

Direction de l'innovation et des relations avec les entreprises

cnrs **formation**
entreprises

Preprocessing des données

Alexandre Boucaud (CNRS/APC)





Imports





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

%matplotlib inline

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





Avant-propos : théorème du *No Free Lunch*

Si on ne fait pas d'hypothèse a priori sur les données, il n'y a aucune raison de préférer un algorithme à un autre.

La seule façon de savoir si un modèle est meilleur qu'un autre, il faut les évaluer et les comparer sur les données.





#1 Déterminer le type de problème auquel on a affaire





- sélection des modèles adéquats (*model selection*)
- choix de la métrique pour évaluer **les** modèles (*metric selection*)





#2 Préparer les données





- visualiser et parcourir les données pour trouver des corrélations
- isoler la/les variables cibles
- vérifier et traiter les données manquantes (*missing data handling*)
- sélectionner les bonnes variables d'entraînement ou en créer d'autres (*feature extraction*)
- normaliser les variables d'entraînement (*rescaling*)
- préparer la/les variables cibles (*encoding*)
- séparer l'ensemble du jeu de données en plusieurs échantillons (train/validation/test)





#3 Entraînement





- choisir les hyperparamètres des modèles (*initialisation*)
- entraîner les modèles avec les échantillons d'entraînement (*training*)
- vérifier la bonne convergence des modèles (*validation*)
- utiliser la validation croisée des données (*cross-validation*)





#4 Evaluation





- évaluer les modèles **une seule fois** sur l'échantillon de test
(*evaluation/scoring*)
- utiliser ces scores comparer les performances des modèles
(*model comparison*)





Principaux challenges à surmonter





- quantité de données insuffisante
- données d'entraînement non représentatives (sampling bias)
- mauvaise qualité des données
- variables d'entraînement inutiles ou décorréliées du problème
- sur-entraînement
- sous-entraînement





Préparation des données





Format des données et exemples





```
In [2]: from sklearn.datasets import fetch_california_housing
```





```
In [2]: from sklearn.datasets import fetch_california_housing
```

```
In [3]: data = fetch_california_housing()
```

```
print(data.DESCR)
```

```
.. _california_housing_dataset:
```

```
California Housing dataset
```

```
-----
```

```
**Data Set Characteristics:**
```

```
:Number of Instances: 20640
```

```
:Number of Attributes: 8 numeric, predictive attributes and the target
```

```
:Attribute Information:
```

- MedInc median income in block
- HouseAge median house age in block
- AveRooms average number of rooms
- AveBedrms average number of bedrooms
- Population block population
- AveOccup average house occupancy
- Latitude house block latitude
- Longitude house block longitude





Jeu de données d'entrée : X

Matrice de la forme `[n_samples, n_features]`





Jeu de données d'entrée : X

Matrice de la forme `[n_samples, n_features]`

```
In [4]: X = data.data  
        X.shape
```

```
Out[4]: (20640, 8)
```





Jeu de données cibles : y

Vecteur de la forme `[n_samples]`





Jeu de données cibles : y

Vecteur de la forme `[n_samples]`

```
In [5]: y = data.target  
        y.shape
```

```
Out[5]: (20640,)
```





```
In [6]: # shorter version  
X, y = fetch_california_housing(return_X_y=True)
```





Encoding des données de type catégories





Il y a deux façon d'encoder les données de type catégorie qui sont par nature des données discrètes : exemple le nom de la ville natale de quelqu'un.

```
In [7]: df = pd.DataFrame({  
    'city': ['Manhattan', 'Queens', 'Manhattan', 'Brooklyn', 'Brooklyn', 'Bronx'],  
    'employed': ['No', 'No', 'No', 'Yes', 'Yes', 'No']})
```





Ordinal encoder

L'encodage ordinal consiste à représenter chaque catégorie par un entier.

L'inconvénient est que cela crée un ordre non naturel entre les catégories qui tend à donner une information non existante, ce qui peut mettre en erreur certains algorithmes linéaires (en particulier).

```
In [8]: from sklearn.preprocessing import OrdinalEncoder
```

```
OrdinalEncoder().fit_transform(df)
```

```
Out[8]: array([[2., 0.],  
               [3., 0.],  
               [2., 0.],  
               [1., 1.],  
               [1., 1.],  
               [0., 0.]])
```





One-hot encoder

L'encodage one-hot permet de préserver l'absence d'ordre entre les catégories, au détriment d'augmenter considérablement le nombre de variables car il crée une nouvelle variable pour chaque catégorie de chaque variable catégorique.

```
In [9]: from sklearn.preprocessing import OneHotEncoder
```

```
OneHotEncoder(sparse=False).fit_transform(df)
```

```
Out[9]: array([[0., 0., 1., 0., 1., 0.],  
               [0., 0., 0., 1., 1., 0.],  
               [0., 0., 1., 0., 1., 0.],  
               [0., 1., 0., 0., 0., 1.],  
               [0., 1., 0., 0., 0., 1.],  
               [1., 0., 0., 0., 1., 0.]])
```





La fonction `get_dummies` de Pandas fait la même chose de manière plus explicite

```
In [10]: pd.get_dummies(df)
```

```
Out[10]:
```

	city_Bronx	city_Brooklyn	city_Manhattan	city_Queens	employed_No	employed_Yes
0	0	0	1	0	1	0
1	0	0	0	1	1	0
2	0	0	1	0	1	0
3	0	1	0	0	0	1
4	0	1	0	0	0	1
5	1	0	0	0	1	0





