

Direction de l'innovation et des relations avec les entreprises

cnrs **formation**
entreprises

Apprentissage non-supervisé

Alexandre Boucaud (CNRS/APC)





L'apprentissage non-supervisé se scinde en deux grandes catégories :

La réduction de dimensions (*dimensionality reduction*)

- Principal Component Analysis (PCA)
- t-SNE, UMAP

Le partitionnement des données (*clustering*)

- K-Means
- DBSCAN
- Gaussian Mixtures





Imports

```
In [1]: import numpy as np
import matplotlib.pyplot as plt

plt.style.use('fivethirtyeight')
```





Préambule

- représentation des données
- `scikit-learn`
- malédiction de la dimension





Représentation des données





Représentation matricielle (convention)

En machine learning, on représente généralement les données présentées aux algorithmes sous forme matricielle.

X est une matrice de taille $(n_samples, n_features)$, représentant des données qui peuvent être associées à une étiquette représentée par un vecteur y de taille $n_samples$.





one sample

$$X = \begin{pmatrix} 1.1 & 2.2 & 3.4 & 5.6 & 1.0 \\ 6.7 & 0.5 & 0.4 & 2.6 & 1.6 \\ 2.4 & 9.3 & 7.3 & 6.4 & 2.8 \\ 1.5 & 0.0 & 4.3 & 8.3 & 3.4 \\ 0.5 & 3.5 & 8.1 & 3.6 & 4.6 \\ 5.1 & 9.7 & 3.5 & 7.9 & 5.1 \\ 3.7 & 7.8 & 2.6 & 3.2 & 6.3 \end{pmatrix}$$

one feature

$$y = \begin{pmatrix} 1.6 \\ 2.7 \\ 4.4 \\ 0.5 \\ 0.2 \\ 5.6 \\ 6.7 \end{pmatrix}$$

outputs / labels





Séparation du jeu de données ab-initio

Afin de pouvoir calculer l'efficacité des algorithmes et les comparer entre eux, on sépare **toujours** le jeu de données en deux parties dénotées `train` et `test`.

L'entraînement des algorithmes `fit()` se fait sur les données d'entraînement `X_train`.

Ces algorithmes sont ensuite appliquées aux données de test `X_test` pour l'évaluation.





training set

$$X = \begin{pmatrix} 1.1 & 2.2 & 3.4 & 5.6 & 1.0 \\ 6.7 & 0.5 & 0.4 & 2.6 & 1.6 \\ 2.4 & 9.3 & 7.3 & 6.4 & 2.8 \\ 1.5 & 0.0 & 4.3 & 8.3 & 3.4 \\ 0.5 & 3.5 & 8.1 & 3.6 & 4.6 \\ 5.1 & 9.7 & 3.5 & 7.9 & 5.1 \\ 3.7 & 7.8 & 2.6 & 3.2 & 6.3 \end{pmatrix}$$

test set

$$y = \begin{pmatrix} 1.6 \\ 2.7 \\ 4.4 \\ 0.5 \\ 0.2 \\ 5.6 \\ 6.7 \end{pmatrix}$$





Evaluer sur des données qui n'ont pas servi à l'entraînement permet d'évaluer la capacité de **généralisation** du modèle aux données.

Les répartitions les plus communes entre `train` et `test` sont de 80%-20% et 70%-30%, en fonction de la taille du jeu de données.

Plus on a de données, plus on peut prendre un jeu de test important.





scikit-learn





Dans ce cours nous allons aborder l'utilisation de la librairie de machine learning principale en Python : `scikit-learn`.

```
import sklearn
```

```
# ou de manière générale
```

```
from sklearn.submodule1 import AlgoA  
from sklearn.submodule2 import AlgoB
```

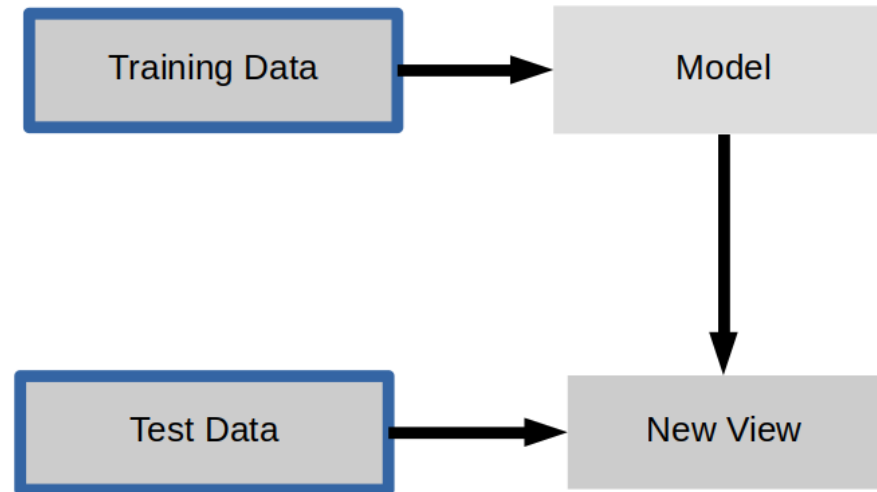
L'avantage de `scikit-learn` est sa simplicité de prise en main. C'est une collection d'algorithmes robustes et éprouvés de machine learning, codés sous forme de classes Python.





L'utilisation est à peu près toujours la même :







Malédiction de la dimension





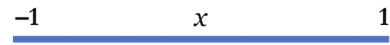
Chaque dimension ajoutée à l'espace des paramètres en fait grandir le volume de manière exponentielle.

100 points équirépartis sur une longueur unitaire $[0, 1]$ sont distants de $\Delta_{1D} = 10^{-2} = 0.01$

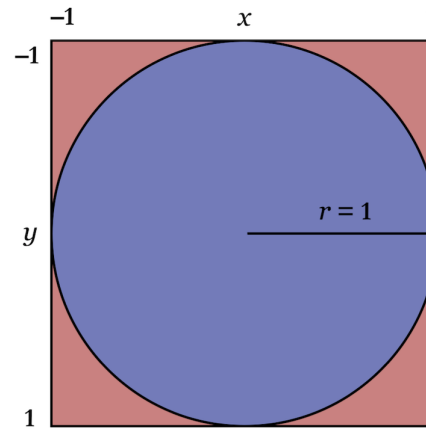
Pour conserver la même densité de points pour un hypercube **unitaire** de dimension 10, il faut $100^{10} = 10^{20}$ points.

A titre de comparaison, on estime à 10^{21} le nombre de grains de sable sur les plages de la Terre combinées.

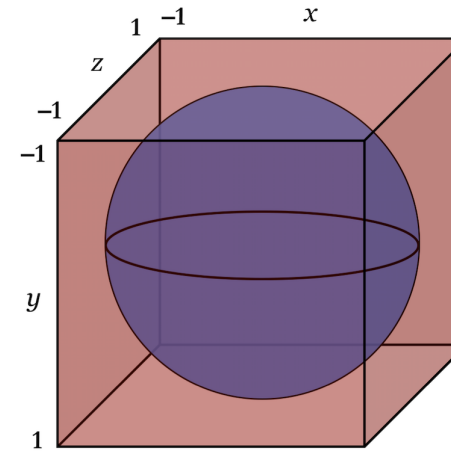




1 dimension ($D = 1$)



2 dimensions ($D = 2$)



3 dimensions ($D = 3$)





La réduction de dimensions





Pour quoi faire ?

- compresser l'information
- réduire le bruit dans les données
- accélérer la convergence du modèle
- éliminer des paramètres inintéressants
- visualiser les données (2D / 3D)





Approches principales

- projections

<https://scikit-learn.org/stable/modules/decomposition.html>

- apprentissage de *manifold* (*manifold learning*)

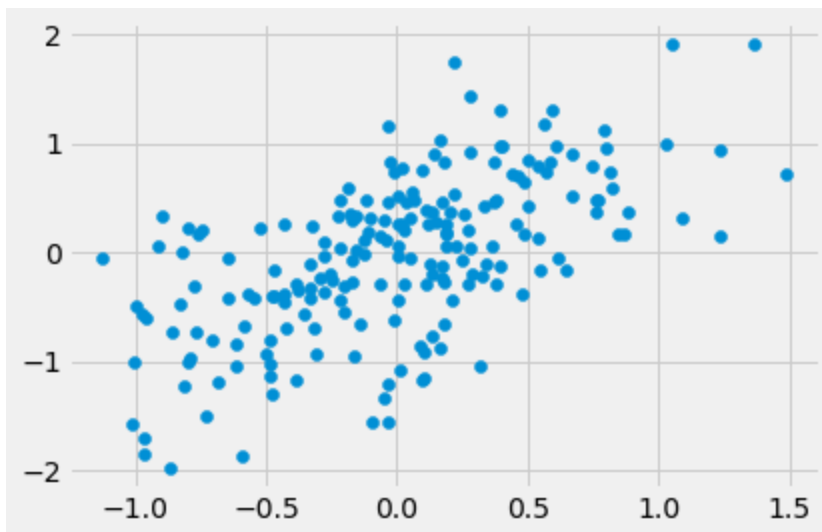
<https://scikit-learn.org/stable/modules/manifold.html>





Création du jeu de données

```
In [2]: X = (np.random.rand(2, 2) @ np.random.randn(2, 200)).T  
  
plt.scatter(X[:, 0], X[:, 1]);
```





Principal Component Analysis or PCA





L'analyse en composantes principales se base sur la recherche des directions dans l'espace à N dimensions dont la variance est maximale.

