Apprentissage supervisé en pratique avec Scikit-Learn

Master parcours SSD - UE Apprentissage Statistique II

Pierre Mahé - bioMérieux & Université de Grenoble-Alpes

Introduction

Cours précédent : bases théoriques de l'apprentissage supervisé

- formalisation, compromis biais-variance, validation croisée, k-ppv

Ce cours : démarche générale pour réaliser une étude de ML [1]

Rien d'universel mais des questions / étapes récurrentes

Objectifs :

1. formaliser ces étapes clés
2. décrire les outils `Scikit-Learn` utiles

1. ML : Machine Learning

Introduction

Quels "outils" `Scikit-Learn` ?

- **pas** des modèles de prédiction à proprement parler
- des outils pour :
  1. optimiser leurs (hyper)paramètres
  2. estimer leurs performances
  3. mettre en forme les jeux de données
- des classes ou des fonctions
- modules clés :
  - `model_selection`
  - `metrics`
  - `preprocessing`

Introduction

Démarche générale d'une étude de ML [2] :

1. spécifier le problème et les objectifs
2. préparer les données
3. choisir le(s) modèle(s) de prédiction
4. optimiser le modèle
5. évaluer ses performances
6. (passage en production / maintenance)

2. inspiré de Géron (2017), chapitre 2.

Spécifier le problème et les objectifs

1. spécifier le problème et les objectifs
2. préparer les données
3. choisir le(s) modèle(s) de prédiction
4. optimiser le modèle
5. évaluer ses performances

Spécifier le problème

Se mettre d'accord avec le client / l'utilisateur sur :

1. les sorties attendues
2. les éventuelles contraintes

$\Rightarrow \sim$ cahier des charges

Spécifier le problème

Se mettre d'accord avec le client / l'utilisateur sur :

1. les sorties attendues
2. les éventuelles contraintes

$\Rightarrow \sim$ cahier des charges

Comprendre la nature des données

- critères de recrutement (échantillons cliniques), protocole de mesure, covariables associées, ...

Spécifier le problème

Se mettre d'accord avec le client / l'utilisateur sur :

1. les sorties attendues
2. les éventuelles contraintes

$\Rightarrow \sim$ cahier des charges

Comprendre la nature des données

- critères de recrutement (échantillons cliniques), protocole de mesure, covariables associées, ...

Pourquoi ?

- contextualiser les données et le problème
  - clé pour développer une solution pertinente
- les sorties et contraintes impactent la modélisation
  - orientent vers / disqualifient certaines méthodes

Sorties attendues : quelques questions à se poser

Niveau d'information à fournir lors de la prédiction ?

- régression ou classification ?
- on mesure parfois une réponse quantitative mais on souhaite in fine prédire une catégorie

$\Rightarrow$ "never solve a more complex problem than you have to" !

Sorties attendues : quelques questions à se poser

Niveau d'information à fournir lors de la prédiction ?

- régression ou classification ?
- on mesure parfois une réponse quantitative mais on souhaite in fine prédire une catégorie

$\Rightarrow$ "never solve a more complex problem than you have to" !

Niveau d'interprétabilité du modèle ?

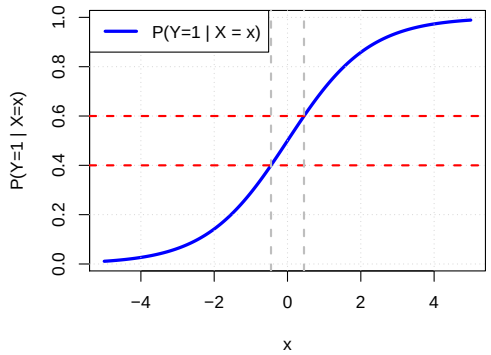
- classifieur "boîte noire" vs modèle inteprétable
- sélection de variables

$\Rightarrow$ pas toujours les mêmes méthodes

Sorties attendues : quelques questions à se poser

Critère de confiance dans la prédiction ?

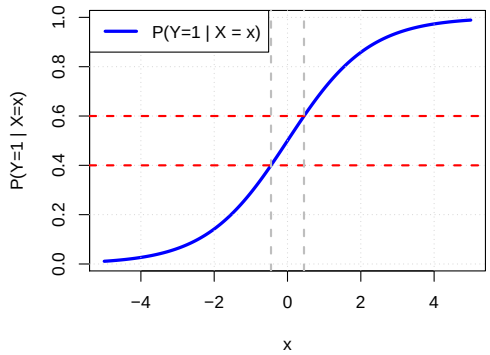
- obligatoire ?
- en termes probabilistes ?
- critère de rejet pour la prédiction ?



Sorties attendues : quelques questions à se poser

Critère de confiance dans la prédiction ?

- obligatoire ?
- en termes probabilistes ?
- critère de rejet pour la prédiction ?



Comment quantifier le coût d'une erreur ?

- classification : coût 0/1 vs coûts asymétriques ?
- régression : erreur quadratique vs erreur absolue ?
  - erreur absolue : limite influence des valeurs extrêmes

$\Rightarrow$ à prendre en compte pour l'optimisation

Contraintes éventuelles

Contraintes liées à la volumétrie des données

- nombre d'instances et/ou de features

$\Rightarrow$ impact fort sur la modélisation

Contraintes éventuelles

Contraintes liées à la volumétrie des données

- nombre d'instances et/ou de features

$\Rightarrow$ impact fort sur la modélisation

Contraintes liées à la cadence de prédiction

- problématiques temps réel

Contraintes éventuelles

Contraintes liées à la volumétrie des données

- nombre d'instances et/ou de features

$\Rightarrow$ impact fort sur la modélisation

Contraintes liées à la cadence de prédiction

- problématiques temps réel

Contraintes liées à la manière d'obtenir les données

- données arrivant en flux : apprentissage "online"
  - vs apprentissage en "batch"
- données étiqueteés à la demande : apprentissage actif
  - problématique du coût de l'annotation

Préparer les données

1. spécifier le problème et les objectifs
2. préparer les données
3. choisir le(s) modèle(s) de prédiction
4. optimiser le modèle
5. évaluer ses performances

Préparer les données

Préparer les données ?

- prise en main & analyse exploratoire
- "feature transformation"
- "feature engineering"

