TP 4
Les forêts aléatoires

Méthodes d'agrégation

Les méthodes ensemblistes (ou d'agrégation) pour les algorithmes d'apprentissage statistique (en anglais : *ensemble learning*) sont basées sur l'idée de combiner les prédictions de plusieurs prédicteurs (ou classifieurs) pour une meilleure généralisation et pour compenser les défauts éventuels de prédicteurs individuels.
En général, on distingue deux familles de méthodes de ce type :

1. Méthodes par moyennage (*bagging*, forêts aléatoires) où le principe est de faire la moyenne de plusieurs prédictions en espérant un meilleur résultat suite à la réduction de variance de l'éstimateur moyenne.
2. Méthodes adaptatives (*boosting*) où les paramètres sont itérativement adaptés pour produire un meilleur mélange.

Dans la suite nous explorerons chacune de ces classes d'algorithme en Scikit-learn et présenterons quelques comparaisons.

Bagging

Les méthodes de type *bagging* construisent plusieurs instances d'un estimateur, calculées sur des échantillons aléatoires tirés de la base d'apprentissage (et éventuellement une partie des attributs, également sélectionnés de façon aléatoire), et ensuite combine les prédictions individuelles en réalisant leur moyenne pour réduire la variance de l'estimateur. Leur avantage principal réside dans le fait qu'ils construisent une version améliorée de l'algorithme de base, sans demander de modification de cet algorithme. Le prix à payer est un coût de calcul plus élevé. Comme elles réduisent le sur-apprentissage, les méthodes *bagging* fonctionnent très bien avec des prédicteurs « forts ». Par contraste, les méthodes *boosting* sont mieux adaptées à des prédicteurs faibles (*weak learners*).
Dans Scikit-learn, les méthodes de *bagging* sont implémentées via la classe `BaggingClassifier` et `BaggingRegressor`. Les constructeurs prennent en paramètres un estimateur de base et la stratégie de sélection des points et attributs :

- `base_estimator` : optionnel (default=None). Si None alors l'estimateur est un arbre de décision.
- `max_samples` : la taille de l'échantillon aléatoire tiré de la base d'apprentissage.
- `max_features` : le nombre d'attributs tirés aléatoirement.
- `bootstrap` : boolean, optionnel (default=True). Tirage des points avec remise ou non.
- `bootstrap_features` : boolean, optionnel (default=False). Tirage des attributs avec remise ou non.
- `oob_score` : boolean. Estimer ou non l'erreur de généralisation OOB (*Out of Bag*).

Le code suivant construit un ensemble des classifieurs. Chaque classifieur de base est un `KNeighborsClassifier` (c'est-à-dire k-plus-proches-voisins), chacun utilisant au maximum 50% des points pour son apprentissage et la moitié des attributs (*features*) :

```
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
bagging = BaggingClassifier(KNeighborsClassifier(), max_samples=0.5,
max_features=0.5)
```

Dans cet exemple nous allons utiliser la base de données digits, qui contient 10 classes (images des chiffres en écriture manuscrite). Il y a 1797 éléments, chaque élément a 64 attributs (8 pixels par 8).

```
from sklearn.datasets import load_digits
digits = load_digits()

# Affichage des 10 premières images
import matplotlib.pyplot as plt
fig = plt.figure()
for i, digit in enumerate(digits.images[:10]):
    fig.add_subplot(1,10,i+1)
    plt.imshow(digit)
plt.show()
```

Pour ce TP, nous allons utiliser comme classifieur de base un arbre de décision DecisionTreeClassifier. Ce classifieur nous permet d'établir des performances de référence (c'est un ensemble à 1 modèle).

```
import numpy as np
from sklearn import tree
from sklearn.ensemble import BaggingClassifier

X, y = digits.data, digits.target
clf = tree.DecisionTreeClassifier()
clf.fit(X, y)
accuracy = clf.score(X,y)
print(accuracy)
```

Sur la base d'apprentissage accuracy = 1 . Pour plus de réalisme, découpons la base de données en un jeu d'apprentissage et un je de test afin de voir le comportement de généralisation de l'arbre sur des données différentes des celles d'apprentissage :

```
from sklearn.model_selection import train_test_split
# 90% des données pour le test, 10% pour l'apprentissage
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.90)

clf = tree.DecisionTreeClassifier()
clf.fit(X_train, y_train)

Z = clf.predict(X_test)
accuracy = clf.score(X_test,y_test)
print(accuracy)
```

Question :

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

Pour comparer, construisons mainntenant un classifieur *bagging* sur nos données, toujours basé sur les DecisionTreeClassifier :

```
clf = BaggingClassifier(tree.DecisionTreeClassifier(), max_samples=0.5, max_features=0.5, n_estimators=200)
```

L'apprentissage et l'évaluation de cet ensemble se font de la façon habituelle :

```
clf.fit(X_train, y_train)
Z = clf.predict(X_test)
accuracy=clf.score(X_test,y_test)
```

Question :

Calculer la variance de la valeur `accuracy` sur 100 tirages pour la séparation apprentissage/test. Comparer avec la variance du classifieur de base. Que pouvons-nous conclure ?

