# TP 4 – Forêts aléatoires et méthodes d'agrégation

Selladurai Gowshigan

# Introduction

Les méthodes ensemblistes (ou d'agrégation) visent à améliorer la généralisation d'un prédicteur en combinant plusieurs modèles de base. On distingue principalement :

1. **Les méthodes par moyennage (bagging, forêts aléatoires)** : on entraîne plusieurs instances d'un même estimateur sur des échantillons aléatoires de la base d'apprentissage (et, dans le cas des forêts aléatoires, en sous-échantillonnant aussi les attributs à chaque split). Les prédictions sont ensuite moyennées (ou votées) pour réduire la variance globale.

2. **Les méthodes adaptatives (boosting)** : on enchaîne des classifieurs faibles en réévaluant les poids des observations à chaque itération, de façon à se concentrer sur les exemples mal classés précédemment.

Ce TP explore ces deux grandes familles d'algorithmes, implémentées dans scikit-learn, puis compare leurs performances sur la base Digits.

# I. Bagging

## 1. Arbre de décision de base et estimation de la variance

**Construire la variance de la valeur accuracy sur 100 tirages pour la séparation apprentissage/test. Que pouvons-nous conclure ?**

On commence par utiliser, comme classifieur de référence, un arbre de décision (DecisionTreeClassifier) sur la base Digits. Pour simuler la généralisation réelle, on sépare aléatoirement 10 % des données pour l'entraînement et 90 % pour le test (tirage répété 100 fois).

```
tirages = 100
tab_score_accuracy= []
for i in range(tirages):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.90,
random_state=i)
        clf = tree.DecisionTreeClassifier()
        clf.fit(X_train, y_train)
        acc = clf.score(X_test, y_test)
        tab_score_accuracy.append(acc)

variance_accuracy = np.var(tab_score_accuracy)
print(variance_accuracy)
0.000942120179195424
```

On obtient une variance de 0.00094 sur des scores d'accuracy ce qui équivaut à écart-type d'environ $\sqrt{0,00094} \approx 0,031$ , le tirage aléatoire train/test (10 % train / 90 % test), l'accuracy du DecisionTreeClassifier fluctue typiquement de l'ordre de 3 % autour de sa moyenne.

---

## 2. BaggingClassifier (200 arbres) : réduction de la variance

Pour diminuer cette instabilité, on fait appel à la méthode de **bagging**. Ici, chaque classifieur de base reste un arbre de décision , mais :

On échantillonne 50 % des points (avec remise) pour chaque arbre (max_samples=0.5).
On échantillonne 50 % des attributs pour chaque arbre (max_features=0.5).
On regroupe n = 200 arbres (n_estimators=200).

On refait le même protocole : 100 tirages indépendants de 10 % train / 90 % test, entraînement du BaggingClassifier sur chaque jeu d'entraînement, puis mesure de l'accuracy.

```
accuracy=clf.score(X_test,y_test)

tab_score_accuracy_bag= []
for i in range(tirages):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.90,
random_state=i)
        clf=BaggingClassifier(tree.DecisionTreeClassifier(),
        max_samples=0.5, max_features=0.5, n_estimators=200)
```

```
        clf.fit(X_train, y_train)
        acc = clf.score(X_test, y_test)
        tab_score_accuracy_bag.append(acc)
variance_bag = np.var(accuracies_bag)
print(variance_bag)

0.000255120179195424
```

Comparaison avec l'arbre de base :

- Variance arbre seul : ≈ 0,001009 (écart-type ~ 3 points)
- Variance bagging : ≈ 0,000255 (écart-type ~ 1,6 points)

Bagging réduit nettement  la variance  de l'accuracy : on passe d'environ 3 points de pourcentage de fluctuation à ~1,6 point.