⇒ très dépendant de l'application et du domaine

⇒ processus exploratoire et itératif

Analyse exploratoire

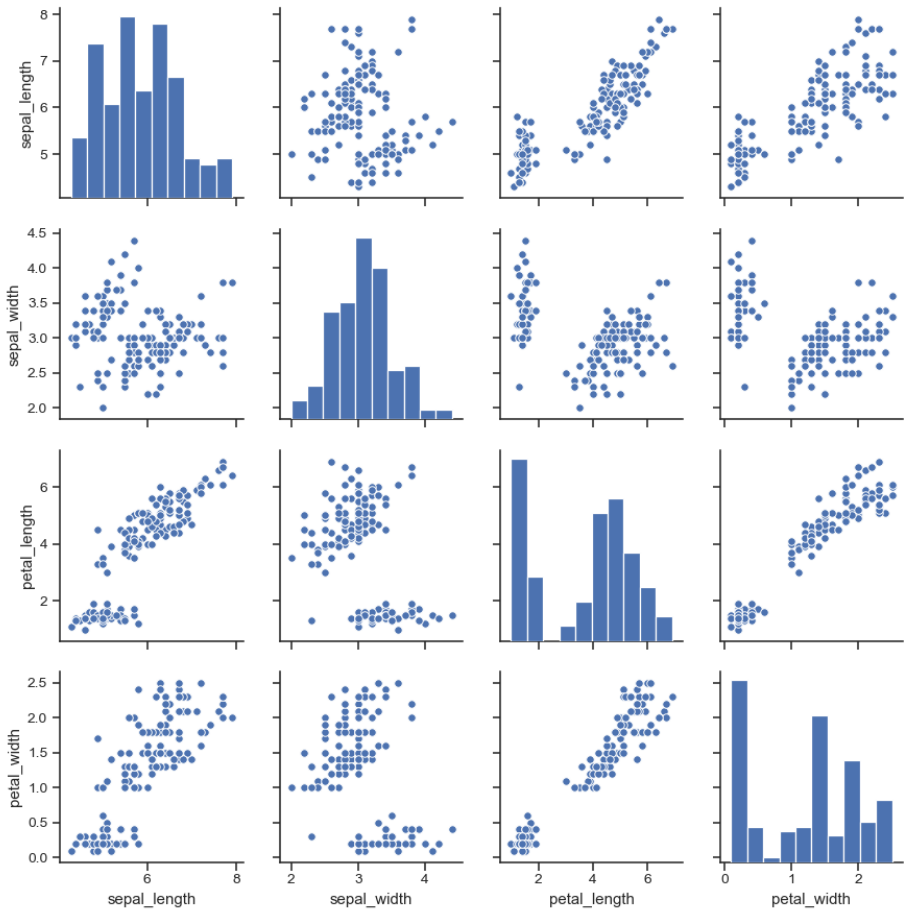
Objectifs généraux :

- s'approprier les données
  - volumétrie
  - caractère qualitatif/quantitatif des descripteurs
  - gammes dynamiques
- mesurer leur qualité
  - données manquantes
  - descripteurs non-informatifs
- détecter d'éventuels "biais"
  - sous-structures dans le jeu d'apprentissage
  - mauvaise distribution de la réponse
- détecter des points aberrants ("outliers")

Analyse exploratoire - distributions

Données en petite dimension : visualiser les distributions

- illustration : package `seaborn` et données "iris" [3] :

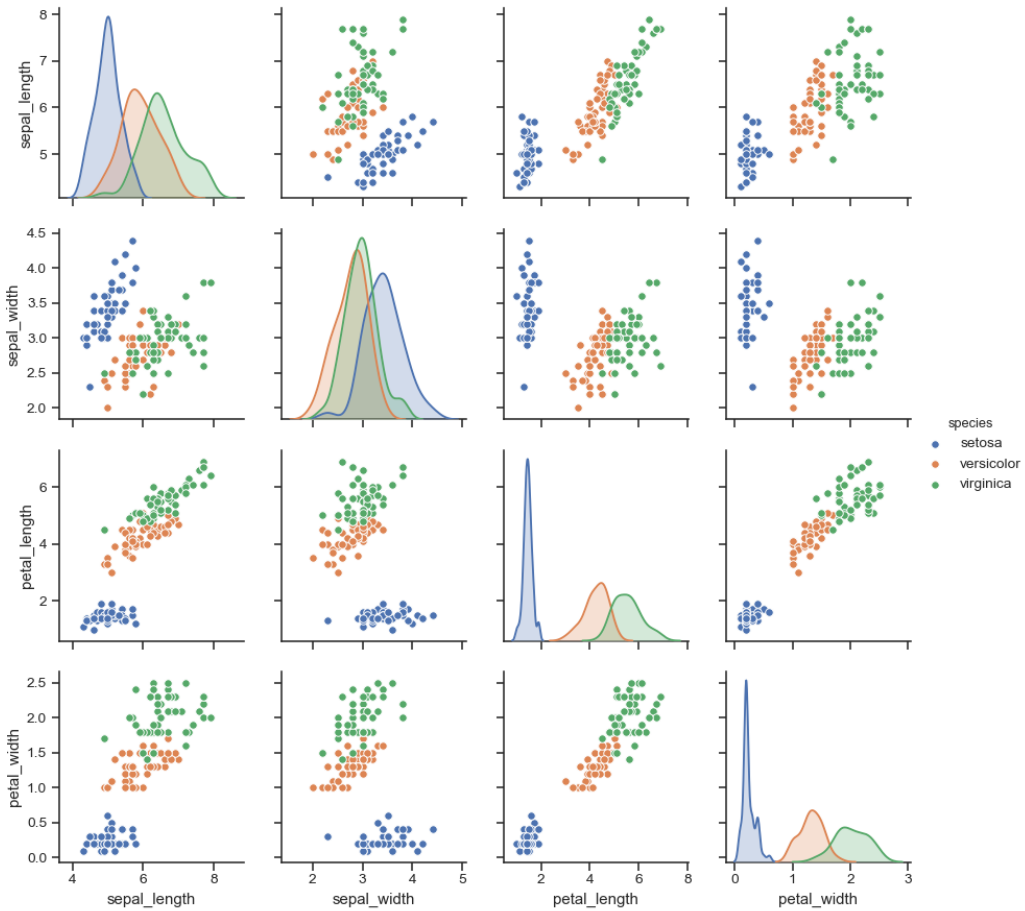


3. https://seaborn.pydata.org/generated/seaborn.pairplot.html

Analyse exploratoire - distributions

Données en petite dimension : visualiser les distributions

- illustration : package `seaborn` et données "iris" [4] :

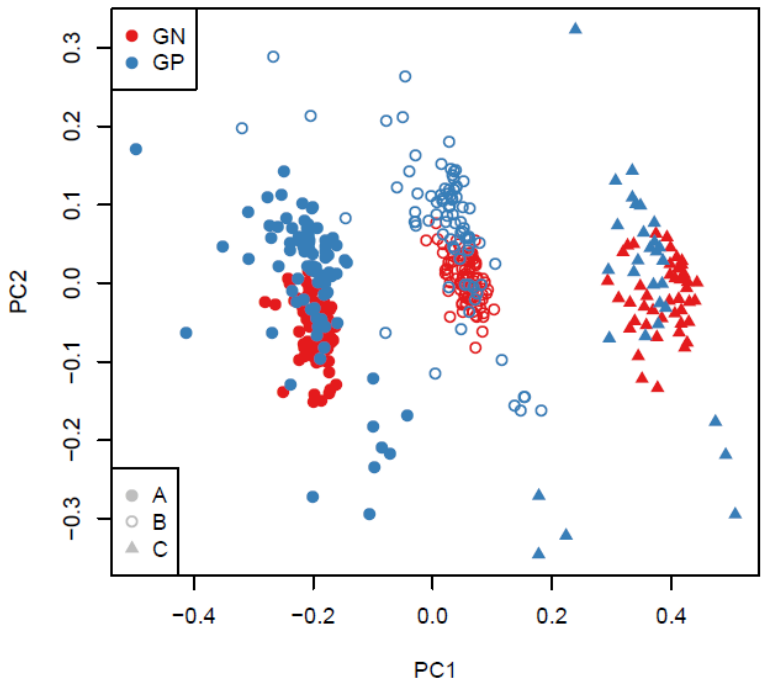


4. https://seaborn.pydata.org/generated/seaborn.pairplot.html

Analyse exploratoire & ACP

ACP : outil central de l'analyse exploratoire

- **très puissant** pour détecter des biais dans les données



⇒ **TOUJOURS** commencer par faire une ACP.

