🎓 Modèles d'Ensemble – Théorie et Pratique Complète

Random Forest, XGBoost, LightGBM, CatBoost et Stacking

📚 Table des Matières

🧩 Partie 1 – Introduction Générale : Biais, Variance et Modèles d'Ensemble

1.1 Contexte et Motivation

En apprentissage automatique, un modèle performant doit trouver un équilibre optimal entre **simplicité** (éviter le surajustement) et **expressivité** (capturer la complexité des données). Cet équilibre fondamental est formalisé par le **dilemme biais-variance**.

Les modèles d'ensemble exploitent ce dilemme en combinant plusieurs modèles faibles pour créer un modèle fort.

1.2 Décomposition Mathématique : Biais-Variance

1.2.1 Formulation Théorique

Soit une variable cible y et une prédiction $\hat{f}(x)$ issue d'un modèle entraîné sur différents échantillons.

L'**erreur quadratique moyenne (MSE)** attendue se décompose en trois composantes :

$$\mathbb{E}[(y - \hat{f}(x))^2] = \underbrace{(\text{Bias}[\hat{f}(x)])^2}_{\text{Erreur systématique}} + \underbrace{\text{Var}[\hat{f}(x)]}_{\text{Sensibilité aux données}} + \underbrace{\sigma^2}_{\text{Bruit irréductible}}$$

1.2.2 Définitions Formelles

Biais : Erreur systématique du modèle par rapport à la vraie fonction cible $f(x)$

$$\text{Bias}[\hat{f}(x)] = \mathbb{E}[\hat{f}(x)] - f(x)$$

- **Biais élevé** → Modèle trop simple → **Underfitting**

- Le modèle ne capture pas les patterns complexes des données

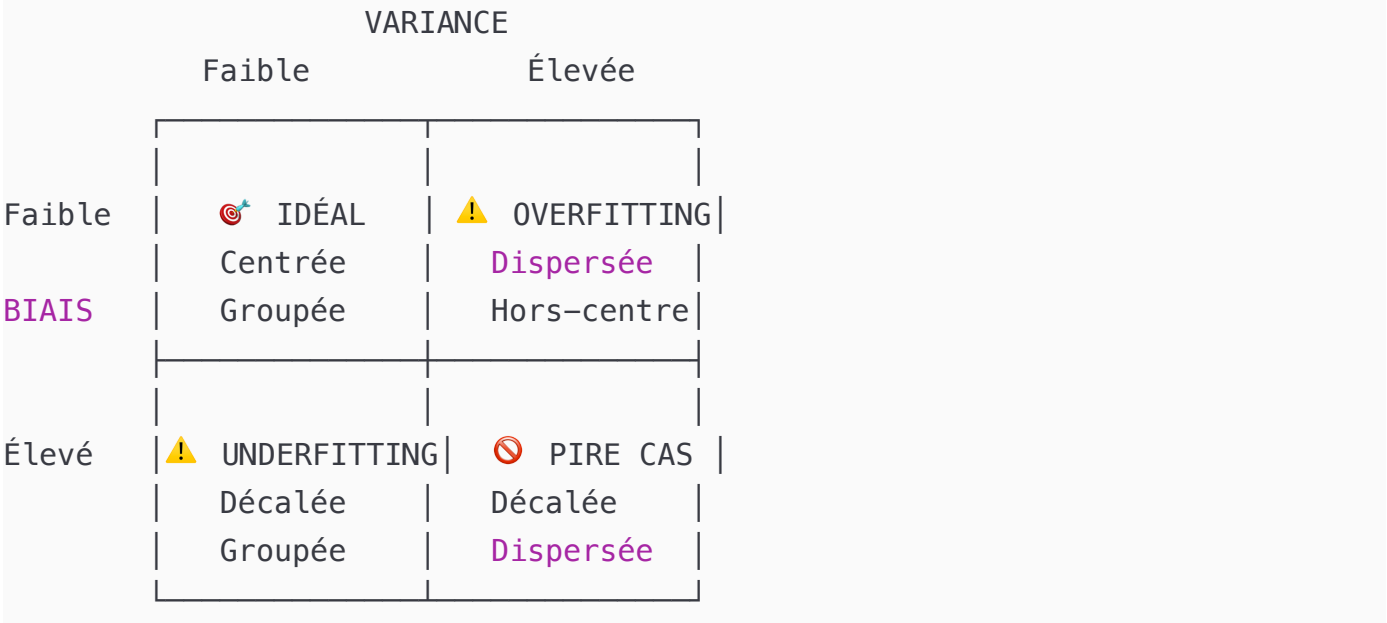
Variance : Sensibilité du modèle aux fluctuations de l'échantillon d'apprentissage

$$\text{Var}[\hat{f}(x)] = \mathbb{E}\left[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2\right]$$

- **Variance élevée** → Modèle trop complexe → **Overfitting**
- Le modèle capture le bruit plutôt que le signal

1.3 Visualisation du Dilemme Biais-Variance

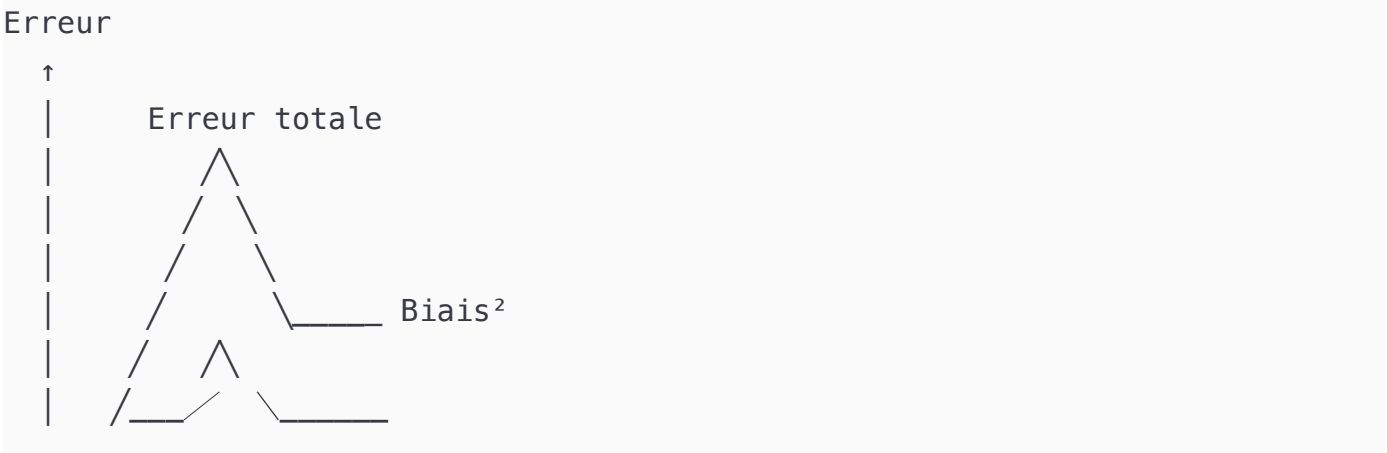
1.3.1 Représentation en Cible (Target Diagram)



Interprétation :

- **Centre de la cible** : Vraie valeur $f(x)$
- **Points noirs** : Prédictions sur différents échantillons
- **Dispersion** : Mesure de la variance
- **Distance au centre** : Mesure du biais

1.3.2 Courbe de Complexité



```
|  Variance
|
└──────────────────────→
   Simple    ↑    Complexe
         Optimal
```

1.4 Stratégies des Modèles d'Ensemble

1.4.1 Trois Approches Fondamentales

Approche	Principe	Objectif Principal	Méthode
Bagging	Parallélisation	Réduire la **variance**	Moyennage de modèles indépendants
Boosting	Séquentialisation	Réduire le **biais**	Correction séquentielle des erreurs
Stacking	Méta-apprentissage	Optimiser la **combinaison**	Apprentissage de la meilleure agrégation