Transformation de la variable cible **y**





The labels are integers





The labels are integers

```
In [11]: from sklearn.preprocessing import LabelBinarizer

number_labels = [1, 2, 3, 1, 3, 2, 2, 1, 1, 3]
lb = LabelBinarizer().fit(number_labels)

print(f"Classes = {lb.classes_}")
```

```
Classes = [1 2 3]
```





```
In [12]: number_y = lb.transform(number_labels)
print(number_y)
```

```
[[1 0 0]
 [0 1 0]
 [0 0 1]
 [1 0 0]
 [0 0 1]
 [0 1 0]
 [0 1 0]
 [1 0 0]
 [1 0 0]
 [0 0 1]]
```





```
In [12]: number_y = lb.transform(number_labels)
print(number_y)
```

```
[[1 0 0]
 [0 1 0]
 [0 0 1]
 [1 0 0]
 [0 0 1]
 [0 1 0]
 [0 1 0]
 [1 0 0]
 [1 0 0]
 [0 0 1]]
```

```
In [13]: all(lb.inverse_transform(number_y) == number_labels)
```

```
Out[13]: True
```





The labels are text





The labels are text

```
In [14]: from sklearn.preprocessing import LabelEncoder

text_labels = ["alice", "bob", "alice", "alice"]
le = LabelEncoder().fit(text_labels)

print(f"Classes = {le.classes_}")

Classes = ['alice' 'bob']
```





```
In [15]: text_y = le.transform(text_labels)
print(text_y)
```

```
[0 1 0 0]
```





```
In [15]: text_y = le.transform(text_labels)
print(text_y)
```

```
[0 1 0 0]
```

```
In [16]: all(le.inverse_transform(text_y) == text_labels)
```

```
Out[16]: True
```





Partitionnement des données





Partitionnement des données

En machine learning il est **primordial** de partitionner nos données d'entrée de jeu.

On gardera la majeure partie des données $(X_{\text{train}} , y_{\text{train}})$ pour entraîner le modèle et valider l'entraînement.

La partie restante, appelée échantillon de test $(X_{\text{test}} , y_{\text{test}})$, servira uniquement à la fin de l'entraînement pour évaluer la performance du modèle.





```
In [17]: from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.20,    # train set 80%, test set 20%
    random_state=42,   # pour reproduire le partitionnement
    shuffle=True       # mélange les données avant de partitionner
)
```





Extraction de nouvelles features

De la même manière que pour les valeurs cibles, on peut transformer les valeurs en entrée pour qu'elles puissent être traitées par les algorithmes de ML.

En science, on traite essentiellement avec des données numériques donc cette partie ne sera pas abordée.

https://scikit-learn.org/stable/modules/feature_extraction.html





Données manquantes





```
[[ 6.   2.9  4.5  1.5]
 [ 5.9  3.   5.1  1.8]
 [ 4.4  3.   1.3  0.2]
 [ 5.1  3.3  nan  nan]
 [ 5.   3.5  1.6  0.6]
 [ 5.4  3.4  nan  nan]
 [ 5.7  3.8  nan  0.3]
 [ 5.6  2.5  3.9  nan]
 [ 7.7  2.6  6.9  2.3]
 [ 5.8  2.7  5.1  1.9]
 [ 6.7  3.1  5.6  2.4]
 [ 4.8  3.4  1.9  nan]
 [ 7.2  3.2  6.   1.8]
 [ 4.4  2.9  nan  nan]
 [ 6.9  3.2  5.7  2.3]
 [ 5.5  4.2  1.4  nan]
 [ 6.3  2.3  4.4  1.3]
 [ 7.   3.2  4.7  1.4]
 [ 5.8  2.7  nan  nan]
 [ 6.8  2.8  4.8  1.4]
 [ 5.4  3.9  1.7  nan]
 [ 7.6  3.   6.6  2.1]
 [ 7.7  2.8  6.7  2. ]
 [ 5.   3.3  nan  0.2]
 [ 5.9  3.   4.2  1.5]
 [ 6.1  2.8  4.   1.3]
 [ 5.   3.6  1.4  0.2]
 [ 7.4  2.8  6.1  1.9]
 [ 6.3  2.5  5.   1.9]
 [ 6.7  3.3  5.7  2.5]]
```

```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(strategy="median").fit(X_train)
X_median_imp = imp.transform(X_train)
```

Imputation

```
array([[ 6.   ,  2.9   ,  4.5   ,  1.5   ],
 [ 5.9   ,  3.     ,  5.1   ,  1.8   ],
 [ 4.4   ,  3.     ,  1.3   ,  0.2   ],
 [ 5.1   ,  3.3   ,  4.116 ,  1.462 ],
 [ 5.    ,  3.5   ,  1.6   ,  0.6   ],
 [ 5.4   ,  3.4   ,  4.116 ,  1.462 ],
 [ 5.7   ,  3.8   ,  4.116 ,  0.3   ],
 [ 5.6   ,  2.5   ,  3.9   ,  1.462 ],
 [ 7.7   ,  2.6   ,  6.9   ,  2.3   ],
 [ 5.8   ,  2.7   ,  5.1   ,  1.9   ],
 [ 6.7   ,  3.1   ,  5.6   ,  2.4   ],
 [ 4.8   ,  3.4   ,  1.9   ,  1.462 ],
 [ 7.2   ,  3.2   ,  6.    ,  1.8   ],
 [ 4.4   ,  2.9   ,  4.116 ,  1.462 ],
 [ 6.9   ,  3.2   ,  5.7   ,  2.3   ],
 [ 5.5   ,  4.2   ,  1.4   ,  1.462 ],
 [ 6.3   ,  2.3   ,  4.4   ,  1.3   ],
 [ 7.    ,  3.2   ,  4.7   ,  1.4   ],
 [ 5.8   ,  2.7   ,  4.116 ,  1.462 ],
 [ 6.8   ,  2.8   ,  4.8   ,  1.4   ],
 [ 5.4   ,  3.9   ,  1.7   ,  1.462 ],
 [ 7.6   ,  3.    ,  6.6   ,  2.1   ],
 [ 7.7   ,  2.8   ,  6.7   ,  2.    ],
 [ 5.    ,  3.3   ,  4.116 ,  0.2   ],
 [ 5.9   ,  3.    ,  4.2   ,  1.5   ],
 [ 6.1   ,  2.8   ,  4.    ,  1.3   ],
 [ 5.    ,  3.6   ,  1.4   ,  0.2   ],
 [ 7.4   ,  2.8   ,  6.1   ,  1.9   ],
 [ 6.3   ,  2.5   ,  5.    ,  1.9   ],
 [ 6.7   ,  3.3   ,  5.7   ,  2.5   ]])
```