- surtout si les données sont en **haute dimension**

Features transformation

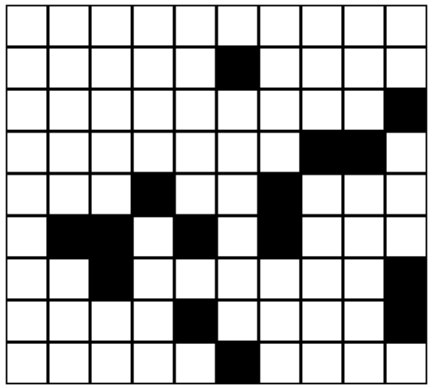
Objectifs généraux :

- gérer les données manquantes
- gérer / normaliser les descripteurs quantitatifs
- gérer / mettre en forme les descripteurs qualitatifs

Features transformation

I) Gestion des données manquantes

Les algorithmes d'apprentissage ne prennent en général pas en compte les données manquantes.



$\Rightarrow$ 3 possibilités :

1. supprimer les instances correspondantes
2. supprimer les variables correspondantes
3. imputer les données manquantes

Features transformation

I) Gestion des données manquantes

Imputation : nombreuses méthodes existantes

- hypothèses sur la cause des données manquantes
- modèles plus ou moins sophistiqués

$\Rightarrow$ solutions les plus élémentaires :

- imputation par la moyenne
- imputation par la médiane
- imputation par la valeur la plus fréquente

Features transformation

I) Gestion des données manquantes

Outil `Scikit-Learn` : classe `Imputer`

- module `preprocessing`
- stratégies de la moyenne, médiane ou plus fréquente

Utilisation :

```
# import Imputer class and instantiate
from sklearn.preprocessing import Imputer
imputer = Imputer(strategy = "median")
# fit imputer
imputer.fit(X)
# apply imputer
X_input = imputer.transform(X)
```

- options = `strategy` et `axis` (selon colonnes ou lignes)
- champ `statistics_` stocke les valeurs imputées

Features transformation

II) Normalisation des descripteurs quantitatifs

Certains algorithmes sont sensibles aux différences d'échelle ou de variance des variables

- e.g., modèles linéaires ou basés sur distance Euclidienne

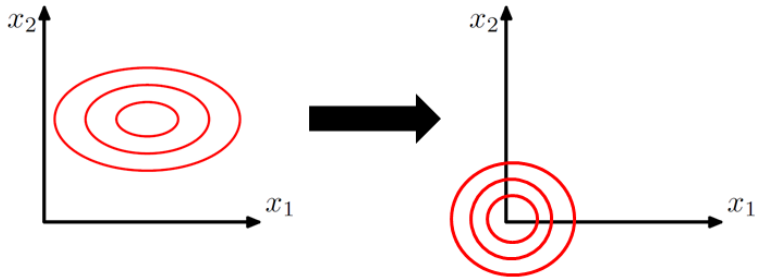
Normalisation : les ramener dans une gamme comparable

- standardisation : centrer / réduire chaque variable
- min-max scaling : ramener les gammes de valeur à [0,1]
- quantile-normalisation : distributions identiques
- ...

Features transformation

II) Normalisation des descripteurs quantitatifs

Standardisation : centrer / réduire chaque variable



Outil `Scikit-Learn` : classe `StandardScaler` :

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
Xnorm = scaler.fit_transform(X)
```

- module `preprocessing`
- champs `means_` et `vars_` stockent les valeurs calculées
- possible de centrer ou réduire uniquement

Features transformation

II) Normalisation des descripteurs quantitatifs

Autres outils `Scikit-Learn` (module `preprocessing`) :

- classe `MaxAbsScaler`
- classe `MinMaxScaler`
- classe `QuantileTransformer`
- classe `RobustScaler`
- ...

Pour plus de détails : `http://scikit-learn.org/stable/modules/preprocessing.html`

Pour des exemples : `http://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html`

Features transformation

III) Gestion des descripteurs qualitatifs

Descripteur qualitatif :

- valeurs entières entre 1 et K indiquant une catégorie
- descripteur textuel
  - e.g., {Lyon, Grenoble, Marseille, ...}

Deux manières de les interpréter :

1. comme une grandeur numérique entre 1 et K
   - hypothèse (implicite) d'une relation d'ordre
   - effet additif dans un modèle linéaire (1 coefficient)
2. comme K différentes modalités
   - encodage comme K variables différentes
   - K coefficients dans un modèle linéaire

Features transformation

III) Gestion des descripteurs qualitatifs

Encodage d'un descripteur textuel comme $\{1, ..., K\}$:

⇒ Outil `Scikit-Learn` : classe `LabelEncoder` :

```
In [11]: from sklearn.preprocessing import LabelEncoder
         encoder = LabelEncoder()
         encoder.fit_transform(["Grenoble", "Marseille", "Grenoble", "Lyon"])
Out[11]: array([0, 2, 0, 1], dtype=int64)

In [7]:  encoder.classes_
Out[7]:  array(['Grenoble', 'Lyon', 'Marseille'],
               dtype='<U9')

In [10]: encoder.inverse_transform([2,0])
Out[10]: array(['Marseille', 'Grenoble'],
               dtype='<U9')
```

⇒ module `preprocessing` ; méthodes `transform` et `inverse_transform` ; attribut `classes_`.

III) Gestion des descripteurs qualitatifs

Encodage d'un descripteur $\{1, ..., K\}$ comme K modalités :

$\Rightarrow$ approche "One Hot Encoding" :

- introduire K variables (colonnes) binaires 0/1
- si $x = k \rightarrow$ on met la k-ième colonne à 1

$$\begin{bmatrix} 1 \\ 2 \\ 1 \\ 3 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$\Rightarrow$ 3 modalités $\rightarrow$ 3 colonnes

$\Rightarrow$ chaque ligne : exactement 1 valeur égale à 1

Features transformation

III) Gestion des descripteurs qualitatifs

Encodage d'un descripteur $\{1, ..., K\}$ comme K modalités :

$\Rightarrow$ approche "One Hot Encoding" :

Outil `Scikit-Learn` : classe `OneHotEncoder` :

```
In [44]: from sklearn.preprocessing import OneHotEncoder
         oneHot = OneHotEncoder()
         x = np.array([0,2,0,1])
         X = oneHot.fit_transform(x.reshape(-1,1))

In [45]: X.toarray()

Out[45]: array([[ 1.,  0.,  0.],
                [ 0.,  0.,  1.],
                [ 1.,  0.,  0.],
                [ 0.,  1.,  0.]])
```

Features transformation

III) Gestion des descripteurs qualitatifs

Encodage d'un descripteur textuel comme K modalités :

⇒ Outil `Scikit-Learn` : classe `LabelBinarizer` :

- module `preprocessing`
- combine `LabelEncoder` et `OneHotEncoder`

```
from sklearn.preprocessing import LabelBinarizer
labBin = LabelBinarizer()
labBin.fit_transform(["Grenoble", "Marseille", "Grenoble", "Lyon"])
```

