Méthodes d'agrégation avec Python

Octobre 2025

Cédric Dangeard
cedric.dangeard@orange.com

Cours 3 : Au programme

- Finir TP 2: Arbres de décision (sklearn)
- Les méthodes d'agrégation
  - Bagging (Random Forests)
  - Boosting (AdaBoost, Gradient Boosting, XGBoost)
- Sur-apprentissage et validation croisée (rappels)
- Métriques de performance
- TP : Bagging et Boosting (sklearn)

Arbres de décisions : Rappels

- Partition récursive de l'espace des features
- Critère de séparation : Gini, Entropie, SSR (somme des carrés des résidus intra-nœud)
- Arbres de classification et de régression
- Arbres peu profonds : Biais élevé, Variance faible
- Arbres profonds : Biais faible, Variance élevée

Arbres de décisions : Limites

- Variance élevé & surapprentissage pour les arbres trop profonds
- Peu performant en général

Solutions :

- Comment améliorer les performances des arbres ?

Bootstrap Aggregating

Soit X_i un ensemble de variables indépendantes de même loi dont la variance est $\sigma^2 < \infty$. La variance de la moyenne est :

$$Var(\frac{\sum_{i=1}^{n} X_i}{n}) = \frac{1}{n^2}\sum_{i=1}^{n} Var(X_i) = \frac{1}{n^2} n\sigma^2 = \frac{\sigma^2}{n}$$

Dans le cas de variable correlés d'un facteur ρ

$$Var(\frac{\sum_{i=1}^{n} X_i}{n}) = \frac{1-\rho}{n}\sigma^2 + \rho\sigma^2$$

Bootstrap Sampling

Objectif : Réduire la variance d'un estimateur

- Idée : Faire la moyenne de plusieurs estimateurs le plus indépendants possibles
- Problème : Nous n'avons qu'un seul jeu de données

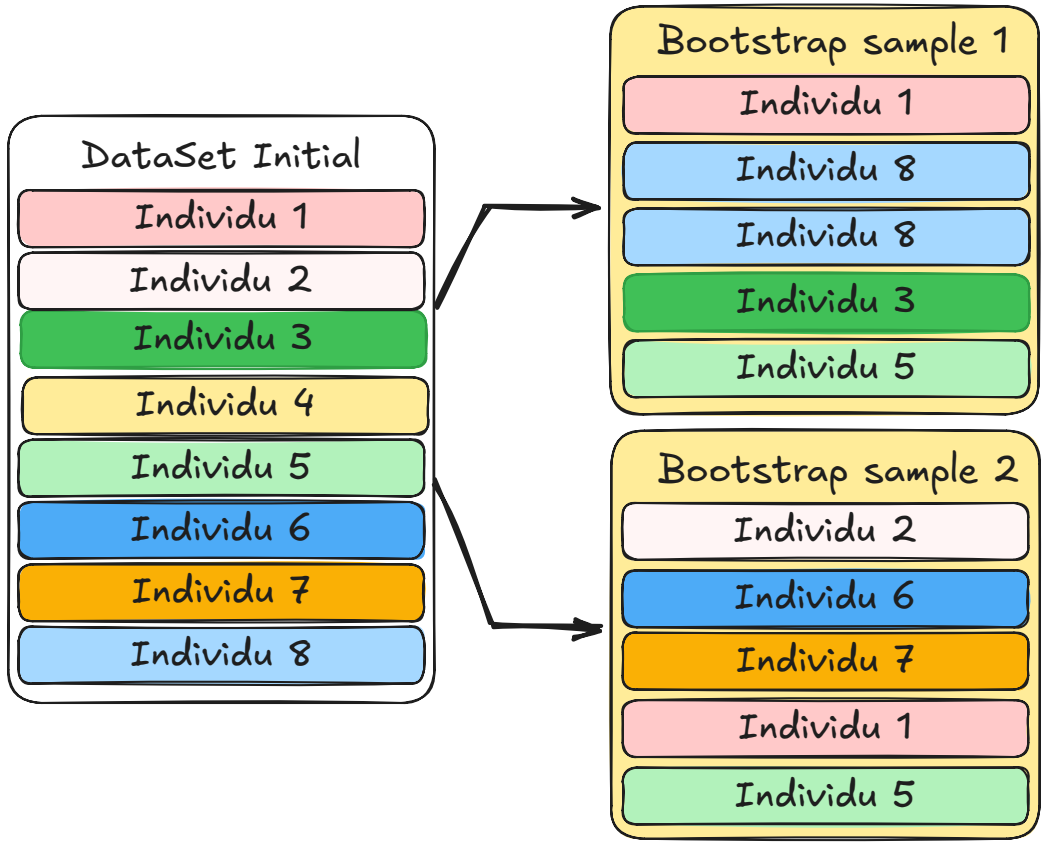
Bootstrap Sampling

Problème :

