

# Ingénierie des données

Dans ce notebook nous présentons l'une des étapes essentielles de la classification : l'ingénierie des données. Nous abordons le traitement des données catégorielles, la mise à l'échelle et le traitement des valeurs manquantes. Il existe de très nombreuses méthodes qui ont forcément un impact sur le résultat de la classification. Il est important de bien les comprendre et de rechercher celle qui est la plus adaptée en fonction du contexte.

## Traitement des données catégorielles ou qualitatives

De très nombreux jeux de données contiennent des données catégorielles comme une couleur, une adresse, etc. Même les classes peuvent être catégorielles (avis positif, avis négatif). De nombreux algorithmes ne sont pas capables de les traiter car ils considèrent uniquement des valeurs numériques. Dans cette section nous présentons différentes manières de transformer les données catégorielles. Elles dépendent bien entendu du contexte.

**Rappel** : les attributs pour lesquels il n'existe pas d'ordre sont appelés nominaux (par exemple les couleurs). En opposition ceux pour lesquels il existe un ordre sont appelés ordinaux (taille XL, L, M).

Considérons l'exemple suivant qui contient des attributs numériques et catégoriels.

In [1]:

```
1
2 import pandas as pd
3
4 df = pd.DataFrame(
5     {'Taille': ['XL', 'L', 'M', 'S'],
6      'Couleur': ['bleu', 'blanc', 'rouge', 'vert'],
7      'Prix': [20.76, 23.5, 40.99, 10.0],
8      'Classe': ['classe1', 'classe1', 'classe2', 'classe3']},
9     columns=['Taille', 'Couleur', 'Prix', 'Classe'])
10
11 print ('Pour connaître les informations qui sont catégorielles, faire un df.
12 print (df.info())
13
14
```

Pour connaître les informations qui sont catégorielles, faire un df.  
info()

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 4 entries, 0 to 3
```

```
Data columns (total 4 columns):
```

```
Taille      4 non-null object
```

```
Couleur     4 non-null object
```

```
Prix        4 non-null float64
```

```
Classe      4 non-null object
```

```
dtypes: float64(1), object(3)
```

```
memory usage: 208.0+ bytes
```

```
None
```

Dans un jeu de données pour connaître les attributs qui ne sont pas numériques, il suffit de faire, pour un dataframe de nom df, un df.info() et les attributs non numériques apparaissent avec le type object.

Attention toutefois, lorsqu'il y a la présence de valeurs manquantes il se peut que l'attribut apparaisse avec le dtype objet. Par exemple en remplaçant le dataframe précédent par :

```
'Prix': [20.76,'nan',40.99,10.0]
```

l'information sur l'attribut prix sera : Prix 4 non-null object

Il est également possible de faire un df.describe() qui affiche des statistiques (moyenne, max, min, etc) uniquement pour les attributs numériques.

In [2]:

```
1 print ('Pour connaître quelques statistiques, faire un df.describe()')
2
3
4 display(df.describe())
```

Pour connaître quelques statistiques, faire un df.describe()

Prix	
count	4.000000
mean	23.812500
std	12.848697
min	10.000000
25%	18.070000
50%	22.130000
75%	27.872500
max	40.990000

Un df.describe() sur un attribut non numérique donne d'autres informations.

In [3]:

```
1 display(df['Couleur'].describe())
```

```
count      4
unique      4
top        bleu
freq        1
Name: Couleur, dtype: object
```

## Remplacement de la valeur

La première approche la plus simple est de remplacer les valeurs. Considérons qu'il y ait un ordre pour les tailles tels que S=1, M=S+1 etc. Il est possible de transformer les valeurs à l'aide de la fonction map appliquée au dataframe

In [4]:

```
1 #creation de la transformation
2 replace_map = {'XL': 4, 'L': 3, 'M': 2, 'S': 1}
```

In [5]:

```
1  #creation d'une copie de df pour ne pas perdre le df initial
2  df_test=df.copy()
3
4  print ( 'Application de la fonction map \n')
5  print (df_test['Taille'].map(replace_map))
6
7  print ("\nAjout d'une colonne au dataframe avec application de la fonction
8  df_test["Taille renommée"]=df_test['Taille'].map(replace_map)
9
10 display(df_test)
11
12 print ("\nRemplacement direct de taille avec les nouvelles valeurs\n")
13
14 df_test.replace(replace_map, inplace=True)
15
16 display(df_test)
```

Application de la fonction map

```
0    4
1    3
2    2
3    1
Name: Taille, dtype: int64
```

Ajout d'une colonne au dataframe avec application de la fonction map

	Taille	Couleur	Prix	Classe	Taille renommée
0	XL	bleu	20.76	classe1	4
1	L	blanc	23.50	classe1	3
2	M	rouge	40.99	classe2	2
3	S	vert	10.00	classe3	1

Remplacement direct de taille avec les nouvelles valeurs

	Taille	Couleur	Prix	Classe	Taille renommée
0	4	bleu	20.76	classe1	4
1	3	blanc	23.50	classe1	3
2	2	rouge	40.99	classe2	2
3	1	vert	10.00	classe3	1

# Label encoding

Une autre approche appelée label encoding consiste à transformer l'ensemble des valeurs par un nombre de 0 au nombre-1 de catégories. Il suffit pour cela d'utiliser la fonction *LabelEncoder*.

```
In [6]:  
  
1  from sklearn.preprocessing import LabelEncoder  
2  
3  #creation d'une copie de df pour ne pas perdre le df initial  
4  df_test=df.copy()  
5  
6  class_label_encoder = LabelEncoder()  
7  print ("\n Affichage des transformations\n")  
8  print(class_label_encoder.fit_transform(df_test['Classe'].values))  
9  
10 print ("\nAjout d'une colonne au dataframe")  
11 df_test["Classe renommée"]=class_label_encoder.fit_transform(df_test["Classe"])  
12  
13 display (df_test)  
14  
15 print ("\nRemplacement direct de classe avec les nouvelles valeurs\n")  
16 df_test["Classe"] = class_label_encoder.fit_transform(df_test["Classe"])  
17  
18 display(df_test)  
19
```

Affichage des transformations

[0 0 1 2]

Ajout d'une colonne au dataframe

	Taille	Couleur	Prix	Classe	Classe renommée
0	XL	bleu	20.76	classe1	0
1	L	blanc	23.50	classe1	0
2	M	rouge	40.99	classe2	1
3	S	vert	10.00	classe3	2

Remplacement direct de classe avec les nouvelles valeurs

	Taille	Couleur	Prix	Classe	Classe renommée
0	XL	bleu	20.76	0	0
1	L	blanc	23.50	0	0
2	M	rouge	40.99	1	1
3	S	vert	10.00	2	2

# One Hot Encoding

Le label encoding a l'avantage d'être direct à obtenir mais le désavantage de donner des valeurs croissantes qui peuvent être mal considérés par les classifieurs. Par exemple dans le cas des tailles il est plus logique que XL qui est le plus grand ait une grande valeur. Par contre pour les couleurs cela ne correspond pas à une réalité. On ne peut pas dire que bleu a 4 fois plus de poids que vert par exemple.

Le principe est de convertir chaque valeur de catégorie dans une nouvelle colonne et de mettre une valeur 1 ou 0 (vrai/faux) à la colonne. Cette approche a l'avantage de ne plus mettre de poids différents pour un attribut mais à l'inconvénient de rajouter un grand nombre de colonnes si le nombre de catégories est important.

Scikit learn propose la fonction `get_dummies` pour effectuer la transformation. Cette dernière prend en paramètre le dataframe, les colonnes sur lesquelles doivent s'effectuer les transformation et un prefixe qui sera utilisé pour nommer les colonnes.

```
pd.get_dummies(df,columns=['colA','colB',..],prefix = ['leprefix'])
```

In [7]:

```
1
2  #creation d'une copie de df pour ne pas perdre le df initial
3  df_test=df.copy()
4
5  print ("\n Affichage des transformations\n")
6  print (pd.get_dummies(df_test[['Couleur']]))
7
8  print ("\nAjout des colonne au dataframe")
9  df_test = pd.get_dummies(df_test,columns=['Couleur'],
10                          prefix = ['Coul'])
11
12  display (df_test)
13
14
```

Affichage des transformations

	Couleur_blanc	Couleur_bleu	Couleur_rouge	Couleur_vert
0	0	1	0	0
1	1	0	0	0
2	0	0	1	0
3	0	0	0	1

Ajout des colonne au dataframe

	Taille	Prix	Classe	Coul_blanc	Coul_bleu	Coul_rouge	Coul_vert
0	XL	20.76	classe1	0	1	0	0
1	L	23.50	classe1	1	0	0	0
2	M	40.99	classe2	0	0	1	0
3	S	10.00	classe3	0	0	0	1

# Transformation de données continues en données discrètes

Parfois il est nécessaire de devoir transformer des données continues en données discrètes. Par exemple si l'on a des revenus très variés il est préférable de les regrouper dans des catégories. Cette étape de groupement des données par classe s'appelle le binning ou la discretisation.

Scikitlearn propose la fonction KBinsDiscretizer qui permet de spécifier le nombre de groupe, le type d'encodage et la transformation.

In [8]:

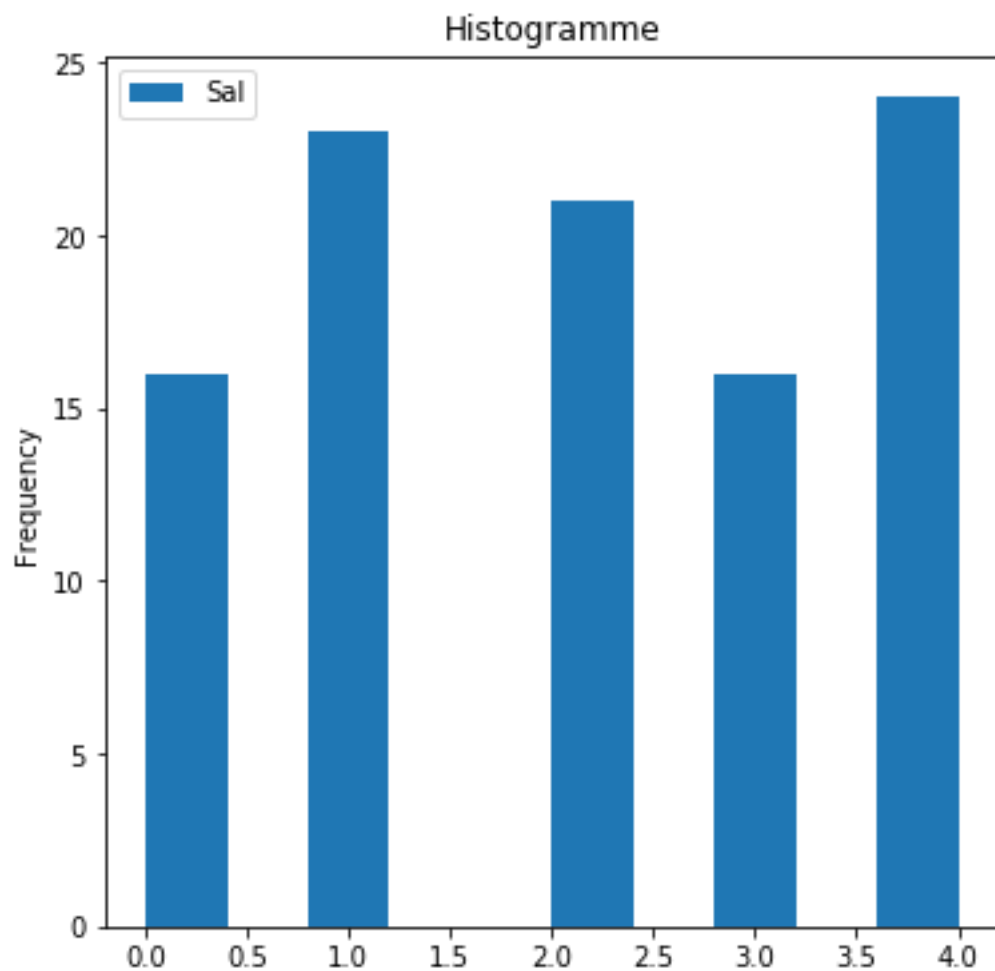
```
1  from sklearn.preprocessing import KBinsDiscretizer
2  import numpy as np
3  from random import uniform
4  import matplotlib.pyplot as plt
5  print ("Creation d'un jeu de données aléatoire de 100 lignes")
6  df_test = pd.DataFrame(
7      {'Sal': np.random.uniform(1000,10000,size=100)},columns=['Sal'])
8
9
10 df_test.plot(kind='hist', figsize=(6,6),title='Histogramme')
11 plt.show()
12
13
14
```

Creation d'un jeu de données aléatoire de 100 lignes

<Figure size 600x600 with 1 Axes>

In [9]:

```
1
2 X=np.array(df_test['Sal']).reshape(-1,1)
3 #KBinsDiscretizer fonctionne sur un array.
4 #Comme il n'y a qu'une colonne il faut
5 #faire un reshape pour lui préciser
6
7 ▼ disc = KBinsDiscretizer(n_bins=5, encode='ordinal',
8                           strategy='uniform')
9 df_test['Sal']=disc.fit_transform(X)
10
11 df_test.plot(kind='hist', figsize=(6,6),title='Histogramme')
12 plt.show()
13
14
```



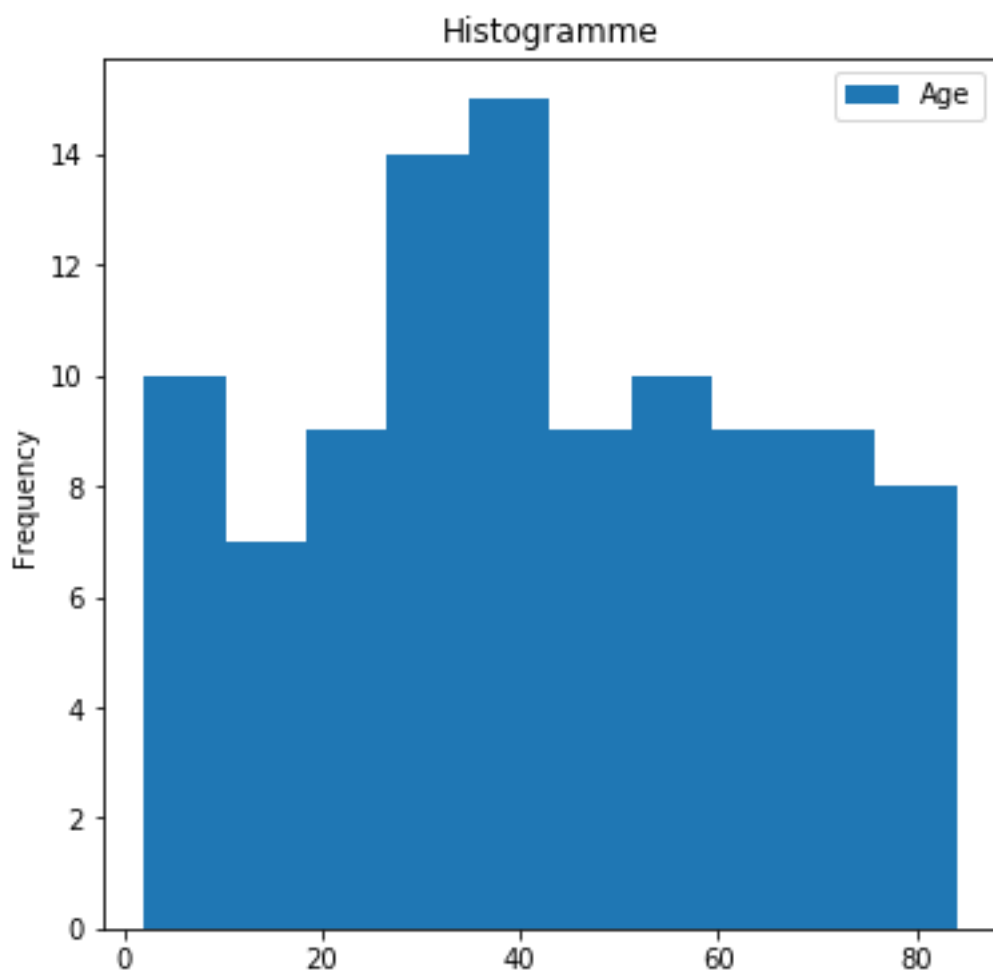
Il est également possible de spécifier des bins spécifiques à l'aide de la fonction cut sur un dataframe.



In [10]:

```
1 import numpy as np
2
3 print ("Creation d'un jeu de données aléatoire de 100 lignes")
4 df_test = pd.DataFrame(
5     {'Age': np.random.randint(1, 85, size=100)}, columns=['Age'])
6
7 df_test.plot(kind='hist', figsize=(6,6), title='Histogramme')
8 plt.show()
9
```

Creation d'un jeu de données aléatoire de 100 lignes



In [11]:

```
1 bins = (0, 25, 65, 85)
2 #Attention le nombre de label doit être inférieur au nombre de bins
3 group_names = ['Jeune', 'Adulte', 'Senior']
4 df_test['Age'] = pd.cut(df_test['Age'], bins, labels=group_names)
5
```

Attention pd.cut transforme les données à l'aide des labels comme le montre l'exemple ci-dessous :

In [12]:

```
1 print (df_test.head())
```

	Age
0	Senior
1	Jeune
2	Jeune
3	Jeune
4	Jeune

Il est donc indispensable de transformer ces données symbolique en numérique, par exemple, à l'aide de LabelEncoder ou tout autre méthode présentée précédemment.

In [13]:

```
1 from sklearn.preprocessing import LabelEncoder
2
3 class_label_encoder = LabelEncoder()
4 print ("\nRemplacement direct d'age avec les nouvelles valeurs\n")
5 df_test["Age"]=class_label_encoder.fit_transform(df_test["Age"])
6
7 display(df_test.head())
```

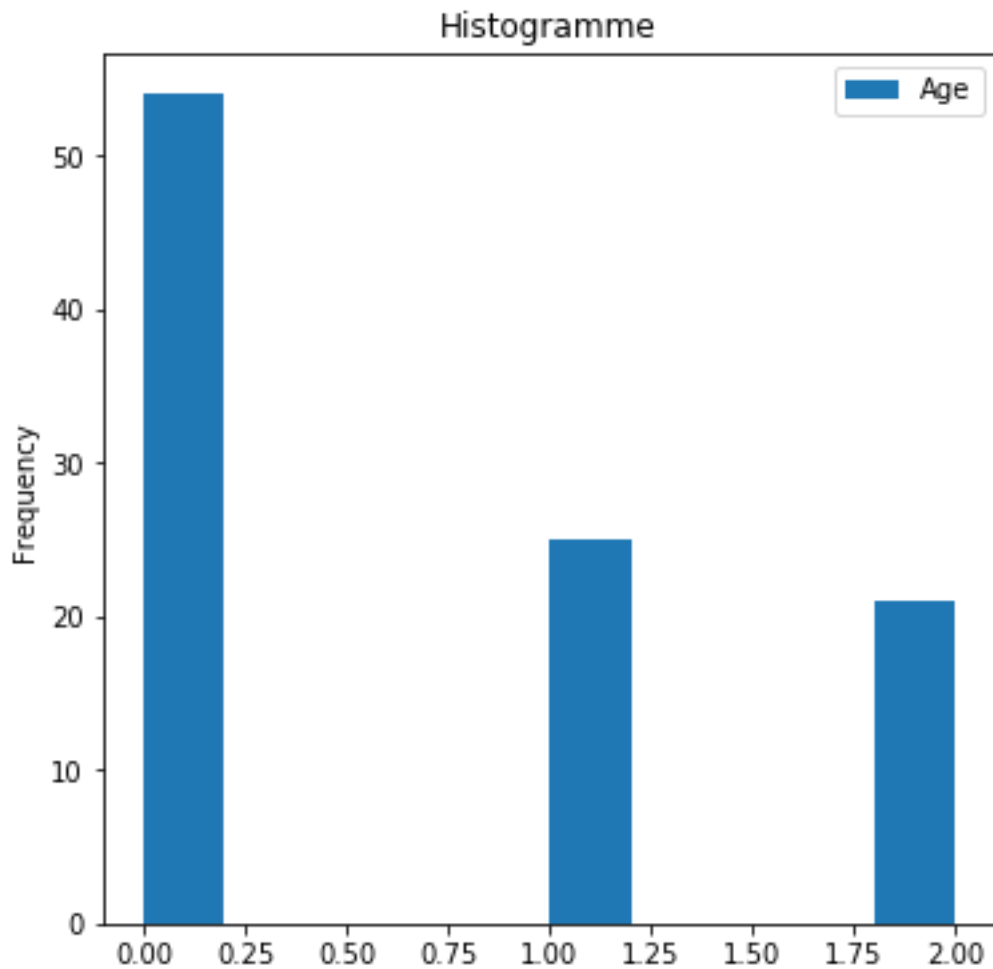
Remplacement direct d'age avec les nouvelles valeurs

	Age
0	2
1	1
2	1
3	1
4	1

Affichage du nouvel histogramme des ages après l'étape de transformation.

In [14]:

```
1 df_test.plot(kind='hist', figsize=(6,6), title='Histogramme')
2 plt.show()
```



## Mise à l'échelle des valeurs attributs (Feature scaling)

Le mise à l'échelle des valeurs des attributs (Feature scaling) est une méthode qui est utilisée pour normaliser les tailles des valeurs des attributs. Elle est aussi appelée **normalisation** (ou **standardisation**) et constitue une étape très importante dans le pré-traitement des données notamment lorsque des distances sont utilisées. C'est le cas par exemple pour KNN, SVM, Regression, ... mais également pour des méthodes de réduction de dimensions comme PCA et même en apprentissage non supervisée (K-Means).

La normalisation est, bien entendu, effectuée attribut par attribut dans le cas où plusieurs attributs doivent être mis à l'échelle.

## Normalisation (ou min-max scaling)

La normalisation permet de mettre toutes les valeurs dans un intervalle de [0,1]. Elle suit la formule :

$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

En scikit learn la normalisation se fait par la fonction `MinMaxScaler()`. Par défaut `MinMaxScaler` normalise entre 0 et 1. Il est possible de changer la valeur : `MinMaxScaler(feature_range=(0, 2))` normalisera les valeurs entre 0 et 2.

In [15]:

```
1 import pandas as pd
```

```

2 import numpy as np
3 #import numpy.random.uniform
4 from sklearn import preprocessing
5
6 print ("Création d'un dataframe de 7 valeurs")
7 data = {'Valeur': [14,-16,34,17,65,-32,5]}
8 df = pd.DataFrame(data,dtype='float')
9 #dtype = float car la normalisation considère
10 #que les objets sont des float
11 display("Max : ",df.max()," Min : ",df.min(),df)
12
13 df.plot(kind='bar')

```

Création d'un dataframe de 7 valeurs

'Max : '

Valeur 65.0  
dtype: float64

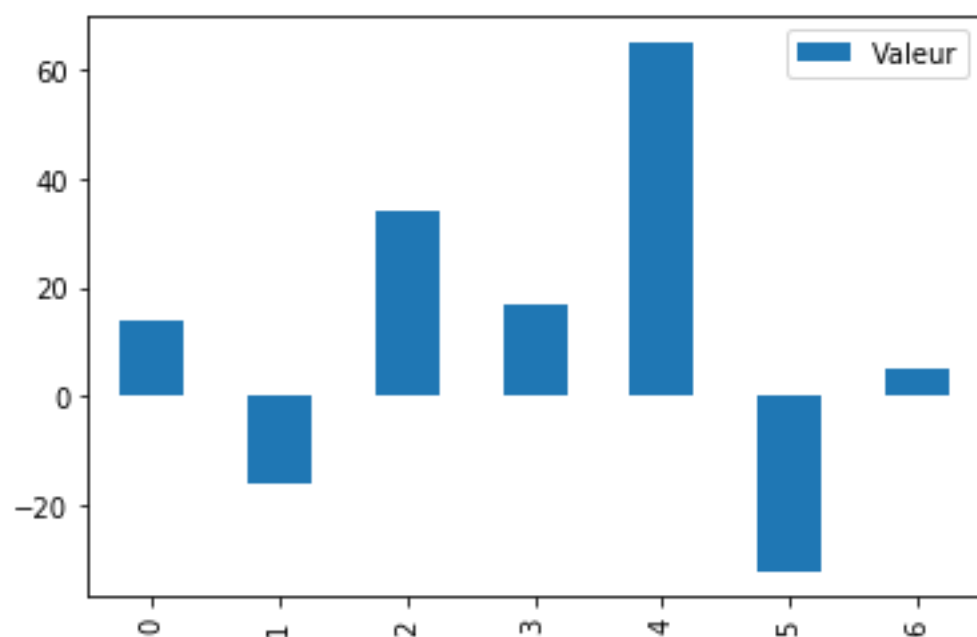
' Min : '

Valeur -32.0  
dtype: float64

	Valeur
0	14.0
1	-16.0
2	34.0
3	17.0
4	65.0
5	-32.0
6	5.0

Out[15]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x114dbe860>



In [16]:

```
1 print ("Création d'un dataframe normalisé")
2 normalise = preprocessing.MinMaxScaler()
3 df_normalise = normalise.fit_transform(df)
4 df_normalise = pd.DataFrame(df_normalise, columns=['Valeur'])
5
6 ▼ display("Max : ",df_normalise.max(),
7          " Min : ",df_normalise.min(),
8          df_normalise)
9
10 df_normalise.plot(kind='bar')
```

Création d'un dataframe normalisé

'Max : '

Valeur 1.0  
dtype: float64

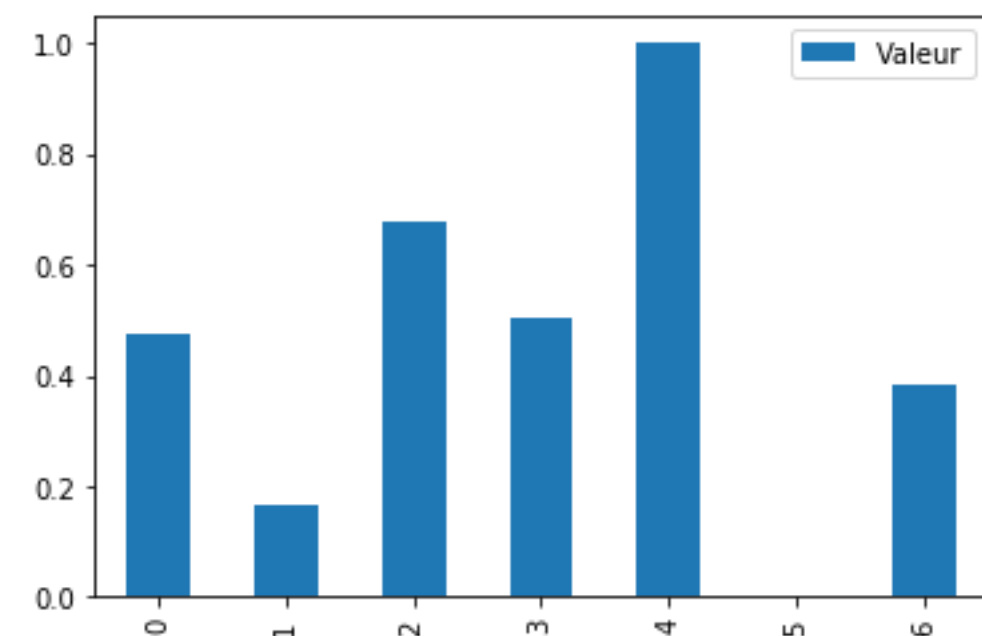
' Min : '

Valeur 0.0  
dtype: float64

	Valeur
0	0.474227
1	0.164948
2	0.680412
3	0.505155
4	1.000000
5	0.000000
6	0.381443

Out[16]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x114db54e0>



In [17]:

```
1  ▼ #scaler = MinMaxScaler(feature_range=(0, 1))
2    normalise = preprocessing.MinMaxScaler(feature_range=(0, 2))
3    df_normalise = normalise.fit_transform(df)
4    df_normalise = pd.DataFrame(df_normalise, columns=['Valeur'])
5
6  ▼ display("Max : ",df_normalise.max(),
7           " Min : ",df_normalise.min(),
8           df_normalise)
9    df_normalise.plot(kind='bar')
```

'Max : '

Valeur 2.0  
dtype: float64

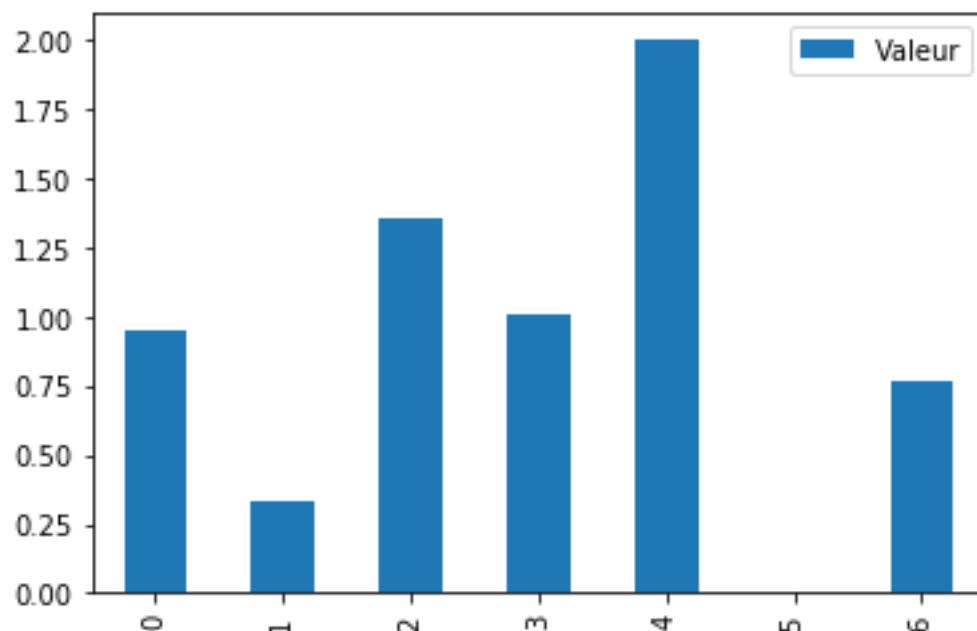
' Min : '

Valeur 0.0  
dtype: float64

	Valeur
0	0.948454
1	0.329897
2	1.360825
3	1.010309
4	2.000000
5	0.000000
6	0.762887

Out[17]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x114ec54e0>



L'exemple suivant illustre l'intérêt de normaliser plusieurs attributs. Il contient trois attributs où chaque valeur est prise au hasard en fonction d'une loi de distribution différente (2 asymétriques et 1 symétrique) : une avec une loi de distribution  $X^2$  ([https://fr.wikipedia.org/wiki/Loi\\_du\\_χ²](https://fr.wikipedia.org/wiki/Loi_du_χ²)), une avec une loi bêta ([https://fr.wikipedia.org/wiki/Loi\\_bêta](https://fr.wikipedia.org/wiki/Loi_bêta)) et une avec une loi normale. L'asymétrie d'une distribution est positive si la queue de droite (à valeurs hautes) est plus longue ou grosse, et négative si la queue de gauche (à valeurs basses) est plus longue ou grosse ([https://fr.wikipedia.org/wiki/Asymétrie\\_\(statistiques\)](https://fr.wikipedia.org/wiki/Asymétrie_(statistiques)))).

In [18]:

```
1 df = pd.DataFrame({
2     # asymétrie positive
3     'ChiSquare': np.random.chisquare(3, 1000)+50,
4     # asymétrie négative
5     'Beta': np.random.beta(20, 1, 1000)*30,
6     # pas d'asymétrie
7     'Normale': np.random.normal(110, 15, 1000)
8 })
```

In [19]:

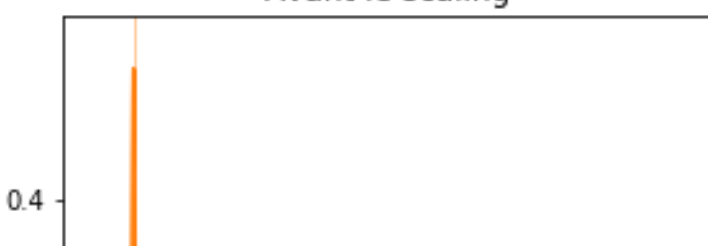
```
1 import matplotlib.pyplot as plt
2 import seaborn as sns
3 scaler = preprocessing.MinMaxScaler()
4 scaled_df = scaler.fit_transform(df)
5 scaled_df = pd.DataFrame(scaled_df,
6                          columns=['ChiSquare', 'Beta', 'Normale'])
7
8 fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(10, 8))
9 ax1.set_title('Avant le scaling')
10 sns.distplot(df['ChiSquare'], ax=ax1, kde=True)
11 sns.distplot(df['Beta'], ax=ax1, kde=True)
12 sns.distplot(df['Normale'], ax=ax1, kde=True)
13
14 ax2.set_title('Après le min-max scaling')
15 sns.distplot(scaled_df['ChiSquare'], ax=ax2, kde=True)
16 sns.distplot(scaled_df['Beta'], ax=ax2, kde=True)
17 sns.distplot(scaled_df['Normale'], ax=ax2, kde=True)
18
19 plt.show()
```

/Users/pascalponcelet/Desktop/Sicki-learn/Tools/tools/lib/python3.6/site-packages/matplotlib/axes/\_axes.py:6521: MatplotlibDeprecationWarning:

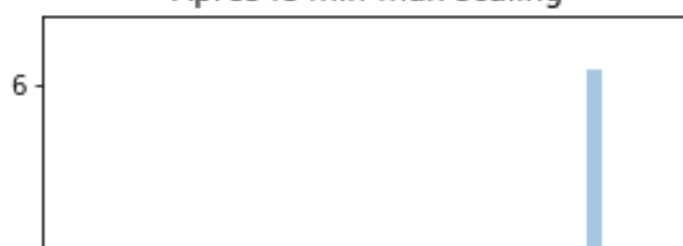
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1. Use 'density' instead.

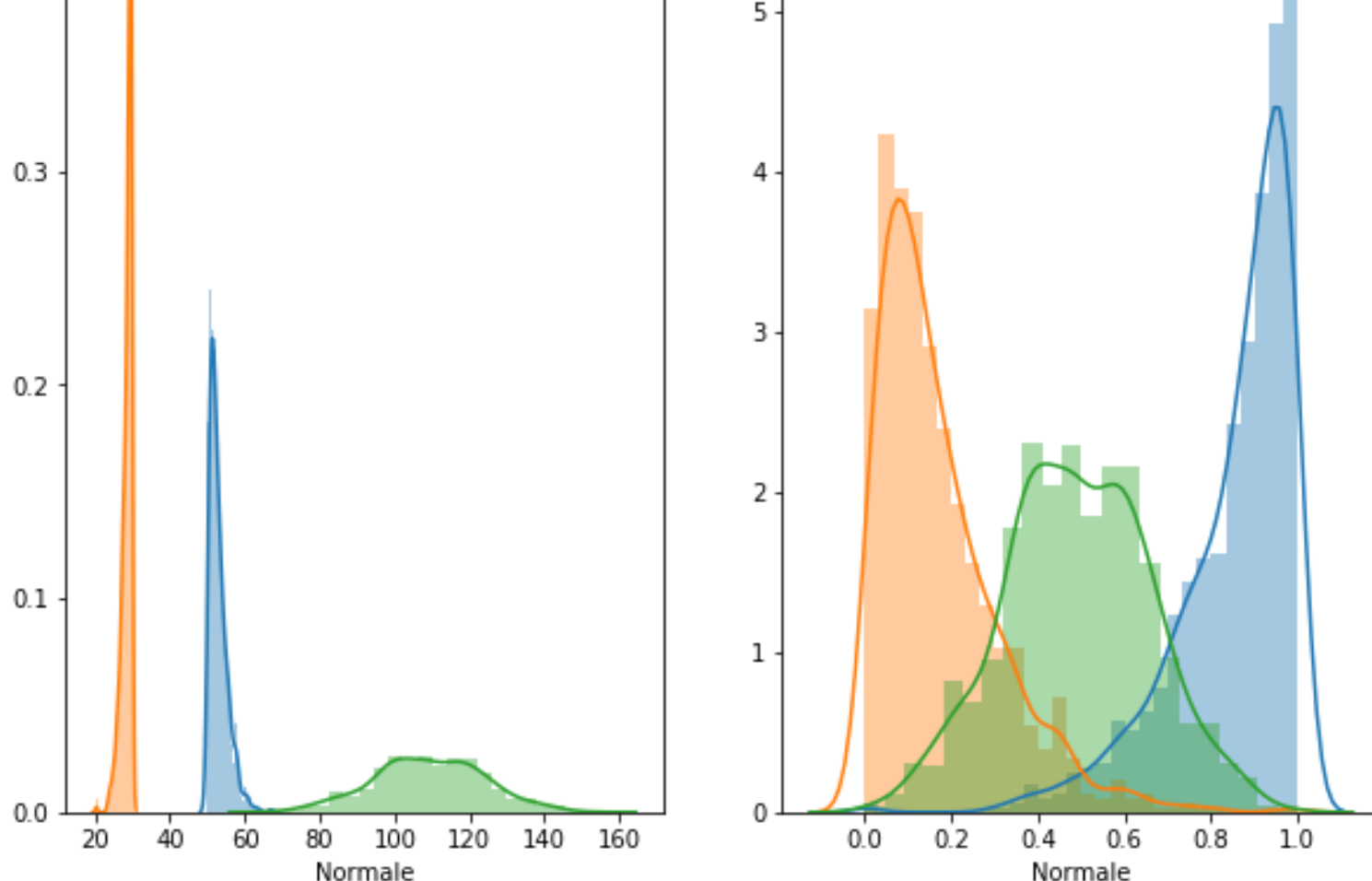
alternative="'density'", removal="3.1")

Avant le scaling



Après le min-max scaling





Nous pouvons constater que les asymétries restent les mêmes mais que maintenant toutes les valeurs sont comprises entre 0 et 1.

## Standardisation

La standardisation est utile lorsque les attributs suivent des lois normales mais avec des moyennes et écarts type différents. Elle permet, par exemple, de rendre les algorithmes moins sensibles aux outliers.

En scikit learn la standardisation se fait par la fonction `StandardScaler()` en appliquant :

$$z = \frac{x_i - \mu}{\sigma}$$

où  $\mu$  représente la moyenne (*mean*) et  $\sigma$  l'écart type (*standard deviation*).

Rappel : ([https://fr.wikipedia.org/wiki/Loi\\_normale](https://fr.wikipedia.org/wiki/Loi_normale) ([https://fr.wikipedia.org/wiki/Loi\\_normale](https://fr.wikipedia.org/wiki/Loi_normale)))

Lorsqu'une variable aléatoire  $X$  suit la loi normale, elle est dite *gaussienne* ou *normale* et il est habituel d'utiliser la notation avec la variance  $\sigma^2$  :

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

`StandardScaler` suppose donc que les données suivent une loi normale et les redimensionne pour que la distribution soit centrée autour de 0 avec un écart-type de 1. Elle vise donc à transformer les valeurs pour qu'elles répondent à la même loi normale

$$X \sim \mathcal{N}(0, 1)$$

Il est toujours intéressant d'afficher la distribution des données pour voir si ces dernières peuvent être standardisées.