1.4.2 Caractéristiques des Arbres de Décision

Les arbres de décision sont la base de la plupart des modèles d'ensemble :

Points forts :

- ✅ Très expressifs (faible biais)
- ✅ Non-linéaires
- ✅ Capturent les interactions complexes

Points faibles :

- ❌ Instables (forte variance)
- ❌ Sensibles au bruit
- ❌ Tendance au surapprentissage

1.5 Tableau Comparatif des Stratégies

Caractéristique	Arbre Unique	Random Forest (Bagging)	Boosting	Stacking
Biais	Faible	Stable	**Réduit**	Optimisé
Variance	**Élevée**	**Réduite**	Légèrement augmentée	Équilibrée

Caractéristique	Arbre Unique	Random Forest (Bagging)	Boosting	Stacking
Parallélisation	N/A	✅ Oui	❌ Non	Partielle
Complexité calcul	Faible	Moyenne	Élevée	Très élevée
Risque overfitting	Très élevé	Faible	Moyen (si mal réglé)	Moyen
Interprétabilité	Excellente	Moyenne	Faible	Faible

1.6 Intuitions Clés

> 🧠 **Bagging (Random Forest)** : "Sagesse de la foule"
>
> - Combine des modèles diversifiés
> - La moyenne stabilise les prédictions
> - Effet : **Variance** ↓

> 🚀 **Boosting (XGBoost, LightGBM, CatBoost)** : "Apprentissage incrémental"
>
> - Chaque modèle corrige les erreurs du précédent
> - Focus sur les exemples difficiles
> - Effet : **Biais** ↓

> 🎯 **Stacking** : "Expert des experts"
>
> - Apprend comment combiner optimalement les prédictions
> - Exploite les forces de chaque modèle
> - Effet : **Performance globale** ↑

🌲 Partie 2 – Modèle de Bagging : Random Forest

2.1 Fondements Théoriques

2.1.1 Principe du Bootstrap Aggregating (Bagging)

Le **Bagging** combine deux idées puissantes :

1. **Bootstrap** : Échantillonnage avec remise
2. **Aggregating** : Agrégation par vote ou moyenne

Algorithme général :

```
Pour t = 1 à T :
    1. Créer un échantillon bootstrap D_t en tirant N exemples avec remise
    2. Entraîner un modèle h_t sur D_t
Prédiction finale : Agrégation des h_t
```

2.1.2 Random Forest : Extension du Bagging

Random Forest ajoute une **randomisation supplémentaire** au niveau des features :

- À chaque nœud, sélection aléatoire de m features parmi p disponibles
- Généralement : $m \approx \sqrt{p}$ (classification) ou $m \approx p/3$ (régression)

Effet : Décorrélation des arbres → Réduction encore plus forte de la variance

---

2.2 Formulations Mathématiques

2.2.1 Bootstrap et Diversité

Soit $D = \{(x_i, y_i)\}_{i=1}^{N}$ l'ensemble d'apprentissage.

Échantillon bootstrap D_t :

$$D_t = \{(x_{i_j}, y_{i_j})\}_{j=1}^{N} \text{ où } i_j \sim \text{Uniform}(\{1, ..., N\})$$

Probabilité qu'un exemple ne soit jamais sélectionné :

$$P(\text{non sélectionné}) = \left(1 - \frac{1}{N}\right)^N \xrightarrow{N \to \infty} \frac{1}{e} \approx 0.368$$

→ Environ **63.2%** des données sont utilisées pour entraîner chaque arbre

2.2.2 Critère de Division : Indice de Gini

Pour un nœud contenant un ensemble S d'exemples :

Indice de Gini :

$$\text{Gini}(S) = 1 - \sum_{k=1}^{K} p_k^2$$

où p_k = proportion d'exemples de la classe k dans S

Gain d'impureté pour une division (S_L, S_R) :

$$\Delta i = i(S) - \frac{|S_L|}{|S|} \cdot i(S_L) - \frac{|S_R|}{|S|} \cdot i(S_R)$$

Le split optimal maximise ce gain.

2.2.3 Agrégation des Prédictions

Classification (vote majoritaire) :

$$\hat{y} = \arg\max_k \sum_{t=1}^{T} 1_{h_t(x)=k}$$

Régression (moyenne) :

$$\hat{y} = \frac{1}{T} \sum_{t=1}^{T} h_t(x)$$

2.2.4 Variance de l'Ensemble

Si les arbres sont **indépendants** avec variance σ^2 :

$$\text{Var}(\text{moyenne}) = \frac{\sigma^2}{T}$$

En pratique, corrélation ρ entre arbres :

$$\text{Var}(\text{Random Forest}) = \rho\sigma^2 + \frac{1-\rho}{T}\sigma^2$$

Objectif : Minimiser ρ via randomisation des features

2.3 Métriques d'Importance des Variables

2.3.1 Mean Decrease Impurity (MDI)

Pour chaque variable X_j :

$$\text{Importance}(X_j) = \frac{1}{T} \sum_{t=1}^{T} \sum_{s \in \text{splits sur } X_j} \frac{|S_s|}{N} \cdot \Delta i_s$$

2.3.2 Out-of-Bag (OOB) Error

Les exemples non utilisés pour entraîner un arbre (≈37%) servent de validation :

$$\text{OOB Error} = \frac{1}{N} \sum_{i=1}^{N} 1_{y_i \neq \hat{y}_i^{\text{OOB}}}$$