- Nous n'avons qu'un seul jeu de données

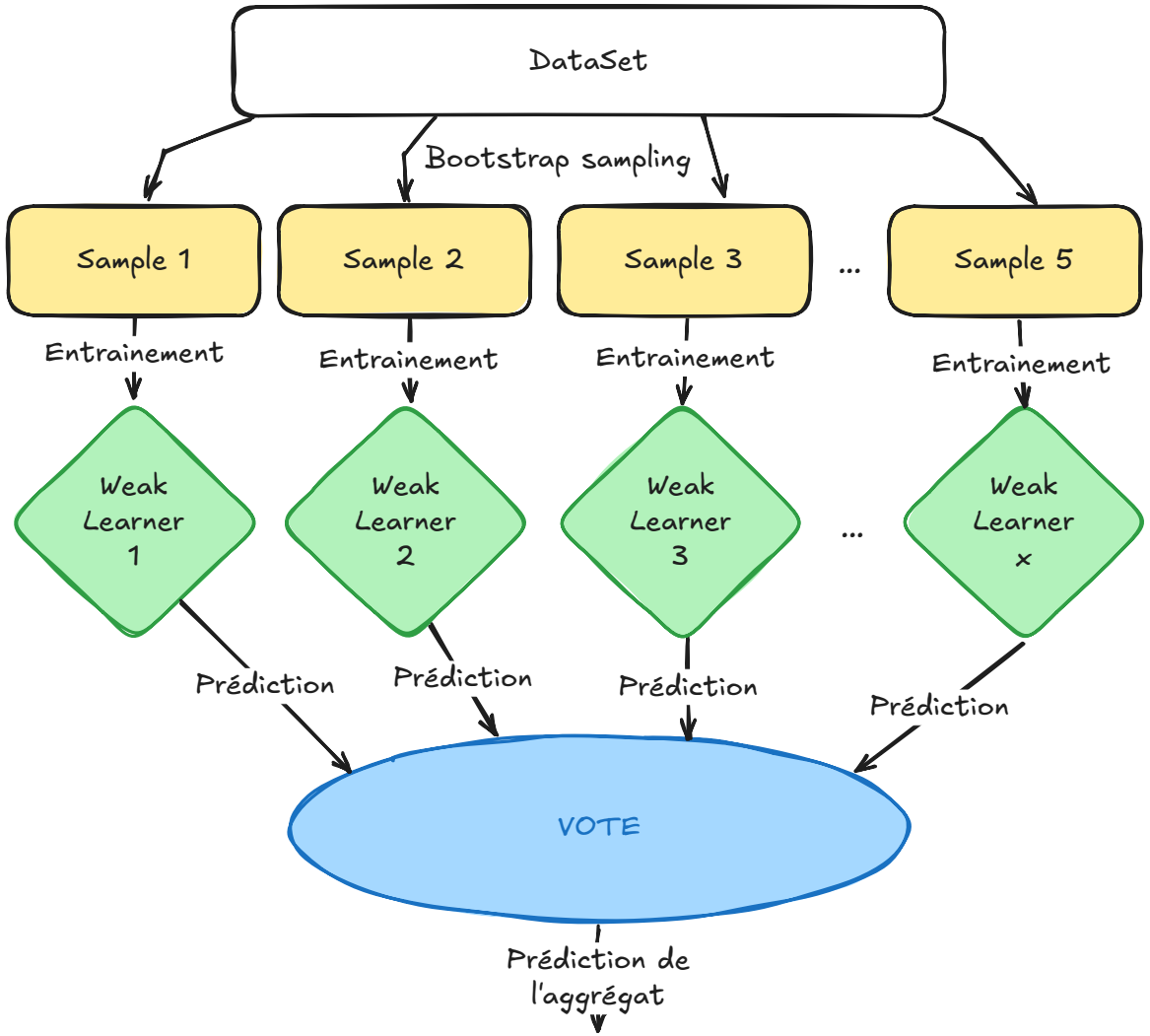
Solution :

- Utiliser notre jeu de données pour produire plusieurs jeux de données



Aggregating

- Le bagging consiste donc à prendre la moyenne d'un ensemble de modèles
- On appelle en général ces modèles *Weak Learner*
- On peut utiliser tout types de modèles, idéalement des modèles faible biais /forte variance.