```

1 import numpy as np
2 from pandas import DataFrame
3
4
5 ▼ df = pd.DataFrame(
6 ▼     {'Valeur': [10,9,8,7,6,5,5,6,7,8,9,10]},
7     dtype='float',
8     columns=[ 'Valeur' ])
9
10 ▼ display(df.head(),"Moyenne ",
11         df['Valeur'].mean(),
12         " Ecart type ",
13         df['Valeur'].std())
14
15 fig, ax1 = plt.subplots(ncols=1, figsize=(8, 6))
16 title='X ~ N(''%0.2f'%df['Valeur'].mean()+",%0.2f"%df['Valeur'].std()+'^2)'
17 ax1.set_title(title)
18
19 sns.kdeplot(df['Valeur'], ax=ax1);

```

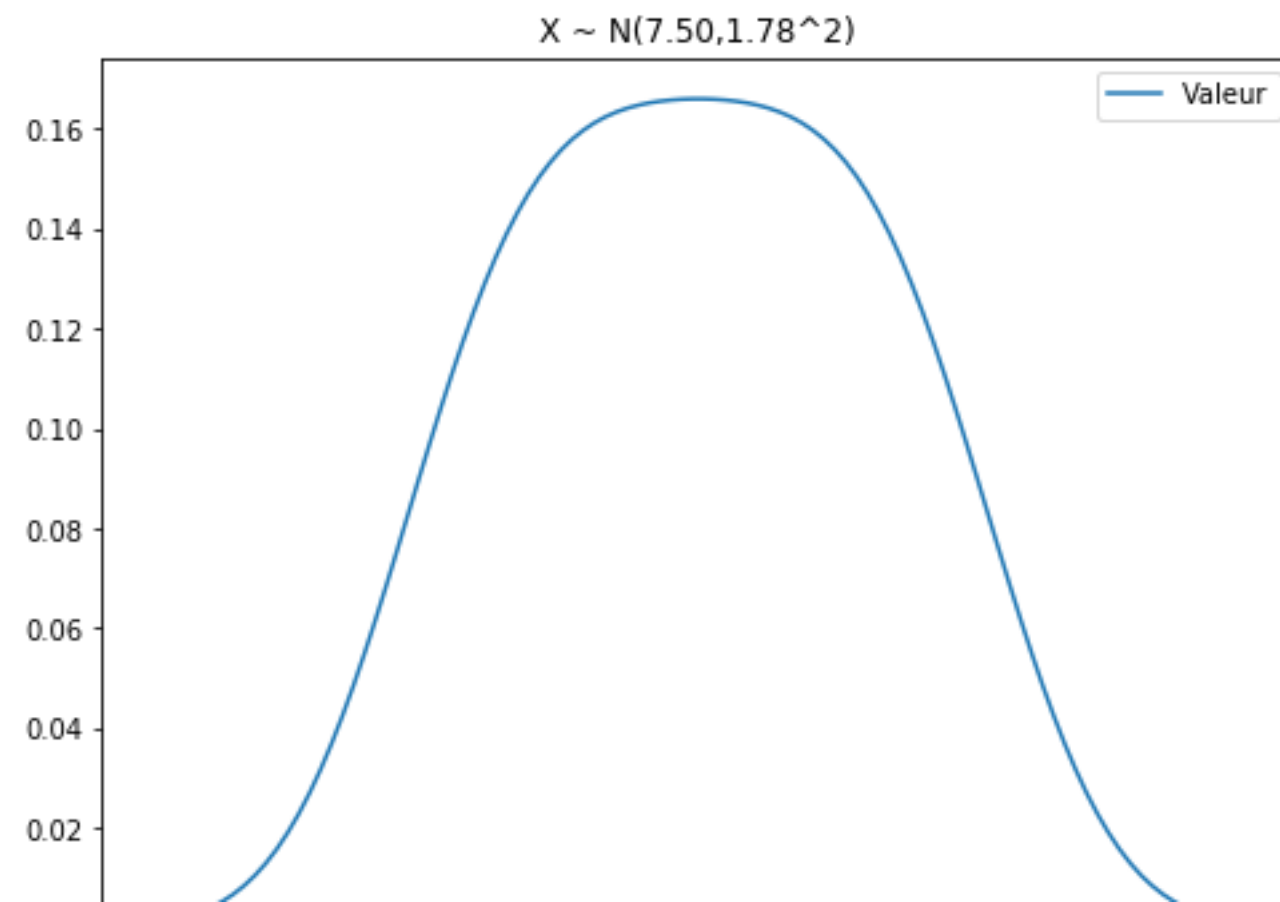
	Valeur
0	10.0
1	9.0
2	8.0
3	7.0
4	6.0

'Moyenne '

7.5

' Ecart type '

1.7837651700316894



In [21]:

```
1  from sklearn.preprocessing import StandardScaler
2
3  print ("Création d'un dataframe avec StandardScaler")
4  standardscaler = preprocessing.StandardScaler()
5  df_standardscale = standardscaler.fit_transform(df)
6  df_standardscale = pd.DataFrame(df_standardscale,
7                                  columns=['Valeur'])
8
9  display(df_standardscale, "Moyenne ",
10          df_standardscale['Valeur'].mean(),
11          " Ecart type ",
12          df_standardscale['Valeur'].std())
13
14  fig, ax1 = plt.subplots(ncols=1, figsize=(8, 6))
15  title='X ~ N(''%0.2f'%df_standardscale['Valeur'].mean()+",%0.2f"%df_standardscale['Valeur'].std())
16  ax1.set_title(title)
17
18  sns.kdeplot(df_standardscale['Valeur'], ax=ax1);
```

Création d'un dataframe avec StandardScaler

	Valeur
0	1.46385
1	0.87831
2	0.29277
3	-0.29277
4	-0.87831
5	-1.46385
6	-1.46385
7	-0.87831
8	-0.29277
9	0.29277
10	0.87831
11	1.46385

'Moyenne '

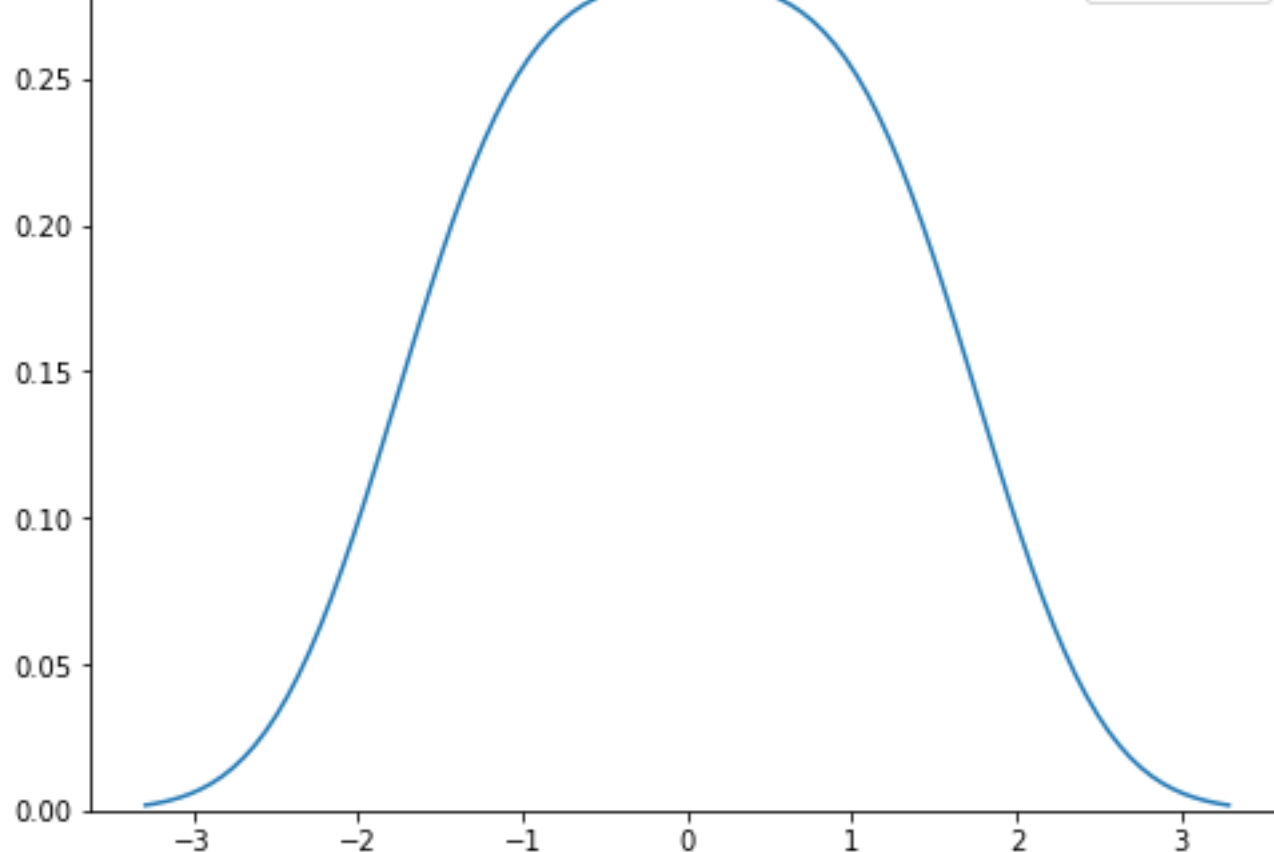
-1.850371707708594e-17

' Ecart type '

1.044465935734187

X ~ N(-0.00,1.04^2)

Valeur



Comme précédemment, le code suivant illustre la standardisation avec

$$X \sim \mathcal{N}(10, 2^2)$$

$$X \sim \mathcal{N}(40, 7^2)$$

$$X \sim \mathcal{N}(110, 15^2)$$

In [22]:

```
1 df = pd.DataFrame({
2     'Normale1': np.random.normal(10, 2, 1000),
3     'Normale2': np.random.normal(40, 7, 1000),
4     'Normale3': np.random.normal(110, 15, 1000)
5 })
```

In [23]:

```
1
2 standardscaler = preprocessing.StandardScaler()
3 df_standardscale = standardscaler.fit_transform(df)
4 df_standardscale=pd.DataFrame(df_standardscale,
5                               columns=['Normale1',
6                                       'Normale2',
7                                       'Normale3'])
8
9
10 fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(10, 8))
11 ax1.set_title('Avant le standard scaling')
12 sns.distplot(df['Normale1'], ax=ax1)
13 sns.distplot(df['Normale2'], ax=ax1)
14 sns.distplot(df['Normale3'], ax=ax1)
15
16 ax2.set_title('Après le standard scaling')
17 sns.distplot(df_standardscale['Normale1'], ax=ax2)
18 sns.distplot(df_standardscale['Normale2'], ax=ax2)
19 sns.distplot(df_standardscale['Normale3'], ax=ax2)
```

```

19 sns.distplot(df_standardscale[ Normales ], ax=ax2)
20
21 plt.show()

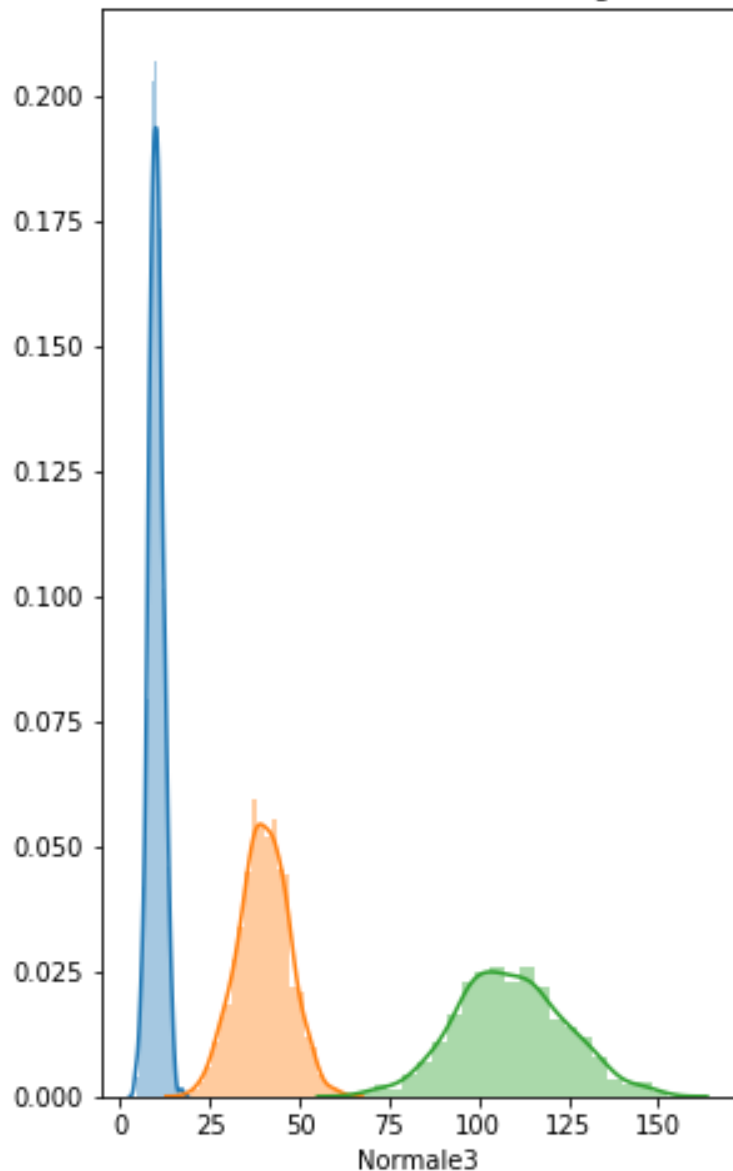
```

/Users/pascalponcelet/Desktop/Sicki-learn/Tools/tools/lib/python3.6/site-packages/matplotlib/axes/\_axes.py:6521: MatplotlibDeprecationWarning:

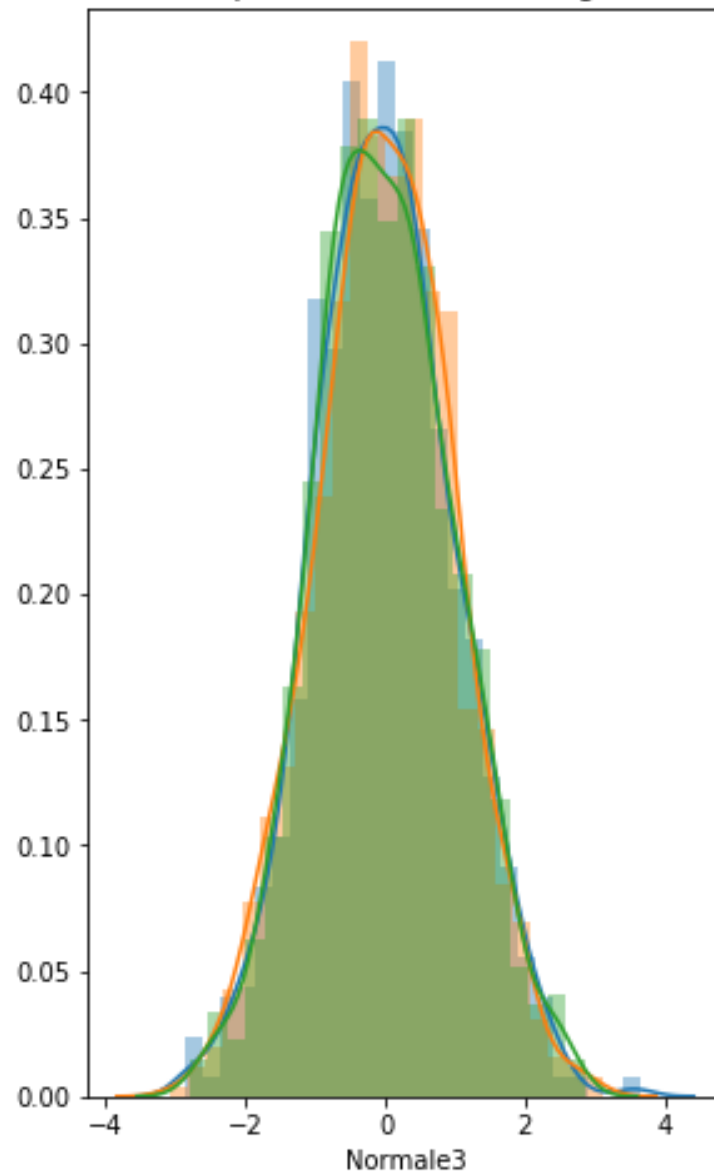
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1. Use 'density' instead.