où $\hat{y}_i^{\text{OOB}}$ = prédiction par les arbres n'ayant pas vu x_i

---

2.4 Implémentation Python Complète

2.4.1 Code de Base

```python
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.datasets import load_breast_cancer, load_diabetes
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import (
    accuracy_score, classification_report,
    mean_squared_error, r2_score
)
import matplotlib.pyplot as plt
import seaborn as sns

# Configuration
sns.set_style("whitegrid")
np.random.seed(42)

# ============================================
# EXEMPLE 1 : Classification
# ============================================

# Chargement des données
data = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    data.data, data.target, test_size=0.2, random_state=42,
stratify=data.target
)

# Modèle Random Forest
rf_clf = RandomForestClassifier(
    n_estimators=200,        # Nombre d'arbres
    max_depth=None,          # Profondeur maximale (None = sans limite)
    min_samples_split=2,     # Min échantillons pour splitter
    min_samples_leaf=1,      # Min échantillons par feuille
    max_features='sqrt',     # Nombre de features par split
    bootstrap=True,          # Échantillonnage bootstrap
    oob_score=True,          # Calculer l'erreur OOB
    n_jobs=-1,               # Parallélisation
    random_state=42
)

# Entraînement
rf_clf.fit(X_train, y_train)
```

```python
# Prédictions
y_pred = rf_clf.predict(X_test)
y_proba = rf_clf.predict_proba(X_test)

# Évaluation
print("="*50)
print("RANDOM FOREST - CLASSIFICATION")
print("="*50)
print(f"Accuracy (Test): {accuracy_score(y_test, y_pred):.4f}")
print(f"OOB Score: {rf_clf.oob_score_:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=data.target_names))

# Cross-validation
cv_scores = cross_val_score(rf_clf, X_train, y_train, cv=5,
scoring='accuracy')
print(f"\nCross-Validation Accuracy: {cv_scores.mean():.4f} (+/-
{cv_scores.std():.4f})")

# ============================================
# EXEMPLE 2 : Régression
# ============================================

# Chargement des données
diabetes = load_diabetes()
X_train_r, X_test_r, y_train_r, y_test_r = train_test_split(
    diabetes.data, diabetes.target, test_size=0.2, random_state=42
)

# Modèle Random Forest
rf_reg = RandomForestRegressor(
    n_estimators=200,
    max_depth=10,
    min_samples_split=5,
    min_samples_leaf=2,
    max_features='sqrt',
    bootstrap=True,
    oob_score=True,
    n_jobs=-1,
    random_state=42
)

# Entraînement et prédictions
```

```python
rf_reg.fit(X_train_r, y_train_r)
y_pred_r = rf_reg.predict(X_test_r)

# Évaluation
print("\n" + "="*50)
print("RANDOM FOREST - RÉGRESSION")
print("="*50)
print(f"R² Score (Test): {r2_score(y_test_r, y_pred_r):.4f}")
print(f"RMSE (Test): {np.sqrt(mean_squared_error(y_test_r, y_pred_r)):.4f}")
print(f"OOB Score (R²): {rf_reg.oob_score_:.4f}")

# ============================================
# VISUALISATIONS
# ============================================

# 1. Importance des features
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Classification
importances_clf = rf_clf.feature_importances_
indices_clf = np.argsort(importances_clf)[-10:]
axes[0].barh(range(len(indices_clf)), importances_clf[indices_clf])
axes[0].set_yticks(range(len(indices_clf)))
axes[0].set_yticklabels(np.array(data.feature_names)[indices_clf])
axes[0].set_xlabel('Importance')
axes[0].set_title('Top 10 Features - Classification')

# Régression
importances_reg = rf_reg.feature_importances_
indices_reg = np.argsort(importances_reg)[-10:]
axes[1].barh(range(len(indices_reg)), importances_reg[indices_reg])
axes[1].set_yticks(range(len(indices_reg)))
axes[1].set_yticklabels(np.array(diabetes.feature_names)[indices_reg])
axes[1].set_xlabel('Importance')
axes[1].set_title('Top 10 Features - Régression')

plt.tight_layout()
plt.savefig('rf_feature_importance.png', dpi=300, bbox_inches='tight')
plt.show()

# 2. Learning Curves
from sklearn.model_selection import learning_curve
```

```
train_sizes, train_scores, val_scores = learning_curve(
    rf_clf, X_train, y_train, cv=5, n_jobs=-1,
    train_sizes=np.linspace(0.1, 1.0, 10), scoring='accuracy'
)

train_mean = train_scores.mean(axis=1)
train_std = train_scores.std(axis=1)
val_mean = val_scores.mean(axis=1)
val_std = val_scores.std(axis=1)

plt.figure(figsize=(10, 6))
plt.plot(train_sizes, train_mean, label='Score Train', marker='o')
plt.plot(train_sizes, val_mean, label='Score Validation', marker='s')
plt.fill_between(train_sizes, train_mean - train_std, train_mean +
train_std, alpha=0.1)
plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std,
alpha=0.1)
plt.xlabel('Nombre d\'exemples d\'entraînement')
plt.ylabel('Accuracy')
plt.title('Learning Curves - Random Forest')
plt.legend()
plt.grid(True, alpha=0.3)
plt.savefig('rf_learning_curves.png', dpi=300, bbox_inches='tight')
plt.show()
```

2.4.2 Analyse de la Profondeur Optimale

```
# Test de différentes profondeurs
depths = range(1, 21)
train_scores = []
test_scores = []
oob_scores = []

for depth in depths:
    rf_temp = RandomForestClassifier(
        n_estimators=100,
        max_depth=depth,
        oob_score=True,
        random_state=42,
        n_jobs=-1
    )
    rf_temp.fit(X_train, y_train)

    train_scores.append(rf_temp.score(X_train, y_train))
```

```
    test_scores.append(rf_temp.score(X_test, y_test))
    oob_scores.append(rf_temp.oob_score_)

# Visualisation
plt.figure(figsize=(10, 6))
plt.plot(depths, train_scores, label='Train', marker='o')
plt.plot(depths, test_scores, label='Test', marker='s')
plt.plot(depths, oob_scores, label='OOB', marker='^')
plt.xlabel('Profondeur Maximale')
plt.ylabel('Accuracy')
plt.title('Impact de la Profondeur sur la Performance')
plt.legend()
plt.grid(True, alpha=0.3)
plt.savefig('rf_depth_analysis.png', dpi=300, bbox_inches='tight')
plt.show()

print(f"Profondeur optimale (Test): {depths[np.argmax(test_scores)]}")
```

2.5 Avantages et Limitations

2.5.1 Avantages

✅ **Robustesse**

- Résistant au surapprentissage
- Stable face au bruit

✅ **Polyvalence**

- Classification et régression
- Variables numériques et catégorielles

✅ **Facilité d'utilisation**

- Peu d'hyperparamètres critiques
- Validation OOB intégrée

✅ **Interprétabilité**

- Importance des variables
- Visualisation des arbres individuels

2.5.2 Limitations

❌ **Performance modérée**

- Moins précis que le boosting sur données complexes

❌ **Mémoire**

- Stockage de nombreux arbres complets

❌ **Prédiction lente**

- Nécessite l'agrégation de tous les arbres

❌ **Variables catégorielles**

- Nécessite un encodage préalable

⚡ Partie 3 – Modèles de Boosting

3.1 Principe Fondamental du Boosting

3.1.1 Philosophie

Le **Boosting** construit séquentiellement des modèles faibles, chacun se concentrant sur les erreurs des précédents.

Algorithme générique :

```
Initialiser : F_0(x) = valeur initiale (souvent 0 ou moyenne)

Pour m = 1 à M :
    1. Calculer les résidus : r_i = y_i - F_{m-1}(x_i)
    2. Entraîner h_m pour prédire r_i
    3. Mettre à jour : F_m(x) = F_{m-1}(x) + η · h_m(x)

Prédiction finale : F_M(x)
```

où η est le **taux d'apprentissage** (learning rate)

3.2 XGBoost (Extreme Gradient Boosting)

3.2.1 Formulation Mathématique Complète

Objectif d'optimisation

On cherche à minimiser une fonction de perte régularisée :

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

où :

- l : fonction de perte (ex: log-loss, MSE)
- $\Omega(f_k)$: régularisation du k-ième arbre

Algorithme itératif

À l'itération t, on ajoute un arbre f_t pour minimiser :

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

Approximation de Taylor (ordre 2)

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

où :

Gradient (premier ordre) :

$$g_i = \frac{\partial l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}$$

Hessien (second ordre) :

$$h_i = \frac{\partial^2 l(y_i, \hat{y}_i^{(t-1)})}{\partial (\hat{y}_i^{(t-1)})^2}$$

Régularisation de l'arbre

$$\Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2$$

où :

- T : nombre de feuilles
- w_j : score de la feuille j
- γ : pénalité par feuille
- λ : régularisation L2

Poids optimal d'une feuille

En supprimant les termes constants et en réorganisant :

$$\mathcal{L}^{(t)} = \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T$$

Le poids optimal de la feuille j est :

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

Gain de split

Pour évaluer si une division est bénéfique :

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right] - \gamma$$

où :

- $G_L = \sum_{i \in I_L} g_i, G_R = \sum_{i \in I_R} g_i$
- $H_L = \sum_{i \in I_L} h_i, H_R = \sum_{i \in I_R} h_i$

Interprétation :

- Si Gain > 0 : le split améliore le modèle
- γ contrôle la complexité de l'arbre