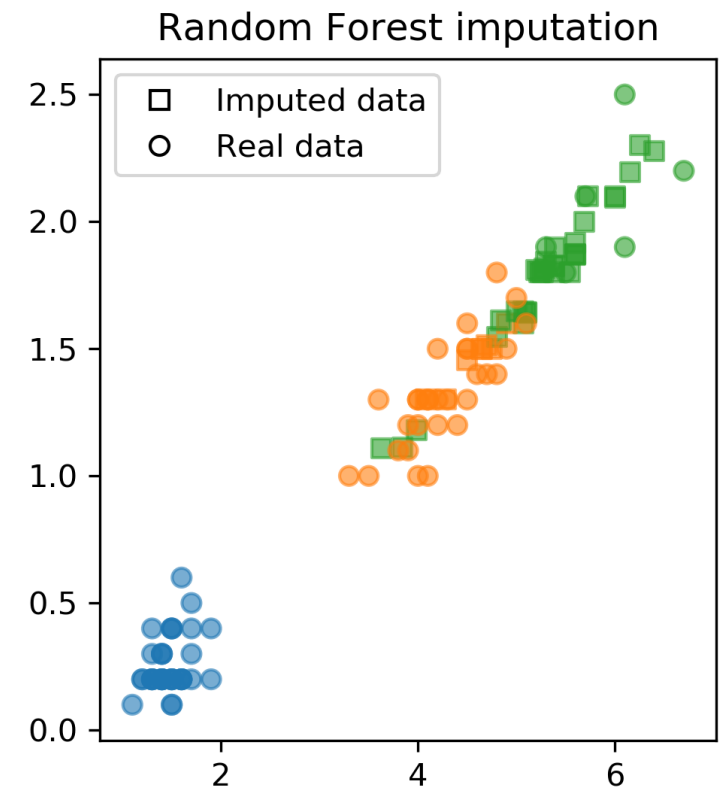
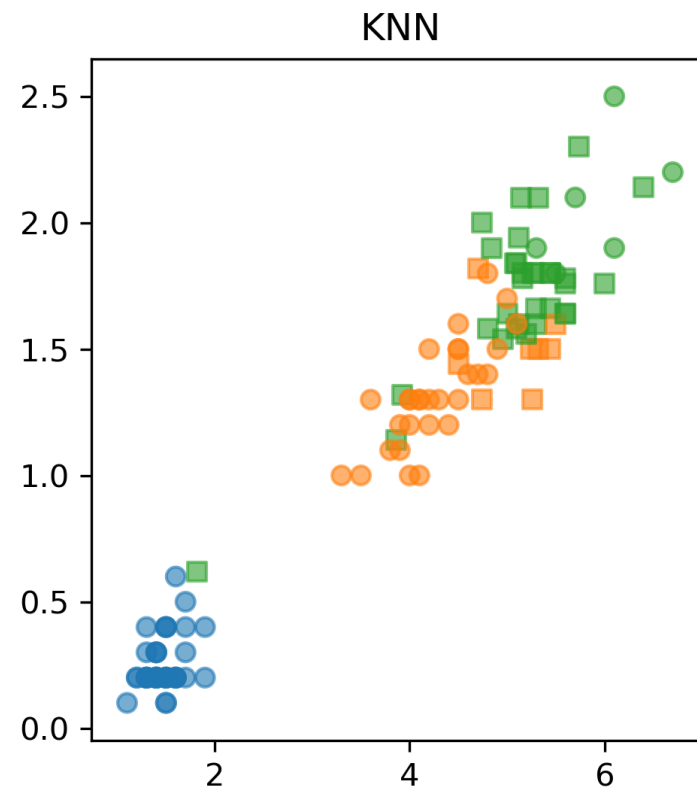
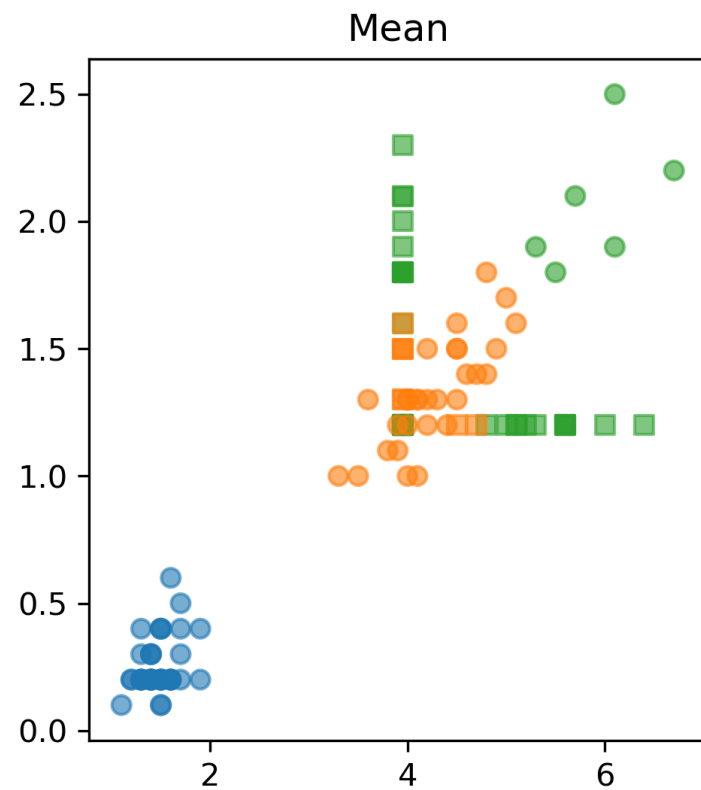