```
array([[1, 0, 0],
       [0, 0, 1],
       [1, 0, 0],
       [0, 1, 0]])
```

⇒ là aussi l'attribut `classes_` contient le nom des classes

- ici {Grenoble,Lyon,Marseille}

Features engineering

Features engineering : générer des features par combinaison des features initiaux.

Par exemple :

- transformations non-linéaires
- interactions
- ACP et méthodes de réduction de dimension

Cas particulier des données structurées (texte, images, ...)

- nécessaire pour interfacer les données et l'algorithme
- voir fin du cours

Choisir le(s) modèle(s) de prédiction

1. spécifier le problème et les objectifs
2. préparer les données
3. choisir le(s) modèle(s) de prédiction
4. optimiser le modèle
5. évaluer ses performances

Quel(s) modèle(s) choisir ?

Beaucoup de modèles....

...mais pas de modèle meilleur dans tous les cas.

$\Rightarrow$ the No Free Lunch theorem !

Quel(s) modèle(s) choisir ?

Stratégie agnostique ?

- considérer (beaucoup de) différents modèles
- que le meilleur gagne !

Quel(s) modèle(s) choisir ?

Stratégie agnostique ?

- considérer (beaucoup de) différents modèles
- que le meilleur gagne !

Néanmoins la nature du problème peut orienter :

- régression vs classification 0/1 ou multiclasse
  - nativement multiclasse ou par combinaisons
- volumétrie des données
  - large scale et/ou haute dimension
- souci d'interprétabilité
  - sélection de variables
- critère de confiance / prédiction probabiliste

Quel(s) modèle(s) choisir ?

Dans ce cours on traitera :

- de l'algorithme des k-PPV
- des méthodes à base d'arbres de décision
- des machines à vecteurs de support (SVMs)
- ~~des méthodes de régression pénalisée (Lasso)~~
- de réseaux de neurones (deep-learning)

$\Rightarrow$ on étudiera leurs propriétés au cas par cas.

Optimiser le modèle

1. spécifier le problème et les objectifs
2. préparer les données
3. choisir le(s) modèle(s) de prédiction
4. optimiser le modèle
5. évaluer ses performances

Optimiser le modèle

Paramètres vs hyper-paramètres :

- paramètres : estimés à partir des données
  - par minimisation d'un critère
- hyper-paramètres : fixés /réglés "à la main"
  - hypothèse de modélisation & complexité du modèle

Optimiser le modèle

Paramètres vs hyper-paramètres :

- paramètres : estimés à partir des données
  - par minimisation d'un critère
- hyper-paramètres : fixés /réglés "à la main"
  - hypothèse de modélisation & complexité du modèle

Exemple : régression polynomiale

- hyper-paramètre : degré d du polynome
- paramètres : poids du modèle linéaire ($\in \mathbb{R}^{d+1}$)

Optimiser le modèle

Paramètres vs hyper-paramètres :

- paramètres : estimés à partir des données
  - par minimisation d'un critère
- hyper-paramètres : fixés /réglés "à la main"
  - hypothèse de modélisation & complexité du modèle

Exemple : régression polynomiale

- hyper-paramètre : degré d du polynome
- paramètres : poids du modèle linéaire ($\in \mathbb{R}^{d+1}$)

Quizz : k-ppv ?

Optimiser le modèle

Paramètres vs hyper-paramètres :

- paramètres : estimés à partir des données
  - par minimisation d'un critère
- hyper-paramètres : fixés /réglés "à la main"
  - hypothèse de modélisation & complexité du modèle

Exemple : régression polynomiale

- hyper-paramètre : degré d du polynome
- paramètres : poids du modèle linéaire ($\in \mathbb{R}^{d+1}$)

Quizz : k-ppv ?

- hyper-paramètre : nombre k de voisins
- pas de paramètres

Optimiser le modèle

Optimiser le modèle : optimiser les hyper-paramètres

$\Rightarrow$ trouver le bon niveau de complexité du modèle

- en général difficile de choisir a priori

Optimiser le modèle

Optimiser le modèle : optimiser les hyper-paramètres

$\Rightarrow$ trouver le bon niveau de complexité du modèle

- en général difficile de choisir a priori

Deux possibilités :

1. utilisation d'un jeu de validation
2. utilisation de techniques de validation croisée

Optimiser le modèle

Optimiser le modèle : optimiser les hyper-paramètres

$\Rightarrow$ trouver le bon niveau de complexité du modèle

- en général difficile de choisir a priori

Deux possibilités :

1. utilisation d'un jeu de validation
2. utilisation de techniques de validation croisée

Au préalable : on définit un **jeu de test** pour évaluer les performances du modèle (une fois optimisé)

Optimiser le modèle & jeu de validation

Avec un jeu de validation :

1. construire les différents modèles sur le jeu train
   - e.g., en faisant varier k pour k-ppv
2. évaluer leurs performances sur le jeu de validation
3. choisir les meilleurs hyperparamètres
4. construire le modèle final sur { train + validation }
5. estimer les performance sur le jeu de test

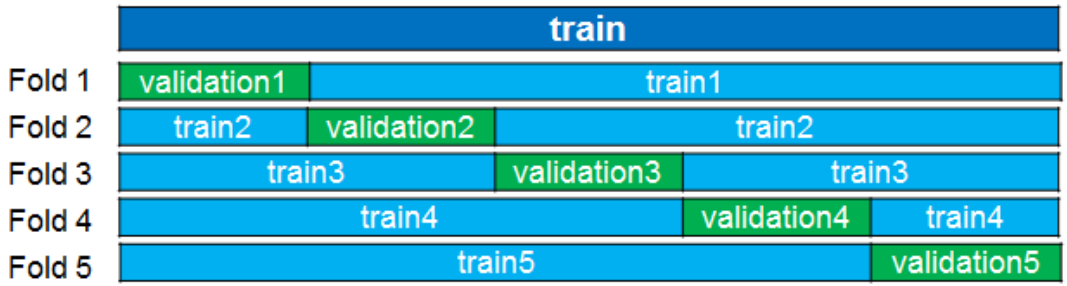
$\Rightarrow$ le cas le plus simple, pas d'outils `Scikit-Learn` dédiés

- appel des fonctions `fit` et `predict` dans une boucle

Optimiser le modèle & validation croisée

Rappel : principe de la validation croisée :

- découper le jeu d'apprentissage en K parties - les folds
  - les données de test sont toujours de côté
- pour $k = 1, ..., K$:
  - fold k = données de validation
  - autres folds = données d'apprentissage



⇒ on évalue les performances sur tout le jeu d'apprentissage

Estimation de performance par validation croisée

Estimer les performances d'un modèle par validation croisée :

1. définir les K folds de validation croisée
   - en pratique : un vecteur de longueur n avec des valeurs entre 1 et K affectant les n observations aux K folds
2. pour $k = 1$ à K :
   - 2.1 mettre de côté la k-ième fold
   - 2.2 apprendre le modèle sur les $(K-1)$ folds restantes
   - 2.3 appliquer le modèle sur les données de la k-ième fold
3. évaluer les performances du modèle

Estimation de performance par validation croisée

Estimer les performances d'un modèle par validation croisée :

1. définir les K folds de validation croisée
   - en pratique : un vecteur de longueur n avec des valeurs entre 1 et K affectant les n observations aux K folds
2. pour $k = 1$ à K :
   - 2.1 mettre de côté la k-ième fold
   - 2.2 apprendre le modèle sur les $(K-1)$ folds restantes
   - 2.3 appliquer le modèle sur les données de la k-ième fold
3. évaluer les performances du modèle

$\Rightarrow$ pour un modèle = 1 configuration de ses hyperparamètres

- e.g., k-ppv avec k=3

$\Rightarrow$ `Scikit-Learn` : `cross_val_score` & `cross_val_predict`

Estimation de performance par validation croisée

Fonction `Scikit-Learn` `cross_val_score()` :

$\Rightarrow$ calcule les performances obtenues dans les différentes folds