Une fois ces directions trouvées, on peut classer les dimensions suivant l'importance relative de leur variance.

Réduire la dimension du problème revient dès lors à supprimer les dimensions de variance minimale pour ne conserver que celles sur lesquelles les données sont éparses.





```
In [3]: from sklearn.decomposition import PCA
```

```
pca = PCA(n_components=2)
```

```
pca.fit(X)
```

```
Out[3]: PCA(n_components=2)
```





In [4]: `print(pca.components_)`

```
[[ 0.50322223  0.86415704]
 [ 0.86415704 -0.50322223]]
```





```
In [5]: print(pca.explained_variance_)
```

```
[0.66846779 0.13530378]
```





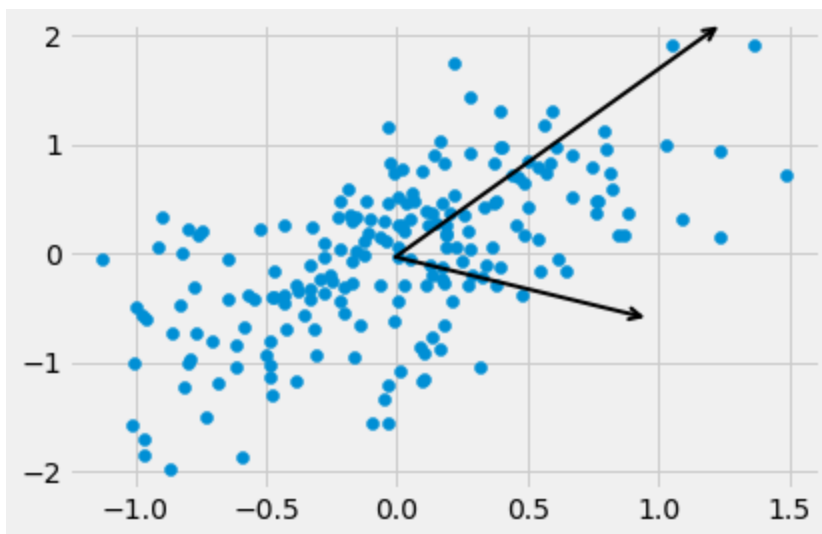
Visualisons ces axes qui maximisent la variance

```
In [6]: def draw_vector(v0, v1, ax=None):  
        ax = ax if ax is not None else plt.gca()  
        arrowprops = dict(arrowstyle='->', color='k',  
                           lw=2, shrinkA=0, shrinkB=0)  
        ax.annotate('', v1, v0, arrowprops=arrowprops)
```





```
In [7]: plt.scatter(X[:, 0], X[:, 1])  
for length, vector in zip(pca.explained_variance_, pca.components_):  
    v = vector * 3 * np.sqrt(length)  
    draw_vector(pca.mean_, pca.mean_ + v)
```



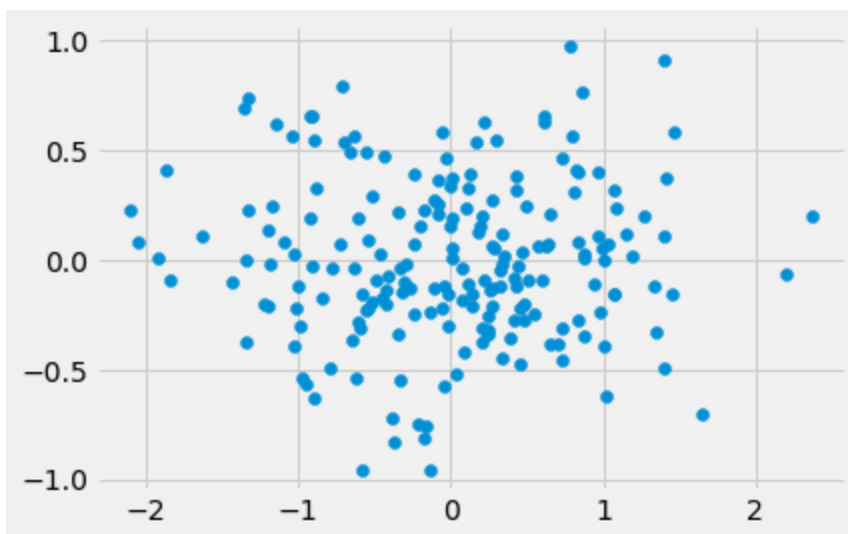


Transformons les données pour les exprimer suivant les nouveaux axes





```
In [8]: X_trans = pca.fit_transform(X)  
plt.scatter(X_trans[:, 0], X_trans[:, 1]);
```





Si maintenant on cherche à réduire la dimension, on projette suivant l'axe de plus faible variance



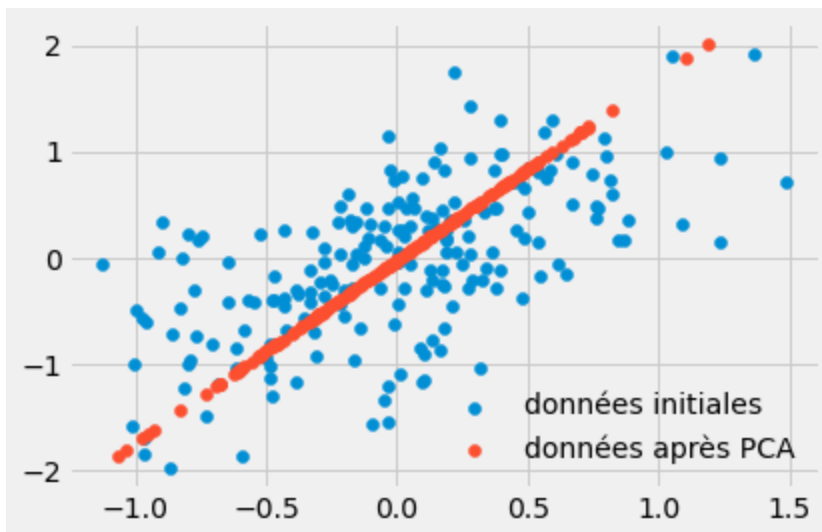


```
In [9]: # On choisit de ne conserver qu'1 dimension  
pca = PCA(n_components=1)  
# On calcule la transformation  
X_trans = pca.fit_transform(X)  
# Puis on reprojette dans l'espace initial pour visualiser  
X2 = pca.inverse_transform(X_trans)
```





```
In [10]: plt.scatter(X[:, 0], X[:, 1], label='données initiales')  
plt.scatter(X2[:, 0], X2[:, 1], label='données après PCA')  
plt.legend(frameon=False, loc=4);
```





Passons à des données plus complexes





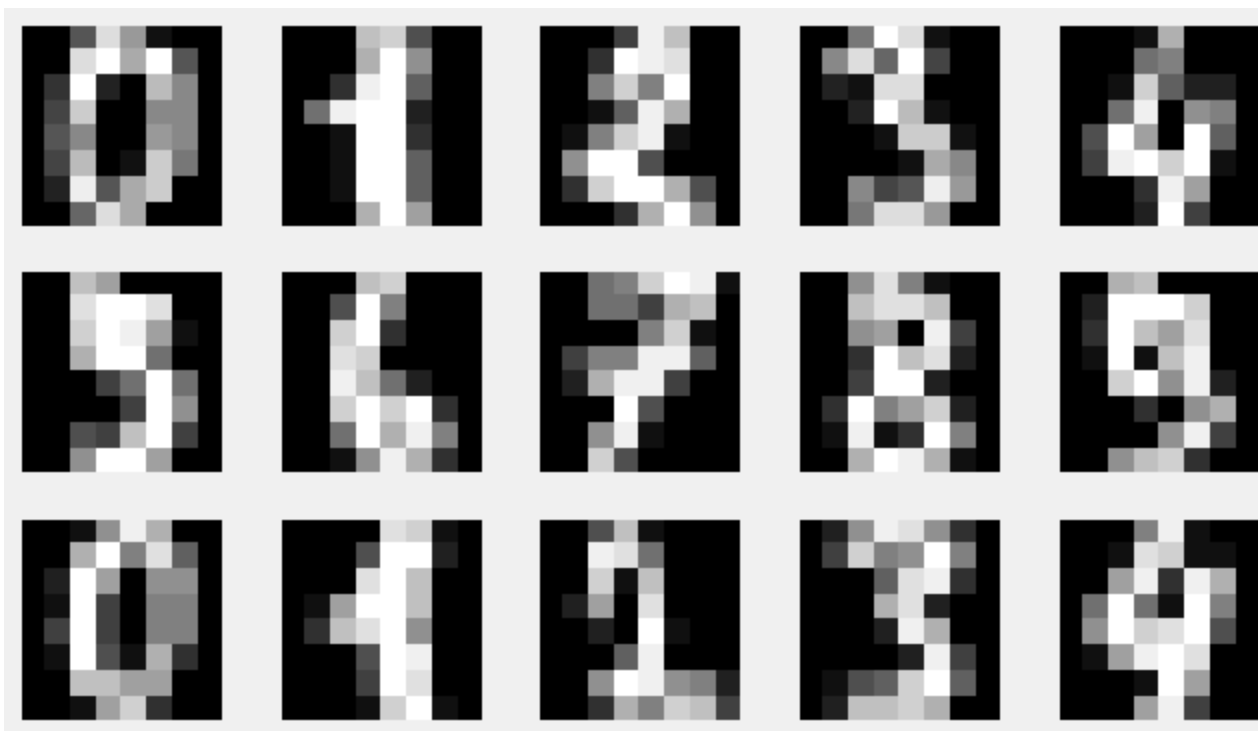
```
In [11]: from sklearn.datasets import load_digits  
digits = load_digits()
```