```
alternative="'density'", removal="3.1")
```

Avant le standard scaling



Après le standard scaling



## Traitement des valeurs manquantes

Dans les données réelles, de très nombreuses fois et pour différentes raisons (données corrompues, données inexistantes, extraction incomplète, etc.) des valeurs peuvent être absentes ou apparaître sous la forme d'outlier. Généralement on parle de **valeurs manquantes** et elles peuvent poser de nombreux problèmes pour certains classifieurs qui y sont très sensibles (e.g. SVM).

En fonction des domaines, elles peuvent apparaître sous la forme de -1, 0, -999 ou NaN (Not a Number). Pandas, numpy et scikit learn utilisent NaN pour les valeurs manquantes et toutes valeurs avec NaN sont ignorées dans les opérations d'agrégation comme sum, count, etc. Il est donc préférable de remplacer toutes les valeurs manquantes par NaN.

Pour remplacer des données sous la forme de NaN il suffit d'utiliser la fonction `replace()` du dataframe.

In [24]:

```
1 import pandas as pd
2 import numpy as np
3 data = [[7, 2, 3], [4, -1, 6], [10, 5, 9]]
4 print('Remplacement valeur -1 en nan\n')
5 df = pd.DataFrame(data)
6 print ('Avant :')
7 display(df)
8 df=df.replace(-1,np.nan)
9 print ('Après :')
10 display(df)
```

Remplacement valeur -1 en nan

Avant :

	0	1	2
0	7	2	3
1	4	-1	6
2	10	5	9

Après :

	0	1	2
0	7	2.0	3
1	4	NaN	6
2	10	5.0	9

Il existe différentes stratégies, en fonction des données et du domaine, pour traiter les valeurs manquantes :

1. Supprimer les lignes contenant des valeurs manquantes
2. Remplacer les valeurs par *mean*, *median*, *mode*
3. Mettre une catégorie unique
4. Prédire la valeur manquante

Chaque stratégie a des avantages et des inconvénients. Au travers de l'exemple suivant nous illustrons les différentes stratégies et les fonctionnalités de sickit learn pour les traiter.

In [25]:

```
1
2  #Création de fichiers exemples
3  fichier = open("exemplenullvalues.csv", "w")
4  fichier.write("Nom;Age;Dept;Sal;Prime\n")
5  fichier.write("Marie;22;;48000;1\n")
6  fichier.write("Isabelle;;Comptable;52000;0\n")
7  fichier.write("Pierre;35;Informatique;;1\n")
8  fichier.write("Paul;43;Commercial;49000;1\n")
9  fichier.write("Jean;;Commercial;;0\n")
10 fichier.write("Michel;35;;51000;0\n")
11 fichier.write("Nancy;45;;66000;1\n")
12 fichier.close()
13
14
```

Pour connaître le nombre de valeurs manquantes :

In [26]:

```
1 df = pd.read_csv('exemplenullvalues.csv',sep=';')
2 display (df)
3
4
5 print ('Par rapport aux colonnes :\n')
6 display (df.info())
7 print ('\nPar rapport aux différentes lignes :\n')
8 display (df.isnull().sum(axis=1))
9
```

	Nom	Age	Dept	Sal	Prime
0	Marie	22.0	NaN	48000.0	1
1	Isabelle	NaN	Comptable	52000.0	0
2	Pierre	35.0	Informatique	NaN	1
3	Paul	43.0	Commercial	49000.0	1
4	Jean	NaN	Commercial	NaN	0
5	Michel	35.0	NaN	51000.0	0
6	Nancy	45.0	NaN	66000.0	1

Par rapport aux colonnes :

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 5 columns):
Nom      7 non-null object
Age      5 non-null float64
Dept     4 non-null object
Sal      5 non-null float64
Prime    7 non-null int64
dtypes: float64(2), int64(1), object(2)
memory usage: 360.0+ bytes
```

None

Par rapport aux différentes lignes :

```
0    1
1    1
2    1
3    0
4    2
5    1
6    1
dtype: int64
```

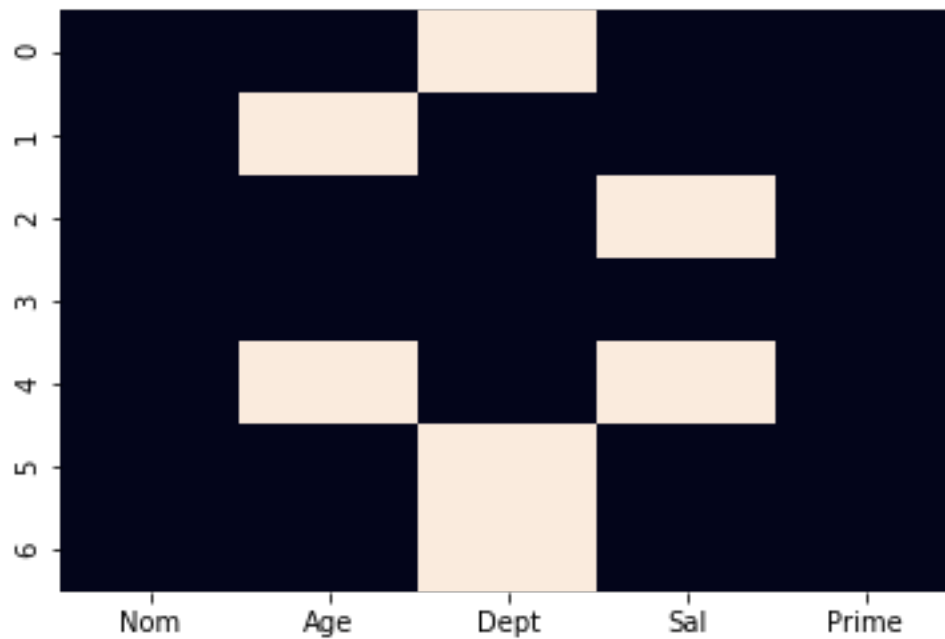
Pour visuellement afficher si le jeu de données contient des valeurs manquantes :

In [27]:

```
1 import seaborn as sns
2 sns.heatmap(df.isnull(), cbar=False)
```

Out[27]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x11590cd30>



## Supprimer les lignes contenant des valeurs manquantes

Pour supprimer simplement les lignes (ou les colonnes) qui contiennent des valeurs manquantes, il est possible d'utiliser la fonction `dropna` sur le dataframe :

*DataFrame.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)*



In [28]:

```
1 df_test = df.copy() #pour tester
2 print ('Suppression des lignes pour lesquelles au moins un élément est manq
3 print (df_test.dropna())
4
5 print ('\nSuppression des colonnes pour lesquelles au moins un élément est
6 print (df_test.dropna(axis="columns"))
7
8 print ('\nSuppression des lignes qui ont au moins 4 valeurs non manquantes\
9 print (df_test.dropna(thresh=4))
10 print ("La ligne 4 n'apparaît plus car elle a 2 valeurs manquantes sur les
11
12 print ('\nRemplacement du dataframe initial en supprimant les lignes manqua
13 df_test.dropna(inplace=True)
14 display(df_test)
```

Suppression des lignes pour lesquelles au moins un élément est manquant

	Nom	Age	Dept	Sal	Prime
3	Paul	43.0	Commercial	49000.0	1

Suppression des colonnes pour lesquelles au moins un élément est manquant

	Nom	Prime
0	Marie	1
1	Isabelle	0
2	Pierre	1
3	Paul	1
4	Jean	0
5	Michel	0
6	Nancy	1

Suppression des lignes qui ont au moins 4 valeurs non manquantes

	Nom	Age	Dept	Sal	Prime
0	Marie	22.0	NaN	48000.0	1
1	Isabelle	NaN	Comptable	52000.0	0
2	Pierre	35.0	Informatique	NaN	1
3	Paul	43.0	Commercial	49000.0	1
5	Michel	35.0	NaN	51000.0	0
6	Nancy	45.0	NaN	66000.0	1

La ligne 4 n'apparaît plus car elle a 2 valeurs manquantes sur les 5 colonnes

Remplacement du dataframe initial en supprimant les lignes manquantes

	Nom	Age	Dept	Sal	Prime
3	Paul	43.0	Commercial	49000.0	1

## Remarques

La suppression des lignes permet de pouvoir utiliser des classifieurs qui sont sensibles aux valeurs manquantes.

L'utilisation de ces méthodes risquent de supprimer de nombreuses informations (C.f. le dernier exemple). Il est conseillé de ne pas les utiliser si le nombre d'objets supprimés est trop grand. Le remplacement est généralement préférable à la suppression des données.

## Remplacer les valeurs par mean, median, mode

Pour les variables numériques, il est possible de remplacer les valeurs manquantes par la moyenne, la médiane, le mode, etc. Rappel : dans une série le mode correspond à une valeur dominante, i.e. la valeur la plus représentée d'une variable quelconque dans une population donnée. Le choix dépend bien entendu du contexte.