En présence de données manquantes, correspondant à des NaN dans les jeux de données, plusieurs choix sont possibles:

- supprimer toute l'entrée (ligne du tableau en question).
Souvent la baseline.
- faire de l'imputation de données (technique statistique de remplissage des données manquantes grâce aux connaissances apportées par l'ensemble des autres valeurs du tableau)







Problématique à part entière, et complexe.

Pour en savoir plus :

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





Mise à l'échelle / normalisation (rescaling)





Un grand nombre d'algorithmes de machine learning sont très sensibles aux valeurs absolues des *features* alors que l'information cruciale se situe plutôt au niveau des valeurs relatives au sein des différentes *features*.

Pour pouvoir se concentrer sur ce dernier aspect, il est **commun et fortement encouragé** de normaliser les données en entrée.





Plusieurs types de normalisation sont possibles, la plus courante étant la normalisation d'échelle standard qui retire la moyenne des données et divise par l'écart-type.





```
In [18]: from sklearn.preprocessing import StandardScaler

standard_scaler = StandardScaler()
# fit sur l'échantillon d'entraînement
X_train_scaled = standard_scaler.fit_transform(X_train)
# applique sur l'échantillon de test
X_test_scaled = standard_scaler.transform(X_test)
```





Une autre normalisation appelée `Quantile` permet de faire une transformation non linéaire des données afin d'obtenir une distribution uniforme ou bien Gaussienne.





```
In [19]: from sklearn.preprocessing import QuantileTransformer  
  
quantile_transformer_n = QuantileTransformer(output_distribution='normal')  
X_train_trans_n = quantile_transformer_n.fit_transform(X_train)  
X_test_trans_n = quantile_transformer_n.transform(X_test)
```





```
In [20]: quantile_transformer_u = QuantileTransformer(output_distribution='uniform')  
X_train_trans_u = quantile_transformer_u.fit_transform(X_train)  
X_test_trans_u = quantile_transformer_u.transform(X_test)
```

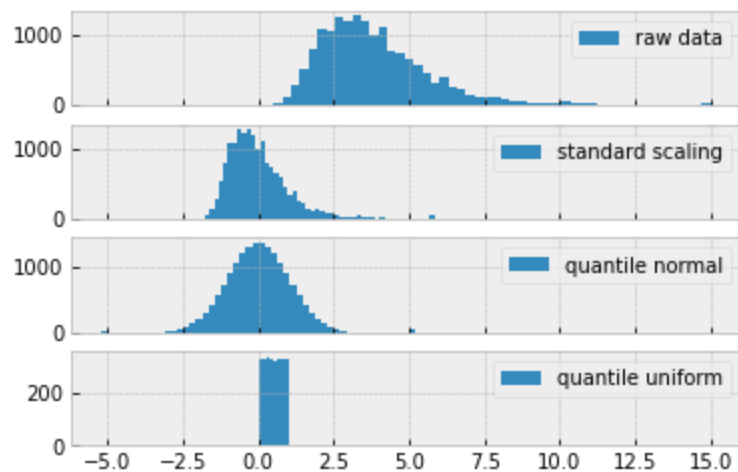




```
In [21]: fig, axes = plt.subplots(4, 1, sharex=True, )

distribs = [X_train, X_train_scaled, X_train_trans_n, X_train_trans_u]
labels = ['raw data', 'standard scaling', 'quantile normal', 'quantile uniform']

for i, data in enumerate(distribs):
    axes[i].hist(data[:, 0], bins=50, label=labels[i])
    axes[i].legend()
```





Données sparses

Pour les données sparses (beaucoup de zéros) seules les valeurs non-nulles sont stockées.

En soustraire une valeur rendra ces données "denses" et remplira la mémoire immédiatement.

Dans ce cas précis, il faut simplement normaliser les valeurs **sans les centrer** en utilisant le `MaxAbsScaler`.