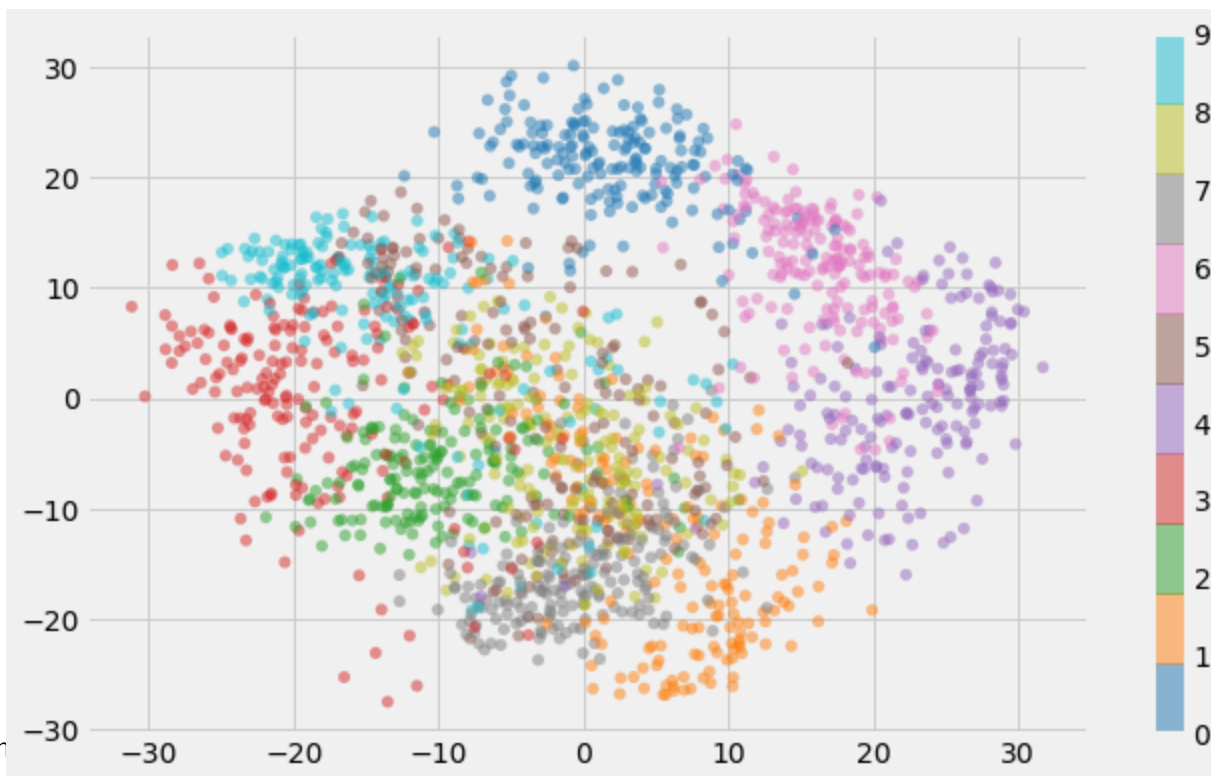
```
In [12]: fig, axes = plt.subplots(3, 5, figsize=(10, 6),
        subplot_kw={'xticks': [],
        'yticks': []})

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary_r')
plt.grid(False)
```





```
In [13]: plt.figure(figsize=(10,6))
projected_data = PCA(2).fit_transform(digits.data)
plt.scatter(projected_data[:, 0],
            projected_data[:, 1],
            c=digits.target,
            edgecolor='none',
            alpha=0.5,
            cmap=plt.cm.get_cmap('tab10'))
plt.colorbar();
```



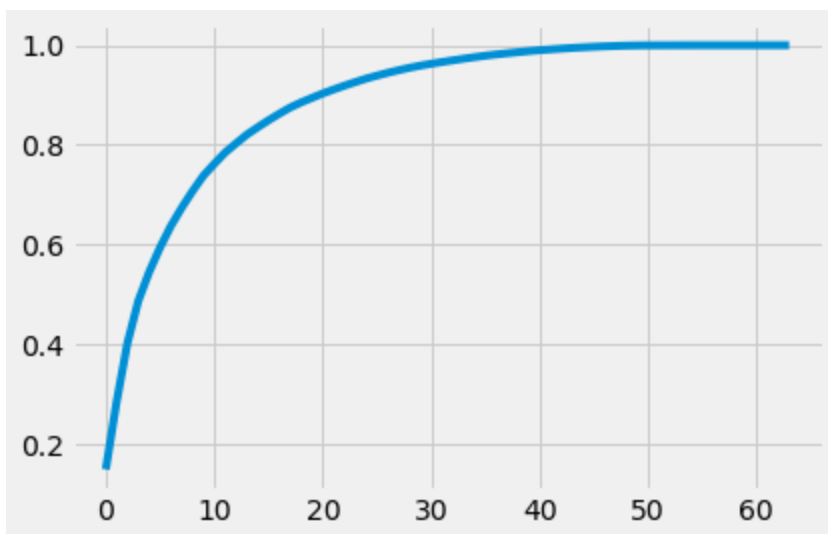


Comment savoir combien de composantes garder ?





```
In [14]: pca = PCA().fit(digits.data)  
plt.plot(np.cumsum(pca.explained_variance_ratio_));
```





Réduction du bruit

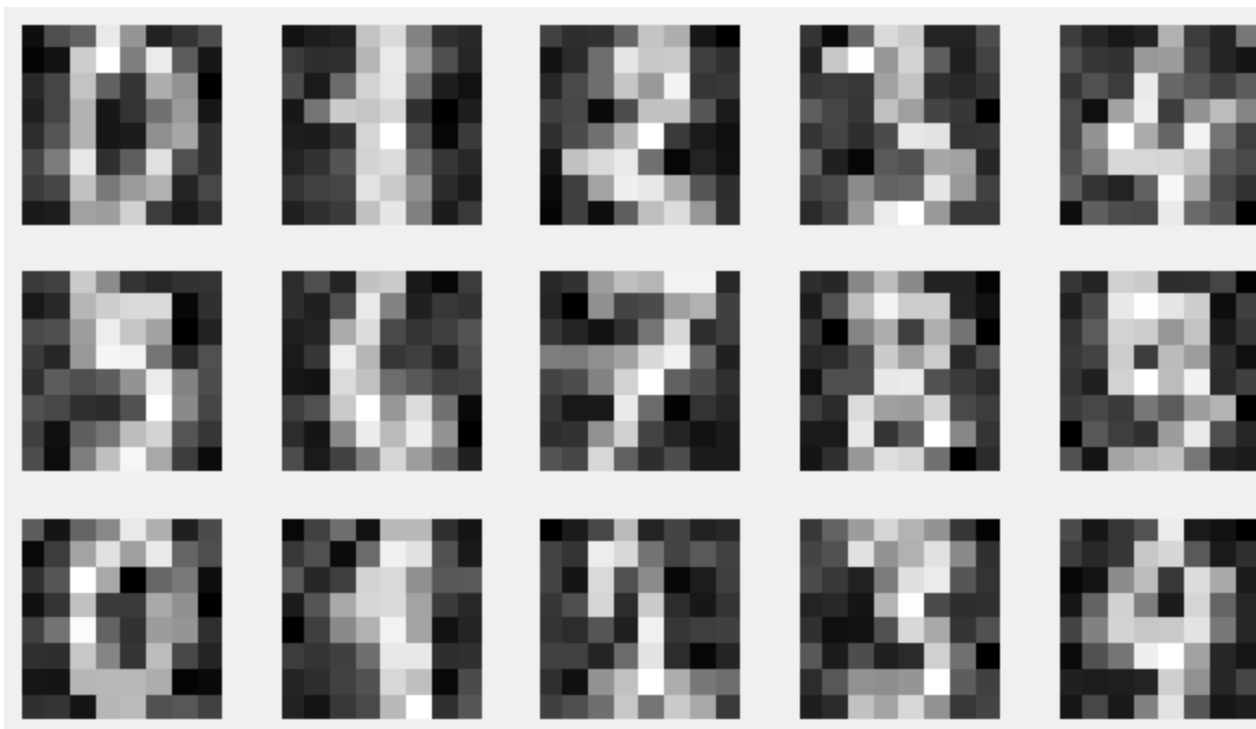




```
In [15]: noisy = np.random.normal(digits.data, 2)

fig, axes = plt.subplots(3, 5, figsize=(10, 6),
                        subplot_kw={'xticks': [],
                                    'yticks': []})

for i, ax in enumerate(axes.flat):
    ax.imshow(noisy[i].reshape(8, 8),
              cmap='binary_r')
plt.grid(False)
```





```
In [16]: pca = PCA(.65).fit(noisy)
          components = pca.transform(noisy)
          components.shape
```