In [29]:

```
1  import numpy as np
2  print ('Rappel de la colonne Age\n', df['Age'])
3
4  print ('\nMoyenne', df['Age'].mean(),
5         ' Median', df['Age'].median(),
6         '\nMode', df['Age'].mode())
7
8  print ('\n Remplacement des valeurs manquantes de Age par la moyenne\n')
9  df_test = df.copy() #pour tester
10 print ('Avant : \n')
11 display(df_test['Age'])
12 df_test['Age']=df_test['Age'].replace(np.NaN,df_test['Age'].mean())
13 print ('\nAprès : \n')
14 display(df_test['Age'])
15
16 print ('\n Remplacement des valeurs manquantes de Age par la valeur la plus
17 df_test = df.copy() #pour tester
18 print ('Valeur la plus fréquente \n',
19         df_test['Age'].value_counts().idxmax(),
20         ' (',df_test['Age'].value_counts().max(),')')
21 newval=df_test['Age'].value_counts().idxmax()
22 print ('Avant : \n')
23 display(df_test['Age'])
24 df_test['Age']=df_test['Age'].replace(np.NaN,newval)
25 print ('\nAprès : \n')
26 display(df_test['Age'])
27
28 print ('\n Données catégorielles. Remplacement des valeurs manquantes de De
29 df_test = df.copy() #pour tester
30 print ('Valeur la plus fréquente',
31         df_test['Dept'].value_counts().idxmax(),
32         ' (',df_test['Dept'].value_counts().max(),')')
33 newval=df_test['Dept'].value_counts().idxmax()
34 print ('Avant : \n')
35 display(df_test['Dept'])
36 df_test['Dept']=df_test['Dept'].replace(np.NaN,newval)
37 print ('Après : \n')
```

38	display(df_test['Dept'])
39	

Rappel de la colonne Age

```
0      22.0
1      NaN
2      35.0
3      43.0
4      NaN
5      35.0
6      45.0
Name: Age, dtype: float64

Moyenne 36.0  Median 35.0
Mode 0      35.0
dtype: float64
```

Remplacement des valeurs manquantes de Age par la moyenne

Avant :

```
0      22.0
1      NaN
2      35.0
3      43.0
4      NaN
5      35.0
6      45.0
Name: Age, dtype: float64
```

Après :

```
0      22.0
1      36.0
2      35.0
3      43.0
4      36.0
5      35.0
6      45.0
Name: Age, dtype: float64
```

Remplacement des valeurs manquantes de Age par la valeur la plus fr  
équente sans utiliser le mode

Valeur la plus fréquente

```
35.0 ( 2 )
```

Avant :

```
0      22.0
1      NaN
2      35.0
3      43.0
4      NaN
5      35.0
6      45.0
Name: Age, dtype: float64
```

Après :

```
0    22.0
1    35.0
2    35.0
3    43.0
4    35.0
5    35.0
6    45.0
```

Name: Age, dtype: float64

Données catégorielles. Remplacement des valeurs manquantes de Dept par la valeur la plus fréquente

Valeur la plus fréquente Commercial ( 2 )

Avant :

```
0      NaN
1    Comptable
2  Informatique
3    Commercial
4    Commercial
5      NaN
6      NaN
```

Name: Dept, dtype: object

Après :

```
0    Commercial
1    Comptable
2  Informatique
3    Commercial
4    Commercial
5    Commercial
6    Commercial
```

Name: Dept, dtype: object

Scikit learn propose aussi une fonction *SimpleImputer* qui permet de remplacer directement les valeurs. Elle s'applique sur un tableau et non pas un dataframe.

In [30]:

```
1 import numpy as np
2 from sklearn.impute import SimpleImputer
3 array = df.values
4 X = array[:,1:2]
5 print ('\n Remplacement des valeurs manquantes de Age par la moyenne\n')
6 print ('Avant : \n')
7 print (X)
8 imputer = SimpleImputer(missing_values=np.nan, strategy = 'mean')
9 imputer = imputer.fit(X)
10 X = imputer.transform(X)
11 print ('\nAprès : \n')
12 print (X)
```

Remplacement des valeurs manquantes de Age par la moyenne

Avant :

```
[[22.0]
 [nan]
 [35.0]
 [43.0]
 [nan]
 [35.0]
 [45.0]]
```

Après :

```
[[22.]
 [36.]
 [35.]
 [43.]
 [36.]
 [35.]
 [45.]]
```

## Remarques

Cette approche est efficace quand le jeu de données est petit et que les valeurs peuvent facilement être remplacées.

Le fait de faire des approximations ajoute des biais dans les données.

## Affecter une catégorie unique

Dans le cas de variables catégorielles, lorsqu'il n'est pas possible de pouvoir connaître la valeur, il est possible d'affecter une valeur similaire aux NaN.

L'avantage est de pouvoir considérer toutes ces données comme étant de la même classe et ainsi elles seront transformées de la même manière que les autres valeurs de l'attribut lors de l'étape d'encodage de données catégorielles.

L'inconvénient est d'avoir une nouvelle classe qui ne correspond pas à grand chose et qui peut donc entraîner des problèmes lors de la classification.

In [31]:

```
1
2 print ( '\n Remplacement des valeurs manquantes de Dept par une valeur commune')
3 df_test = df.copy() #pour tester
4 print ( 'Avant: \n')
5 display(df_test['Dept'])
6 df_test['Dept']=df_test['Dept'].fillna("Inconnu")
7 print ( '\nAprès: \n')
8 display(df_test['Dept'])
```

Remplacement des valeurs manquantes de Dept par une valeur commune

Avant:

```
0      NaN
1  Comptable
2  Informatique
3  Commercial
4  Commercial
5      NaN
6      NaN
Name: Dept, dtype: object
```

Après:

```
0      Inconnu
1  Comptable
2  Informatique
3  Commercial
4  Commercial
5      Inconnu
6      Inconnu
Name: Dept, dtype: object
```

## Prédire les valeurs manquantes

Le principe est d'utiliser les autres attributs pour appliquer un algorithme d'apprentissage en considérant que la valeur à prédire est la colonne qui contient des NaN.

L'exemple suivant illustre comment utiliser KNN pour prédire des valeurs.

In [32]:

```
1  import numpy as np
2
3  print ("Creation d'un jeu de données aléatoire de 100 lignes")
4  df_test = pd.DataFrame(
5      {'Age': np.random.randint(30, 35, size=100),
6       'Sal': np.random.randint(3, size=100)*1000,
7       'Prime': np.random.randint(2, size=100),
8       'Dept': np.random.randint(3, size=100)},
9      columns=['Age', 'Sal', 'Prime', 'Dept'])
10
11 print (df_test.shape)
12 display(df_test.head(5))
13 array = df_test
14
15 print ('Sélection de 3% du jeu de données pour mettre NaN dans les Dept')
16 echantillon = df_test.sample(frac=0.03)
17 display(echantillon)
18
19 print("Remplacement par NaN pour l'échantillon")
20 index = echantillon.index.values
21 for ind in index:
22     echantillon.at[ind, 'Dept']=-1
23     echantillon=echantillon.replace(-1,np.nan)
24     df_test.at[ind, 'Dept']=-1
25     df_test=df_test.replace(-1,np.nan)
26
27
28
```

Creation d'un jeu de données aléatoire de 100 lignes  
(100, 4)

	Age	Sal	Prime	Dept
0	31	0	1	1
1	32	2000	1	2
2	34	2000	1	1
3	30	1000	1	2
4	31	2000	0	2

Sélection de 3% du jeu de données pour mettre NaN dans les Dept

	Age	Sal	Prime	Dept
78	30	1000	1	0
35	31	0	1	2
33	32	2000	1	0

Remplacement par NaN pour l'échantillon

In [33]:

```
1  from sklearn.model_selection import train_test_split
2  from sklearn.metrics import accuracy_score
3  from sklearn.neighbors import KNeighborsClassifier
4
5  print ('Récupération des lignes sans NAN')
6  sans_nan = df_test[['Sal', 'Prime', 'Age', 'Dept']].notnull().all(axis=1)
7  print ("Creation d'un dataframe sans NaN")
8  df_sansnan = df_test[sans_nan]
9
10 print ("Apprentissage sur le dataframe sans les nan")
11 array = df_sansnan.values
12 X = array[:,0:3]
13
14 y= array[:,3]
15
16 validation_size=0.25 #30% du jeu de données pour le test
17 testsize= 1-validation_size
18 seed=2
19
20 X_train,X_test,y_train,y_test=train_test_split(X, y,
21                                                  train_size=validation_size,
22                                                  random_state=seed,
23                                                  test_size=testsize)
24
25
26 clf=KNeighborsClassifier(n_neighbors=4)
27 clf.fit(X, y)
28
29 result = clf.predict(X_test)
30 print('\n accuracy :', accuracy_score(result, y_test),'\n')
31
32 #recuperation dans df_avecnan de toutes
33 #les lignes qui ont un nan (noter la negation)
34 print ('Remplacement des valeurs NaN par les valeurs prédites')
35 df_avecnan = df_test.loc[~sans_nan].copy()
36 df_avecnan['Dept'] = clf.predict(df_avecnan[['Sal', 'Prime', 'Age']])
37 display(df_avecnan)
```

Récupération des lignes sans NAN

Creation d'un dataframe sans NaN

Apprentissage sur le dataframe sans les nan

accuracy : 0.5753424657534246

Remplacement des valeurs NaN par les valeurs prédites

	Age	Sal	Prime	Dept
33	32	2000	1	0.0
35	31	0	1	1.0
78	30	1000	1	0.0



