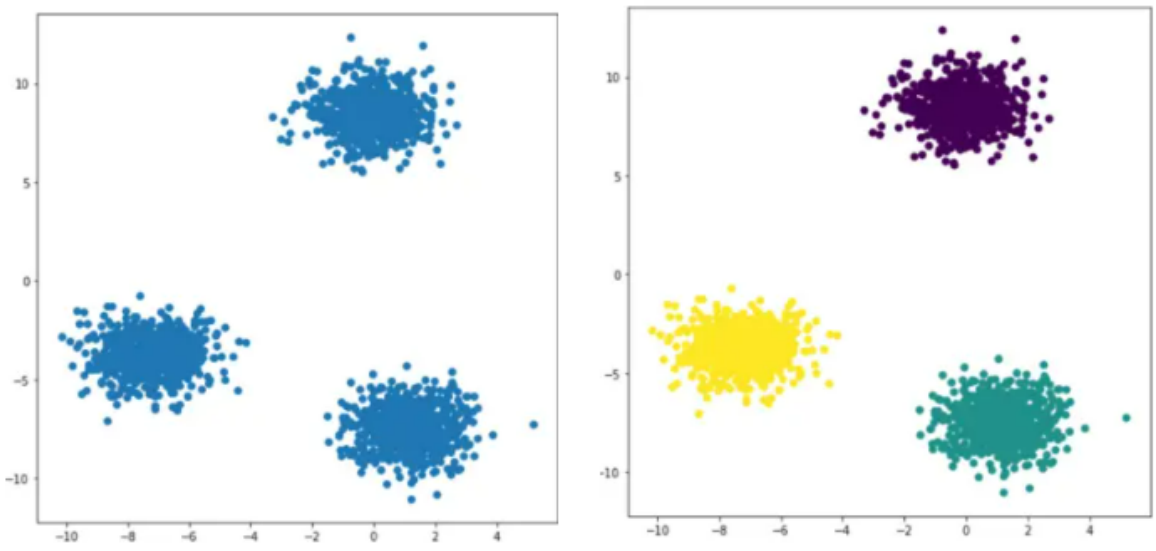
Le système d'association permet de trier et regrouper les données qui peuvent être liées grâce à certaines caractéristiques. L'objectif est donc de **trouver des objets liés les uns aux autres sans qu'il s'agisse néanmoins d'objets identiques**. À titre d'illustration, en fournissant à l'algorithme des images de chats et d'accessoires pour chats, alors l'algorithme d'apprentissage non supervisé ne regrouperait pas tous les chats ensemble mais par exemple une pelote de laine avec un chat. Un exemple d'algorithme très souvent utilisé dans l'association est l'algorithme A-priori.

L'apprentissage non supervisé est très souvent utilisé dans le domaine de la reconnaissance vocale (Alexa, Siri, etc)

K-means

Le principe de l'algorithme

Étant donnés des points et un entier k, l'algorithme vise à diviser les points en k groupes, appelés clusters, homogènes et compacts. Regardons l'exemple ci-dessous :



L'idée est assez simple et intuitive.

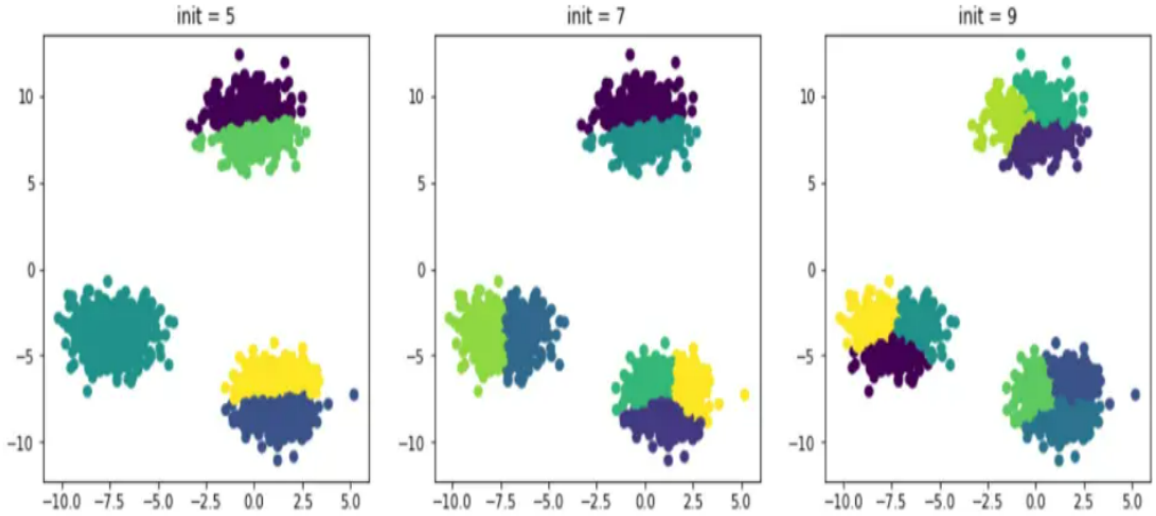
- La première étape consiste à définir 3 centroïdes aléatoirement auxquels on associe 3 étiquettes par exemple 0,1,2.
- Ensuite nous allons pour chaque point regarder leur distance aux 3 centroïdes et nous associons le point au centroïde le plus proche et l'étiquette correspondante. Cela revient à étiqueter nos données.
- Enfin on recalcule 3 nouveaux centroïdes qui seront les centres de gravité de chaque nuage de points labellisés.
- On répète ces étapes jusqu'à ce que les nouveaux centroïdes ne bougent plus des précédents. Le résultat final se trouve sur la figure de droite.