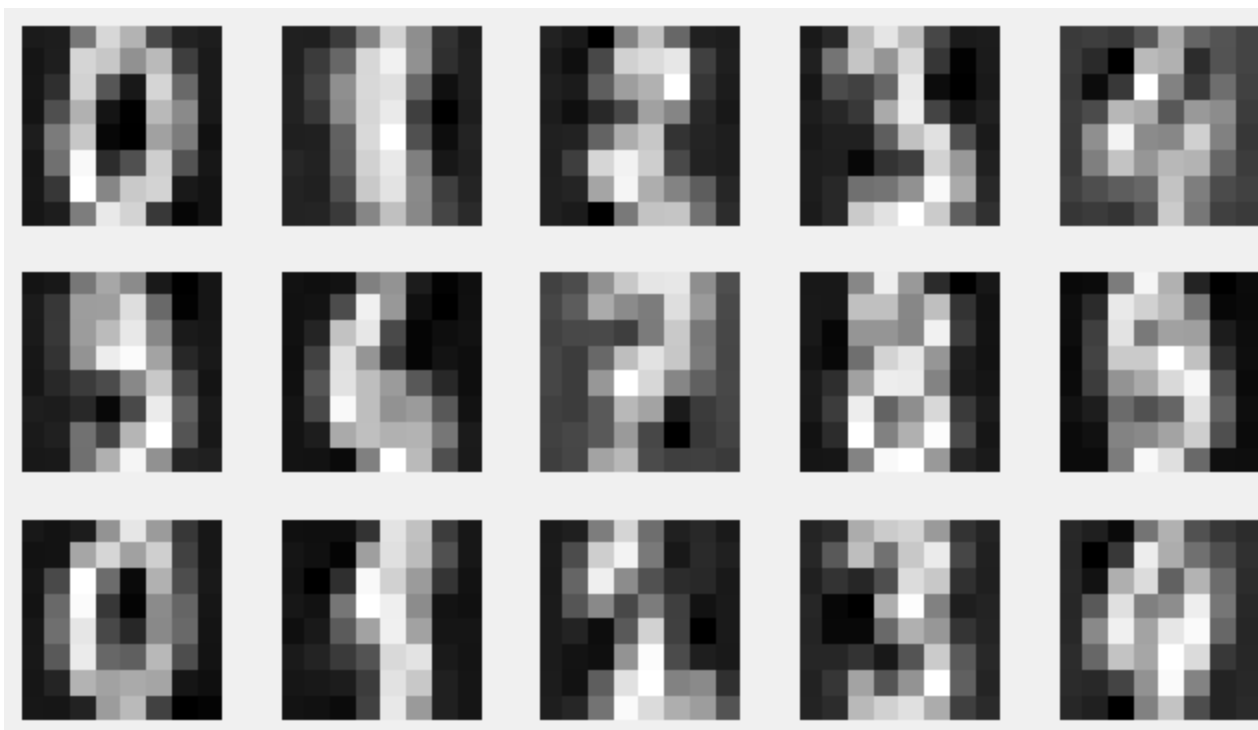
```
Out[16]: (1797, 11)
```





```
In [17]: filtered = pca.inverse_transform(components)

fig, axes = plt.subplots(3, 5, figsize=(10, 6),
                        subplot_kw={'xticks': [], 'yticks': []})
for i, ax in enumerate(axes.flat):
    ax.imshow(filtered[i].reshape(8, 8),
              cmap='binary_r')
plt.grid(False)
```





Avantages

- très rapide
- permet de visualiser les données facilement

Inconvénients

- très sensible aux valeurs extrêmes
- moins efficace en présence de données sparses (beaucoup de zéros)





Autres algorithmes à voir sur <https://scikit-learn.org/stable/modules/decomposition.html>





Manifold Learning



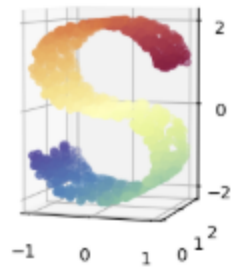


L'idée derrière l'apprentissage de manifold est de trouver un espace de faible dimension (généralement 2D) dans lequel on peut réexprimer des données qui ne sont pas linéairement séparables dans leur espace à N -dim, $N > 2$.

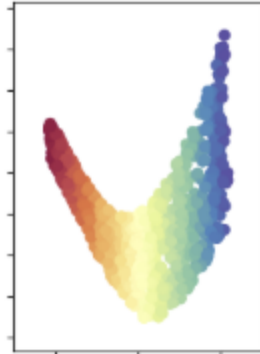




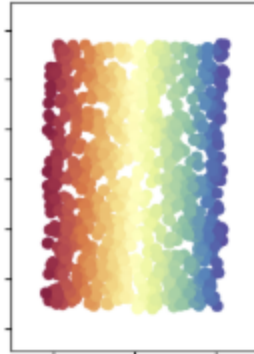
Manifold Learning with 1000 points, 10 neighbors



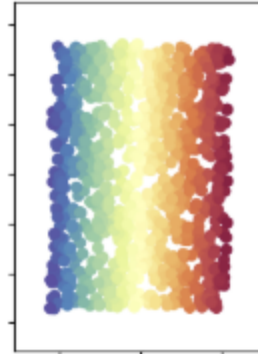
LLE (0.11 sec)



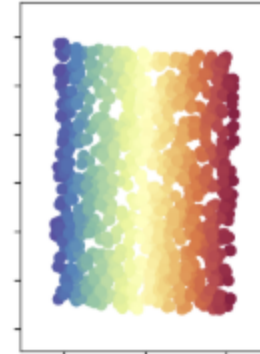
LTSA (0.18 sec)



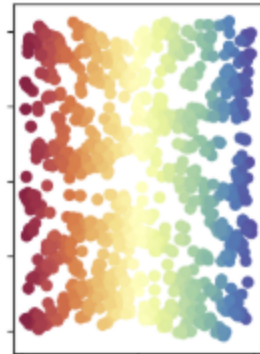
Hessian LLE (0.28 sec)



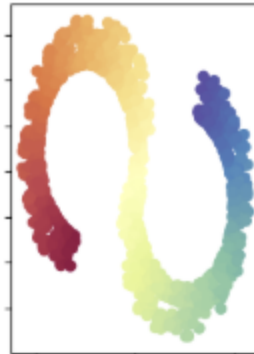
Modified LLE (0.22 sec)



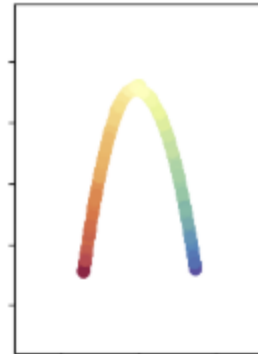
Isomap (0.39 sec)



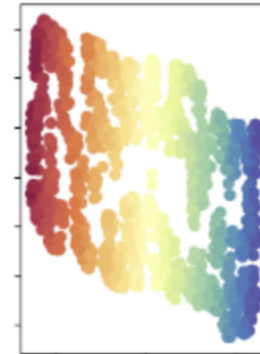
MDS (1.2 sec)



SE (0.072 sec)



t-SNE (7 sec)





Example with t-SNE

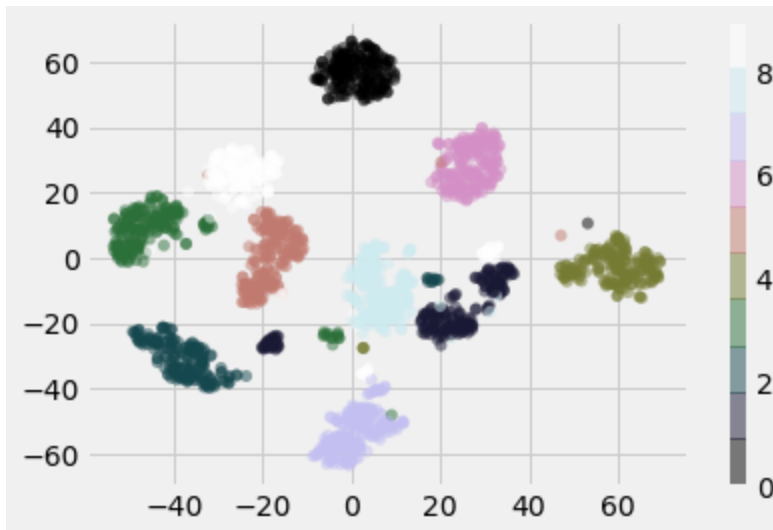




```
In [18]: from sklearn.manifold import TSNE

projected_data = TSNE(n_components=2,
                      init='pca').fit_transform(digits.data)
plt.scatter(projected_data[:, 0],
            projected_data[:, 1],
            c=digits.target,
            edgecolor='none',
            alpha=0.5,
            cmap=plt.cm.get_cmap('cubehelix', 10))
plt.colorbar()
```

Out[18]: <matplotlib.colorbar.Colorbar at 0x139cd7100>





Comprendre et utiliser efficacement T-SNE

<https://distill.pub/2016/misread-tsne/>





Autres approches

- plusieurs algorithmes sont présents dans scikit-learn
<https://scikit-learn.org/stable/modules/manifold.html>
- un algorithme récent (pas encore dans `sklearn`) fait beaucoup parler de lui: **UMAP**

<https://umap-learn.readthedocs.io/en/latest/index.html>





Clustering





- K-means, DBSCAN

<https://scikit-learn.org/stable/modules/clustering.html>

- composition de gaussiennes (*Gaussian mixture*)

<https://scikit-learn.org/stable/modules/mixture.html>





Jeu de données

```
In [19]: import sklearn.datasets as sds

X, labels = sds.make_blobs(n_samples=200,
                           centers=3,
                           cluster_std=0.9)
```