```
from sklearn.model_selection import cross_val_score
knn = KNeighborsClassifier(n_neighbors = 3)
cv_perf = cross_val_score(knn, X, y, cv = 10)
```

- une fonction du module `model_selection`
- paramètres (en plus de l'estimateur, de X et de y) :
  - `cv` = nombre de folds
    - par défaut les folds sont **stratifiées**
    - on peut également utiliser des folds pré-définies
  - `scoring` = critère de performance considéré
    - par défaut, la méthode `score` de l'estimateur
- sortie = vecteur de `scores` de taille K (nombre de folds)

Estimation de performance par validation croisée

Fonction `Scikit-Learn` `cross_val_predict()` :

$\Rightarrow$ calcule les prédictions obtenues par validation croisée

```
from sklearn.model_selection import cross_val_predict
knn = KNeighborsClassifier(n_neighbors = 3)
cv_preds = cross_val_predict(knn, X, y, cv = 10)
```

- même structure / utilisation que `cross_val_score()`
- sortie : vecteur de `predictions` de taille n
- permet de connaître la nature des prédictions
  - e.g., pour calculer une **matrice de confusion**

Sélection de modèle par validation croisée

La validation croisée est notamment utile pour choisir le meilleur modèle entre plusieurs modèles candidats.

- e.g., des modèles + ou - complexes

Pseudo-code :

1. Définir un ensemble de modèles candidats
   - régression polynomiale : différents degrés de polynôme
   - k-PPV : différentes valeurs de k
   - ...
2. Pour chaque modèle :
   - 2.1 Appliquer la procédure de validation croisée
   - 2.2 Mesurer les performances de prédiction
3. Choisir le meilleur modèle.

$\Rightarrow$ outil `Scikit-Learn` : classe `GridSearchCV`

Sélection de modèle par validation croisée

Classe `Scikit-Learn` `GridSearchCV` :

- évalue la performance d'un modèle par validation croisée
- pour différentes configurations de ses hyperparamètres
- et renvoie le meilleur modèle
  - appris sur **l'ensemble du jeu d'apprentissage**

Exemple : optimisation du nombre de voisins pour k-ppv

```
# define model
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
# define grid of parameters
param_grid = {'n_neighbors' : [1,3,5,7,9]}
# define grid search model
from sklearn.model_selection import GridSearchCV
grid_search = GridSearchCV(knn, param_grid, cv = 10)
# fit
grid_search.fit(X_train, y_train)
```

Sélection de modèle par validation croisée

Classe `Scikit-Learn` `GridSearchCV` :

- une classe du module `model_selection` :
  - un constructeur (`GridSearchCV()`)
  - la méthode `fit` pour optimiser le modèle
- paramètres d'entrée (pour le constructeur) :
  - l'estimateur (ici, l'objet `KNeighborsClassifier`)
  - les paramètres à considérer : un dictionnaire
    - champs = nom(s) **du ou des paramètres**
  - les options de la validation croisée (e.g. # de folds)
- en sortie :
  - la meilleure configuration : `best_params_`
    - et `best_score_`
  - le meilleur modèle : `best_estimator_`
  - le score de toutes les configurations : `cv_results_`

Sélection de modèle par validation croisée

Classe `Scikit-Learn` `GridSearchCV` :

⇒ Dernière remarque :

- `best_estimator_` contient le meilleur modèle
  - reconstruit sur l'ensemble du jeu de données
  - NB : si option `refit=True` (défaut)
- pour prédire on peut appeler sa fonction `predict`
  - i.e., `grid_search.best_estimator_.predict()`
- ...ou la fonction `predict` de l'objet `GridSearchCV`
  - i.e., `grid_search.predict()`
  - $\sim$ raccourci

(NB : valable aussi pour les fonctions `decision_function`, `predict_proba` et `predict_log_proba` si définies par l'estimateur)

Jeu de test et évaluation de performance

Une règle fondamentale pour évaluer les performances :

> les données de test ne doivent **jamais** intervenir dans la construction du modèle

Jeu de test et évaluation de performance

Une règle fondamentale pour évaluer les performances :

> les données de test ne doivent **jamais** intervenir dans la construction du modèle

Inclut notammment l'optimisation des hyperparamètres mais aussi la mise en forme des données

- e.g., standardisation des variables ou projection ACP

Jeu de test et évaluation de performance

Une règle fondamentale pour évaluer les performances :

> les données de test ne doivent **jamais** intervenir dans la construction du modèle

Inclut notammment l'optimisation des hyperparamètres mais aussi la mise en forme des données

- e.g., standardisation des variables ou projection ACP

Risque : avoir une estimation optimiste des performances

- on parle parfois de **biais de sélection**

Jeu de test et évaluation de performance

Une règle fondamentale pour évaluer les performances :

> les données de test ne doivent **jamais** intervenir dans la construction du modèle

Inclut notammment l'optimisation des hyperparamètres mais aussi la mise en forme des données

- e.g., standardisation des variables ou projection ACP

Risque : avoir une estimation optimiste des performances

- on parle parfois de **biais de sélection**

$\Rightarrow$ toujours traiter les données de test en "mode production"

- modèle figé, toutes les étapes de traitement à réaliser

Jeu de test et évaluation de performance

Exemple classique (et évident) de biais de sélection :

On dispose de p variables pour représenter nos observations et on veut construire un prédicteur n'en utilisant que $q << p$.

Erreur typique :

1. calculer la corrélation entre chaque variable et la réponse sur l'ensemble du jeu de données
2. sélectionner les q variables les plus corrélées à la réponse
3. appliquer une procédure de validation croisée pour estimer les performances du modèle.

Jeu de test et évaluation de performance

Exemple classique (et évident) de biais de sélection :

On dispose de p variables pour représenter nos observations et on veut construire un prédicteur n'en utilisant que $q << p$.

Erreur typique :

1. calculer la corrélation entre chaque variable et la réponse sur l'ensemble du jeu de données
2. sélectionner les q variables les plus corrélées à la réponse
3. appliquer une procédure de validation croisée pour estimer les performances du modèle.

$\Rightarrow$ les données de test ont servi à sélectionner les variables

$\Rightarrow$ procédure à suivre : sélectionner les variables dans les folds

Jeu de test et évaluation de performance

Exemples plus subtils de procédure (peut être) optimiste :

- standardiser les données avant découpage train/test
- projection ACP avant découpage train/test

$\Rightarrow$ les données de test contribuent à construire le modèle

Jeu de test et évaluation de performance

Exemples plus subtils de procédure (peut être) optimiste :

- standardiser les données avant découpage train/test
- projection ACP avant découpage train/test

$\Rightarrow$ les données de test contribuent à construire le modèle

Procédure correcte :

1. découper le jeu de données en train/test
2. standardiser les variables (ou les transformer via l'ACP) en utilisant uniquement le jeu d'apprentissage
3. estimer les paramètres du modèle ("fitter" le modèle)
4. transformer les données de test en appliquant les paramètres obtenus sur le jeu d'apprentissage
   - i.e., $\{\mu_i, \sigma_i\}$ ou axes ACP
5. obtenir les prédictions

Jeu de test et évaluation de performance

Pour éviter l'optimisme ⇒ procédures plus complexes

- séparation train / test de A à Z
- s'appliquent également pour la validation croisée.

⇒ outil `Scikit-Learn` : classe `Pipeline`

- du module `pipeline`
- définit une séquence d'opérations (i.e., d'estimateurs)
- s'apelle comme un estimateur "classique"
  - méthodes `fit` et `predict` (ou `transform`)
- compatible avec la classe `GridSearchCV`
  - et fonctions `cross_val_score`, etc...

"Pipelines" Scikit-Learn

Exemple : 3-ppv dans l'espace des 2 premières PCs