Notion de distance et initialisation

Vous l'aurez compris dans cet algorithme deux points sont clé : **Quelle est la métrique utilisée pour évaluer la distance entre les points et les centroïdes ? Quel est le nombre de clusters à choisir ?**
Pour la première question, **la distance Euclidienne** est généralement utilisée.

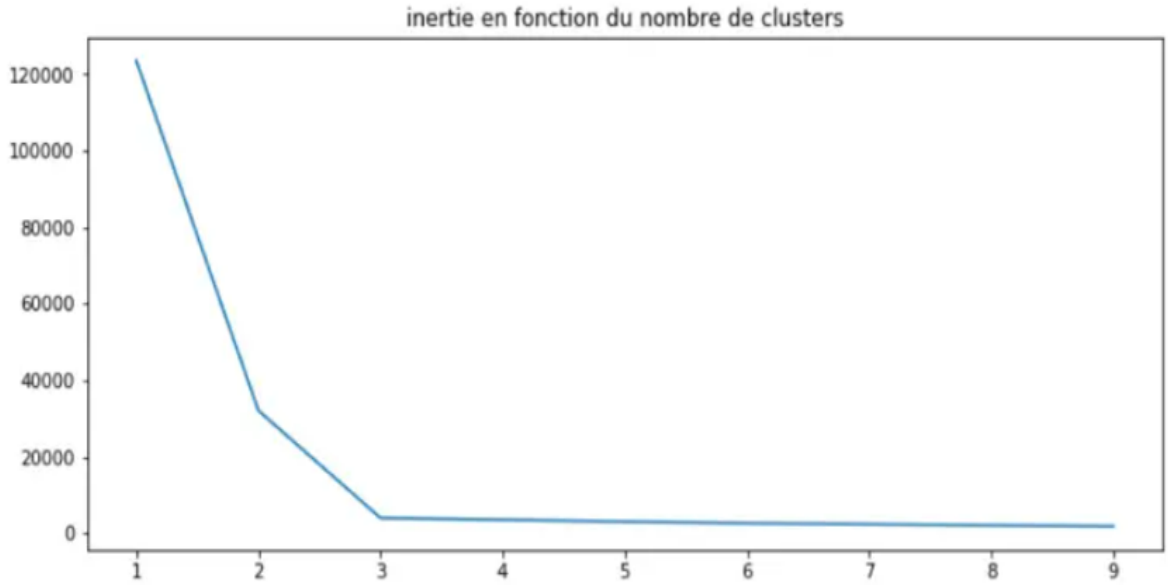
Elle permet d'évaluer la distance entre chaque point et les centroïdes. Pour chaque point on calcule la distance euclidienne entre ce point et chacun des centroïdes puis on l'associe au centroïde le plus proche c'est-à-dire celui avec la plus petite distance.

Dans l'exemple précédent il était aisé de trouver le nombre idéal de clusters simplement en visualisant graphiquement. Généralement **les jeux de données ont plus de deux dimensions et il est donc difficile de visualiser le nuage de points et d'identifier rapidement le nombre de clusters optimal**.
Supposons dans l'exemple précédent que nous n'avons pas visualiser les données avant et décidons de tester différentes fois avec un nombre de clusters initiaux différents. Voici les résultats obtenus :



Le partitionnement est inexact car le nombre de clusters initiaux est bien supérieur au nombre idéal en l'occurrence 3.

Il existe des méthodes pour déterminer le nombre de clusters idéal. La plus connu est la méthode du coude. Elle s'appuie sur la notion d'inertie : la somme des distances euclidiennes entre chaque point et son centroïde associé. (plus on fixe un nombre initial de clusters élevés et plus on réduit l'inertie : les points ont plus de chance d'être à côté d'un centroïde.)



On remarque que l'inertie stagne à partir de 3 clusters. Cette méthode est concluante (Pas besoin d'aller plus dans les détails concernant cette méthode et d'autres, ce n'est pas le but de notre TP, l'essentiel ici c'est que vous soyez au courant de cette mesure).

Enfin, il est également primordial en plus de ces deux méthodes de faire une analyse poussée des clusters créés. J'entends par là une analyse descriptive précise et approfondie pour déterminer les caractéristiques communes de chaque cluster. Cela vous permettra de comprendre les profils types de chaque cluster.