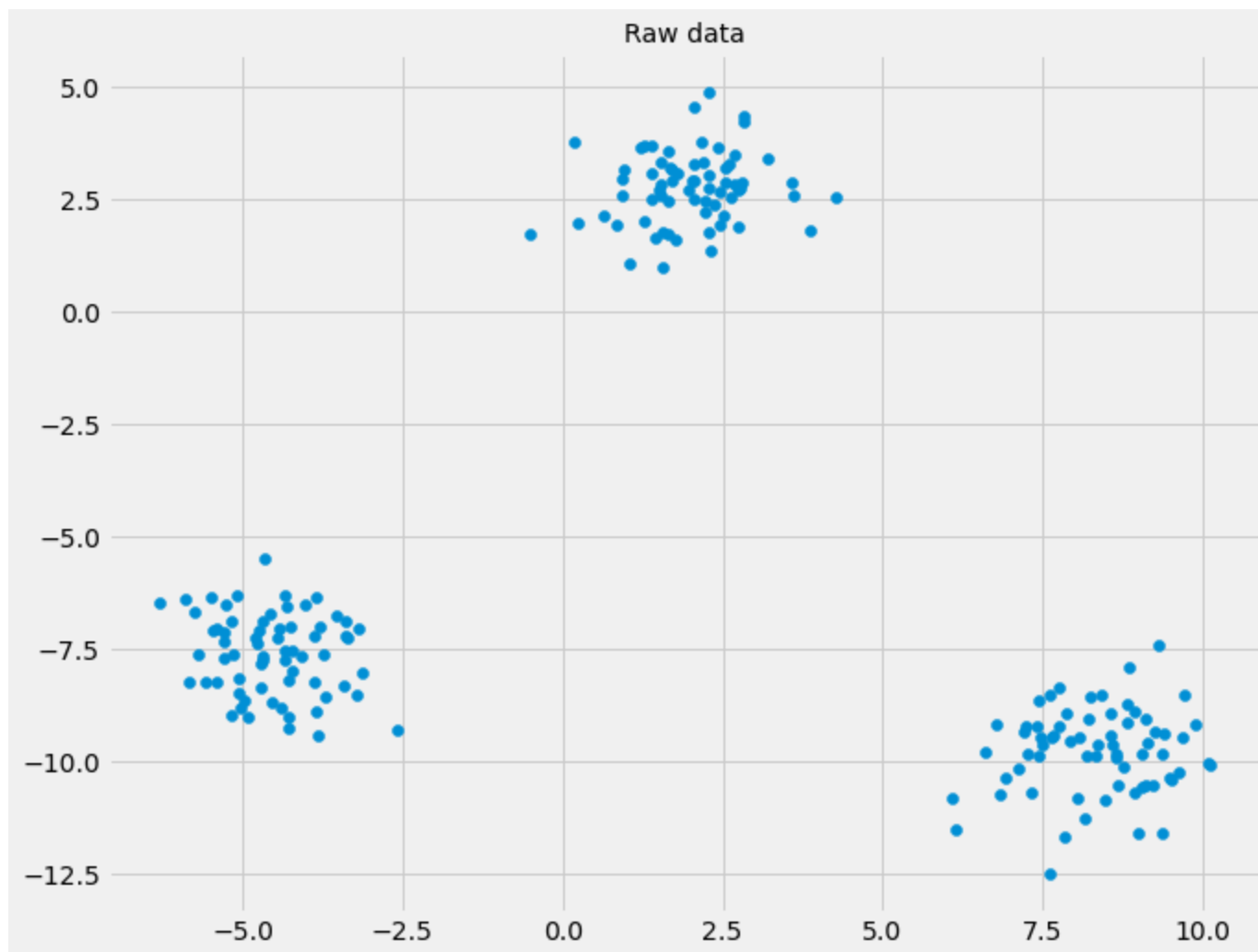


```
In [20]: def plot_blobs(X, y=None, n=3):  
    fig, ax = plt.subplots(figsize=(10, 8))  
    if y is None:  
        ax.scatter(X[:, 0], X[:, 1])  
        ax.set_title("Raw data", fontsize=14)  
    else:  
        im = ax.scatter(X[:, 0], X[:, 1],  
                        c=y,  
                        cmap=plt.cm.get_cmap('Paired'))  
        plt.colorbar(im, ax=ax, values=range(n))  
        ax.set_title("Labeled data", fontsize=14)
```