3.2.2 Implémentation Python

```
import xgboost as xgb
from sklearn.model_selection import GridSearchCV
import numpy as np
import matplotlib.pyplot as plt

# ============================================
# CLASSIFICATION AVEC XGBOOST
# ============================================

# Préparation des données
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Paramètres de base
params = {
    'objective': 'binary:logistic',
    'eval_metric': 'logloss',
    'max_depth': 4,
    'learning_rate': 0.05,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'lambda': 1.0,  # L2 regularization
    'alpha': 0.0,   # L1 regularization
    'gamma': 0.0,   # Minimum loss reduction
```

```
    'seed': 42
}

# Entraînement avec early stopping
evals = [(dtrain, 'train'), (dtest, 'test')]
evals_result = {}

xgb_model = xgb.train(
    params,
    dtrain,
    num_boost_round=1000,
    evals=evals,
    early_stopping_rounds=50,
    evals_result=evals_result,
    verbose_eval=100
)

# Prédictions
y_pred_proba = xgb_model.predict(dtest)
y_pred = (y_pred_proba > 0.5).astype(int)

print("="*50)
print("XGBOOST - RÉSULTATS")
print("="*50)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(f"\nMeilleure itération: {xgb_model.best_iteration}")
print(f"Meilleur score: {xgb_model.best_score:.4f}")

# ============================================
# AVEC SKLEARN API
# ============================================

xgb_clf = xgb.XGBClassifier(
    n_estimators=300,
    learning_rate=0.05,
    max_depth=4,
    subsample=0.8,
    colsample_bytree=0.8,
    reg_lambda=1.0,
    reg_alpha=0.0,
    gamma=0.0,
    random_state=42,
    n_jobs=-1
```

```
)

xgb_clf.fit(
    X_train, y_train,
    eval_set=[(X_test, y_test)],
    early_stopping_rounds=50,
    verbose=False
)

# ============================================
# TUNING DES HYPERPARAMÈTRES
# ============================================

param_grid = {
    'max_depth': [3, 4, 5, 6],
    'learning_rate': [0.01, 0.05, 0.1],
    'n_estimators': [100, 200, 300],
    'subsample': [0.7, 0.8, 0.9],
    'colsample_bytree': [0.7, 0.8, 0.9],
    'gamma': [0, 0.1, 0.2]
}

grid_search = GridSearchCV(
    xgb.XGBClassifier(random_state=42, n_jobs=-1),
    param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train, y_train)

print("\nMeilleurs paramètres:")
print(grid_search.best_params_)
print(f"Meilleur score CV: {grid_search.best_score_:.4f}")

# ============================================
# VISUALISATIONS
# ============================================

# 1. Courbes d'apprentissage
fig, axes = plt.subplots(1, 2, figsize=(15, 5))
```

```python
# Loss
epochs = len(evals_result['train']['logloss'])
x_axis = range(epochs)
axes[0].plot(x_axis, evals_result['train']['logloss'], label='Train')
axes[0].plot(x_axis, evals_result['test']['logloss'], label='Test')
axes[0].legend()
axes[0].set_ylabel('Log Loss')
axes[0].set_xlabel('Itération')
axes[0].set_title('Évolution de la Loss')
axes[0].grid(True, alpha=0.3)

# Feature importance
xgb.plot_importance(xgb_model, max_num_features=10, ax=axes[1],
importance_type='gain')
axes[1].set_title('Importance des Features (Gain)')

plt.tight_layout()
plt.savefig('xgboost_analysis.png', dpi=300, bbox_inches='tight')
plt.show()

# 2. Arbre individuel
fig, ax = plt.subplots(figsize=(20, 10))
xgb.plot_tree(xgb_model, num_trees=0, ax=ax)
plt.savefig('xgboost_tree.png', dpi=300, bbox_inches='tight')
plt.show()
```

3.3 LightGBM

3.3.1 Innovations Clés

Croissance Leaf-wise vs Level-wise

Level-wise (XGBoost traditionnel) :

```
      0              0
     / \            / \
    0   0    →     0   0
                   /|   |\
                  0 0   0 0
```

Leaf-wise (LightGBM) :

```
      0              0
     / \            / \
```

```
    0   0    →      0   0
                        \
                         0
```

LightGBM choisit la feuille avec le **plus grand gain** à diviser.

GOSS (Gradient-based One-Side Sampling)

Problème : Tous les exemples ne sont pas également informatifs.

Solution :

1. Trier les exemples par gradient absolu $|g_i|$
2. Garder les $a \times 100\%$ premiers (grands gradients)
3. Échantillonner aléatoirement $b \times 100\%$ parmi le reste
4. Amplifier les petits gradients par facteur $\frac{1-a}{b}$

Gain estimé :

$$\tilde{\text{Gain}} = \frac{1}{n}\left[\frac{(\sum_{i \in A} g_i + \frac{1-a}{b}\sum_{i \in B} g_i)^2}{H_L} + \ldots\right]$$

EFB (Exclusive Feature Bundling)

Objectif : Réduire la dimensionnalité pour les features rares

Idée : Regrouper les features mutuellement exclusives (peu de valeurs non-nulles communes)

3.3.2 Formulation du Gain

Identique à XGBoost :

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right] - \gamma$$

Mais avec optimisations algorithmiques pour le calcul.

3.3.3 Implémentation Python

```python
import lightgbm as lgb
from sklearn.model_selection import train_test_split

# ============================================
# CLASSIFICATION AVEC LIGHTGBM
# ============================================

# Préparation Dataset LightGBM
train_data = lgb.Dataset(X_train, label=y_train)
```

```python
test_data = lgb.Dataset(X_test, label=y_test, reference=train_data)

# Paramètres
params = {
    'objective': 'binary',
    'metric': 'binary_logloss',
    'boosting_type': 'gbdt',
    'num_leaves': 31,
    'learning_rate': 0.05,
    'feature_fraction': 0.8,
    'bagging_fraction': 0.8,
    'bagging_freq': 5,
    'lambda_l1': 0.0,
    'lambda_l2': 1.0,
    'min_gain_to_split': 0.0,
    'verbose': -1,
    'seed': 42
}

# Entraînement
evals_result = {}
lgb_model = lgb.train(
    params,
    train_data,
    num_boost_round=1000,
    valid_sets=[train_data, test_data],
    valid_names=['train', 'test'],
    callbacks=[
        lgb.early_stopping(stopping_rounds=50),
        lgb.log_evaluation(period=100),
        lgb.record_evaluation(evals_result)
    ]
)

# Prédictions
y_pred_proba = lgb_model.predict(X_test,
num_iteration=lgb_model.best_iteration)
y_pred = (y_pred_proba > 0.5).astype(int)

print("="*50)
print("LIGHTGBM - RÉSULTATS")
print("="*50)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

```
print(f"Meilleure itération: {lgb_model.best_iteration}")

# ============================================
# AVEC SKLEARN API
# ============================================

lgb_clf = lgb.LGBMClassifier(
    n_estimators=300,
    learning_rate=0.05,
    num_leaves=31,
    max_depth=-1,
    subsample=0.8,
    colsample_bytree=0.8,
    reg_lambda=1.0,
    reg_alpha=0.0,
    min_split_gain=0.0,
    random_state=42,
    n_jobs=-1
)

lgb_clf.fit(
    X_train, y_train,
    eval_set=[(X_test, y_test)],
    callbacks=[lgb.early_stopping(stopping_rounds=50),
lgb.log_evaluation(period=0)]
)

# ============================================
# VISUALISATIONS
# ============================================

fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# 1. Loss curves
ax = axes[0, 0]
ax.plot(evals_result['train']['binary_logloss'], label='Train')
ax.plot(evals_result['test']['binary_logloss'], label='Test')
ax.set_xlabel('Itération')
ax.set_ylabel('Log Loss')
ax.set_title('Évolution de la Loss')
ax.legend()
ax.grid(True, alpha=0.3)
```