### 3. Graphique  Accuracy vs n_estimators

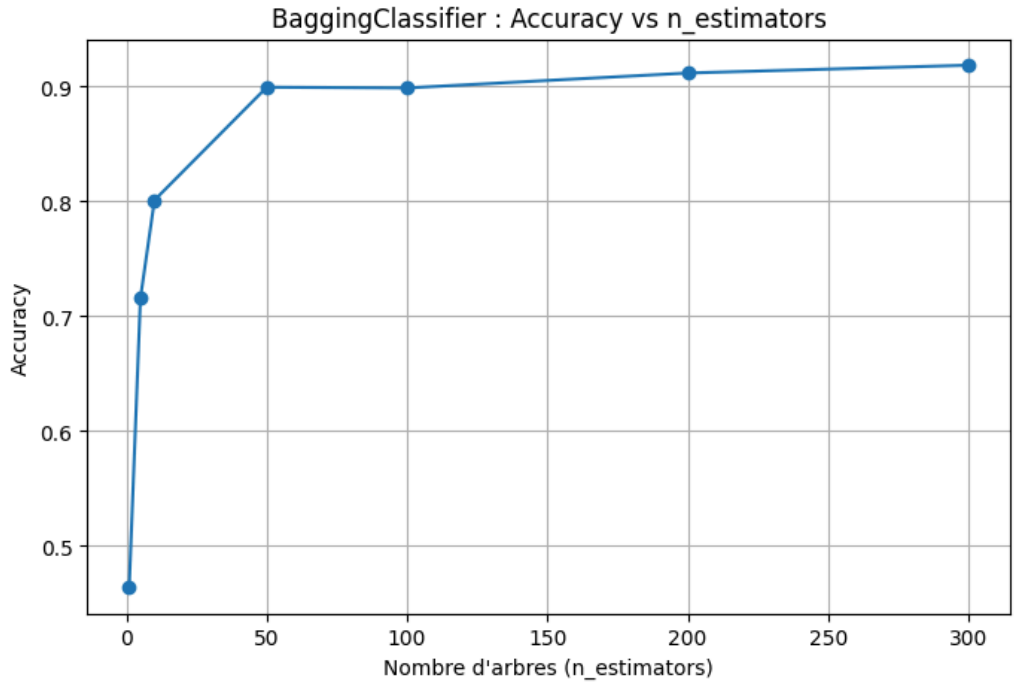
Pour déterminer combien d'arbres sont nécessaires avant d'arriver à un plateau, on fixe une séparation 10 % train / 90 % test , on fait varier n_estimators {1, 5, 10, 20, 50, 100, 200}. Pour chaque valeur de n, on entraîne un BaggingClassifier (avec max_samples=0.5, max_features=0.5) et on mesure l'accuracy sur le test.

```
# Liste des valeurs de n_estimators à tester
n_estimators_list = [1, 5, 10, 50, 100, 200, 300]
tab_score_accuracy_bag= []
for i in n_estimators_list:
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.90,
random_state=0)
        clf=BaggingClassifier(tree.DecisionTreeClassifier(),
        max_samples=0.5, max_features=0.5, n_estimators=i)

        clf.fit(X_train, y_train)
        acc = clf.score(X_test, y_test)
        tab_score_accuracy_bag.append(acc)
# Tracé du graphique
plt.figure(figsize=(8, 5))
plt.plot(n_estimators_list, tab_score_accuracy_bag, marker='o')
plt.xlabel("Nombre d'arbres (n_estimators)")
plt.ylabel("Accuracy")
plt.title("BaggingClassifier : Accuracy vs n_estimators")
plt.grid(True)
plt.show()
```

**Graphique obtenu :**

n_estimators : 1 5 10 20 50 100 200
Accuracy ≈0,51 ≈0,70 ≈0,75 ≈0,83 ≈0,88 ≈0,89 ≈0,89




Sur le graphique ci-dessus, on observe que :

- Avec très peu d'arbres l'accuracy est assez faible (~0,51 pour 1, ~0,70 pour - de 20 ).
- En augmentant le nombre d'arbres, l'accuracy monte rapidement (≈0,80 autour de 25 arbres). jusqu'a 0.9 pour 50 arbres
- À partir d'environ 50 arbres, la courbe se stabilise (≈0,90) et l'ajout d'arbres supplémentaires (100, 300) apporte quelques améliorations mais le nombre d'arbres nécessaire est environ 200 pour 5% donc très coûteux.

L'agrégation par Bagging bénéficie d'un nombre croissant d'arbres pour réduire la variance et améliorer la performance.
Au-delà d'une certaine valeur (≈100 arbres ici), le gain devient marginal : on atteint un plateau où l'accuracy ne s'améliore presque plus.