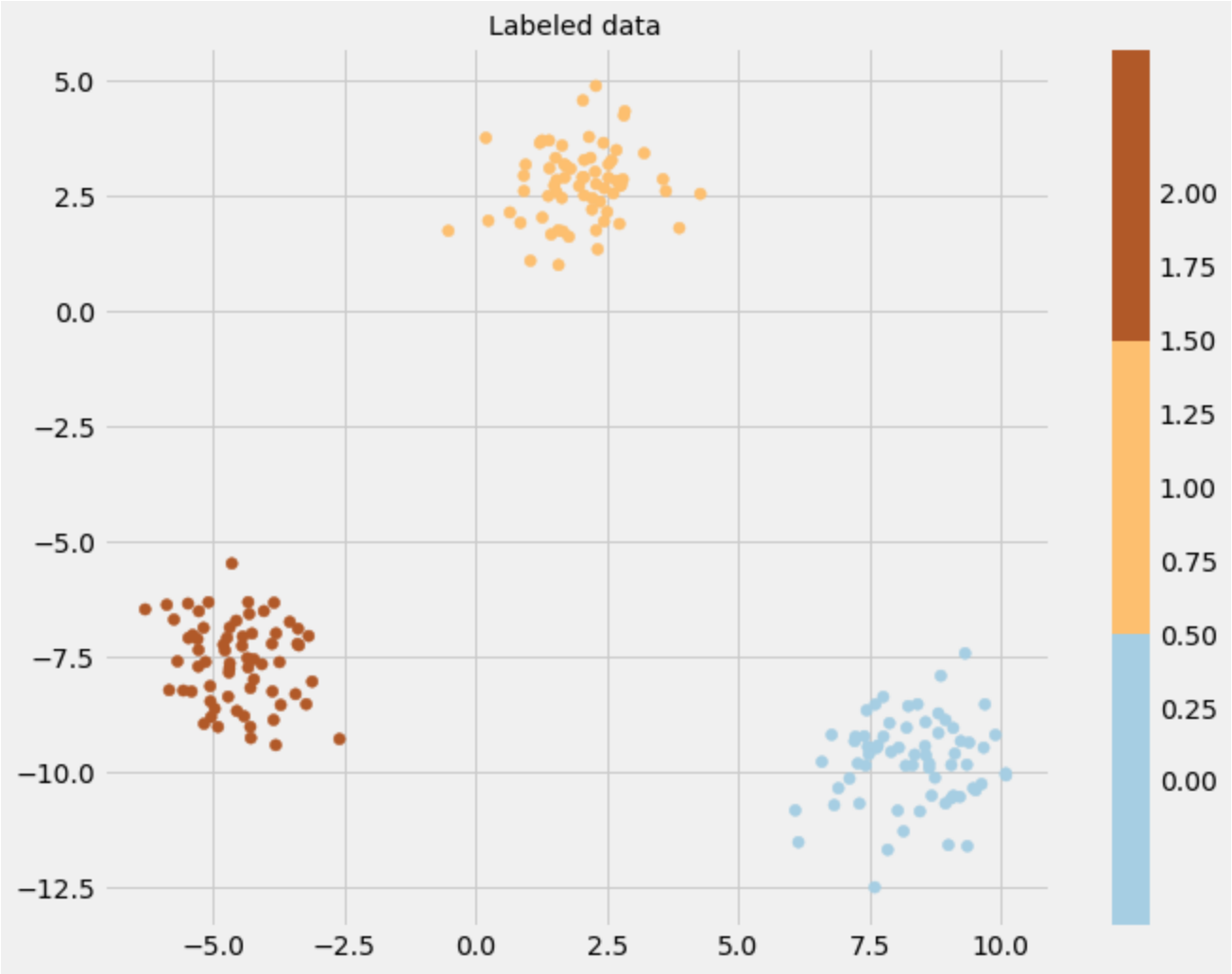


In [21]: `plot_blobs(X)`





```
In [22]: plot_blobs(X, labels, 3)
```





K-Means

<https://www.naftaliharris.com/blog/visualizing-k-means-clustering/>



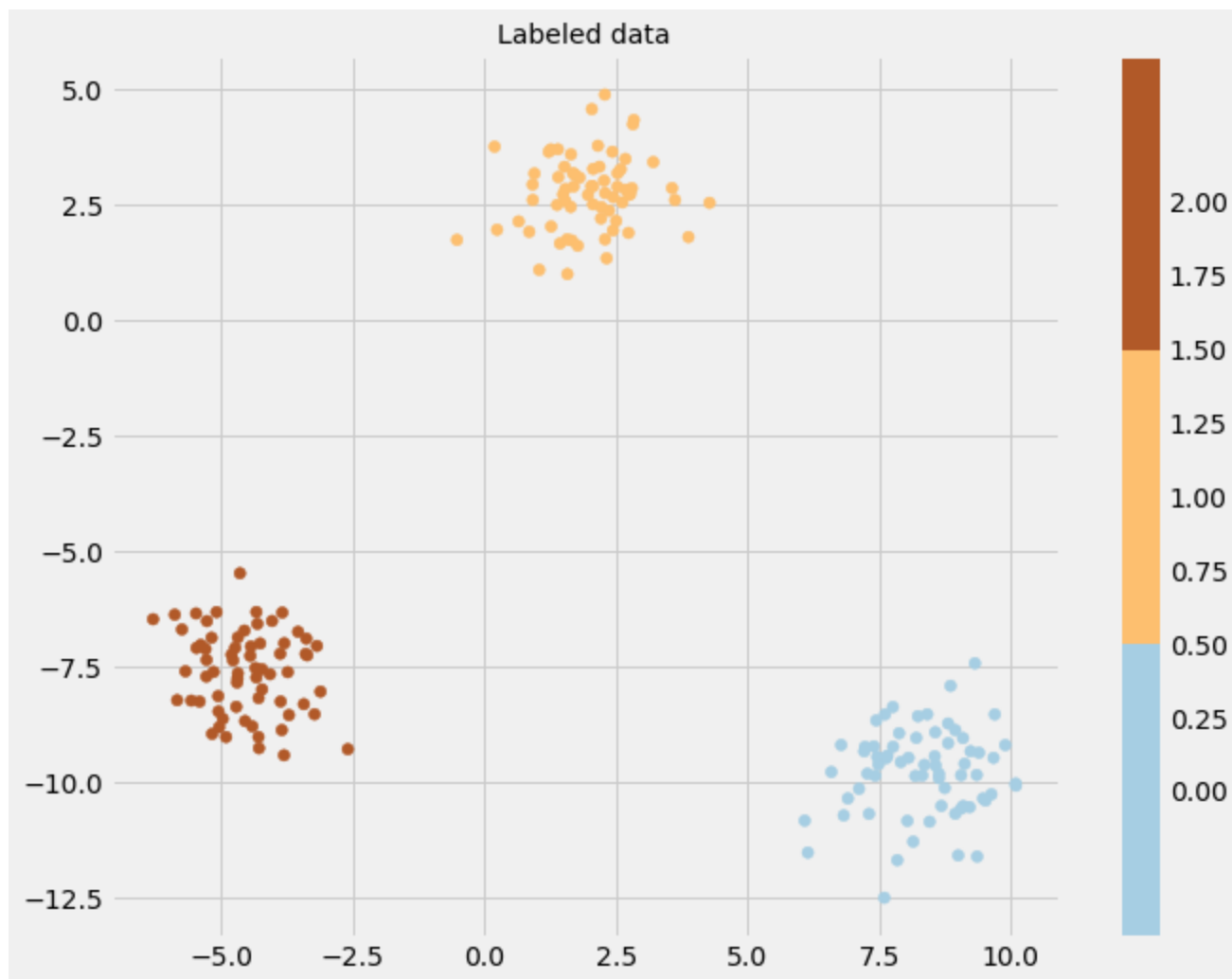


```
In [23]: from sklearn.cluster import KMeans  
  
kmeans = KMeans(n_clusters=3).fit(X)  
y_kmeans = kmeans.predict(X)
```



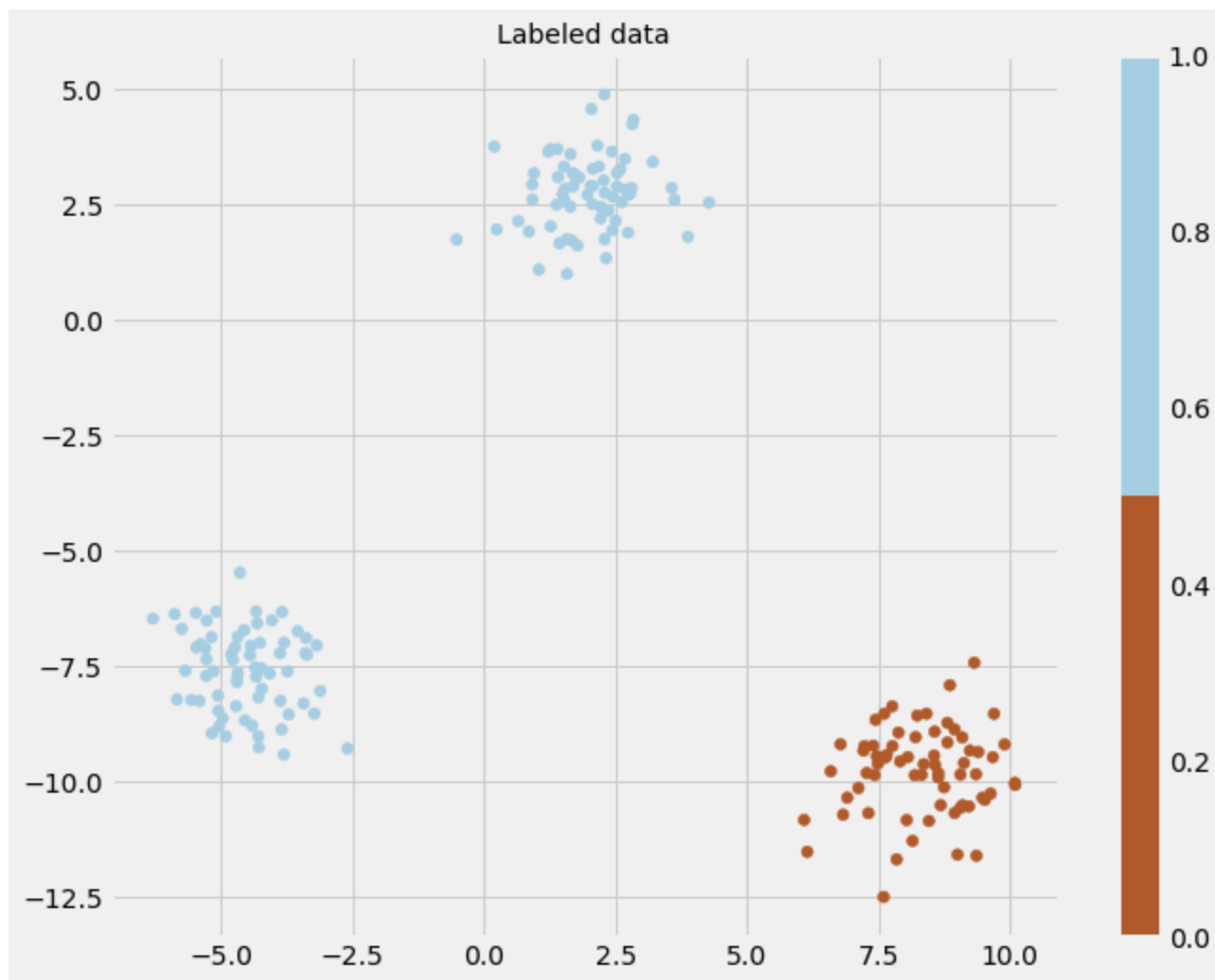


In [24]: `plot_blobs(X, y_kmeans, 3)`





```
In [25]: kmeans = KMeans(n_clusters=2).fit(X)  
y_kmeans = kmeans.predict(X)  
plot_blobs(X, y_kmeans, 2)
```





DBSCAN

<https://www.naftaliharris.com/blog/visualizing-dbscan-clustering/>





```
In [26]: from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=.8, min_samples=6)
y_dbscan = dbscan.fit_predict(X)

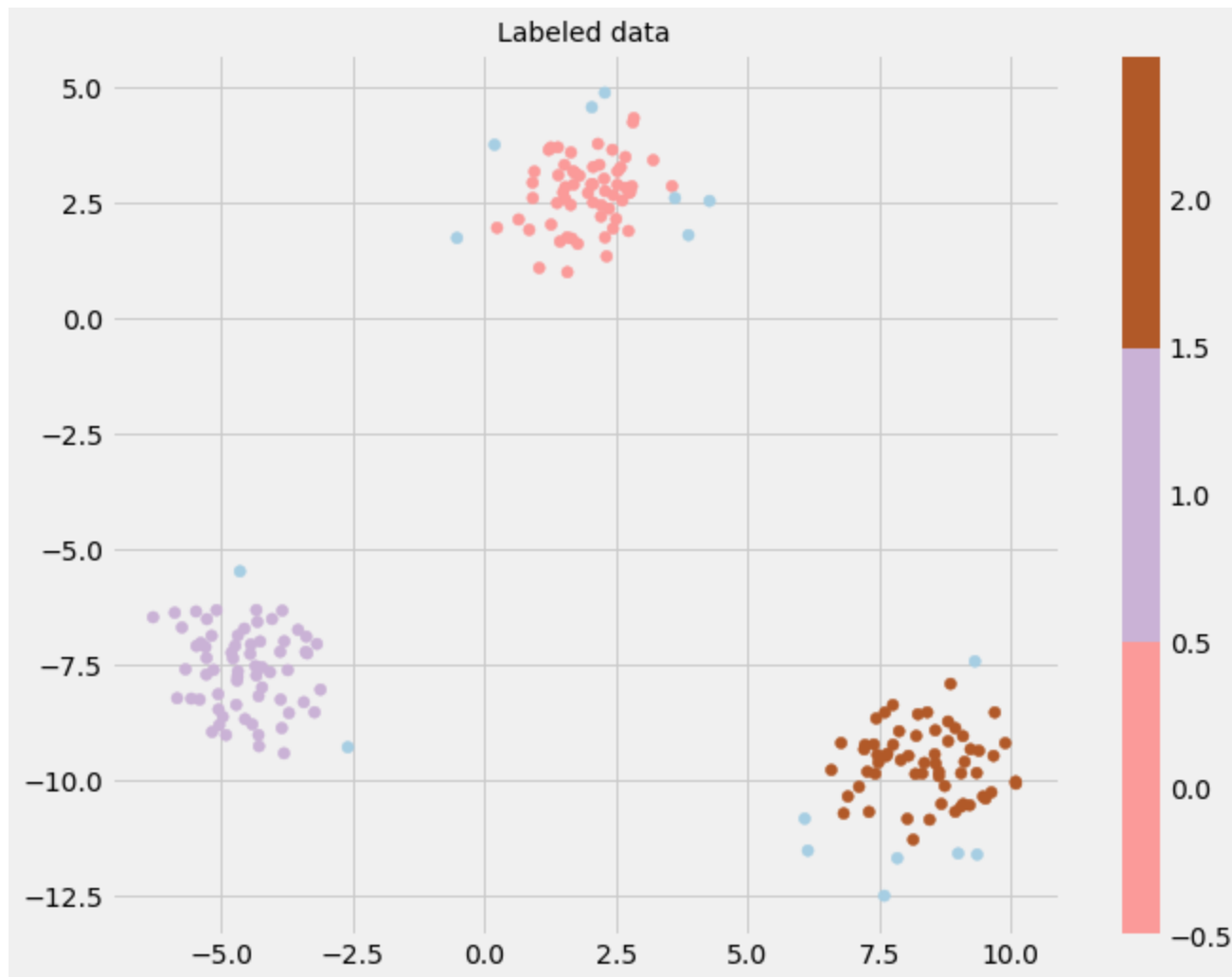
print(f"n_clusters = {np.unique(labels).size}")

n_clusters = 3
```