```
# 2. Feature importance (split)
ax = axes[0, 1]
lgb.plot_importance(lgb_model, max_num_features=10, importance_type='split',
ax=ax)
ax.set_title('Importance (Nombre de Splits)')

# 3. Feature importance (gain)
ax = axes[1, 0]
lgb.plot_importance(lgb_model, max_num_features=10, importance_type='gain',
ax=ax)
ax.set_title('Importance (Gain)')

# 4. Arbre
ax = axes[1, 1]
lgb.plot_tree(lgb_model, tree_index=0, ax=ax, figsize=(15, 10))
ax.set_title('Visualisation du Premier Arbre')

plt.tight_layout()
plt.savefig('lightgbm_analysis.png', dpi=300, bbox_inches='tight')
plt.show()
```

3.4 CatBoost

3.4.1 Innovations Principales

Ordered Target Statistics (Encodage Catégoriel)

Problème : Encodage naïf cause du target leakage

Solution : Encodage basé sur un ordre artificiel

Pour l'observation i et catégorie c :

$$\text{Enc}(c_i) = \frac{\sum_{j=1}^{i-1} 1_{c_j=c_i} \cdot y_j + \text{prior}}{\sum_{j=1}^{i-1} 1_{c_j=c_i} + \alpha}$$

où :

- prior : valeur a priori (souvent la moyenne globale)
- α : poids du prior (régularisation)

Avantage : Pas de data leakage, encodage dynamique

Ordered Boosting

Problème : Prédictions biaisées si on utilise les mêmes données pour entraîner et prédire

Solution :

1. Permuter aléatoirement l'ordre des données
2. Pour chaque exemple i, construire le modèle sur $\{1, ..., i-1\}$
3. Utiliser ce modèle pour prédire i

En pratique : Maintien de plusieurs modèles avec différentes permutations

Symmetric Trees

CatBoost utilise des arbres **symétriques** :

- Même condition de split à chaque niveau
- Structure équilibrée
- Plus rapide en prédiction

3.4.2 Formulation Mathématique

Objectif similaire à XGBoost :

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

Mais avec :

- Encodage catégoriel natif
- Boosting ordonné pour éviter le biais

3.4.3 Implémentation Python

```
from catboost import CatBoostClassifier, Pool
import pandas as pd

# ============================================
# EXEMPLE AVEC VARIABLES CATÉGORIELLES
# ============================================

# Créer un dataset avec catégories
df_train = pd.DataFrame(X_train, columns=data.feature_names)
df_train['target'] = y_train
df_test = pd.DataFrame(X_test, columns=data.feature_names)

# Pour cet exemple, simulons quelques variables catégorielles
# (normalement, les données Breast Cancer sont toutes numériques)
df_train['cat_feature_1'] = pd.cut(df_train.iloc[:, 0], bins=5, labels=['A',
'B', 'C', 'D', 'E'])
```

```
df_test['cat_feature_1'] = pd.cut(df_test.iloc[:, 0], bins=5, labels=['A',
'B', 'C', 'D', 'E'])

cat_features = ['cat_feature_1']

# Préparation Pool CatBoost
train_pool = Pool(
    df_train.drop('target', axis=1),
    df_train['target'],
    cat_features=cat_features
)

test_pool = Pool(
    df_test,
    y_test,
    cat_features=cat_features
)

# ============================================
# MODÈLE CATBOOST
# ============================================

cat_clf = CatBoostClassifier(
    iterations=300,
    learning_rate=0.05,
    depth=4,
    l2_leaf_reg=3.0,
    loss_function='Logloss',
    eval_metric='Accuracy',
    random_seed=42,
    verbose=100,
    early_stopping_rounds=50,
    task_type='CPU',  # 'GPU' si disponible
    bootstrap_type='Bernoulli',
    subsample=0.8
)

# Entraînement
cat_clf.fit(
    train_pool,
    eval_set=test_pool,
    plot=False
)
```

```python
# Prédictions
y_pred = cat_clf.predict(test_pool)
y_pred_proba = cat_clf.predict_proba(test_pool)

print("="*50)
print("CATBOOST - RÉSULTATS")
print("="*50)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(f"Meilleure itération: {cat_clf.best_iteration_}")
print(f"Meilleur score: {cat_clf.best_score_['validation']
['Accuracy']:.4f}")

# ============================================
# ANALYSE DES FEATURES
# ============================================

# Feature importance
feature_importance = cat_clf.get_feature_importance(train_pool)
feature_names = df_train.drop('target', axis=1).columns

importance_df = pd.DataFrame({
    'feature': feature_names,
    'importance': feature_importance
}).sort_values('importance', ascending=False)

print("\nTop 10 Features:")
print(importance_df.head(10))

# Visualisation
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Feature importance
axes[0].barh(importance_df.head(10)['feature'], importance_df.head(10)
['importance'])
axes[0].set_xlabel('Importance')
axes[0].set_title('Top 10 Features - CatBoost')
axes[0].invert_yaxis()

# Learning curves
evals_result = cat_clf.get_evals_result()
axes[1].plot(evals_result['learn']['Logloss'], label='Train')
axes[1].plot(evals_result['validation']['Logloss'], label='Validation')
```

