

Python pour les statistiques

Fiche 4 : Scikit-learn

M2 Modélisation statistique, 2023–2024

Table des matières

1	Ce qu’il faut savoir	1
1.1	Jeux de données	1
1.1.1	Jeux de données fournis	1
1.1.2	Générer des jeux de données	2
1.2	Fonctionnement général	2
1.3	Choix des hyperparamètres	4
2	Exercices	6

1 Ce qu’il faut savoir

Scikit-learn (nom du module : `sklearn`) est une librairie Python d’apprentissage automatique (*machine learning*). Elle définit des fonctions qui permettent d’attaquer de nombreux problèmes d’apprentissage supervisé ou non-supervisé : régression et classification, *clustering*, estimation de densité, détection d’anomalie, en utilisant des modèles linéaires, des arbres de décisions, des méthodes d’ensembles, des réseaux de neurones, et bien plus de méthodes. Dans son fonctionnement général, elle se rapproche de la librairie R `caret` (facilité d’utilisation pour les tâches habituelles d’apprentissage). Le *workflow* de Scikit-learn est aussi très similaire à la librairie Python Keras (une interface pour TensorFlow), qui se concentre sur les réseaux de neurones profonds – il sera donc d’autant plus facile de prendre en main cette autre librairie par la suite.

Pour avoir un aperçu de ce qui est possible avec Scikit-learn, on réfère encore et toujours à [la documentation](#). Évidemment, on ne va pas tout aborder ici. Ce document vise à montrer quelques exemples et à fournir quelques bases sur l’utilisation de la librairie. Un [tutoriel](#) officiel succinct est également disponible en ligne. Enfin, un pense-bête de Scikit-learn est sur [cette page](#), pour bien choisir son algorithme en fonction du problème.

1.1 Jeux de données

1.1.1 Jeux de données fournis

Scikit-learn fournit un [ensemble de jeux de données classiques](#) dans le sous-module `sklearn.datasets`. Ces jeux peuvent être utilisés dans de nombreux exemples de la documentation, et on les utilisera aussi dans les exercices. Ainsi pour charger le jeu de données de classification des iris, on utilise :

```
from sklearn.datasets import load_iris
iris = load_iris()
```

Notons que ces fonctions `load_...` peuvent renvoyer des objets Pandas si on passe l’argument optionnel `as_frame=True`.

1.1.2 Générer des jeux de données

On peut aussi générer son propre jeu de données, par exemple pour tester une méthode. On pourra utiliser les fonctions suivantes (importer avec `from sklearn.datasets import ...`) :

- `make_regression` pour un problème de régression.
- `make_classification` pour un problème de régression.
- `make_circles` ou `make_moons` pour un problème de clustering.

Toutes les fonctions de génération de données disponibles se trouvent [ici](#).

1.2 Fonctionnement général

Les modèles Scikit-learn sont des objets que l'on utilise de la manière suivante :

- a) **On prépare nos jeux de données.** En pratique, cette phase-là peut être longue : gestion des valeurs manquantes, des valeurs aberrantes, normalisation des données. On va ignorer les deux premières étapes dans cette introduction, et se concentrer sur la normalisation des données. La plupart des algorithmes d'apprentissage supposent que les données sont normalisées (chaque variable est sur une échelle similaire). Si on ne normalise pas les données, on risque par exemple de donner trop de poids à une variable qui a une grande variance. Quelques exemples d'algorithmes dont les résultats (où la convergence vers la solution) varient en fonction de la normalisation : ACP, régression linéaire, logistique (en particulier quand on ajoute une pénalisation), algorithmes basés sur les distances (k plus proches voisins, K-means, SVM).

Ainsi le premier composant du modèle sera en général un *scaler* :

```
from sklearn.preprocessing import StandardScaler
```

Cet outil (on va voir comment le mettre en place dans la suite) permet de centrer-réduire chacune de nos variables explicatives.