K-means avec sklearn

L'entraînement d'un algorithme K-means est facilité avec la librairie Scikit-Learn. Notre jeu de données qui a servi d'exemples se crée facilement avec la fonctionalité make_blobs de Scikit-Learn. Voici comment l'implémenter en code :

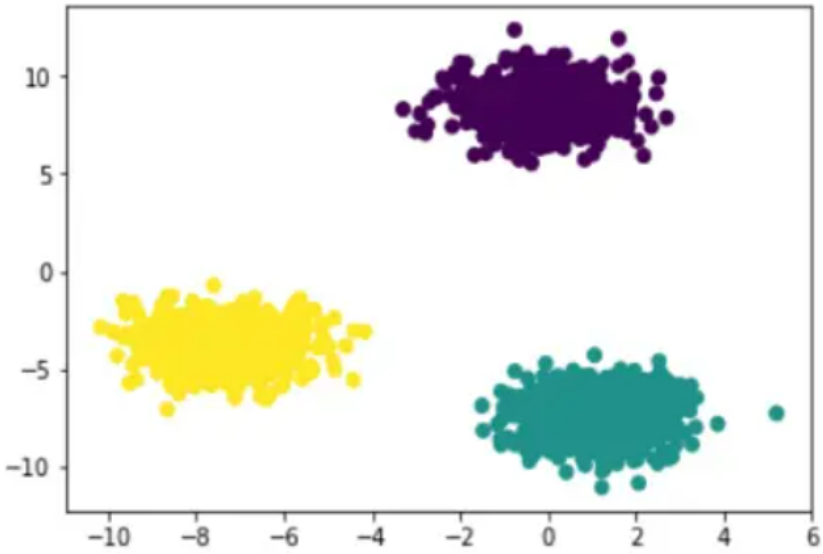
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

n_samples = 2000
random_state = 130
X,y = make_blobs(n_samples=n_samples, random_state=random_state)
```

Comme vous l'avez surement deviné, la fonctionalité make_blobs nous sert à créer l'ensemble de données sur lequel on appliquera K-means. Elle prend deux paramètres : *n_samples* il s'agit du nombre total de points répartis également entre les clusters (par défaut = 100). *random_state* détermine la génération de nombres aléatoires pour la création de l'ensemble de données (par défaut = aucun).

Le retour de make_blobs est dans les variables : X Les échantillons générés, y les étiquettes entières pour l'appartenance au cluster de chaque échantillon.

Maintenant que nous avons notre jeu de données on peut passer à l'implémentation d'un K-means. Nous allons utiliser la librairie Scikit-Learn et visualiser le résultat :

```
model = KMeans(n_clusters=3, random_state=random_state).fit(X)
y_pred = model.predict(X)
plt.scatter(X[:, 0], X[:, 1], c = y_pred);
```



L'attribut n_clusters de la classe KMeans de Scikit-Learn permet de fixer le nombre de centroïdes que l'on souhaite. Par conséquent elle définit aussi le nombre de centroïdes initiaux.

Cette classe utilise par défaut non pas une méthode d'initialisation aléatoire mais une méthode développée en 2007 par David Arthur et Sergei Vassilvitskii dénommée K-MEANS ++.

Elle consiste à sélectionner des centroïdes distincts les uns des autres ce qui limite les risques de convergence vers une solution non-optimale. La distance par défaut utilisée est bien évidemment la distance euclidienne.

Exercice 1 :

On va essayer de déterminer les cluster_centers d'un nuage de points.

Tout d'abord, on commence par positionner nos points dans un repère cartésiens (2D). pour cela il suffit de préciser les coordonnées des points dans une liste. (penser à utiliser numpy).

Par la suite, on lance l'apprentissage de notre k-means avec 2 clusters pour commencer.

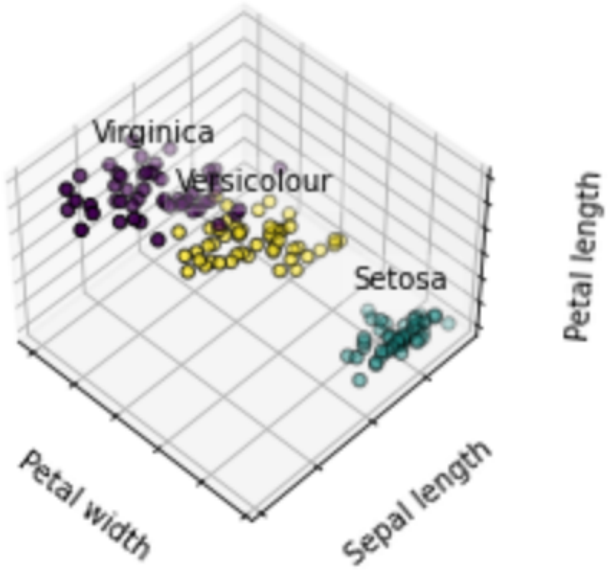
Pour finir, on demande la prédiction de 2 nouveaux points et on affiche les centres des clusters avec la fonction : cluster_centers_

Exercice 2 :

Pour cette exercice on va reprendre la base de données IRIS. Le but est visualiser et clusteriser les différents points dans l'espace (chaque point représentant une fleur de la base IRIS). La visualisation se fera avec la bibliothèque Axes3D.

La séparation en cluster se fera en appliquant k-means sur cette base avec différents paramètres pour voir son comportement.

Avant de commencer, voici à quoi ressemble le clustering de la base Iris avec 3 clusters :



Commençant la réponse à cet exercice d'abord par l'importation de nos data :