```
axes[1].set_xlabel('Itération')
axes[1].set_ylabel('Log Loss')
axes[1].set_title('Évolution de la Loss')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('catboost_analysis.png', dpi=300, bbox_inches='tight')
plt.show()
```

3.5 Comparaison des Modèles de Boosting

Caractéristique	XGBoost	LightGBM	CatBoost
Croissance arbre	Level-wise	Leaf-wise	Symmetric
Vitesse	Rapide	**Très rapide**	Moyen
Précision	Excellente	Excellente	**Excellente**
Catégorielles	Non natif	Non natif	**Natif**
Overfitting	Risque moyen	**Risque élevé**	Faible
GPU	✅ Oui	✅ Oui	✅ Oui
Mémoire	Moyenne	**Faible**	Moyenne
Hyperparamètres	Nombreux	Nombreux	**Moins nombreux**
Target leakage	Risque	Risque	**Protégé**

🎯 Partie 4 – Stacking : Méta-apprentissage

4.1 Principe et Motivation

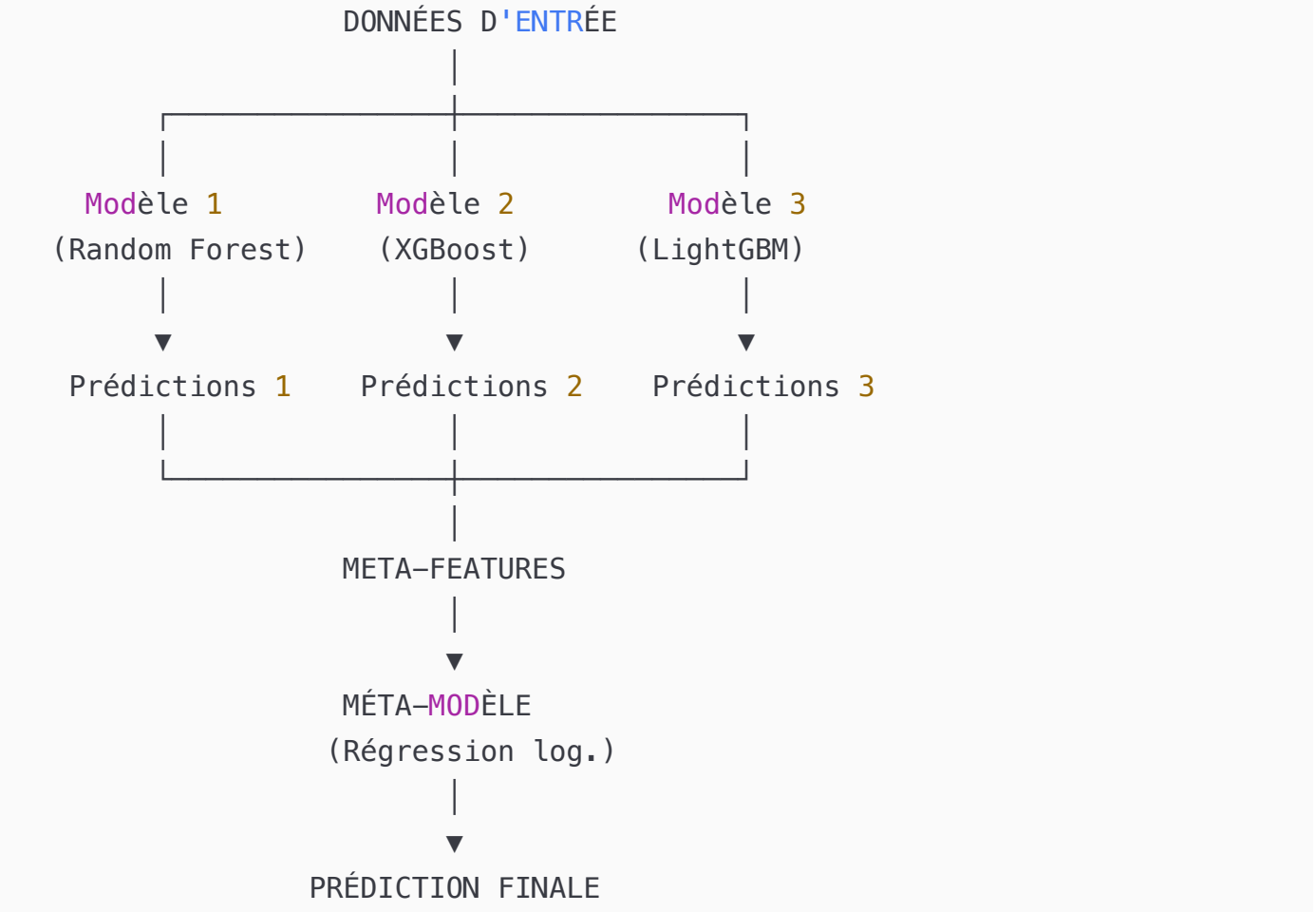
4.1.1 Concept Fondamental

Le **Stacking** (ou **Stacked Generalization**) est une méthode d'ensemble qui :

1. Entraîne plusieurs modèles de base (**base learners**)
2. Utilise leurs prédictions comme features pour un **méta-modèle**
3. Le méta-modèle apprend la meilleure façon de combiner les prédictions

Analogie : Un jury d'experts où un "super-expert" apprend à pondérer les avis.

4.1.2 Architecture



4.2 Formulation Mathématique

4.2.1 Notations

Soit :

- $D = \{(x_i, y_i)\}_{i=1}^{N}$: dataset d'entraînement
- $\{h_1, h_2, ..., h_K\}$: ensemble de K modèles de base
- g : méta-modèle

4.2.2 Processus en Deux Étapes

Niveau 1 : Entraînement des modèles de base

Pour chaque modèle h_k :

$$h_k : \mathcal{X} \to \mathcal{Y}$$

Niveau 2 : Construction des méta-features

Pour éviter l'overfitting, on utilise la **validation croisée** :

1. Diviser D en M folds : $D = D_1 \cup D_2 \cup ... \cup D_M$

2. Pour chaque fold m et modèle k :
   - Entraîner h_k sur $D \setminus D_m$
   - Prédire sur D_m : $\hat{y}_{k,m} = h_k(x_{D_m})$
3. Concaténer pour obtenir les méta-features :

$$Z = \begin{bmatrix} \hat{y}_{1,1} & \hat{y}_{2,1} & \cdots & \hat{y}_{K,1} \\ \hat{y}_{1,2} & \hat{y}_{2,2} & \cdots & \hat{y}_{K,2} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{y}_{1,N} & \hat{y}_{2,N} & \cdots & \hat{y}_{K,N} \end{bmatrix} \in \mathbb{R}^{N \times K}$$

Niveau 3 : Entraînement du méta-modèle

$$g : \mathbb{R}^K \to \mathcal{Y}$$

Le méta-modèle apprend à combiner les prédictions :

$$\hat{y}_{\text{final}} = g(h_1(x), h_2(x), ..., h_K(x))$$

4.2.3 Objectif d'optimisation

Le méta-modèle minimise :

$$\arg\min_{g} \sum_{i=1}^{N} L(y_i, g(h_1(x_i), h_2(x_i), ..., h_K(x_i)))$$

où L est la fonction de perte.

4.2.4 Cas particulier : Combinaison linéaire

Si g est linéaire :

$$g(z) = w_0 + \sum_{k=1}^{K} w_k z_k$$

Le problème devient :

$$\arg\min_{w_0, w_1, ..., w_K} \sum_{i=1}^{N} L\left(y_i, w_0 + \sum_{k=1}^{K} w_k h_k(x_i)\right)$$

Avec contrainte de normalisation (optionnelle) :

$$\sum_{k=1}^{K} w_k = 1, \quad w_k \geq 0$$

4.3 Variantes de Stacking

4.3.1 Stacking Classique

- Méta-features = prédictions brutes des modèles de base
- CV pour éviter l'overfitting

4.3.2 Blending

- Division simple train/validation (pas de CV)
- Plus rapide mais moins robuste

4.3.3 Multi-Layer Stacking

Empilement de plusieurs niveaux :