Pour en savoir plus

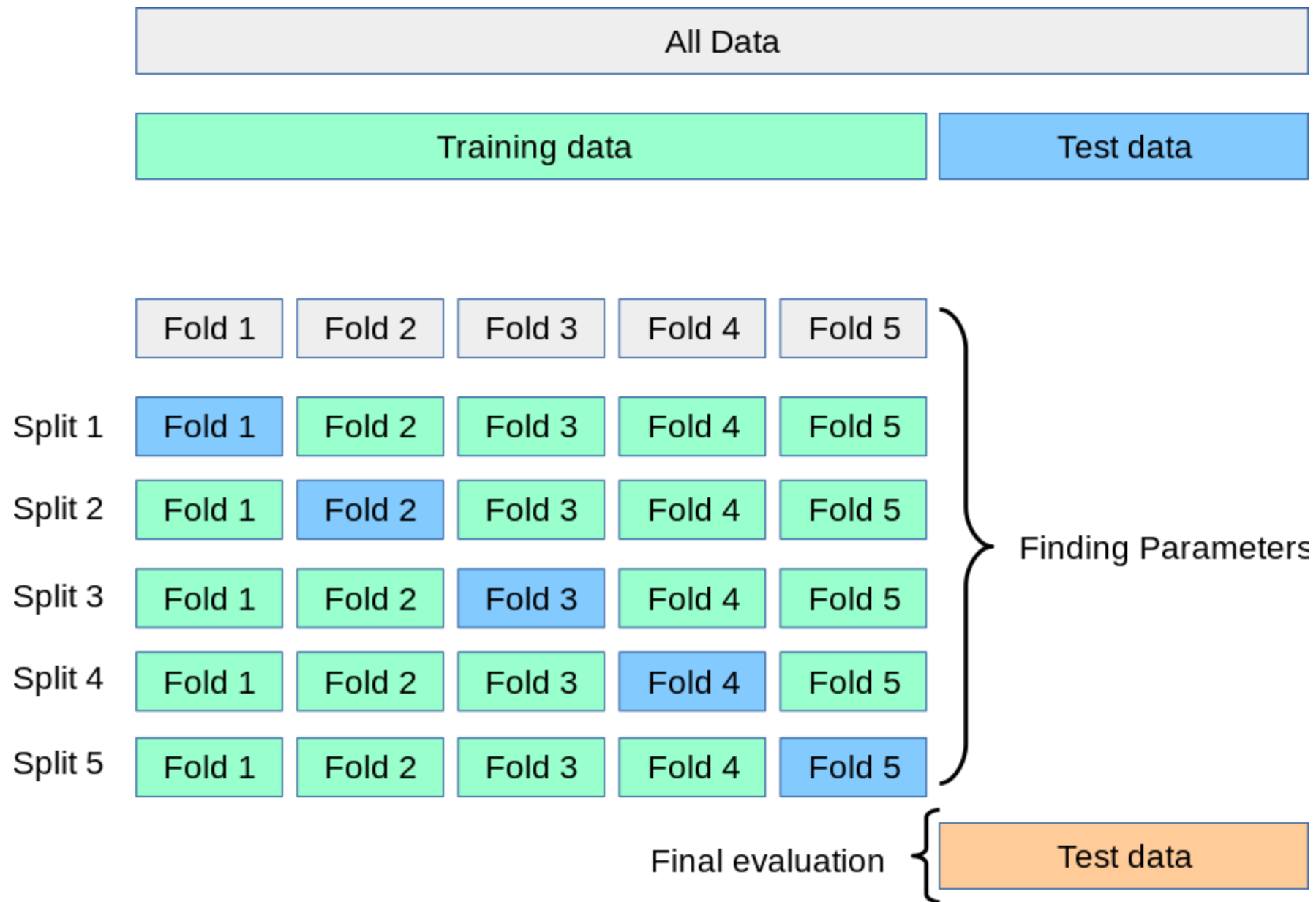
discussion sur les avantages/inconvénients des différents scalers





Optimisation des modèles avec cross-validation







```
from sklearn.model_selection import KFold

cv = KFold(n_splits=5)

for (idx_train, idx_test) in cv.split(X, y):
    X_train, y_train = X[idx_train], y[idx_train]
    X_test, y_test = X[idx_test], y[idx_test]

    model.fit(X_train, y_train)
    model.score(X_test, y_test)
```





Il existe un nombre important de méthodes pour réaliser cette validation croisée, la plupart sont décrites ici.

https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation-iterators





```
In [25]: def plot_cv_indices(cv, X, y, ax, lw=20):
          """Create a sample plot for indices of a cross-validation object."""
          import numpy as np
          import seaborn as sns
          from matplotlib.pyplot import cm
          from matplotlib.patches import Patch

          splits = list(cv.split(X=X, y=y))
          n_splits = len(splits)

          # Generate the training/testing visualizations for each CV split
          for ii, (train, test) in enumerate(splits):
              # Fill in indices with the training/test groups
              indices = np.zeros(shape=X.shape[0], dtype=np.int32)
              indices[train] = 1

              # Visualize the results
              ax.scatter(range(len(indices)), [ii + .5] * len(indices),
                          c=indices, marker='_', lw=lw, cmap=cm.coolwarm,
                          vmin=-.2, vmax=1.2)

          # Formatting
          yticklabels = list(range(n_splits))
          ax.set(yticks=np.arange(n_splits) + .5,
                  yticklabels=yticklabels,
```