```
import matplotlib.pyplot as plt
# pour la projection 3D des clusters
from mpl_toolkits.mplot3d import Axes3D

from sklearn.cluster import KMeans
from sklearn import datasets


iris = datasets.load_iris()
X = iris.data
```

Question 1 : Ajouter la commande pour lancer l'apprentissage avec k-means en donnant 8 comme nombre de clusters. (n'oubliez pas l'apprentissage sur les labels)

Une fois que c'est fait, on visualise le résultat grâce aux commandes suivante (en utilisant la biblio Axes3D) :

```
fig = plt.figure(1, figsize=(4, 3))
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

ax.scatter(X[:, 3], X[:, 0], X[:, 2],
           c=labels.astype(float), edgecolor='k')

ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])
ax.set_xlabel('Petal width')
ax.set_ylabel('Sepal length')
ax.set_zlabel('Petal length')
ax.set_title('8 clusters')
ax.dist = 12
fig.show()
```

Question 2 : refaire l'apprentissage avec uniquement 3 clusters, puis afficher la figure.

Question 3 : ajouter les paramètres nécessaires (n_init=1,init='random') pour faire une mauvaise initialisation, puis afficher la figure et remarquer s'il y a des changements d'étiquetage.

DBScan

Le principe de l'algorithme : DBScan considère les clusters comme des zones de haute densité séparées par des zones de faible densité. En raison de cette vue plutôt générique, les clusters trouvés par DBScan peuvent avoir n'importe quelle forme, par opposition à k-means qui suppose que les clusters sont de forme convex. Les distances entre les éléments d'une même zone de haute densité (Cluster) sont précisées en passant des paramètres à l'algorithme.

Bon pour les données qui contiennent des clusters de densité similaire.

```
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs

centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=750, centers=centers, cluster_std=0.4,
                            random_state=0)
# DBSCAN
db = DBSCAN(eps=0.3, min_samples=10).fit(X)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True

# Libellé des clusters estimés : 0 1 2..etc
# Si une donnée est pas classée dans un cluster : labels = -1
labels = db.labels_

# Nombre  de clusters dans labels, En ignorant données non classée (si présente).
# set(labels) : afficher en un ensemble le contenu de lables sans redondances
#len : pour la taille
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)

#nombre de données non classées = compter le nombre de data classé en -1
nb_data_non_classée = list(labels).count(-1)

print('Nombre de clusters estimés : %d' % n_clusters_)
print('Nombre de données non classées: %d' % nb_data_non_classée)

print(set(labels))
```

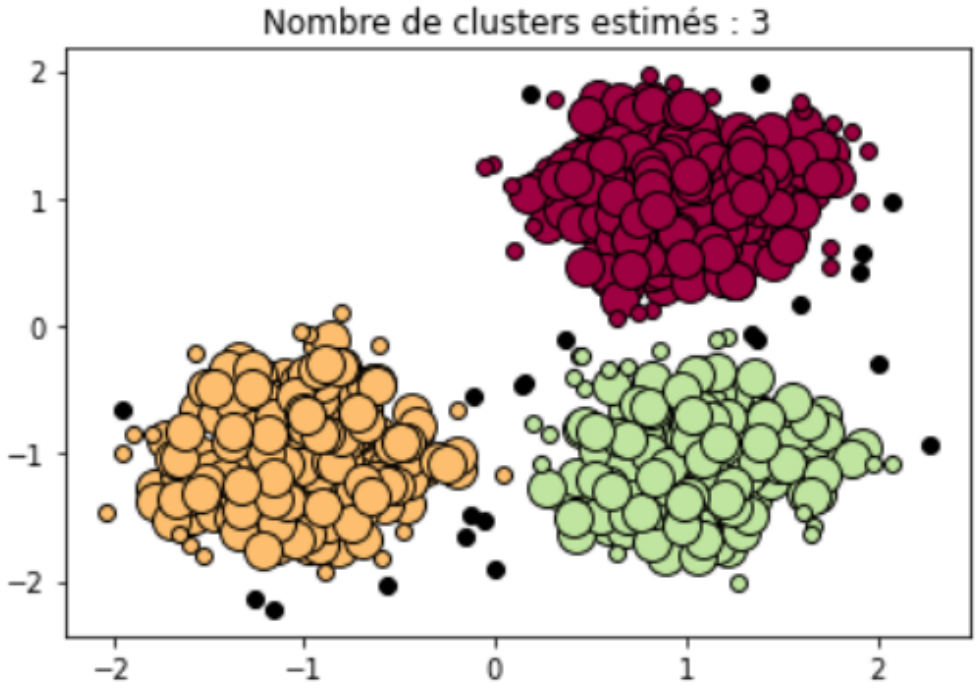
```
Nombre de clusters estimés : 3
Nombre de données non classées: 22
{0, 1, 2, -1}
```

Les paramètres passés pour notre algorithme de clustering : *esp* La distance maximale entre deux échantillons pour que l'un soit considéré comme au voisinage de l'autre, et le nombre d'échantillons dans un quartier pour qu'un point soit considéré comme un point central.

Tandis que la fonction make_blobs a comme paramètres : le nombre d'échantillons, le nombre de centres à générer ou les emplacements des centres fixes, l'écart type des clusters, et Détermine la génération de nombres aléatoires pour la création de l'ensemble de données.

Afin d'afficher le résultat en un scatter plot :