```
Niveau 0 : Données originales
     ↓
Niveau 1 : Modèles de base (5 modèles)
     ↓
Niveau 2 : Méta-modèles niveau 1 (2 modèles)
     ↓
Niveau 3 : Méta-modèle final (1 modèle)
```

4.3.4 Feature-Weighted Linear Stacking

Concaténer features originales et méta-features :

$$Z_{\text{augmented}} = [X, \hat{Y}_1, \hat{Y}_2, ..., \hat{Y}_K]$$

4.4 Implémentation Python Complète

4.4.1 Stacking Manuel avec Validation Croisée

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, roc_auc_score
import xgboost as xgb
import lightgbm as lgb
from catboost import CatBoostClassifier

# ============================================
# FONCTION DE STACKING GÉNÉRIQUE
# ============================================
```

```python
def get_oof_predictions(model, X_train, y_train, X_test, n_folds=5):
    """
    Génère les prédictions out-of-fold pour le stacking.

    Returns:
        oof_train : prédictions sur l'ensemble d'entraînement (via CV)
        oof_test : moyenne des prédictions sur le test
    """
    n_train = X_train.shape[0]
    n_test = X_test.shape[0]

    # Initialisation
    oof_train = np.zeros((n_train,))
    oof_test = np.zeros((n_test,))
    oof_test_skf = np.zeros((n_folds, n_test))

    # Validation croisée stratifiée
    skf = StratifiedKFold(n_splits=n_folds, shuffle=True, random_state=42)

    for fold_idx, (train_idx, val_idx) in enumerate(skf.split(X_train,
y_train)):
        print(f"  Fold {fold_idx + 1}/{n_folds}...")

        # Division des données
        X_tr, X_val = X_train[train_idx], X_train[val_idx]
        y_tr, y_val = y_train[train_idx], y_train[val_idx]

        # Clone du modèle pour éviter les effets de bord
        from sklearn.base import clone
        model_fold = clone(model)

        # Entraînement
        model_fold.fit(X_tr, y_tr)

        # Prédictions OOF sur validation
        oof_train[val_idx] = model_fold.predict_proba(X_val)[:, 1]

        # Prédictions sur test
        oof_test_skf[fold_idx, :] = model_fold.predict_proba(X_test)[:, 1]

    # Moyenne des prédictions test
    oof_test[:] = oof_test_skf.mean(axis=0)
```

```
    return oof_train, oof_test

# ============================================
# ÉTAPE 1 : DÉFINIR LES MODÈLES DE BASE
# ============================================

# Chargement des données
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

data = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    data.data, data.target, test_size=0.2, random_state=42,
stratify=data.target
)

print("="*60)
print("STACKING - IMPLÉMENTATION COMPLÈTE")
print("="*60)
print(f"Taille train: {X_train.shape}")
print(f"Taille test: {X_test.shape}")

# Définition des modèles de base
base_models = {
    'Random Forest': RandomForestClassifier(
        n_estimators=200,
        max_depth=10,
        random_state=42,
        n_jobs=-1
    ),
    'XGBoost': xgb.XGBClassifier(
        n_estimators=200,
        learning_rate=0.05,
        max_depth=4,
        subsample=0.8,
        colsample_bytree=0.8,
        random_state=42,
        n_jobs=-1
    ),
    'LightGBM': lgb.LGBMClassifier(
        n_estimators=200,
        learning_rate=0.05,
```

```
        num_leaves=31,
        subsample=0.8,
        colsample_bytree=0.8,
        random_state=42,
        n_jobs=-1,
        verbose=-1
    ),
    'CatBoost': CatBoostClassifier(
        iterations=200,
        learning_rate=0.05,
        depth=4,
        random_seed=42,
        verbose=0
    ),
    'SVM': SVC(
        kernel='rbf',
        C=1.0,
        probability=True,
        random_state=42
    ),
    'KNN': KNeighborsClassifier(
        n_neighbors=5,
        n_jobs=-1
    )
}

# ============================================
# ÉTAPE 2 : GÉNÉRER LES MÉTA-FEATURES
# ============================================

print("\n" + "="*60)
print("GÉNÉRATION DES MÉTA-FEATURES (Niveau 1)")
print("="*60)

meta_train = np.zeros((X_train.shape[0], len(base_models)))
meta_test = np.zeros((X_test.shape[0], len(base_models)))

for i, (name, model) in enumerate(base_models.items()):
    print(f"\n[{i+1}/{len(base_models)}] Modèle: {name}")

    # Génération OOF
    oof_train, oof_test = get_oof_predictions(
        model, X_train, y_train, X_test, n_folds=5
```

```
    )

    # Stockage
    meta_train[:, i] = oof_train
    meta_test[:, i] = oof_test

    # Performance du modèle de base
    auc_train = roc_auc_score(y_train, oof_train)
    print(f"  AUC OOF: {auc_train:.4f}")

# ============================================
# ÉTAPE 3 : ENTRAÎNER LE MÉTA-MODÈLE
# ============================================

print("\n" + "="*60)
print("ENTRAÎNEMENT DU MÉTA-MODÈLE (Niveau 2)")
print("="*60)

# Méta-modèle : Régression logistique avec régularisation
meta_model = LogisticRegression(
    C=1.0,
    penalty='l2',
    solver='liblinear',
    random_state=42,
    max_iter=1000
)

meta_model.fit(meta_train, y_train)

# Prédictions finales
y_pred_train = meta_model.predict_proba(meta_train)[:, 1]
y_pred_test = meta_model.predict_proba(meta_test)[:, 1]

# ============================================
# ÉTAPE 4 : ÉVALUATION
# ============================================

print("\n" + "="*60)
print("RÉSULTATS FINAUX")
print("="*60)

# Performance du stacking
auc_train_stack = roc_auc_score(y_train, y_pred_train)
```

```python
auc_test_stack = roc_auc_score(y_test, y_pred_test)
acc_test_stack = accuracy_score(y_test, (y_pred_test > 0.5).astype(int))

print(f"\nStacking:")
print(f"  AUC Train: {auc_train_stack:.4f}")
print(f"  AUC Test:  {auc_test_stack:.4f}")
print(f"  Acc Test:  {acc_test_stack:.4f}")

# Comparaison avec les modèles de base
print("\nComparaison avec modèles individuels:")
for i, name in enumerate(base_models.keys()):
    auc_individual = roc_auc_score(y_test, meta_test[:, i])
    print(f"  {name:15s}: AUC = {auc_individual:.4f}")

# Poids du méta-modèle
print("\nPoids du méta-modèle:")
for i, name in enumerate(base_models.keys()):
    weight = meta_model.coef_[0][i]
    print(f"  {name:15s}: {weight:+.4f}")

# ============================================
# VISUALISATIONS
# ============================================

import matplotlib.pyplot as plt
import seaborn as sns

fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# 1. Corrélation entre prédictions des modèles de base
ax = axes[0, 0]
corr_matrix = np.corrcoef(meta_train.T)
sns.heatmap(
    corr_matrix,
    annot=True,
    fmt='.3f',
    cmap='coolwarm',
    xticklabels=base_models.keys(),
    yticklabels=base_models.keys(),
    ax=ax,
    center=0
)
ax.set_title('Corrélation entre Modèles de Base')
```

```
# 2. Distribution des prédictions
ax = axes[0, 1]
for i, name in enumerate(base_models.keys()):
    ax.hist(meta_test[:, i], bins=30, alpha=0.5, label=name)
ax.set_xlabel('Probabilité prédite')
ax.set_ylabel('Fréquence')
ax.set_title('Distribution des Prédictions (Test)')
ax.legend()
ax.grid(True, alpha=0.3)

# 3. Poids du méta-modèle
ax = axes[1, 0]
weights = meta_model.coef_[0]
colors = ['green' if w > 0 else 'red' for w in weights]
ax.barh(list(base_models.keys()), weights, color=colors)
ax.axvline(x=0, color='black', linestyle='--', linewidth=0.8)
ax.set_xlabel('Poids')
ax.set_title('Poids du Méta-Modèle')
ax.grid(True, alpha=0.3)

# 4. Comparaison AUC
ax = axes[1, 1]
auc_scores = [roc_auc_score(y_test, meta_test[:, i]) for i in
range(len(base_models))]
auc_scores.append(auc_test_stack)
labels = list(base_models.keys()) + ['Stacking']
colors_auc = ['steelblue'] * len(base_models) + ['orange']
ax.barh(labels, auc_scores, color=colors_auc)
ax.set_xlabel('AUC Score')
ax.set_title('Comparaison des Performances (Test)')
ax.set_xlim([min(auc_scores) - 0.02, 1.0])
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('stacking_analysis.png', dpi=300, bbox_inches='tight')
plt.show()

print("\n✅ Analyse complète terminée !")
```

4.4.2 Stacking avec Scikit-Learn

```
from sklearn.ensemble import StackingClassifier
```

```python
# ============================================
# STACKING AVEC SKLEARN
# ============================================

# Définition des estimateurs
estimators = [
    ('rf', RandomForestClassifier(n_estimators=200, max_depth=10,
random_state=42, n_jobs=-1)),
    ('xgb', xgb.XGBClassifier(n_estimators=200, learning_rate=0.05,
max_depth=4, random_state=42)),
    ('lgb', lgb.LGBMClassifier(n_estimators=200, learning_rate=0.05,
random_state=42, verbose=-1)),
    ('cat', CatBoostClassifier(iterations=200, learning_rate=0.05,
random_seed=42, verbose=0))
]

# Méta-modèle
final_estimator = LogisticRegression(C=1.0, random_state=42)

# Stacking Classifier
stacking_clf = StackingClassifier(
    estimators=estimators,
    final_estimator=final_estimator,
    cv=5,                    # Validation croisée
    stack_method='predict_proba',  # Utiliser les probabilités
    n_jobs=-1,
    verbose=1
)

# Entraînement
print("\n" + "="*60)
print("STACKING AVEC SKLEARN")
print("="*60)
stacking_clf.fit(X_train, y_train)

# Prédictions
y_pred_sklearn = stacking_clf.predict(X_test)
y_pred_proba_sklearn = stacking_clf.predict_proba(X_test)[:, 1]

# Évaluation
print(f"\nAccuracy: {accuracy_score(y_test, y_pred_sklearn):.4f}")
print(f"AUC: {roc_auc_score(y_test, y_pred_proba_sklearn):.4f}")
```

```
# Coefficients du méta-modèle
print("\nCoefficients du méta-modèle:")
for name, coef in zip([name for name, _ in estimators],
stacking_clf.final_estimator_.coef_[0]):
    print(f"  {name:5s}: {coef:+.4f}")
```