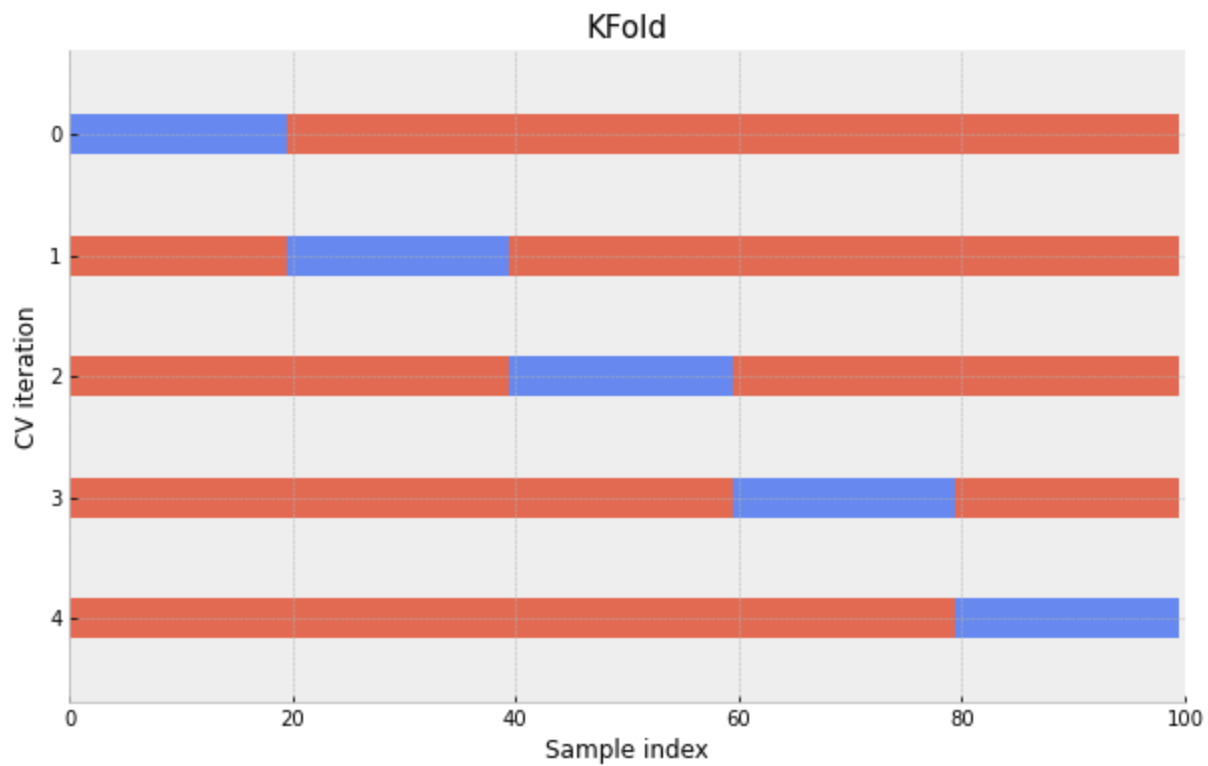
```
In [26]: from sklearn.model_selection import KFold, StratifiedKFold, ShuffleSplit
n_splits = 5

# Some random data points
n_points = 100
X = np.random.randn(n_points, 10)
y = np.zeros(n_points)
y[:20] = 1
```



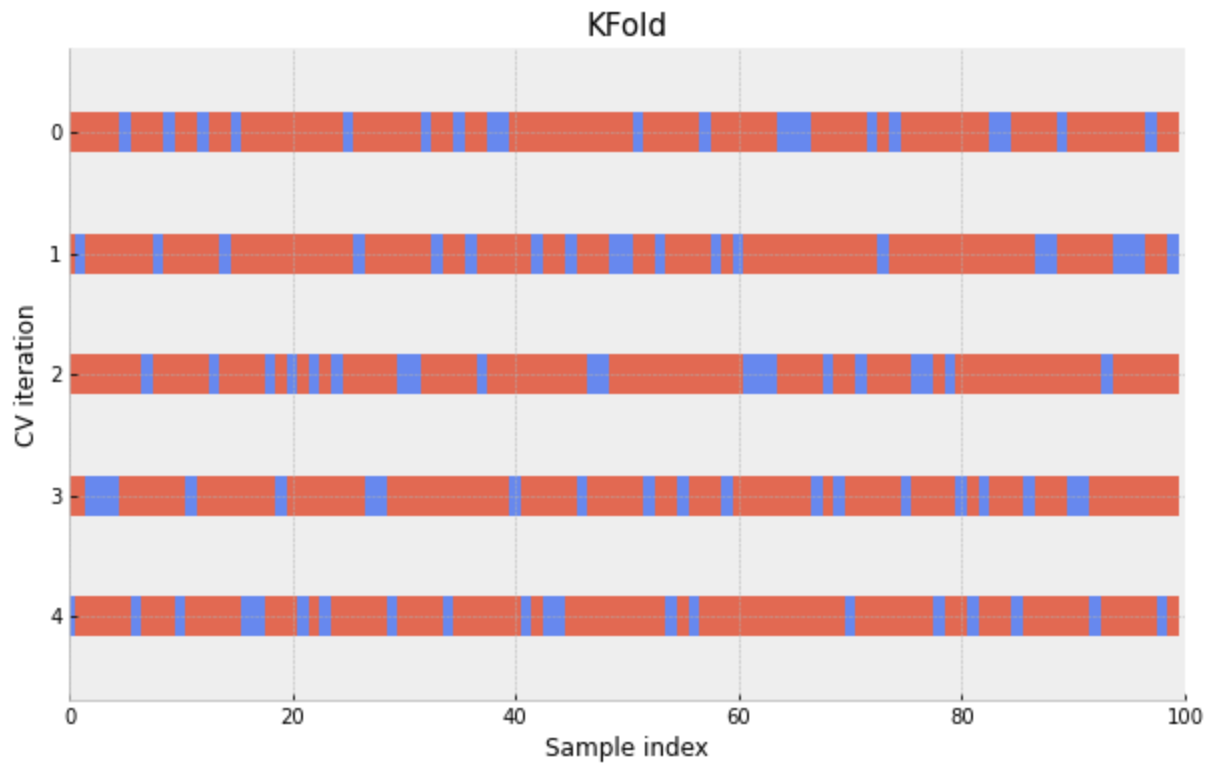


```
In [27]: fig, ax = plt.subplots(figsize=(10, 6))  
cv = KFold(n_splits, shuffle=False)  
_ = plot_cv_indices(cv, X, y, ax)
```