```
import matplotlib.pyplot as plt

# choix des couleurs
# la couleur noire est utilisée pour donnée non classées (label = -1)
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]


for k, col in zip(unique_labels, colors):
    if k == -1:
        # Noire pour data non classée.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    #attribution des couleurs aux données et affichage
    xy = X[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)

    #affichage des données non classée (en noire)
    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Nombre de clusters estimés : %d' % n_clusters_)
plt.show()
```



Dans la figure ci-dessus, la couleur indique l'appartenance au cluster, avec de grands cercles indiquant les échantillons de base trouvés par l'algorithme. Les cercles plus petits sont des échantillons non-noyaux qui font toujours partie d'un cluster. Les données en noire, sont dans aucun cluster.

Réduction de dimension : PCA

L'analyse en composantes principales (ACP ou PCA en anglais pour principal component analysis), est une méthode de la famille de l'analyse des données, qui consiste à transformer des variables liées entre elles (dites " corrélées " en statistique) en nouvelles variables décorrélées les unes des autres.

Ces nouvelles variables sont nommées " composantes principales ", ou axes principaux. Elle permet au praticien de réduire le nombre de variables et de rendre l'information moins redondante.

Pour illuster cette technique, nous réutilisons notre base Iris à nouveau : iris.data contient les informations concernant chaque fleur et iris.target contient les classes de chaque fleur.

Réduction de dimension

Dans le cas des algorithmes non supervisés il est très fréquent de disposer de très grandes quantités de paramètres. Ne sachant pas encore qui est responsable de quoi on a tendance à tout livrer à la machine.

Cela pose 2 problèmes:

- La visualisation des données, au delà de 3 paramètres notre cerveau est bien mal outillé pour se représenter les données
- La complexité des calculs, plus le nombre de paramètres est grand, plus nous aurons des calculs complexes et longs

Pour contourner ces problèmes il est courant de réduire la dimension du vecteur de données à quelque chose de plus simple. La difficulté est alors de réduire le nombre de paramètres tout en conservant l'essentiel de l'information, notamment les variations succeptibles de permettre le regroupement des données.

PCA est une technique linéaire de réduction de dimension qui a l'avantage d'être très rapide. Elle s'utilise simplement:

- Vous définissez le nombre de paramètres
- Vous alimentez l'algorithme avec les données à réduire
- Vous lancez la prédiction ici appelée réduction/transformation

```
from sklearn.decomposition import PCA
# Définition de l'hyperparamètre du nombre de composantes voulues
model = PCA(n_components=2)
# Alimentation du modèle
model.fit(iris.data)
# Transformation avec ses propres données
reduc = model.transform(iris.data )
```

Nous venons de réduire notre vecteur de 4 paramètres en 1 vecteur de 2 paramètres dont les variations sont censées être similaires.

Autrement dit nous devrions être capable de classer nos fleurs avec ces vecteurs réduits en ayant une qualité proche de celle utilisant les vecteurs originaux !

On consulte les valeurs originaux et les valeurs réduites :

```
iris.data[:5]
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
```

```
reduc[:5]
```

```
array([[-2.68412563,  0.31939725],
       [-2.71414169, -0.17700123],
       [-2.88899057, -0.14494943],
       [-2.74534286, -0.31829898],
       [-2.72871654,  0.32675451]])
```

Avant d'ajouter les réductions, rappelons le contenu de notre base :

```
import pandas as pd
data = iris.data
target=iris.target
df = pd.DataFrame(data, columns=iris['feature_names'] )
df['target'] = target
df['label'] = df.apply(lambda x: iris['target_names'][int(x.target)], axis=1)
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	label
0	5.1	3.5	1.4	0.2	0	setosa
1	4.9	3.0	1.4	0.2	0	setosa
2	4.7	3.2	1.3	0.2	0	setosa
3	4.6	3.1	1.5	0.2	0	setosa
4	5.0	3.6	1.4	0.2	0	setosa