Random Forests

- Le Bagging appliqué aux arbres de décisions
- Chaque arbre est construit sur un échantillon bootstrapé des données
- A chaque nœud, on ne considère qu'un sous-ensemble aléatoire des variables
- Très robuste au sur-apprentissage
- Très performant en général
- Très facile à paralléliser
- Permet d'estimer l'importance des variables (cf cours 4)

Random Forest

- **n_estimators** : *(int)* Nombre d'arbres dans la forêt
- **max_features** : *(int/float/string)* Nombre de variables à considérer à chaque nœud
- **bootstrap** : *(bool)* Tirage avec remise des données

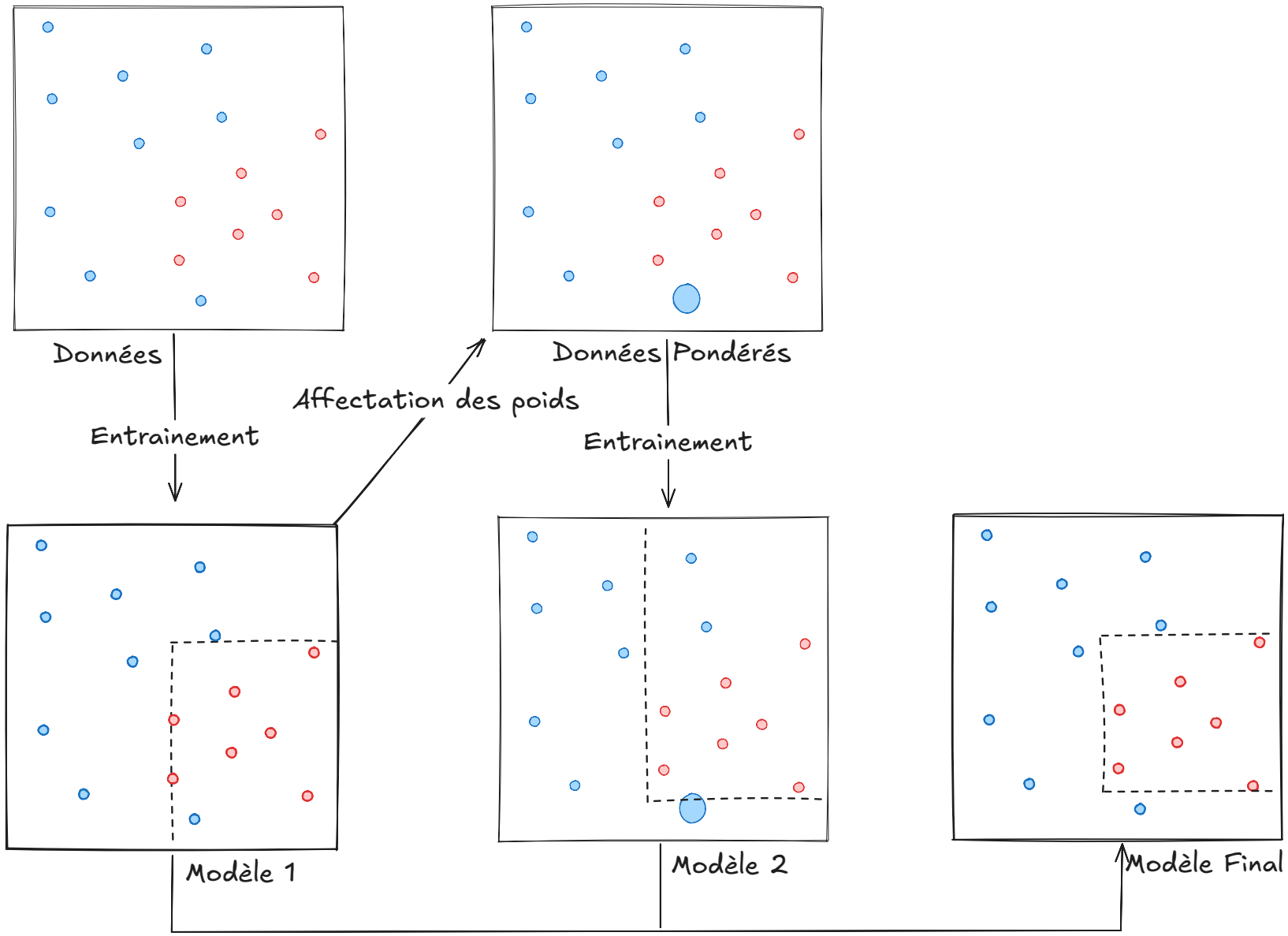
```
from sklearn.ensemble
import RandomForestClassifier

# Définition du modèle
rf = RandomForestClassifier(
    n_estimators=10,
    bootstrap=True,
    max_features='sqrt',
  n_jobs=-1
)

# Entrainement du modèle
rf.fit(X, y)
```

Boosting

A l'instar du Bagging, concept du Boosting est d'améliorer la performance de "weak Learners".