```
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
# define pipeline
knn_pca = Pipeline( [
 ('proj_pca', PCA(n_components = 2)),
 ('knn', KNeighborsClassifier(n_neighbors = 3)),
])
# fit & predict
knn_pca.fit(X,y)
preds = knn_pca.predict(Xtest)
```

- `fit` : calcule les axes ACPs et construit le modèle
- `predict` : projette les données de test dans l'espace ACP (déjà construit) et calcule les prédictions

"Pipelines" Scikit-Learn

Plus formellement : `Pipeline` = succession d' estimateurs :

- les premiers sont des "transformeurs"
  - i.e., ils ont les méthodes `transform` et `fit_transform`
- le dernier peut être un transformeur ou un prédicteur
  - deux types / objectifs de pipeline

$\Rightarrow$ il "expose" les mêmes méthodes que son dernier élément :

- si prédicteur : `fit` et `predict`
- si transformeur : `fit`, `transform` et `fit_transform`

"Pipelines" Scikit-Learn

Appel à la méthode `fit` :

1. appel à `fit_transform` des premiers estimateurs
2. appel à `fit` du dernier

$\Rightarrow$ construit les estimateurs (sur les données d'apprentissage)

"Pipelines" Scikit-Learn

Appel à la méthode `fit` :

1. appel à `fit_transform` des premiers estimateurs
2. appel à `fit` du dernier

⇒ construit les estimateurs (sur les données d'apprentissage)

Appel à la méthode `predict` (pipeline de prédiction) :

1. appel à `transform` des premiers estimateurs
2. appel à `predict` du dernier

⇒ applique un pipeline de prédiction

"Pipelines" Scikit-Learn

Appel à la méthode `fit` :

1. appel à `fit_transform` des premiers estimateurs
2. appel à `fit` du dernier

⇒ construit les estimateurs (sur les données d'apprentissage)

Appel à la méthode `predict` (pipeline de prédiction) :

1. appel à `transform` des premiers estimateurs
2. appel à `predict` du dernier

⇒ applique un pipeline de prédiction

Appel à la méthode `transform` (pipeline de transform.) :

1. appel à `transform` des premiers estimateurs
2. appel à `transform` du dernier

⇒ applique un pipeline de transformation

Optimisation de "pipelines" `Scikit-Learn`

La classe `Pipeline` est compatible avec `GridSearchCV`

- on peut optimiser les paramètres du pipeline comme ceux d'un estimateur individuel

Optimisation de "pipelines" `Scikit-Learn`

La classe `Pipeline` est compatible avec `GridSearchCV`

- on peut optimiser les paramètres du pipeline comme ceux d'un estimateur individuel

Il faut definir la grille de paramètres ainsi :

```
param_grid = {
    'proj_pca__n_components': [2,3,4],
    'knn__n_neighbors' : [1,3,5,7,9]
}
```

$\Rightarrow$ nom du paramètre à optimiser :

- nom du composants du pipeline (e.g., `proj_pca`)
- "`__`" (2 "underscore")
- nom du paramètre (e.g., `n_components`)

Optimisation de "pipelines" Scikit-Learn

Exemple global :

```
# define pipeline
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
knn_pca = Pipeline( [
 ('proj_pca', PCA()),
 ('knn', KNeighborsClassifier()),
])

# define grid of parameters
param_grid = {
    'proj_pca__n_components': [2,3,4],
    'knn__n_neighbors' : [1,3,5,7,9]
}

# define GridSearchCV
from sklearn.model_selection import GridSearchCV
grid_search = GridSearchCV(knn_pca, param_grid, cv = 10)

# optimize parameters
grid_search.fit(X, y)
```

Evaluer les performances

1. spécifier le problème et les objectifs
2. préparer les données
3. choisir le(s) modèle(s) de prédiction
4. optimiser le modèle
5. évaluer ses performances

Critères de performance de prédiction

Mesure de performance les plus simples (et classiques) :

- régression : erreur quadratique moyenne (MSE) :

$$\text{MSE}(f) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(x_i)\right)^2$$

(à minimiser)

- classification : taux de bonne classification (accuracy) :

$$\text{Acc}(f) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}\left(y_i = f(x_i)\right)$$

(à maximiser)

Critères de performance de prédiction

La notion de meilleur modèle peut être liée à l'application

- classification $\Rightarrow$ éviter certains types d'erreur
- e.g., contexte médical, ne pas déclarer un malade sain

Critères de performance de prédiction

La notion de meilleur modèle peut être liée à l'application

- classification ⇒ éviter certains types d'erreur
- e.g., contexte médical, ne pas déclarer un malade sain

Classification binaire : matrice de confusion

- **table de contingence** valeurs réelles / valeurs prédites

		Prédiction	
		+	-
Réalité	+	TP	FN
	-	FP	TN

- TP = True Positive
- TN = True Negative
- FP = False Positive
- FN = False Negative

⇒ permet de définir de nombreux indicateurs

Critères de performance de prédiction

Matrice de confusion et critères de performance (1/3) :

		Prédiction	
		+	-
Réalité	+	TP	FN
	-	FP	TN

- TP = True Positive
- TN = True Negative
- FP = False Positive
- FN = False Negative

- accuracy = (TP + TN)/ n
  - taux de bonne classification global
- sensibilité (sensitivity) = TP / (TP + FN)
  - taux de bonne classification des instances positives
- spécificité (specificity) = TN / (TN + FP)
  - taux de bonne classification des instances négatives

Critères de performance de prédiction

Matrice de confusion et critères de performance (2/3) :

		Prédiction	
		+	-
Réalité	+	TP	FN
	-	FP	TN

- TP = True Positive
- TN = True Negative
- FP = False Positive
- FN = False Negative

- valeur prédictive positive (VPP) = TP / (TP + FP)
  - taux d'instances positives dans les prédictions positives
- valeur prédictive négative (VPN) = TN / (TN + FN)
  - taux d'instances négatives dans les prédictions négatives

Critères de performance de prédiction

Matrice de confusion et critères de performance (3/3) :

		Prédiction +	Prédiction -
Réalité	+	TP	FN
	-	FP	TN

- TP = True Positive
- TN = True Negative
- FP = False Positive
- FN = False Negative

⇒ Indicateurs utilisés en recherche d'information :

- rappel (recall) = TP / (TP + FN)
  - proportion de "documents" retrouvés
  - taux de bonne classification des instances positives

  ⇒ équivalent à la sensibilité
- précision = TP / (TP + FP)
  - proportion de "documents" au sein des résultats
  - taux de vraies positives au sein des prédictions positives

  ⇒ équivalent à la Valeur Prédictive Positive

Critères de performance de prédiction

Le module `metrics` implémente de nombreux indicateurs

Pour la classification :

- `accuracy_score()`
- `recall_score()`
- `precision_score()`
- `f1_score()`
  - moyenne harmonique de precision et recall

Pour la regression :

- `mean_squared_error()`
- `median_absolute_error()`

$\Rightarrow$ des fonctions prenant en entrée deux vecteurs de labels

- (1) référence, et (2) prédiction

Critères de performance de prédiction

Le module `metrics` implémente aussi :

- la fonction `confusion_matrix()`
  - calcule la matrice de confusion
  - (cadre binaire ou multiclasse)
- la fonction `classification_report()`
  - génère un rapport (texte) avec plusieurs indicateurs

```
# compute confusion matrix
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y, cv_preds))
# show "classification report"
from sklearn.metrics import classification_report
print(classification_report(y, cv_preds))
```