Précisions :

- Suivant les données que vous avez et les algorithmes utilisés, vous pourrez aussi choisir par exemple `MinMaxScaler` qui répartit vos données entre 0 et 1 (le minimum est envoyé sur 0, le maximum sur 1), ou `MaxAbsScaler` qui les répartit entre -1 et 1 (en divisant toutes les valeurs par le maximum des valeurs absolues).
- Pour les données non numériques, on peut les vectoriser par exemple ainsi :

```
from sklearn.preprocessing import OneHotEncoder
df = pd.DataFrame({'classe': ['riche', 'pauvre', 'moyenne',
                             'pauvre', 'moyenne', 'pauvre']},
                  index=['Fernand', 'Bernard', 'Alice',
                        'Roger', 'Fabienne', 'Josette'])
enc = OneHotEncoder().fit(df)
print(enc.categories_) # ['moyenne', 'pauvre', 'riche']
print(enc.transform(df).toarray())
# (toarray permet de récupérer un array, le tableau étant sparse par défaut.)
# [[0. 0. 1.]
#  [0. 1. 0.]
#  [1. 0. 0.]
#  [0. 1. 0.]
#  [1. 0. 0.]
#  [0. 1. 0.]]
```

Voir aussi la classe `OrdinalEncoder` pour des valeurs qui peuvent être ordonnées. C'est le cas avec l'exemple ci-dessus, on pourrait donc utiliser :

```
from sklearn.preprocessing import OrdinalEncoder
enc = OrdinalEncoder(categories=[['pauvre', 'moyenne', 'riche']]).fit(df)
print(enc.transform(df))
# [[2.]
#  [0.]
#  [1.]
#  [0.]
#  [1.]
#  [0.]]
```

- On peut aussi appliquer des **transformations différentes à chaque colonne**, de la manière suivante :

```
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import StandardScaler, MinMaxScaler, MaxAbsScaler
# On génère des données.
data = np.random.randn(3, 5)
print(data)
# [[-0.16134259  0.3933524  0.91403367 -0.4089818  1.05817181]
#  [ 0.51478034  0.56703474 -0.37515188  0.06629088 -0.61836441]
#  [ 0.36099046 -1.40790902 -1.5263465  -0.0767628  0.30350023]]

# Un transformateur adapté à chaque colonne.
scaler = make_column_transformer(
    (StandardScaler(), [0]),
    (MinMaxScaler(), [1, 2]),
    (MaxAbsScaler(), [3, 4]))
print(scaler.fit_transform(data))
# [[-1.38052519  0.91205707  1.          -1.          1.          ]
#  [ 0.955993    1.          0.47172758  0.16208761 -0.58437052]
#  [ 0.4245322   0.          0.          -0.18769245  0.28681564]]
```

- b) **On construit le modèle en fixant des paramètres.** Ils déterminent comment se fera l'apprentissage, mais pour l'instant les données sont absentes.

Exemple, pour construire un modèle de régression linéaire avec régularisation L^1 (LASSO) où l'on fixe le paramètre de régularisation à 0.025, on utilise :

```
from sklearn.linear_model import Lasso
model = Lasso(alpha=.025)
```

Dans la ligne ci-dessus, on ne précise pas que l'on voudrait normaliser nos données. Pour faire ça, on crée le modèle comme un **pipeline** (un enchaînement d'étapes) :

```
from sklearn.pipeline import make_pipeline
model = make_pipeline(StandardScaler(), Lasso(alpha=.025))
```

- c) **On divise nos données en jeu d'entraînement (*train*) et jeu de validation (*test*).** Par exemple, on va ajuster le modèle précédent sur le jeu de données "diabète".

```
from sklearn import datasets
diab = datasets.load_diabetes()
```

En examinant `diab.data.shape`, on voit que l'on dispose de 442 observations (lignes) pour 10 variables (colonnes). Pour la validation, on met de côté 25% des données aléatoirement :

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(diab.data, diab.target)
```

Note : l'argument optionnel `test_size` permet de contrôler le pourcentage de données qui se retrouvent dans l'ensemble de validation (par défaut 25%). Si le jeu de données est ordonné, on peut rendre la sélection aléatoire avec l'argument `shuffle=True`.

d) **On ajuste (*fit*) le modèle sur nos données d'entraînement :**