Cependant, cette fois on procède de façon itérative, chaque modèle viendra renforcer les faiblesses du modèles précédent.

Il existe plusieurs algorithmes pour les problèmes de régression et de classification. Les plus utilisés sur python sont : adaboost, gradient boosting, XGBoost


Données
Entrainement
Modèle 1
Affectation des poids
Données Pondérés
Entrainement
Modèle 2
Modèle Final

L'algorithme d'Adaptive Boosting

- On procède par itérations successives
- A chaque itération on va chercher à corriger les erreurs du modèle précédent
- On va pour cela pondérer les individus mal classés
- On va aussi pondérer les modèles en fonction de leur performance
- Le modèle final est une combinaison linéaire des modèles intermédiaires

Algorithme d'Adaboost :

h^s : modèle à l'itération s
w_i^s : poids de l'individu i à l'itération s
y_i : label de l'individu i (+1 ou -1)
x_i : features de l'individu i

- On initialise de façon **uniforme** les poids de chaque individu i : w_i^0
- Pour chaque itération s :
  - On choisis le modèle qui minimise l'erreur en fonction de w_i^s.
  - On utilise le modèle h^s pour calculer le taux d'erreur E^s

- On calcule : $\alpha^s = \frac{1}{2}\ln\left(\frac{1-E^s}{E^s}\right)$
  - α^s est une mesure de la performance du modèle h^s
  - Plus E^s est petit, plus α^s est grand
- On met à jour les poids: $w_i^{s+1} = w_i^s . \exp^{-\alpha^s y_i h^s(x_i)}$
  - Si h^s classe correctement x_i, w_i^{s+1} diminue (car $y_i h^s(x_i) > 0$)
  - Sinon w_i^{s+1} augmente
  - On normalise les poids w_i^{s+1}
- Condition d'arret : $E^s < seuil$ ou $s >$ max_iter
- Le modèle final $H(x) = signe(\sum_{i=1}^{n} \alpha^i h^i(x))$

Adaboost sur Sklearn

base_estimator : *(Estimator)*:
defaut : **DecisionTreeClassifier**
n_estimators : *(int)*:
Nombre d'iterations maximale
learning_rate : *(float)*:
Augmente la valeur de α^s lors de la mise à jour des poids
généralement entre 0.01 et 1.0, plus petit = plus d'itérations

```python
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
#Definition du modèle
ada = AdaBoostClassifier(
    base_estimator=DecisionTreeClassifier(max_depth=1),
    learning_rate=1.0,
    n_estimators=50
)

# Entrainement du modèle
ada.fit(X, y)
```

Gradient Boosting

L'idée du boosting est de prédire itérativement les résidus du modèle précedent.

On va utiliser une fonction de perte type (expodentielle, log-loss).