In [27]: `plot_blobs(X, y_dbscan, np.unique(labels).size)`





DBSCAN a l'avantage de ne pas demander d'avance le nombre de clusters, mais nécessite de trouver les valeurs adéquates pour epsilon et n_{\min} .





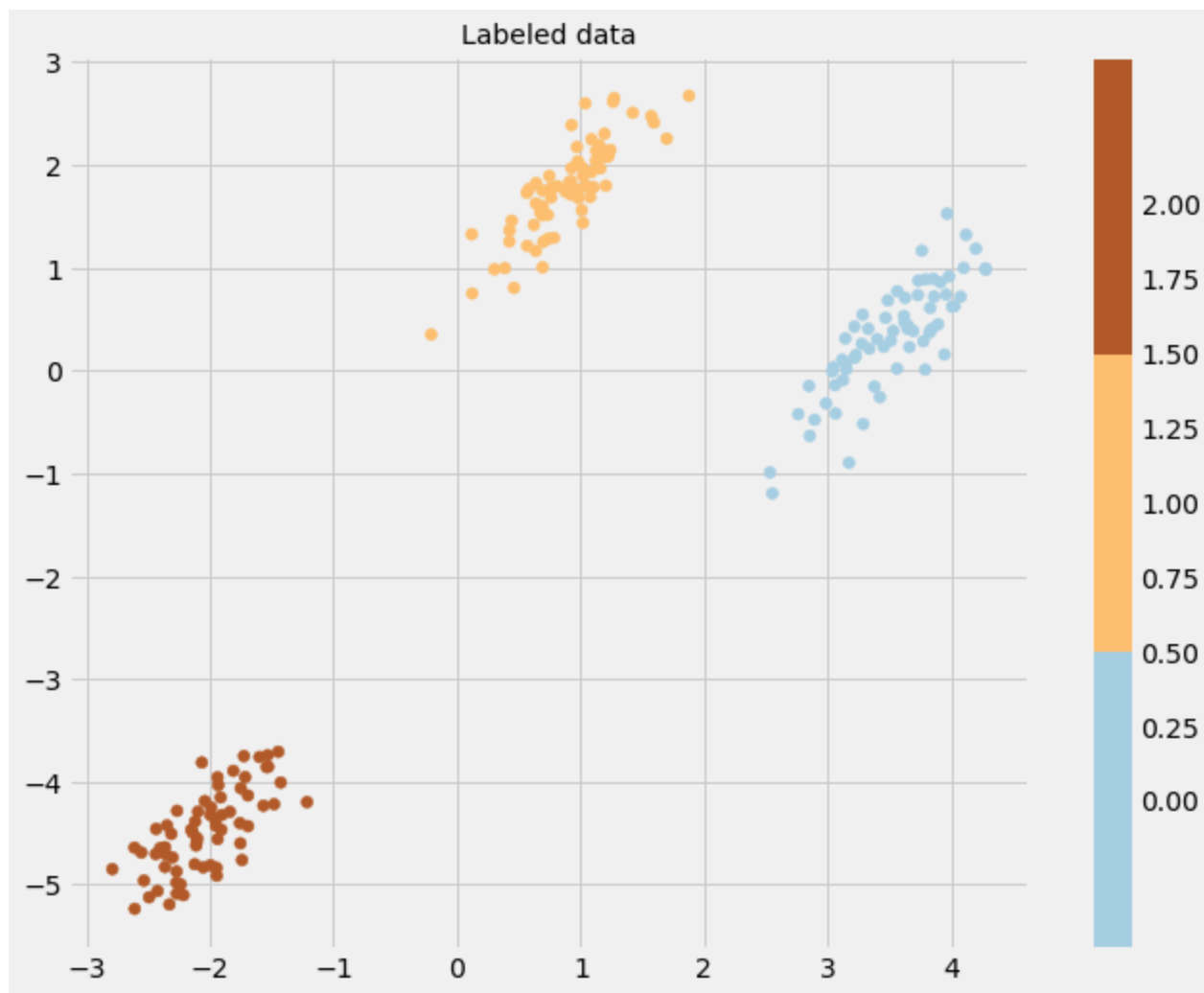
Ajoutons de la corrélation à nos données





```
In [28]: X2 = X @ np.random.rand(2, 2)
```

```
plot_blobs(X2, labels)
```



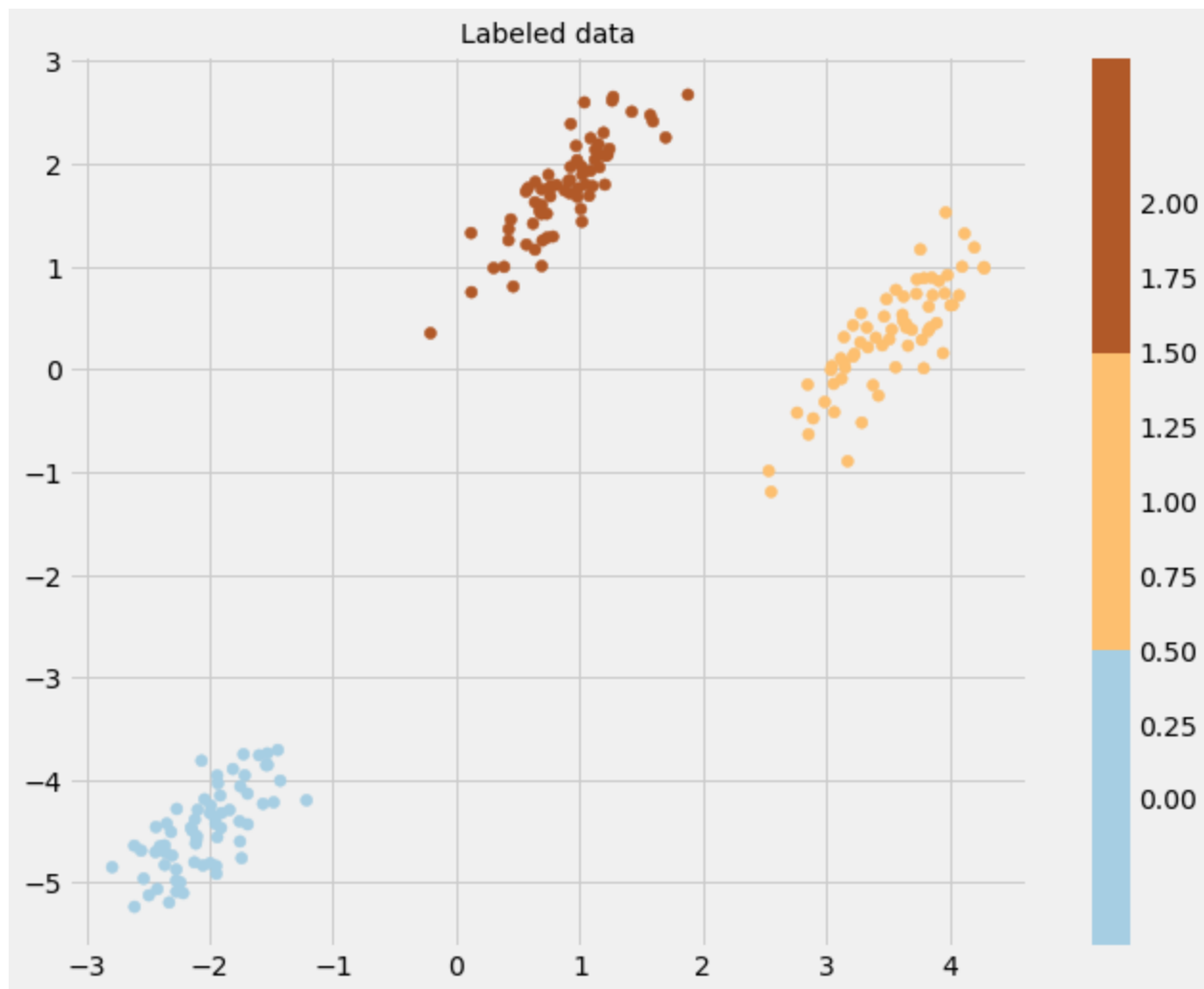


Comment se comporte un algo comme K-Means sur données très corrélées ?





```
In [29]: y2_kmeans = KMeans(n_clusters=3).fit_predict(X2)  
plot_blobs(X2, y2_kmeans)
```





Comme on aurait pu l'imaginer, la distance euclidienne sur des données très corrélées n'est pas forcément un bon estimateur de groupe. Voyons si d'autres algorithmes ne seraient pas plus appropriés.