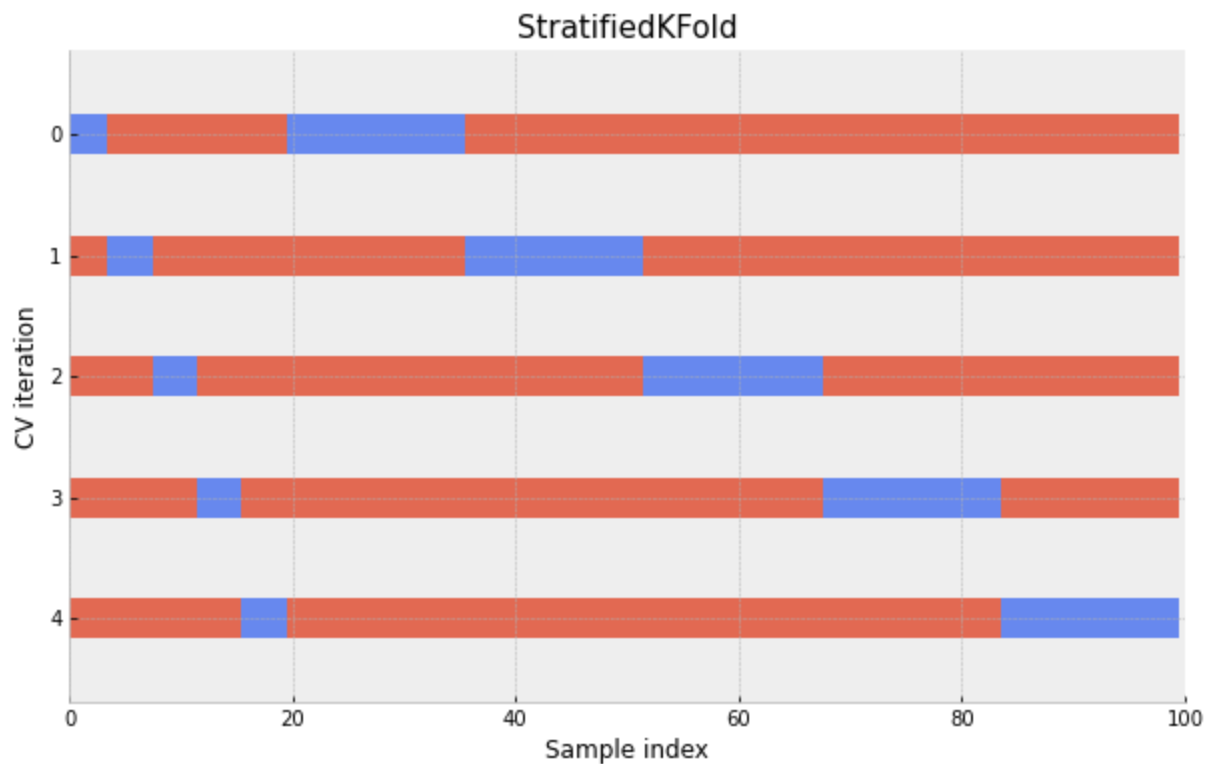


```
In [28]: fig, ax = plt.subplots(figsize=(10, 6))  
cv = KFold(n_splits, shuffle=True)  
_ = plot_cv_indices(cv, X, y, ax)
```



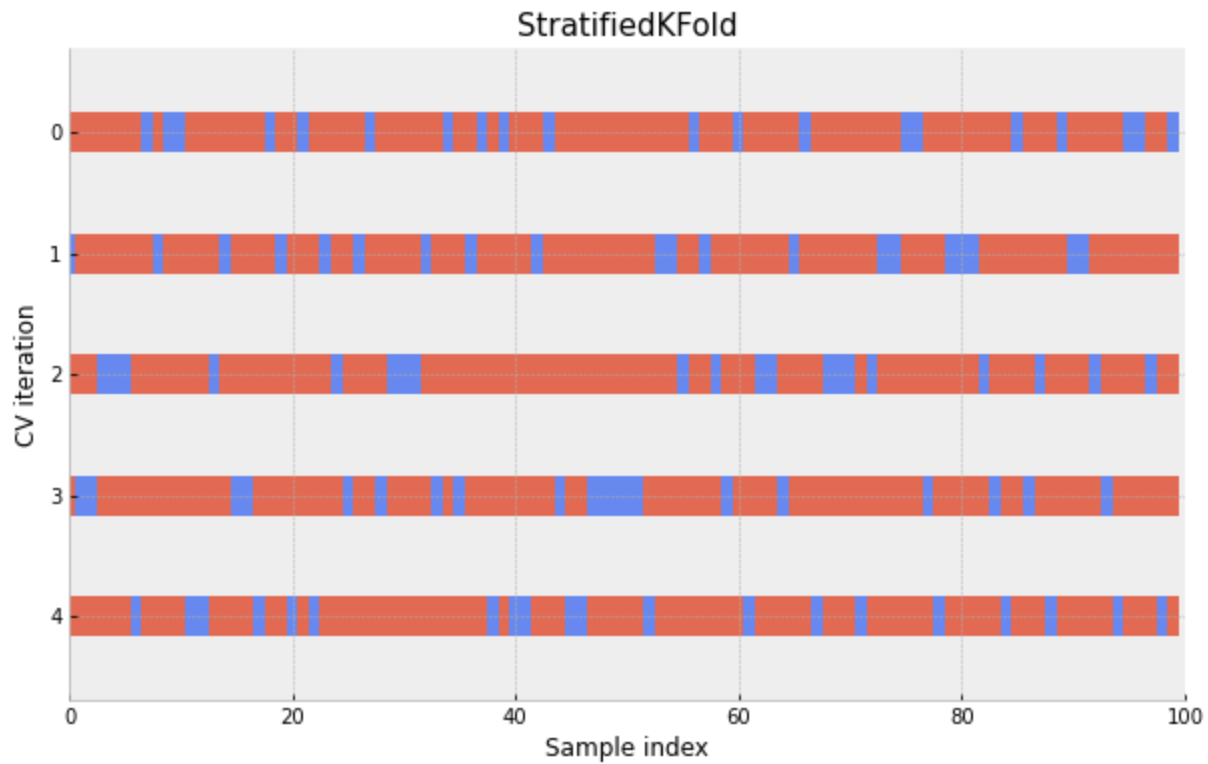


```
In [29]: fig, ax = plt.subplots(figsize=(10, 6))  
cv = StratifiedKFold(n_splits, shuffle=False)  
_ = plot_cv_indices(cv, X, y, ax)
```