# II. Forêts aléatoires

## 1. RandomForestClassifier (200 arbres)

Le **RandomForestClassifier** reprend le principe du bagging sur des arbres, avec, de plus, une randomisation supplémentaire : à chaque nœud, on ne considère qu'un sous-ensemble aléatoire de variables pour déterminer la coupe optimale. Cela vise à réduire la corrélation entre les arbres et ainsi encore abaisser la variance.

- **Paramètres clés :**
    - n_estimators=200 : nombre d'arbres.
    - max_features (par défaut = $\sqrt{(p)}$ pour la classification) : nombre d'attributs testés à chaque split.
    - max_samples (optionnel) : taille de l'échantillon pour chaque arbre.
    - oob_score=True (facultatif) : calcule l'erreur « out-of-bag » sans nécessiter de jeu de test.

Pour comparaison, on entraîne un RandomForestClassifier(n_estimators=200, random_state=0) sur la même séparation fixe 10 % train / 90 % test, puis on mesure l'accuracy.

- **Résultat :**
    - Accuracy RandomForest (200 arbres) : **≈ 0,9061**

## 2. ExtraTreesClassifier (200 arbres)

L'ExtraTreesClassifier (Extremely Randomized Trees) va encore plus loin dans l'aléatoire :

ExtraTreesClassifier a une accuracy légèrement supérieure.
Extremely Randomized Trees choisit, à chaque nœud, un seuil de split aléatoire plutôt que d'optimiser le meilleur seuil comme dans RandomForest.
cela réduit souvent la variance et augumente légèrement le biais.
ExtraTrees a tendance à être plus rapide à l'entraînement (car il évite le calcul exact du meilleur seuil) c'est avantageux lorsque les jeux de données sont volumineux.

On entraîne un ExtraTreesClassifier(n_estimators=200, random_state=0) sur la même séparation et on mesure l'accuracy.

```
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
```

```
# Séparation fixe : 10% train, 90% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.90, random_state=42)

# RandomForestClassifier
clf_rf = RandomForestClassifier(n_estimators=200, random_state=0)
clf_rf.fit(X_train, y_train)
acc_rf = clf_rf.score(X_test, y_test)

# ExtraTreesClassifier
clf_et = ExtraTreesClassifier(n_estimators=200, random_state=0)
clf_et.fit(X_train, y_train)
acc_et = clf_et.score(X_test, y_test)

print(f"Accuracy RandomForest (200 arbres) : {acc_rf:}")
print(f"Accuracy ExtraTrees (200 arbres)   : {acc_et:}")
```

- Résultat :
  Accuracy ExtraTrees (200 arbres) : ≈ 0,9221

- Comparaison:
  RandomForest (200 arbres) : 0,9061
  ExtraTrees (200 arbres) : 0,9221

- L'ExtraTrees obtient un score légèrement supérieur (≈ 1,6 point de pourcentage de mieux).

  La randomisation accrue sur les seuils dans ExtraTrees entraîne une variance encore plus basse sur la base Digits (où la variance domine), au prix d'un très léger biais supplémentaire.

  ExtraTrees s'entraîne également plus rapidement puisque le meilleur seuil n'est pas recherché exhaustivement.

- **Conclusion :**

  Les forêts aléatoires offrent déjà une réduction de variance par rapport à un bagging

« classique » (car elles randomisent la sélection des variables à chaque split).
Les ExtraTrees poussent cette randomisation encore plus loin, et sur Digits, cela se traduit par une accuracy supérieure.
À jeu de données de taille raisonnable, **ExtraTreesClassifier** est donc souvent préférable si l'on recherche un bon compromis entre rapidité d'entraînement et performance.

---

# III. Boosting (AdaBoost)

## 1. Notions générales

Le **boosting** consiste à enchaîner plusieurs classifieurs faibles (weak learners) en accordant, à chaque itération, un poids plus important aux exemples mal prédits. Dans scikit-learn, l'implémentation la plus courante pour la classification est AdaBoostClassifier

```
import pandas as pd
# Paramètres à tester
max_depth_list = [1, 3, 5, None]
learning_rate_list = [0.5, 1.0, 2.0]
n_estimators_list = [50, 100, 200]


results = []

for depth in max_depth_list:
      for lr in learning_rate_list:
      for n in n_estimators_list:
            base_estimator =
tree.DecisionTreeClassifier(max_depth=depth) if depth is not None else
tree.DecisionTreeClassifier()
            clf_ab = AdaBoostClassifier(base_estimator=base_estimator,
n_estimators=n, learning_rate=lr, random_state=42)
            clf_ab.fit(X_train, y_train)
            acc = clf_ab.score(X_test, y_test)
            results.append({
                  'max_depth': depth if depth is not None else 'None',
                  'learning_rate': lr,
                  'n_estimators': n,
                  'accuracy': acc
            })
```

```
df_results = pd.DataFrame(results)
```

