

Méthodes d'ensemble

TP 2 – Adaboost en pratique

AdaBoost est particulièrement sensible aux valeurs aberrantes. Les valeurs aberrantes sont souvent mal classées par les apprenants faibles (les arbres). Rappelons qu'AdaBoost augmente le poids des exemples mal classés. Cela signifie que le poids attribué aux valeurs aberrantes continue d'augmenter. Lorsque le prochain apprenant faible est formé, il continue à mal classer la valeur aberrante, auquel cas AdaBoost augmentera encore son poids, ce qui, à son tour, amène les apprenants faibles suivants à toujours mal classer et continuer à augmenter son poids.

Nous visualisons ce comportement dans l'exemple ci-dessous, où nous observons que les valeurs aberrantes forcent AdaBoost à consacrer une quantité disproportionnée d'efforts à des exemples d'apprentissage bruyants. Autrement dit, les valeurs aberrantes ont tendance à confondre AdaBoost et à le rendre moins robuste.

Exercice 1 – Création du modèle Adaboost

1. Importer les librairies nécessaires

```
import matplotlib.pyplot as plt
%matplotlib inline

#from visualization import plot_2d_data, plot_2d_classifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

import numpy as np
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
```

2. Importer les données avec la fonction make_moons, et ajouter un outlier

```
# Generate some data
X, y = make_moons(n_samples=500, noise=0.02)

# Add an outlier
X = np.concatenate((X, [[-1.15, 0.8]]))
y = np.concatenate((y, [1]))
# Convert to -1/+1 labels from 0/1 labels
y = 2 * y - 1
```

3. Visualiser les données

```
# Separate data points based on their class label
class_1 = X[y == -1]
class_2 = X[y == 1]

#subplots
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(12, 5))

# Create a scatter plot
ax.scatter(class_1[:, 0], class_1[:, 1], label='Class -1', c='blue', marker='o')
ax.scatter(class_2[:, 0], class_2[:, 1], label='Class 1', c='red', marker='x')

# Plot the outlier
ax.scatter(-1.15, 0.8, label='Outlier', c='green', marker='s')
```

4. Implémenter un Adaboost : initialisation des variables

```
#initialize variables
n_samples, n_features = X.shape
n_estimators = 20           #number of trees
D = np.ones((n_samples, ))   # Initialize example weights
ensemble = []                # Initialize an empty ensemble

fig, ax = plt.subplots(nrows=2, ncols=4, figsize=(16, 8))

# Plot the data set
ax[0, 0].scatter([-1.15], [0.8], marker='o', s=200, c='w', edgecolors='k')
| axis_index = 0

# Create a scatter plot
ax[0, 0].scatter(class_1[:, 0], class_1[:, 1], label='Class -1', c='blue', marker='o')
ax[0, 0].scatter(class_2[:, 0], class_2[:, 1], label='Class 1', c='red', marker='x')

# Plot the outlier
ax[0, 0].scatter(-1.15, 0.8, label='Outlier', c='green', marker='s')
```

5. Implémenter un Adaboost : implémentation de l'algorithme à la main

```
for t in range(n_estimators):
    D = D / np.sum(D)          # Normalize the sample weights

    # -- Plot the training examples in different sizes proportional to their weights
    s = D / np.max(D)
    s[(0.00 <= s) & (s < 0.25)] = 2
    s[(0.25 <= s) & (s < 0.50)] = 16
    s[(0.50 <= s) & (s < 0.75)] = 64
    s[(0.75 <= s) & (s <= 1.00)] = 128

    if t in [0, 1, 2, 4, 7, 11, 14]:
        axis_index += 1
        r, c = np.divmod(axis_index, 4)
        title = 'Iteration {}: Sample weights'.format(t + 1)

        # Create a scatter plot
        ax[r, c].scatter(class_1[:, 0], class_1[:, 1], s=s[y == -1], label='Class -1', c='blue', marker='o')
        ax[r, c].scatter(class_2[:, 0], class_2[:, 1], s=s[y == 1], label='Class 1', c='red', marker='x')

        # Plot the outlier
        ax[r, c].scatter(-1.15, 0.8, label='Outlier', c='green', marker='s')

        #plot_2d_data(ax[r, c], X, y, alpha=0.75, s=s, title=title, colormap='RdBu')
        ax[r, c].set_xticks([])
        ax[r, c].set_yticks([])

    ##### MODELLING
    h = DecisionTreeClassifier(max_depth=1, criterion = 'entropy') # Initialize a decision stump
    h.fit(X, y, sample_weight=D)                                     # Train a weak learner using sample weights
    ypred = h.predict(X)                                           # Predict using the weak learner

    #calculating error and new weights
    e = 1 - accuracy_score(y, ypred, sample_weight=D) # Weighted error of the weak learner
    a = 0.5 * np.log((1 - e) / e) # Weak learner weight

    #update sample weights
    m = (y == ypred) * 1 + (y != ypred) * -1 # Identify correctly classified and misclassified points
    D *= np.exp(-a * m) # Update the sample weights

    ensemble.append((a, h)) # Save the weighted weak hypothesis

fig.tight_layout()
```

Exercice 2 – Prunning

Objectif : Déterminer le nombre d'arbres optimaux et appliquer les critères d'élagage

1. Hyperparamètres à optimiser

```
from sklearn.datasets import load_breast_cancer
X, y = load_breast_cancer(return_X_y=True)
%matplotlib inline

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import StratifiedKFold
import numpy as np

#hyperparameters to optimize
n_estimator_steps, n_folds = 5, 10
number_of_stumps = np.arange(5, 50, n_estimator_steps)
splitter = StratifiedKFold(n_splits=n_folds, shuffle=True)

#vector for saving results
trn_err = np.zeros((len(number_of_stumps), n_folds))
val_err = np.zeros((len(number_of_stumps), n_folds))
```

2. Définition de l'apprenant faible

Quel est la profondeur de cet arbre ? pourquoi est avantageux d'avoir cette profondeur ?

```
#Decision stump for the modelling
stump = DecisionTreeClassifier(max_depth=1)
```

3. Entrainement avec plusieurs combinaisons des hyperparamètres à tester

Tester la performance avec un learning_rate = {0.5, 0.1}

```
#looping on several values of number of models and splitting
for i, n_stumps in enumerate(number_of_stumps):
    for j, (trn, val) in enumerate(splitter.split(X, y)):

        #Initialize Adaboost model
        model = AdaBoostClassifier(algorithm='SAMME', base_estimator=stump,
                                    n_estimators=n_stumps, learning_rate=1.0)
        model.fit(X[trn, :], y[trn])

        #calculating error
        trn_err[i, j] = 1 - accuracy_score(y[trn],
                                             model.predict(X[trn, :]))
        val_err[i, j] = 1 - accuracy_score(y[val],
                                             model.predict(X[val, :]))

trn_err = np.mean(trn_err, axis=1)
val_err = np.mean(val_err, axis=1)
```

4. Visualisation des résultats

Quelle est la meilleure profondeur ?

```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(4, 4))

ax.plot(number_of_stumps, trn_err, marker='o', c=cm[0], markeredgecolor='w', linewidth=2)
ax.plot(number_of_stumps, val_err, marker='s', c=cm[1], markeredgecolor='w', linewidth=2)
ax.legend(['Train err', 'Validation err'])
ax.set_xlabel('Number of decision stumps')
ax.set_ylabel('Error (%)')

fig.tight_layout()
```