```
[[185  27]
 [ 15 342]]
             precision    recall  f1-score   support

          0       0.93      0.87      0.90       212
          1       0.93      0.96      0.94       357

avg / total       0.93      0.93      0.93       569
```

Indicateurs de classification multiclasse

Cadre de la classification multiclasse :

- par défaut les fonctions `***_score()` calculent la valeur de l'indicateur pour chaque classe
- on peut néanmoins en obtenir une valeur unique via l'option `average` :
  - `micro` : performance globale, toutes classe confondues
  - `macro` : moyenne de la performance par classe
  - `weighted` : moyenne pondérée par effectifs des classes

$\Rightarrow$ stratégie `macro` moins sensible aux déséquilibres de classes

Remarque :

- la fonction `precision_recall_fscore_support()` calcule précision, recall, Fscore et support par classe
  - support = nombre d'instances

Courbe ROC - principe

On dispose parfois d'un classifieur fournissant un score $f(x)$

- e.g., $f(x) = P(y = +1|x)$

$\Rightarrow$ convention : score élevé = classe positive

Courbe ROC - principe

On dispose parfois d'un classifieur fournissant un score $f(x)$

- e.g., $f(x) = P(y = +1|x)$

$\Rightarrow$ convention : score élevé = classe positive

Critère de décision = seuil sur le score

- e.g., $\hat{y}(x) = +1$ si $P(y = +1|x) >$ seuil

Courbe ROC - principe

On dispose parfois d'un classifieur fournissant un score $f(x)$

- e.g., $f(x) = P(y = +1|x)$

$\Rightarrow$ convention : score élevé = classe positive

Critère de décision = seuil sur le score

- e.g., $\hat{y}(x) = +1$ si $P(y = +1|x) >$ seuil

Sensi/speci "nominales" basées sur un seuil par défaut

- e.g., $\hat{y}(x) = +1$ si $P(y = +1|x) > 0.5$

Courbe ROC - principe

Principe de la courbe ROC : faire varier le seuil par défaut pour obtenir différents compromis sensi/speci

Courbe ROC - principe

Principe de la courbe ROC : faire varier le seuil par défaut pour obtenir différents compromis sensi/speci

Exemple : réduction du seuil :

$$\hat{y}(x) = +1 \text{ si } P(y = +1|x) > 0.3$$

$\Rightarrow$ + de prédictions positives : ne peut qu'améliorer la sensi
... mais risque de faire + de faux positifs = dégrader la speci

Courbe ROC - principe

Principe de la courbe ROC : faire varier le seuil par défaut pour obtenir différents compromis sensi/speci

Exemple : réduction du seuil :

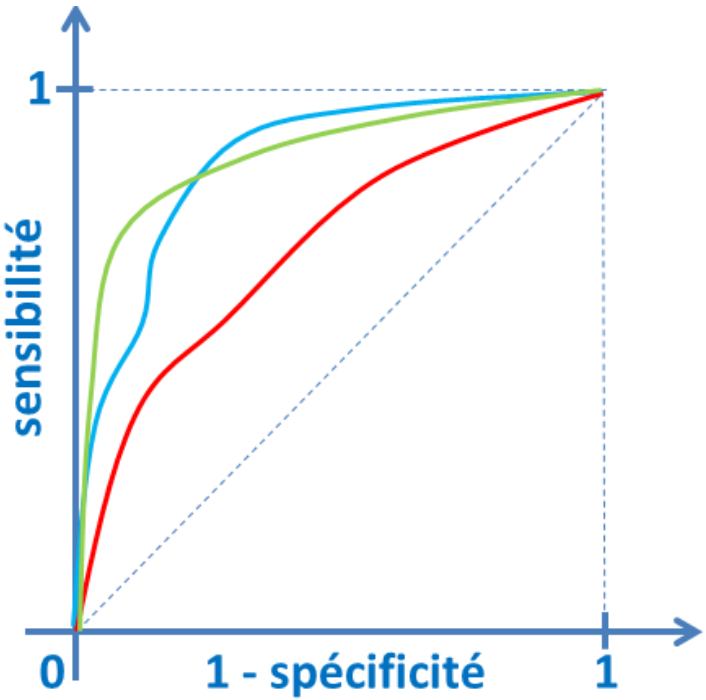
$$\hat{y}(x) = +1 \text{ si } P(y = +1|x) > 0.3$$

$\Rightarrow$ + de prédictions positives : ne peut qu'améliorer la sensi
... mais risque de faire + de faux positifs = dégrader la speci

Procédure :

1. on part à la valeur max du score
   - toutes prédictions négatives : sensi = 0 / speci = 1
2. on réduit graduellement le score
3. on termine à sa valeur minimum
   - toutes prédictions positives : sensi = 1 / speci = 0

Courbe ROC & comparaison de modèles

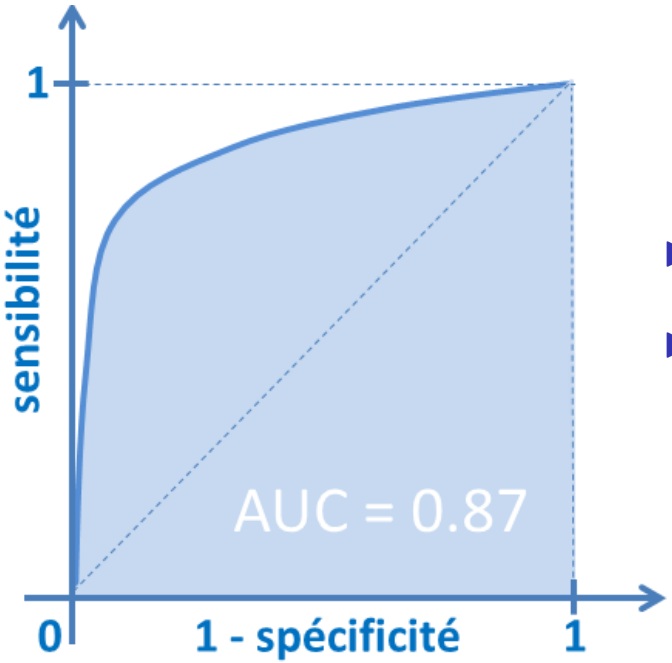


- comparaison de performances aux différents niveaux de sensi/speci : on s'affranchit du choix du seuil
- modèles bleu et vert : optimaux à différents niveaux
- modèle rouge : jamais optimal

Aire sous la courbe ROC (AUC)

Critère AUC : Area Under the (ROC) Curve

- une manière de résumer une courbe ROC



- modèle aléatoire : AUC = 0.5
- modèle parfait : AUC = 1

Interprétation :

$$\text{AUC} = P\big(f(x_1) > f(x_2) \mid y_1 = +1,\ y_2 = -1\big)$$

Courbe ROC dans `Scikit-Learn`

Le module `metrics` implémente le calcul de courbes ROC :

- la fonction `roc_curve()` calcule la courbe ROC
  - entrée : vecteurs de labels et de scores de prédiction
  - sortie : vecteurs des TPR et FPR (+ seuils)
    - **True Positive Rate** = sensibilité
    - **False Positive Rate** = 1 - spécificité

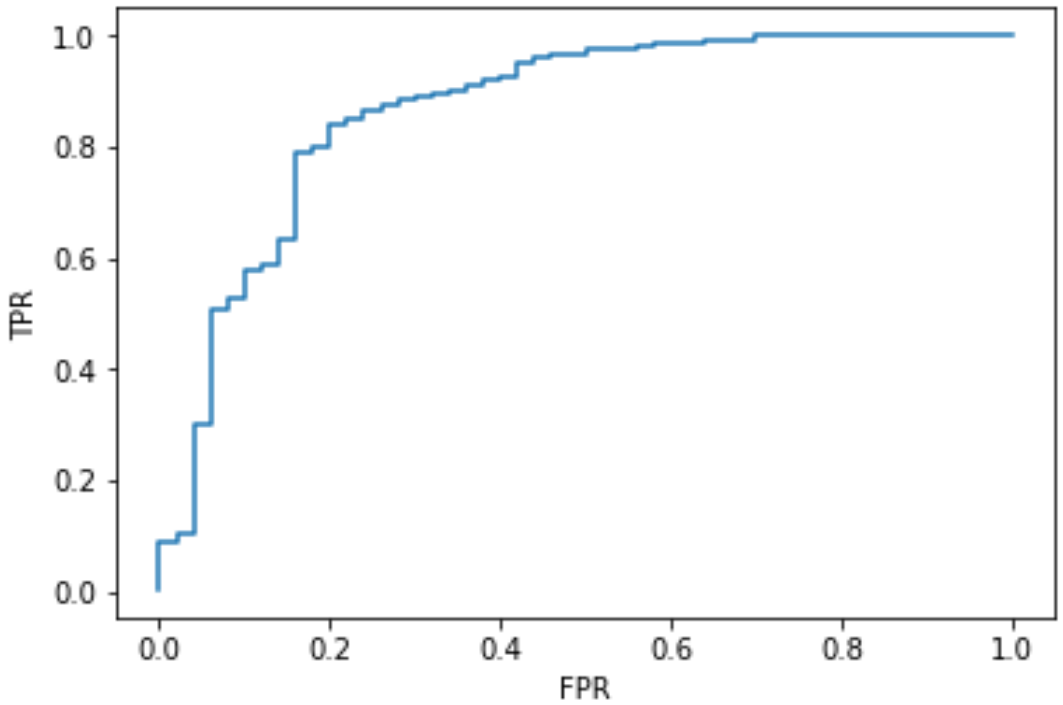
  $\Rightarrow$ courbe ROC = TPR en fonction de FPR
- la fonction `roc_auc_score()` calcule l'AUC
  - entrée : vecteurs de labels et de scores de prédictions
  - sortie : AUC

$\Rightarrow$ la fonction `precision_recall_curve()` permet de générer de la même manière une courbe précision / recall

- même philosophie qu'une courbe ROC

Courbe ROC dans Scikit-Learn