```
model.fit(X_train, y_train)
```

Ici, les deux arguments de `fit(X, y)` doivent être :

- `X` : une matrice de données (ou un dataframe Pandas), une ligne par observation, une colonne par variable.
- `y` : un array (ou une série Pandas) qui donne, pour chaque observation, la valeur à estimer.

Note : après avoir utilisé `fit`, le modèle a ajusté certains paramètres, que l'on peut récupérer comme des attributs de l'objet `model[-1]` (l'étape de LASSO est le dernier composant du pipeline). Par exemple, pour le LASSO, on récupère le vecteur des coefficients (β , ou plutôt w dans la doc) en tapant `model[-1].coef_` (l'underscore final est important). On peut voir tous les paramètres que l'on peut récupérer ainsi en regardant la section "Attributes" de la documentation de chaque modèle (ici pour le LASSO).

Note : à noter qu'ici le vecteur `model[-1].coef_` correspond aux coefficients estimés dans le modèle après normalisation.

e) **On "valide" le modèle.** Chaque modèle incorpore une méthode `score` qui donne un résultat entre 0 (pas bon) et 1 (bon). Pour le LASSO comme pour la régression linéaire classique, elle calcule le coefficient $R^2 = 1 - \left(\frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \right)^{-1}$, où les \hat{y}_i sont les valeurs estimées par le modèle. Le jeu de données sur lequel on calcule ce score est fourni en argument, ce qui permet de comparer le score sur le jeu d'entraînement avec le score sur le jeu de validation.

```
print(model.score(X_train, y_train)) # 0.5307137836874501
```

```
print(model.score(X_test, y_test))   # 0.4565293039024748
```

Note : on voit ici que ce n'est pas forcément très satisfaisant...

f) Si l'on est satisfait du modèle construit, **on peut enfin l'utiliser pour faire des prédictions :**

```
# Prédiction du modèle pour les trois premières observations
```

```
# du jeu de validation, différence avec les vraies valeurs.
```

```
print(model.predict(X_test[:3]) - y_test[:3])
```

```
# [ 8.05550193 -26.56615244  4.45766164] (aléatoire)
```

C'est le fonctionnement général : les algorithmes classiques sont déjà codés, il suffit de bien préparer nos données pour pouvoir les traiter avec Scikit-learn.

1.3 Choix des hyperparamètres

La question de choisir les meilleurs paramètres d'un modèle (par exemple, le α du LASSO ci-dessus) est une question primordiale en apprentissage automatique. Il y a plusieurs méthodes pour faire cela.

On va illustrer ceci sur des données de régression générées automatiquement :

```
from sklearn.datasets import make_regression
```

```
X, y, beta = make_regression(noise=10, coef=True, random_state=0)
```

Grid search. La méthode la plus basique consiste à fournir un ensemble de valeurs à explorer. Exemple d'utilisation de l'objet GridSearchCV pour cela :

```
from sklearn.model_selection import GridSearchCV
model = GridSearchCV(Lasso(), {'alpha': np.linspace(.1,2,50)}, cv=10).fit(X, y)
print(model.best_params_, model.best_score_)
# {'alpha': 0.7591836734693878} 0.9903391985293328
```

On voit que l'on doit spécifier trois choses :

- Le modèle à utiliser (ici Lasso()).
- Les paramètres à explorer, sous la forme d'un dictionnaire dont les clés correspondent aux paramètres à fixer dans le modèle.
- (optionnel) le nombre de *splits* pour la validation croisée (ici cv=10, par défaut 5). En effet, le score est ici calculé en effectuant une validation croisée.

Méthodes plus optimisées. Suivant les modèles, une méthode automatique d'optimisation des hyperparamètres est déjà implémentée dans Scikit-learn. Par exemple le LASSO propose la classe LassoCV, qui permet d'effectuer une recherche du paramètre α sans avoir à spécifier de grille de valeurs. Les solutions sont calculées successivement pour diverses valeurs de α le long du "Lasso path", ce qui peut être fait efficacement en utilisant un "warm start" : pour chaque nouvelle valeur de α , on cherche une solution en partant de la précédente solution.

Utilisation :

```
from sklearn.linear_model import LassoCV
model = LassoCV().fit(X, y)
print(model.alpha_)
# 0.7915956134197346
```

Notons que cette méthode permet de choisir facilement un "bon" coefficient α . Cependant si l'on veut valider le modèle pour cette valeur de α fixée, le comparer à d'autre, il est alors utile de pouvoir **calculer rapidement un score par validation croisée** :

```
from sklearn.model_selection import cross_val_score
model2 = Lasso(alpha=model.alpha_) # On utilise le alpha précédemment trouvé.
print(np.mean(cross_val_score(model2, X, y, cv=10)))
# 0.9896014558714242
```

Attention cependant, si l'on avait utilisé ici model à la place de model2, alors un α différent est cherché par l'algorithme, à chaque *fold* (chaque division du jeu de données) de la validation croisée.

Autre chose d'important : le jeu de données est divisé en "folds" sans randomisation (blocs de lignes qui se suivent). Pour randomiser, on peut utiliser la méthode suivante :