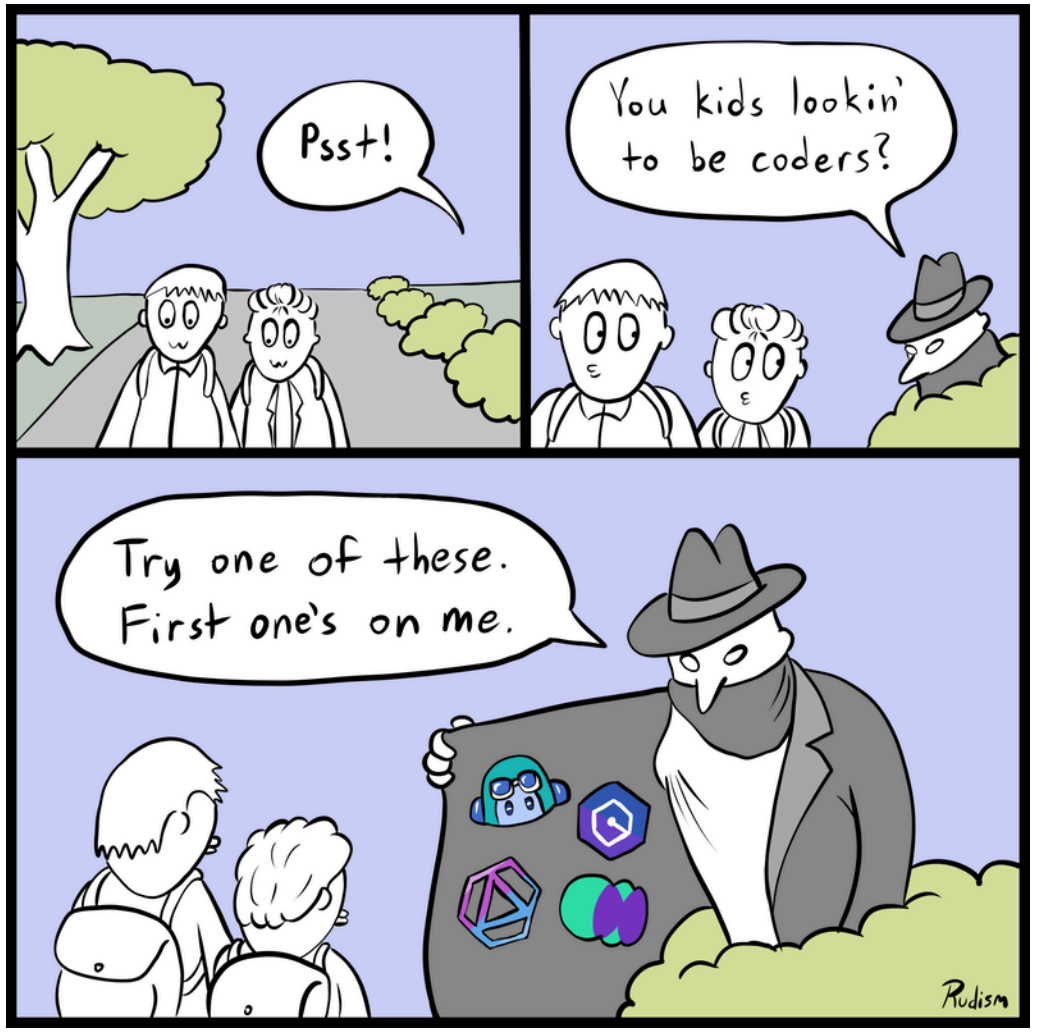
Et chercher à prédire de façon additive les résidus laissés par les modèles précédents.

Bagging sur Sklearn : BaggingClassifier

- **max_features** / **max_samples** : *(float/int)*
  Nombre d'individus et de variables tirés avec remise
- **bootstrap_features** / **bootstrap** :*(bool)*:
  Tirage avec remise ou non des variables/données.
- **oob_score** *(bool)* : Si **bootstrap** erreur out of bag ou

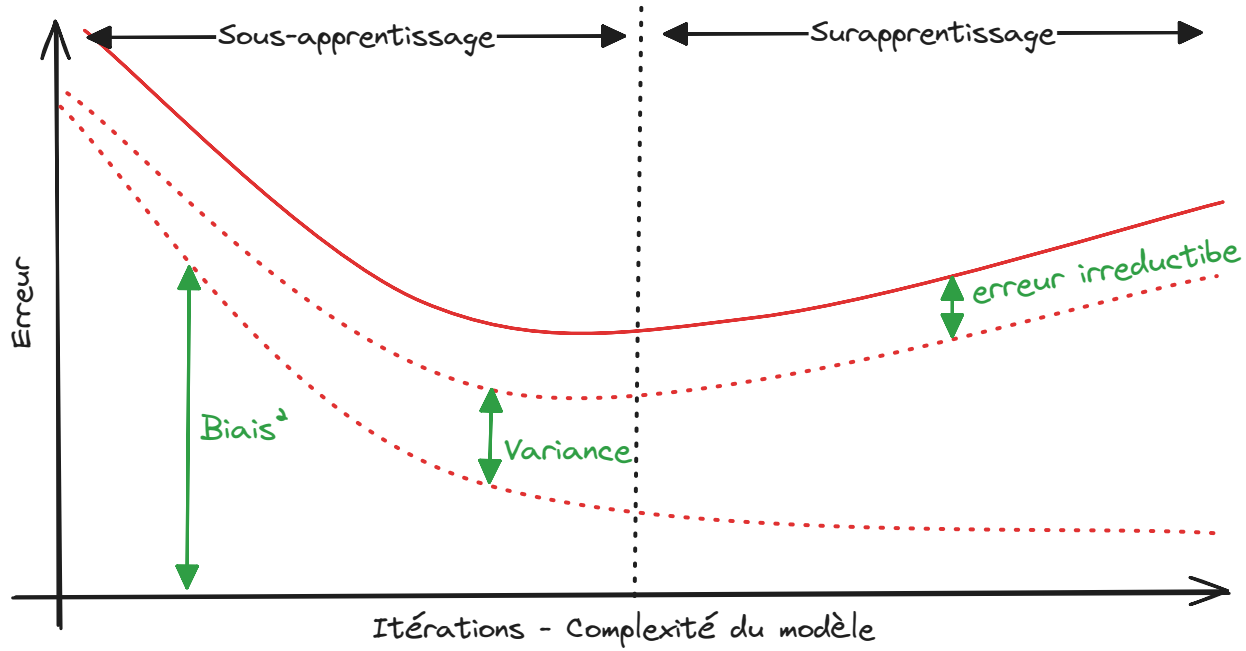
```python
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier

#Definition du modèle
bagging = BaggingClassifier(
    base_estimator=KNeighborsClassifier(),
    n_estimators=10,
    max_samples=0.5,
    max_features=0.5,
    bootstrap=True,
    bootstrap_features=False,
    oob_score=False,
    n_jobs=-1,
)

# Entrainement du modèle
bagging.fit(X, y)
```