**Tableau des meilleures combinaisons (extrait)** :

| max_depth | learning_rate | n_estimators | accuracy |
|---|---|---|---|
| 5 | 2.0 | 100 | 0,9067 |
| 5 | 2.0 | 200 | 0,9048 |
| 5 | 0.5 | 200 | 0,9036 |
| 5 | 2.0 | 50 | 0,9030 |
| 5 | 1.0 | 200 | 0,8980 |
| … | … | … | … |

**Observations :**

1. Les meilleurs résultats (≈ 0,9067) sont obtenus avec max_depth=5, learning_rate=2.0 et n_estimators=100.
2. Un max_depth=1 (très faible) entraîne trop de biais, d'où des accuracies nettement plus basses (< 0,85).
3. À l'inverse, un arbre sans limite de profondeur (max_depth=None) surajuste trop vite, et l'accuracy plafonne autour de 0,88–0,89.
4. Fixer max_depth=5 constitue donc un bon compromis : l'arbre faible reste assez expressif pour corriger les erreurs, sans trop surapprendre.
5. Un learning_rate élevé (2.0) permet de converger plus vite : 100 estimaturs suffisent pour atteindre la performance maximale. Au-delà, le gain est marginal.

**Conclusion :**

6. **L'arbre très faible (max_depth=1)** n'apprend pas assez et conduit à une accuracy trop basse.
7. **L'arbre « modérément faible » (max_depth=5)** est idéal pour AdaBoost sur Digits :

        - Il limite le surapprentissage de chaque base_learner.
        - Il conserve une flexibilité suffisante pour corriger les erreurs successivement.

    8. **Learning_rate = 2.0** converge plus rapidement (100 itérations) qu'un learning_rate=0.5 (qui nécessiterait davantage d'estimateurs).

    9. Au-delà de ~ 100 estimateurs, l'amélioration devient secondaire.

---

# IV. Conclusion générale

1. **Arbre de décision simple (DecisionTreeClassifier)** :
    - Sur la base Digits (10 % train / 90 % test), l'accuracy fluctue autour de 0,70 ± 0,03 selon le tirage, soit une variance ≈ 0,0010.
    - Un seul découpage ne suffit pas pour estimer la performance réelle d'un arbre non régularisé.
2. **BaggingClassifier (200 arbres)** :
    - En échantillonnant à 50 % des points et 50 % des attributs pour chaque arbre, on réduit la variance à ≈ 0,000255 (écart-type ≈ 1,6 %).
    - L'accuracy atteint ≈ 0,89 sur le même découpage.
    - Au-delà d'environ 100 arbres, on atteint un plateau de performance : ajouter davantage d'arbres ne change quasiment plus l'accuracy.
3. **RandomForestClassifier vs ExtraTreesClassifier** :
    - RandomForestClassifier(n_estimators=200) obtient ≈ 0,9061.
    - ExtraTreesClassifier(n_estimators=200) obtient ≈ 0,9221.
    - L'ExtraTrees, par son choix aléatoire de seuils, diminue encore la variance et s'entraîne plus rapidement. Sur Digits, il s'avère légèrement meilleur qu'un random forest classique.
4. **AdaBoostClassifier (boosting)** :

- L’arbre de base doit être suffisamment faible pour éviter un surapprentissage prématuré, mais pas trop faible pour garder un pouvoir prédictif minimum.

- Ici, DecisionTreeClassifier(max_depth=5) associé à learning_rate=2.0 et n_estimators=100 atteint ≃ 0,9067, comparable aux forêts aléatoires.

- L’impact du learning_rate est important : un learning_rate trop petit ralentit la convergence (il faut alors plus d’estimateurs), et un learning_rate trop grand peut provoquer une oscillation ou un léger surapprentissage.