```
kf = KFold(shuffle=True, cv=10) # ou StratifiedKFold pour la classification.
cross_val_score(model, X, y, cv=kf)
```

NB : pour chaque méthode qui se randomise (comme KFold, souvent grâce à shuffle=True), on peut spécifier une graine de générateur de nombres pseudo-aléatoire grâce à l'argument random_state=<seed>. Ceci permet, par exemple, de comparer différents algorithmes sur le même split "aléatoire" du jeu de données.

2 Exercices

Exercice 1 : le modèle logistique.

- Charger le jeu de données “Iris” dans une ou plusieurs variables Python en utilisant les fonctions prévues dans Scikit-learn.
- Décrire le jeu de données : combien de lignes pour chaque classe, combien de variables explicatives quantitatives, de variables catégorielles ?
- Construire un modèle de **régression logistique sans régularisation**.
- Calculer son score par validation croisée sur les données Iris. À votre avis à quoi correspond le score par défaut pour un modèle de classification ? (Réponse dans la doc.)
- Remarquer dans la documentation du modèle logistique qu’il y a plusieurs manières de gérer le fait d’avoir plus de deux classes. Adapter le modèle précédent pour tester la méthode ‘**ovr**’ (*one vs. rest*), et comparer son score avec la première méthode.
- On adapte encore le modèle pour inclure de la **régularisation L^2** . Pourquoi faut-il faire maintenant bien attention à normaliser les données ?
Sélectionner par validation croisée le meilleur paramètre de régularisation C, puis comparer à nouveau le score du dernier modèle avec les précédents.
- On adapte finalement le modèle pour remplacer la régularisation par la **régularisation L^1** . Même démarche que pour la régularisation L^2 .
Attention : il faudra choisir un “solver” adapté selon la documentation, et probablement augmenter le nombre d’itérations maximales de l’algorithme d’optimisation...
- Qualitativement, que peut-on déduire de la matrice des coefficients que l’on obtient comme attribut `coefs_` de ce dernier modèle avec régularisation L^1 ?

Exercice 2 : un peu de préparation de données.

- Charger le jeu de données “Titanic” avec le code suivant (besoin d’internet pour charger la première fois).

```
from sklearn import datasets
titanic = datasets.fetch_openml(data_id=40945, as_frame=True, parser='auto')
```
- Décrire le jeu de données : combien de variables explicatives quantitatives, de variables catégorielles ? Pourcentage de survivants au global ?
Informations sur le nom de quelques features :
 - `sibsp` désigne le nombre de personnes embarquées parmi : les époux·se, les frères et sœurs.
 - `parch` désigne le nombre de personnes embarquées parmi : les parents et les enfants.
- Combien y a-t-il de données manquantes pour chaque colonne ? Nettoyer le jeu en supprimant les colonnes qui vous paraissent superflues pour le problème (qui consiste à prédire la survie des passagers). Pour les données manquantes restantes, choisissez une méthode d’imputation et appliquez-la aux données, pour avoir un jeu complet.
- À l’aide de `make_column_transformer`, transformer le jeu de données en :
 - appliquant un `OneHotEncoder(drop='first')` aux colonnes `sex` et `embarked`,
 - appliquant un `scaler` adapté aux données numériques.Quel est l’intérêt d’utiliser l’argument optionnel `drop='first'` ici ?

- e) Remarquez que le jeu transformé est un array Numpy et plus un dataframe Pandas. En utilisant `transfo.get_feature_names_out()`, construire un nouveau dataframe avec des noms de colonnes pertinents.

Les noms de colonnes obtenus sont sans doute trop longs; le plus simple pour obtenir des noms faciles à manipuler est de modifier l'argument optionnel `verbose_feature_names_out` de `make_column_transformer`.

- f) Évaluer la performance d'une régression logistique sur ces données.

Exercice 3 : une ACP et des images.

- a) Charger le jeu de données “Digits”. Il contient 1797 lignes d'images de chiffres. Ces images de 8 pixels par 8 sont encodées en un vecteur de longueur 64, et peuvent être visualisées de la manière suivante (si `X` est la matrice de données) :