TP : Aggrégations



Surapprentissage (rappels)

$$\vec{E}^2_{new} = \underbrace{\mathbb{E}[\hat{f}(x_*) - f(x_*)]^2}_{Biais^2} + \underbrace{\mathbb{E}[(\hat{f}(x) - \mathbb{E}[f(x)])^2]}_{Variance} + \underbrace{\sigma^2}_{\substack{erreur \\ irreductible}}$$

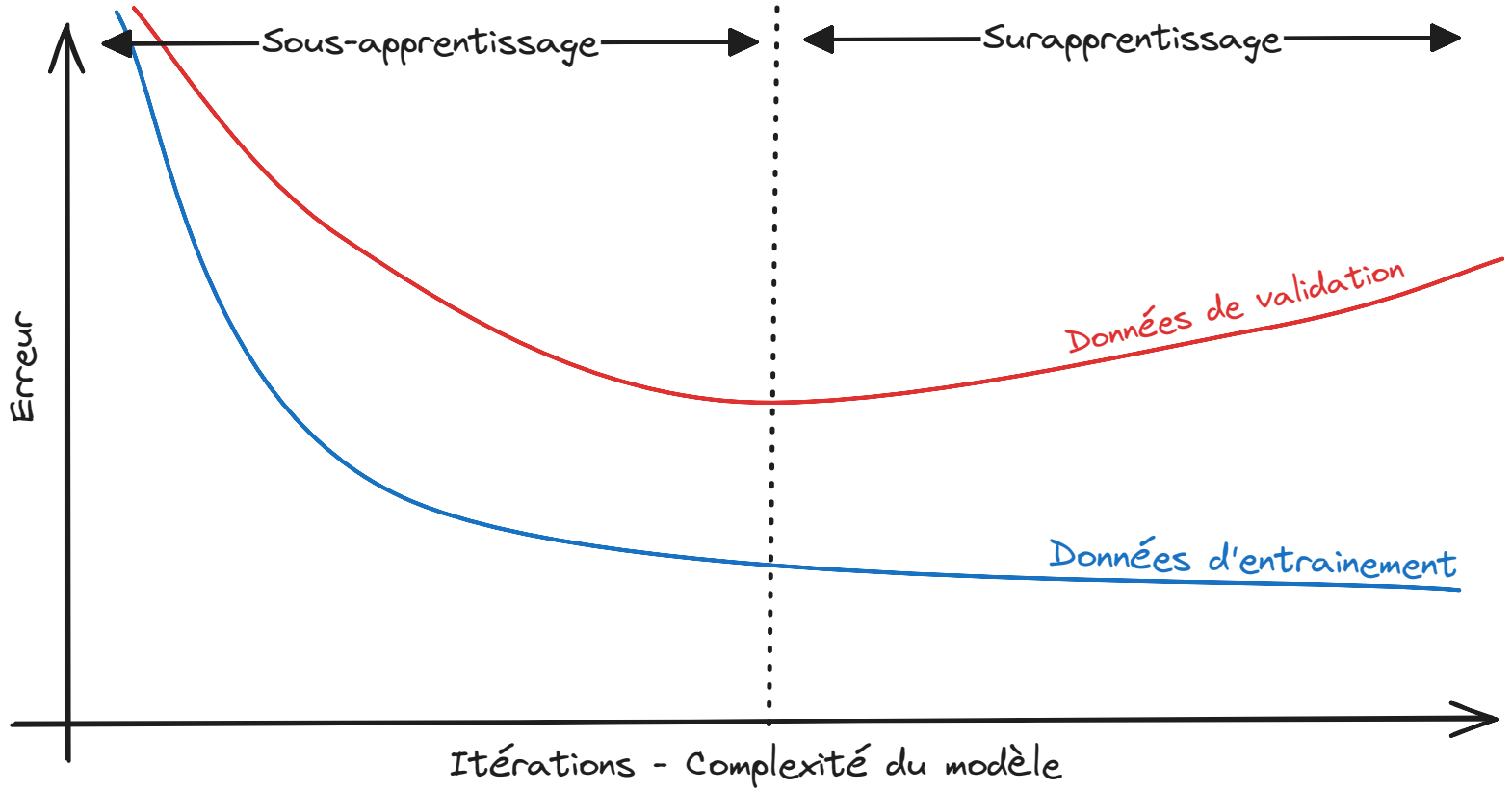


Surapprentissage

Causes possibles

- Complexité élevée du modèle
  - Trop d'Epochs
  - Hyper-Paramétrage
  - ...
- Données d’entraînement peu représentatives