Question :

Construire le graphique `accuracy` vs `n_estimators`. Que constatez-vous ?

Question :

Faites varier les parametres `max_samples` et `max_features`. Pour quelles valeurs on obtient le meilleur résultat ? On pourra notamment utiliser `GridSearchCV` pour réaliser une recherche systématique.

Forêts aléatoires

L'algorithme des forêts aléatoires propose une optimisation des arbres de décision. Il utilise le même principe que le *bagging*, mais avec une étape supplémentaire de randomisation dans la sélection des attributs des nœuds dans le but de réduire la variance de l'estimateur obtenu. Les deux objets Python qui implémentent les forêts aléatoires sont `RandomForestClassifier` et `RandomForestRegressor`.
Les paramètres les plus importants sont :

- `n_estimators` : integer, optional (default=10). Le nombre d'arbres.
- `max_features` : le nombre d'attributs à considérer à chaque split.
- `max_samples` : la taille de l'échantillon aléatoire tiré de la base d'apprentissage.
- `min_samples_leaf` : le nombre minimal d'éléments dans un nœud feuille.
- `oob_score` : boolean. Estimer ou non l'erreur de généralisation OOB (*Out of Bag*).

Par la suite nous allons refaire la classification sur la base Digits en utilisant un classifieur `RandomForestClassifier`. Comme d'habitude, on sépare les données en gardant 10% pour l'apprentissage et 90% pour le test.

```
digits = load_digits()
X, y = digits.data, digits.target

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.90)
```

On peut désormais créer et entraîner notre modèle :

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators=200)
clf.fit(X_train, y_train)
```

Puis réaliser les prédictions et calculer le score de test :

```
y_pred = clf.predict(X_test)
accuracy = clf.score(X_test,y_test)
print(accuracy)
```

Question :

Comment la valeur de la variable `accuracy` se compare avec le cas *bagging* qui utilise le même nombre d'arbres (200 dans notre cas) ?

Question :

Construire la variance de la valeur accuracy sur 100 tirages pour la séparation apprentissage/test. Que pouvons-nous conclure en comparant avec la séction précedente (*bagging*) ?

Question :

Construire le graphique accuracy vs n_estimators. Que constatez-vous ? A partir de quelle valeur on n'améliore plus ?

Question :

Regardez dans la documentation les *ExtraTreesClassifier* et refaites la classification avec ce type de classifieur. Comparez avec *RandomForestClassifier*.

Boosting

Le principe du *boosting* est d'évaluer une séquence de classifieurs faibles (*weak learners*) sur plusieurs versions légèrement modifiées des données d'apprentissage. Les décisions obtenues sont alors combinées par une somme pondérée pour obtenir le modèle final.
Avec scikit-learn, c'est la classe AdaBoostClassifier qui implémente cet algorithme. Les paramètres les plus importants sont :

- n_estimators : integer, optional (default=10). Le nombre de classifieurs faibles.
- learning_rate : contrôle la vitesse de changement des poids par itération.
- base_estimator : (default=DecisionTreeClassifier) le classifieur faible utilisé.

Dans la suite nous allons refaire la classification sur la base Digits en utilisant un classifieur RandomForestClassifier :

```
from sklearn.ensemble import AdaBoostClassifier

digits = load_digits()
X, y = digits.data, digits.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.90)

# AdaBoost basé sur 200 arbres de décision
clf                                                                    =
AdaBoostClassifier(base_estimator=tree.DecisionTreeClassifier(max_depth=5),
n_estimators=200, learning_rate=2)
clf.fit(X_train, y_train)
accuracy = clf.score(X_test, y_test)
print(accuracy)
```

Question :

Le paramètre max_depth contrôle la profondeur de l'arbre. Essayez plusieurs valeurs pour voir l'impact de l'utilisation d'un classifieur faible vs plus fort (max_depth élevé ou éliminer le paramètre). Testez aussi l'effet du paramètre learning_rate et le nombre de classifieurs.