Gaussian mixture models (GMM)





Un *Gaussian mixture model* tente de représenter la distribution des données par un ensemble de gaussiennes à N dimensions. Il devrait naturellement être plus flexible pour représenter des données très corrélées.





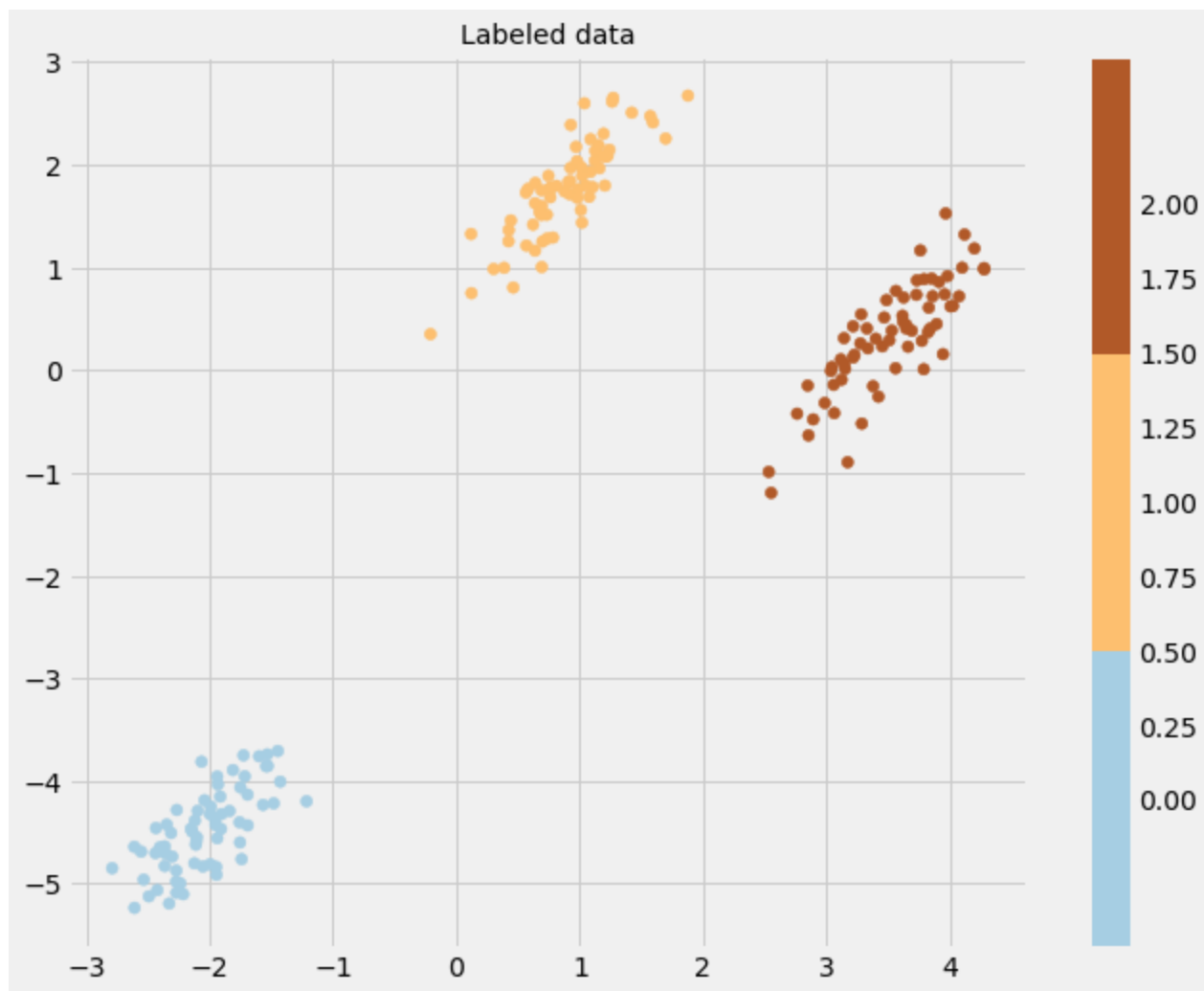
```
In [30]: from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=3).fit(X2)
y2_gmm = gmm.predict(X2)
```





In [31]: `plot_blobs(X2, y2_gmm)`





On retrouve le même résultat qu'avec le K-Means sur nos données très corrélées. Notre cas est assez particulier car la corrélation a été ajoutée artificiellement avec une matrice de covariance et appliquée à l'ensemble des groupes. Si on le spécifie à l'algorithme, on lui permet de s'adapter.





```
In [32]: gmm = GaussianMixture(n_components=3,  
                                covariance_type='tied').fit(X2)  
y2_gmm = gmm.predict(X2)
```





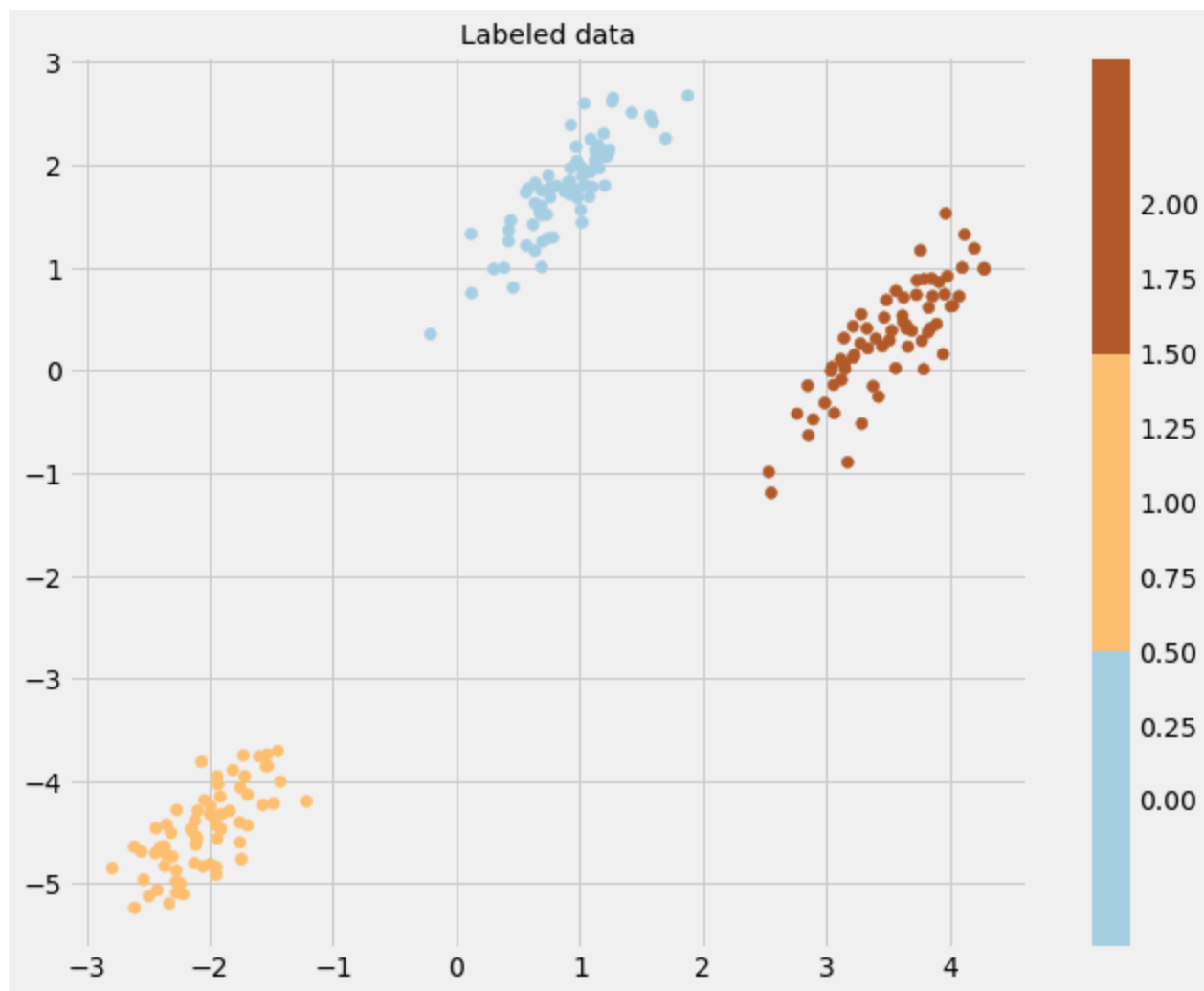
```
In [33]: gmm.predict_proba(X2)[:5].round(3)
```

```
Out[33]: array([[1., 0., 0.],  
                [0., 1., 0.],  
                [0., 0., 1.],  
                [1., 0., 0.],  
                [0., 1., 0.]])
```





In [34]: `plot_blobs(X2, y2_gmm)`





Conclusion





Les algorithmes d'apprentissage non-supervisé sont très faciles d'utilisation et permettent d'**explorer** la distribution de nos données ainsi que les groupes et corrélations sous-jacentes. Comme nous le verrons par la suite, ils sont très utilisés pour **préparer les données aux tâches d'apprentissage supervisé**.