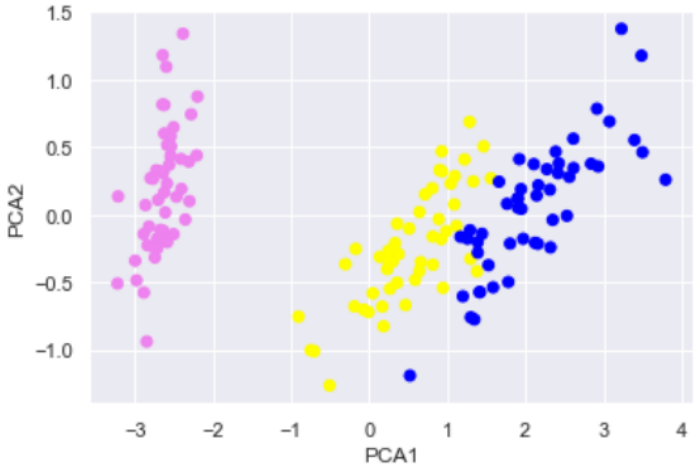
Ajoutons maintenant les nouveaux paramètres dans le dataframe d'origine :

```
df['PCA1'] = reduc[:, 0]
df['PCA2'] = reduc[:, 1]
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	label	PCA1	PCA2
0	5.1	3.5	1.4	0.2	0	setosa	-2.684126	0.319397
1	4.9	3.0	1.4	0.2	0	setosa	-2.714142	-0.177001
2	4.7	3.2	1.3	0.2	0	setosa	-2.888991	-0.144949
3	4.6	3.1	1.5	0.2	0	setosa	-2.745343	-0.318299
4	5.0	3.6	1.4	0.2	0	setosa	-2.728717	0.326755

Puis affichons les nouveaux couples de points (PCA1, PCA2) avec la couleur de l'espèce associée :

```
colors = ['violet', 'yellow', 'blue']
plt.scatter(df['PCA1'], df['PCA2'], c=[ colors[c] for c in df['target'] ]);
plt.xlabel('PCA1')
plt.ylabel('PCA2');
```



Nous obtenons 3 groupes plutôt bien dissociés ! Le résultat est plutôt satisfaisant, et ce qui est assez bluffant c'est que l'algorithme ne connaîssait pas du tout les types de fleurs !

Maintenant, ce nouveau classement peut-il permettre un bon regroupement des 3 espèces ? le graphique semble le confirmer, vérifions cela avec une autre méthodes de clustering.

Gaussian Mixture Models

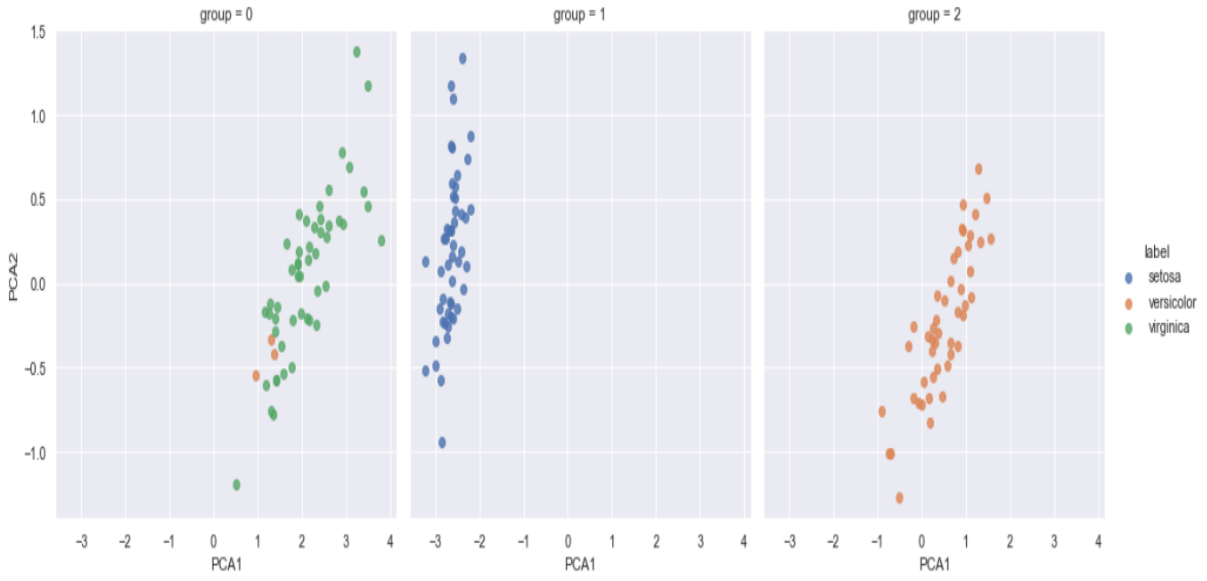
On peut considérer les modèles de mélange gaussien comme une généralisation de l'algorithme de clustering k-means. Gaussian Mixture Models est plus complexe mais s'adapate très bien à différentes formes de clusters.

Nous utilisons ici comme algorithme de clustering le GMM, qui apprend à partir des données et créer les clusters par la suite.