4.4.3 Multi-Level Stacking

```
# ============================================
# STACKING À 3 NIVEAUX
# ============================================

# NIVEAU 1 : Modèles de base diversifiés
level1_estimators = [
    ('rf', RandomForestClassifier(n_estimators=200, random_state=42)),
    ('xgb', xgb.XGBClassifier(n_estimators=200, random_state=42)),
    ('lgb', lgb.LGBMClassifier(n_estimators=200, random_state=42,
verbose=-1)),
    ('svm', SVC(probability=True, random_state=42)),
    ('knn', KNeighborsClassifier(n_neighbors=5))
]

# NIVEAU 2 : Meta-learners
level2_left = StackingClassifier(
    estimators=level1_estimators[:3],
    final_estimator=LogisticRegression(random_state=42),
    cv=5
)

level2_right = StackingClassifier(
    estimators=level1_estimators[3:],
    final_estimator=LogisticRegression(random_state=42),
    cv=5
)

# NIVEAU 3 : Stacking final
level3_estimators = [
    ('left_stack', level2_left),
    ('right_stack', level2_right),
    ('direct_xgb', xgb.XGBClassifier(n_estimators=100, random_state=42))
]

final_stacking = StackingClassifier(
    estimators=level3_estimators,
```

```
    final_estimator=LogisticRegression(C=0.5, random_state=42),
    cv=5,
    n_jobs=-1
)

# Entraînement
print("\n" + "="*60)
print("MULTI-LEVEL STACKING (3 niveaux)")
print("="*60)
final_stacking.fit(X_train, y_train)

# Évaluation
y_pred_multi = final_stacking.predict(X_test)
y_pred_proba_multi = final_stacking.predict_proba(X_test)[:, 1]

print(f"\nAccuracy: {accuracy_score(y_test, y_pred_multi):.4f}")
print(f"AUC: {roc_auc_score(y_test, y_pred_proba_multi):.4f}")
```

4.5 Conseils Pratiques

4.5.1 Choix des Modèles de Base

Principes :

- ✅ Diversité : Utiliser des modèles de familles différentes
- ✅ Performance : Chaque modèle doit être individuellement bon
- ✅ Complémentarité : Les modèles doivent faire des erreurs différentes

Exemples de bonnes combinaisons :

1. Random Forest + XGBoost + SVM + Réseau de neurones
2. LightGBM + CatBoost + Régression logistique + KNN
3. Arbres + Linéaires + Voisins + Ensembles

4.5.2 Choix du Méta-Modèle

Modèles recommandés :

- **Régression logistique** : Simple, interprétable, évite l'overfitting
- **Ridge/Lasso** : Avec régularisation pour stabilité
- **XGBoost léger** : max_depth=2, peu d'arbres
- **Réseau de neurones peu profond** : 1-2 couches

À éviter :

- ❌ Modèles trop complexes (risque d'overfitting sur méta-features)
- ❌ Mêmes modèles que le niveau 1

4.5.3 Éviter l'Overfitting

Stratégies :

1. **Validation croisée** pour générer les méta-features
2. **Régularisation** du méta-modèle
3. **Hold-out set** séparé pour validation finale
4. **Feature selection** si trop de modèles de base

4.5.4 Optimisation

Coût computationnel :

$$\text{Coût total} = K \times (\text{coût modèle base} \times M) + \text{coût méta-modèle}$$

où K = nombre de modèles de base, M = nombre de folds CV

Accélération :

- Parallélisation des modèles de base
- Utilisation de modèles rapides (LightGBM)
- Réduction du nombre de folds (3 au lieu de 5)

4.6 Avantages et Limitations

4.6.1 Avantages

✅ **Performance** : Généralement meilleur que tout modèle individuel

✅ **Flexibilité** : Combine n'importe quels types de modèles

✅ **Robustesse** : Réduit la variance grâce à l'agrégation

✅ **Exploitation de la diversité** : Capitalise sur les forces de chaque modèle

4.6.2 Limitations

❌ **Complexité** : Plus difficile à implémenter et maintenir

❌ **Coût computationnel** : Entraînement beaucoup plus long

❌ **Interprétabilité** : Modèle "boîte noire"

❌ **Risque d'overfitting** : Si mal implémenté (sans CV)

❌ **Diminishing returns** : Gain marginal parfois faible

4.7 Formule Récapitulative

Stacking complet :

$$\begin{aligned}
&\text{Niveau 0: } x \in \mathbb{R}^p \\
&\text{Niveau 1: } \hat{y}_k = h_k(x), \quad k = 1, ..., K \\
&\text{Meta-features: } z = [\hat{y}_1, \hat{y}_2, ..., \hat{y}_K] \in \mathbb{R}^K \\
&\text{Niveau 2: } \hat{y}_{\text{final}} = g(z) = g(h_1(x), ..., h_K(x))
\end{aligned}$$

Avec CV (pour éviter overfitting) :

$$z_i = [h_1^{-fold(i)}(x_i), h_2^{-fold(i)}(x_i), ..., h_K^{-fold(i)}(x_i)]$$

où $h_k^{-fold(i)}$ = modèle k entraîné sans le fold contenant i

---

📊 Conclusion Générale

Récapitulatif des Stratégies

Méthode	Objectif	Force	Idéal pour
Random Forest	Réduire variance	Robustesse, simplicité	Baseline solide
XGBoost	Réduire biais	Haute précision	Compétitions, production
LightGBM	Optimisation vitesse	Très rapide	Grandes données
CatBoost	Gérer catégories	Pas de preprocessing	Données mixtes
Stacking	Combiner forces	Performance maximale	Compétitions, ensembles critiques

Arbre de Décision pour Choisir un Modèle

```
Ai-je beaucoup de données (>100k lignes) ?
├─ Oui → LightGBM ou XGBoost
└─ Non → Random Forest ou CatBoost

Ai-je des variables catégorielles ?
├─ Oui → CatBoost (ou encodage + XGBoost)
```

```
└─ Non → XGBoost ou LightGBM

Ai-je besoin de la meilleure performance possible ?
├─ Oui → Stacking de plusieurs modèles
└─ Non → XGBoost seul

Ai-je des contraintes de temps ?
├─ Entraînement → LightGBM
└─ Prédiction → Random Forest ou CatBoost
```

Équation Unificatrice

Tous ces modèles cherchent à minimiser :

$$\mathcal{L}_{\text{total}} = \underbrace{\sum_{i=1}^{n} L(y_i, \hat{y}_i)}_{\text{Erreur empirique}} + \underbrace{\Omega(\text{modèle})}_{\text{Régularisation}}$$

Mais diffèrent par :

- **Structure** : Bagging (parallèle) vs Boosting (séquentiel) vs Stacking (hiérarchique)
- **Optimisation** : Gradient descent ordre 1/2, échantillonnage, encodage
- **Régularisation** : L1/L2, profondeur, nombre de feuilles

Bonne modélisation ! 🚀