Validation


Sous-apprentissage
Surapprentissage
Erreur
Données de validation
Données d'entrainement
Itérations - Complexité du modèle

Validation Hold-Out

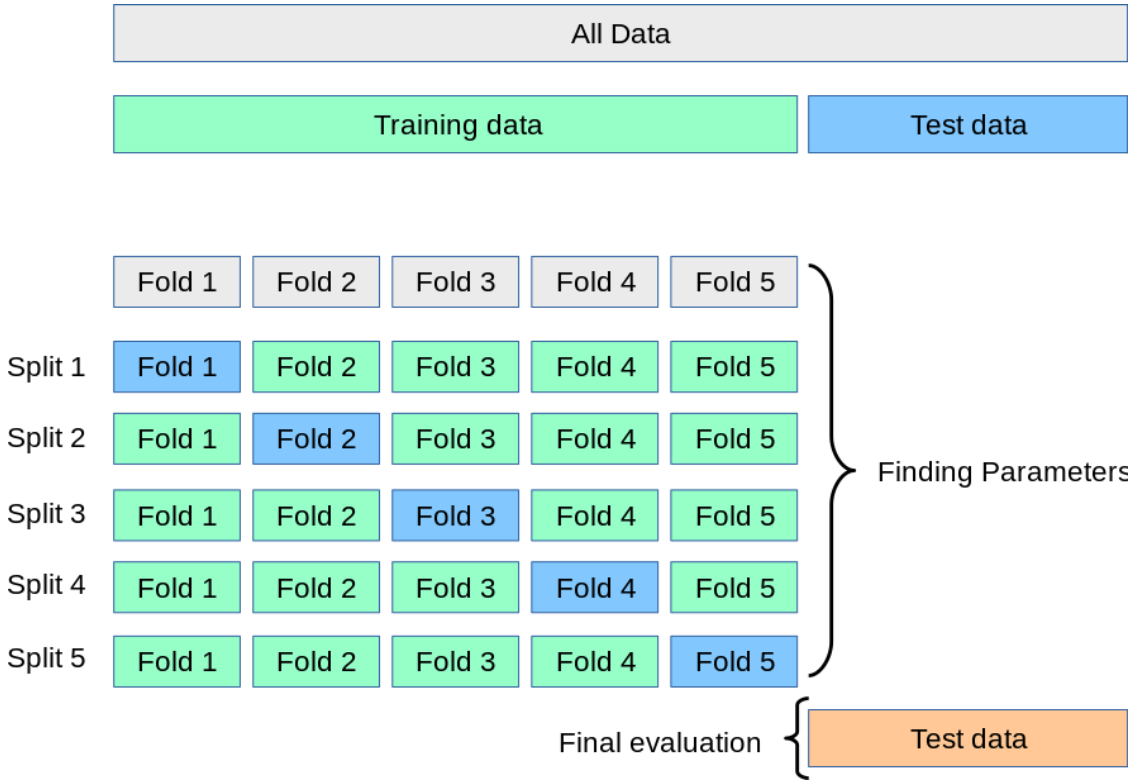
- On découpe les données en 2 ou 3
  - On entraîne sur le jeux d'entraînement
  - On choisis les hyper-paramètre à l'aide du jeu de validation
  - On valide la qualité du modèle sur le jeux de test

par exemple avec sklearn :