```python
from sklearn.mixture import GaussianMixture
# Création du modèle avec 3 groupes de données
model = GaussianMixture (n_components=3, covariance_type='full')
# Apprentissage, il n'y en a pas vraiment
model.fit(df[['PCA1', 'PCA2']])
# Prédiction
groups = model.predict(df[['PCA1', 'PCA2']])
```

La prédiction étant faîte, pour chaque groupe généré nous affichons la couleur réelle des espèces, si le clustering a été efficace, il n'y aura qu'une seule couleur par groupe :

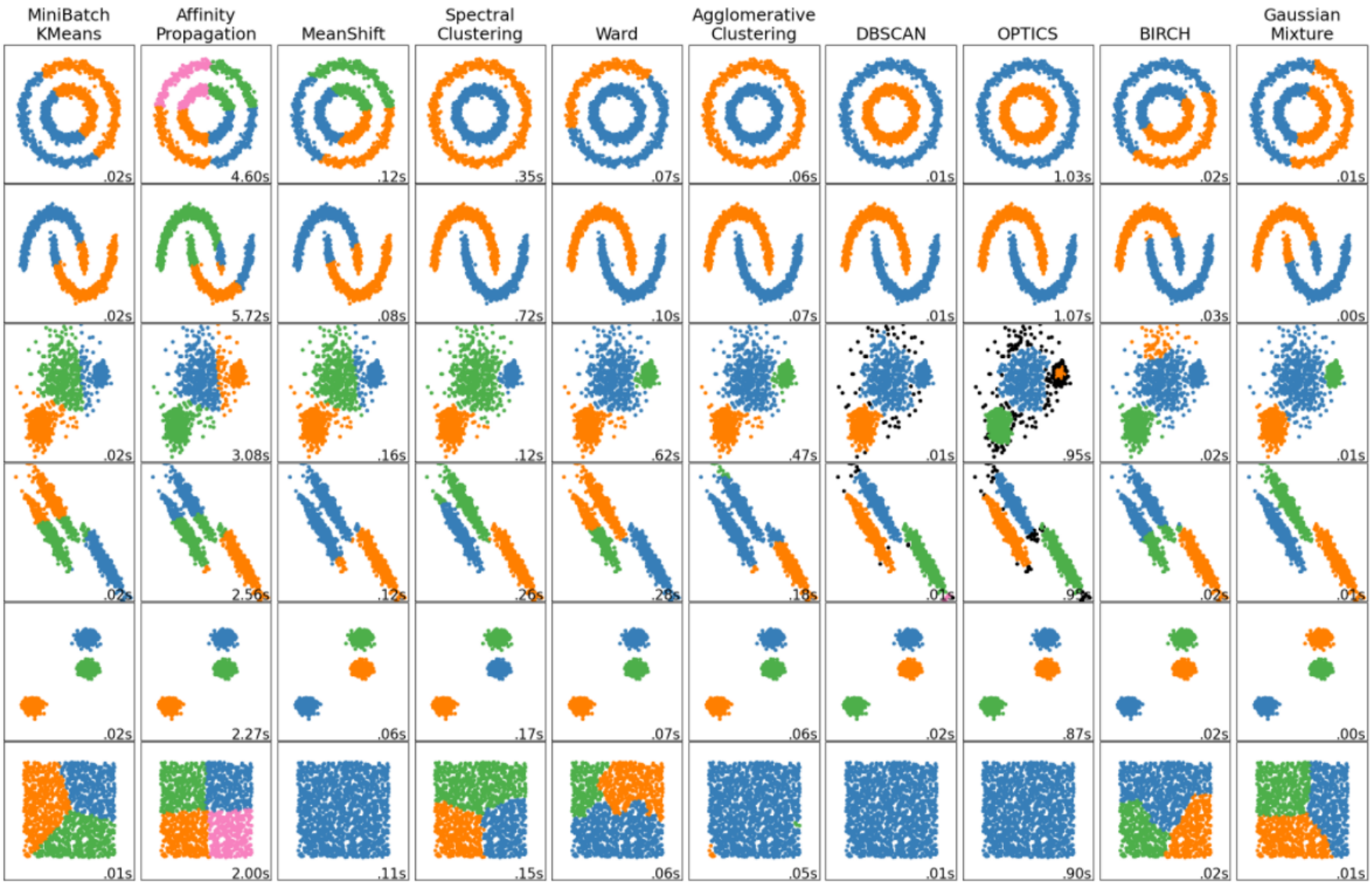
```python
df['group'] = groups
sns.lmplot("PCA1", "PCA2", data=df, hue='label',
           col='group', fit_reg=False);
```



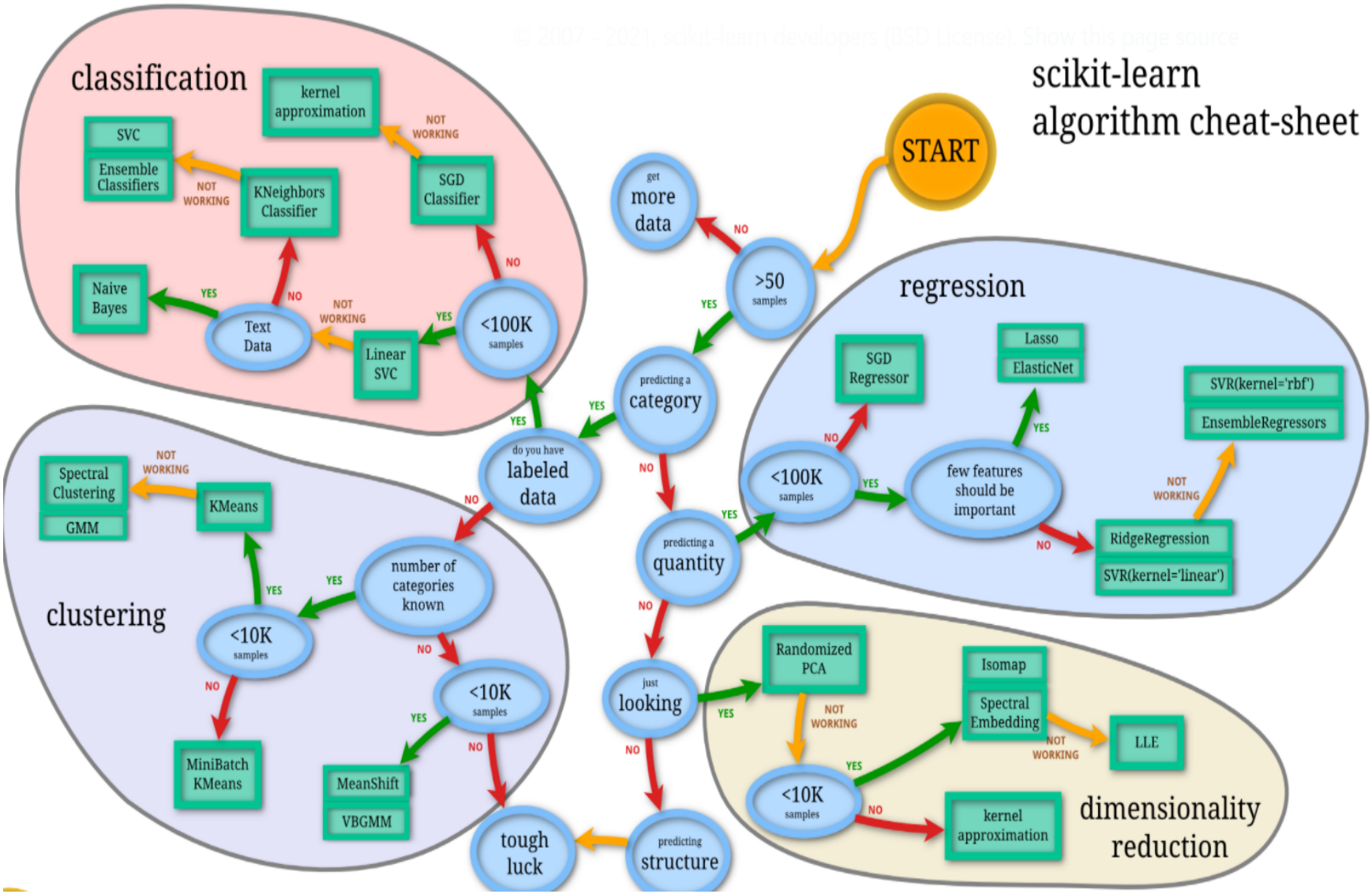
Le groupe *setosa* est très bien identifié, il y a toujours une imprécision entre les classes *virginica* et *versicolor* mais le résultat reste remarquable sur ce petit échantillon.

Références :

- Hands on machine learning with scikit-learn, Keras and TensorFlow. De Aurélien Géron.
- Practical statistics for Data scientists. De PEter Bruce, Andrew Bruce et Peter Gedeck.
- Python Data Science-The Ultimate Handbook for Beginners on How to Explore NumPy for Numerical Data, Pandas for Data Analysis, IPython, Scikit-Learn and Tensorflow for Machine Learning