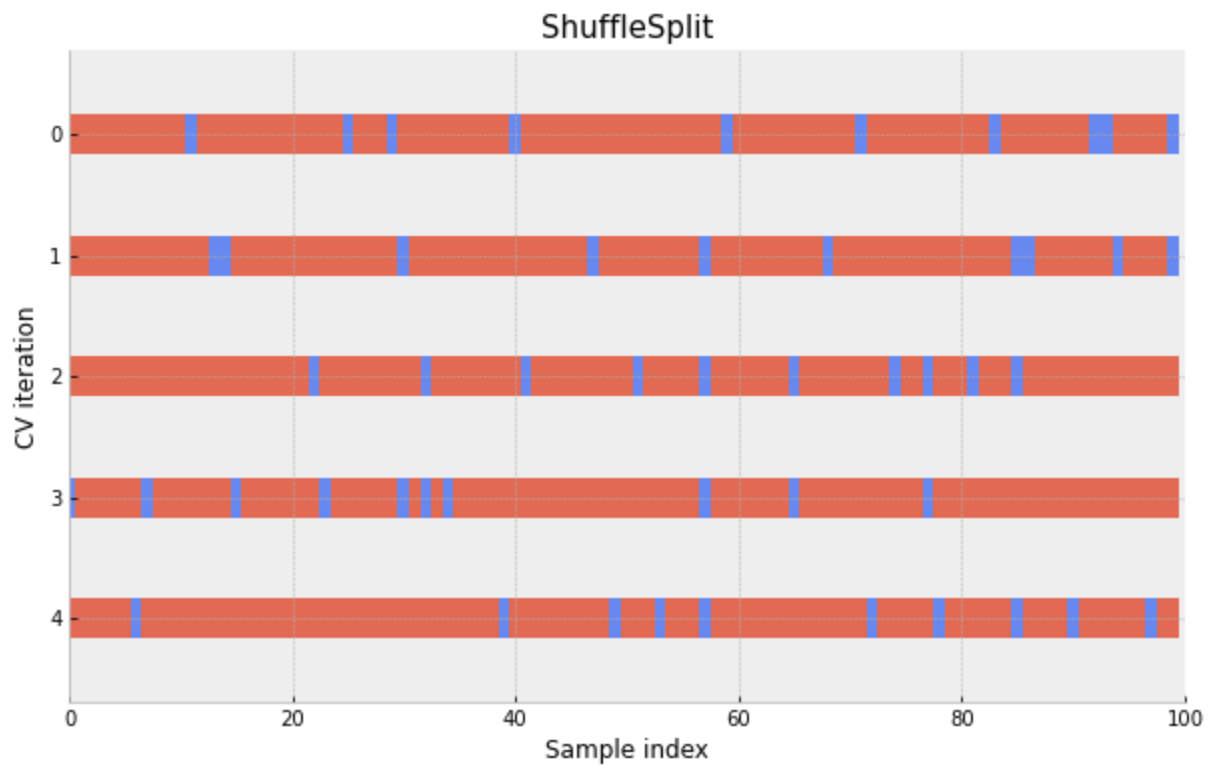


```
In [30]: fig, ax = plt.subplots(figsize=(10, 6))  
cv = StratifiedKFold(n_splits, shuffle=True)  
_ = plot_cv_indices(cv, X, y, ax)
```





```
In [31]: fig, ax = plt.subplots(figsize=(10, 6))  
cv = ShuffleSplit(n_splits)  
_ = plot_cv_indices(cv, X, y, ax)
```





Pour aller au plus court, on peut calculer directement les métriques en utilisant la cross-validation avec des fonctions pratiques comme `cross_val_score`

```
from sklearn.model_selection import cross_val_score

# On initialise un algo de classification
classifier = MyClassifier(**myargs)
# On initialise une méthode de cross-validation
cv = StratifiedKFold(n_splits=5)
# On laisse scikit-learn calculer le score du classifieur sur les données avec notre méthode de CV.
scores = cross_val_score(classifier, X, y, cv=cv, scoring="scoring_method")
```





La taille de score est égale au nombre de splits de la CV ; les méthodes de scoring étant définies sur

https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