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,random_state=42)
```

Validation K-Fold

- Coupe du set en k blocs
- Entraînement sur $k-1$ blocs
- Evaluation sur la partie non utilisée
- Répétition de l'opération k fois
- Score: moyenne des k scores
- Cas Particulier : Leave one Out
  - (K-Fold avec $k = n$)



source : scikit-learn.org

En pratique

```
from sklearn import metrics
scores = cross_val_score(model, X, y, cv=5, scoring='f1_macro')
```

- Plus de blocs meilleurs l'approximation
- Mais plus temps de traitement est long
- k=10 en général

Indicateurs de performance

- Accuracy (Exactitude)

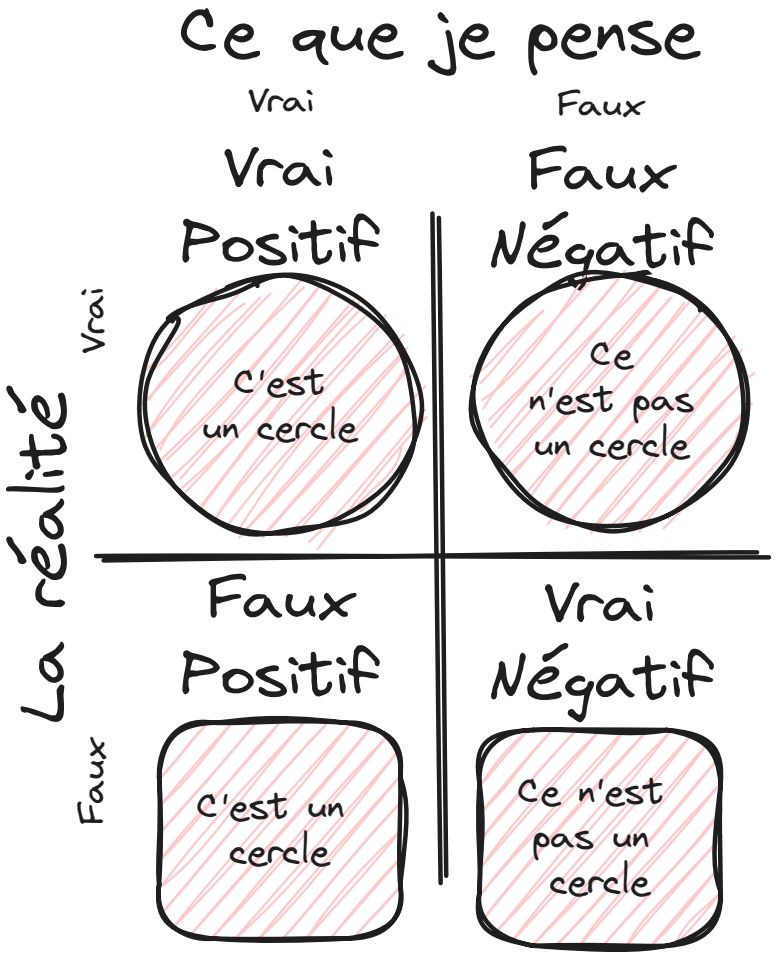
$$Acc = \frac{VP + VN}{FP + FN}$$

- Precision

$$Prec = \frac{VP}{VP + FP}$$

- Recall (Rappel)

$$Recall = \frac{VP}{VP + FN}$$



F Score

- F Score

$$F_\beta = (1 + \beta^2).\frac{Prec.Recall}{(\beta^2.Prec) + Recall}$$

Cette mesure permet de faire varier l'importance que l'on accorde à la Précision ou au Rappel.
Un β plus important $=$ avantage au Recall

En Python

```
#Données exemple
Yvrai = ['bleu', 'rouge', 'bleu', 'bleu', 'rouge', 'bleu', 'bleu', 'rouge']
Ypred = ['bleu', 'rouge', 'bleu', 'bleu', 'bleu', 'bleu', 'rouge', 'bleue']

# Métriques disponibles dans sklearn.metrics
from sklearn.metrics
    import confusion_matrix, accuracy_score, precision_score, recall_score

confusion_matrix(Yvrai, Ypred)
accuracy_score(Yvrai, Ypred)
precision_score(Yvrai, Ypred, pos_label='rouge') #attention au label
recall_score(Yvrai, Ypred, pos_label='rouge') #attention au label
```