```
plt.imshow(X[0].reshape((8,8)), cmap='gray')
```

Définir une fonction `compare_random(X1, X2)` qui étant donné `X1` et `X2` deux matrices de tailles identiques (et de 64 colonnes), permet de visualiser côte-à-côte les images correspondant à `X1[i]` et à `X2[i]`, où `i` est aléatoire.

- b) En utilisant la classe PCA du sous-module `sklearn.decomposition`, trouver le nombre minimum de composantes principales nécessaires pour expliquer 75% de la variance.
- c) Utiliser à nouveau la classe PCA pour projeter les données originales “Digits” sur les composantes principales trouvées. On stocke les données projetées dans une variable `X2`. On pourra utiliser `transform` et `inverse_transform`.
- d) Utiliser le code de visualisation entre le jeu de données original et `X2` pour vérifier que visuellement, on ne perd pas trop d'information.
- e) Utiliser votre modèle favori de classification pour obtenir le meilleur score possible sur le jeu original, puis comparer le même modèle, entraîné sur les données transformées par l'ACP (je dis bien **transformées et non projetées**).

Exercice 4 : Même consigne que l'exercice précédent mais avec les visages (64 par 64 pixels) du jeu de données obtenu grâce à la fonction `fetch_olivetti_faces` de `sklearn.datasets`.

Dans ce cas, le nombre de *features* étant très grand (pour nos petits ordinateurs), on voit bien l'intérêt de faire l'apprentissage sur un jeu de données de dimension réduite.

Exercice 5 : clustering simple, introduction.

- a) Charger les données du fichier `blobs_td4.txt` avec la fonction Numpy `np.loadtxt`. Visualiser les données comme un nuage de points. Combien de clusters s'attend-on trouver ?
- b) On va chercher à appliquer un algorithme de clustering simple (**K-means**). On va faire comme si l'on ne connaissait pas le bon nombre de clusters, pour tenter de le trouver, de manière “automatique”.
- c) Construire une liste de modèles `KMeans` de la manière suivante : chaque élément de la liste est un modèle construit pour un nombre de clusters k donné, pour $2 \leq k \leq 9$, ajusté sur les données.

- d) On présente une première méthode “à la main” pour choisir le bon nombre de clusters : l’inertie d’un modèle K-means (c’est ce qu’on cherche à minimiser) est calculée comme

$$\sum_{C \text{ cluster}} \sum_{x \in C} \|x - \bar{C}\|^2, \quad \text{avec} \quad \bar{C} = \frac{1}{|C|} \sum_{x \in C} x.$$

L’inertie du modèle est stockée dans l’attribut `inertia_` de la classe `KMeans`. Tracer l’inertie obtenue en fonction de k .

C’est une fonction décroissante et “le bon k ” est celui où l’on observe une cassure brusque après une décroissance forte.

- e) La méthode précédente donne une idée d’un bon k (s’il y en a), mais elle est subjective. Pour valider le choix de k , on peut aussi tracer le score de [silhouette](#) des modèles : en utilisant la fonction `silhouette_score` du module `sklearn.metrics`, tracer ces scores en fonction de k . *Il faudra utiliser la répartition en clusters déterminée par le modèle, que l’on obtient dans l’attribut `labels_`. Dans ce problème, le bon nombre de clusters devrait apparaître comme `argmax` de cette courbe.*
- f) Définir une fonction permettant de visualiser la répartition des données en clusters de différentes couleurs. Visualiser le clustering pour $k = 2, 3$ et 4 .

Exercice 6 : arbres de décision.

- a) Charger le jeu de données “vins” grâce à la fonction `load_wine` de `sklearn.datasets`.
On va utiliser ce jeu de données de classifications (3 sépages différents à retrouver en fonction de 13 features, toutes numériques – acidité, taux d’alcool, etc.) pour illustrer les méthodes basées sur les arbres.
- b) Calculer un score par validation croisée d’un arbre de décision seul (`DecisionTreeClassifier`) sur les données.
- c) Utiliser l’attribut `feature_importance_` de ce modèle pour classer les noms des features par ordre décroissant d’importance, selon ce modèle.
- d) Utiliser les fonctions `plot_tree` et `export_text` pour visualiser l’arbre de décision.
- e) On va essayer d’améliorer le score obtenu par un seul arbre en utilisant les modèles ensemblistes de *boosting* (`GradientBoostingClassifier`) et de *bagging* (`RandomForestClassifier`).
Pour chacun de ces deux modèles, calculer un score par validation croisée, puis, en ajustant finalement le modèle sur toutes les données, classer les features obtenues par ordre décroissant d’importance.
Commenter les résultats obtenus par les différents modèles.

Exercice 7 : Trouver son propre problème parmi les jeux de données “sympas” de [Kaggle](#) et proposer une solution, évaluer sa pertinence.