```
# build roc curve
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y, scores)
# plot
plt.plot(fpr, tpr)
plt.xlabel('FPR'); plt.ylabel('TPR'); plt.show()
# compute corresponding AUCs
from sklearn.metrics import roc_auc_score
roc_auc_score(y,scores)
```




```
0.8673599999999991
```

Analyse par "Learning Curve"

On a atteint un niveau de performance avec notre jeu de données : comment savoir si on avait assez de données ?

- pourrait-on avoir un meilleur modèle avec + de données ?

Analyse par "Learning Curve"

On a atteint un niveau de performance avec notre jeu de données : comment savoir si on avait assez de données ?

- pourrait-on avoir un meilleur modèle avec + de données ?

Analyse par Learning Curve = estimer (par validation croisée) les performances pour des tailles d'échantillon croissantes

Analyse par "Learning Curve"

On a atteint un niveau de performance avec notre jeu de données : comment savoir si on avait assez de données ?

- pourrait-on avoir un meilleur modèle avec + de données ?

Analyse par Learning Curve = estimer (par validation croisée) les performances pour des tailles d'échantillon croissantes

Algorithme :

1. définir des folds de validation croisée
2. appliquer une procédure de validation croisée où :
   - on sous-échantillonne les données d'apprentissage pour considérer des jeux de données +/- grands
   - on évalue les modèles sur les folds de tests (complètes)

$\Rightarrow$ un bon diagnostic à l'issue de l'optimisation du modèle

Analyse par "Learning Curve"

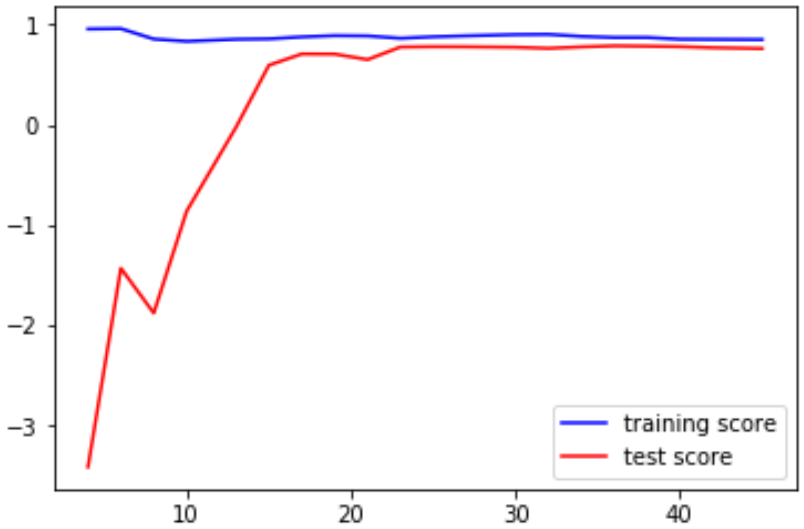
Outil `Scikit-Learn` : fonction `learning_curve()`

- module `model_selection`
- en entrée :
  - un modèle (i.e., un estimateur et ses hyperparamètres)
  - les données (i.e., X et y)
  - les paramètres de validation croisée (e.g., # de folds)
  - les tailles d'échantillon à considérer (définies comme des fractions ($\in]0, 1]$) du jeu global)
- en sortie :
  - les tailles d'échantillons considérées
  - la performance sur les données de test
  - la performance sur les données d'apprentissage
  - (performance = performance par fold)

Analyse par "Learning Curve"

Exemple :

```
from sklearn.model_selection import learning_curve
size_grid = np.linspace(0.1, 1, 20)
N, perf_train, perf_test = learning_curve(model, x, y, train_sizes = size_grid, cv = 10)
plt.plot(N, np.median(perf_train, 1), color = 'blue', label = 'training score')
plt.plot(N, np.median(perf_test, 1), color = 'red', label = 'test score')
plt.legend()
plt.show()
```



Conclusion

Remarques et conclusion

Démarche générale d'une étude de ML [5] :

1. spécifier le problème et les objectifs
2. préparer les données
3. choisir le(s) modèle(s) de prédiction
4. optimiser le modèle
5. évaluer ses performances
6. (passage en production / maintenance)

5. inspiré de Géron (2017), chapitre 2

Remarques et conclusion

De nombreux outils dans `Scikit-Learn` en plus des algorithmes d'apprentissage à proprement parler.

Modules importants :

- `preprocessing` : préparation des données
- `model_selection` : optimisation de modèle
- `metrics` : évaluation de modèle

Classes importantes :

- `GridSearchCV`
- `Pipeline`

$\Rightarrow$ classes fondamentales, utilisées largement par la suite

Remarques et conclusion

Suite du cours : méthodes d'apprentissage à proprement parler :

- arbres de décision & forêts aléatoires
- Support Vector Machines
- ~~méthodes pénalisées (Lasso)~~
- réseaux de neurones

TP :

- `cross_val_score()` & `cross_val_predict()`
- `GridSearchCV`
- `Pipeline`

Références

A. Géron. Hands-On Machine Learning with Scikit-Learn & TensorFlow. O'Reilly, 2017.