- et plein d'autres livres dans : knowledgeisle.com



A comparison of the clustering algorithms in scikit-learn


scikit-learn
algorithm cheat-sheet
© 2007 - 2021, scikit-learn developers (BSD License). Show this page source
START
get more data
NO
>50 samples
YES
predicting a category
YES
do you have labeled data
YES
<100K samples
YES
Linear SVC
NOT WORKING
Text Data
NO
Naive Bayes
YES
KNeighbors Classifier
NO
Ensemble Classifiers
NOT WORKING
SVC
SGD Classifier
NOT WORKING
kernel approximation
classification
NO
predicting a quantity
YES
<100K samples
NO
SGD Regressor
YES
few features should be important
YES
Lasso
ElasticNet
NO
RidgeRegression
NOT WORKING
SVR(kernel='rbf')
EnsembleRegressors
SVR(kernel='linear')
regression
NO
number of categories known
YES
<10K samples
YES
KMeans
NOT WORKING
Spectral Clustering
GMM
NO
MiniBatch KMeans
NO
<10K samples
YES
MeanShift
VBGMM
NO
tough luck
clustering
NO
predicting structure
NO
just looking
YES
Randomized PCA
NOT WORKING
<10K samples
YES
Isomap
Spectral Embedding
NOT WORKING
LLE
NO
kernel approximation
dimensionality reduction