Td : Algorithmes de classification

Introduction

Les notions de reconnaissance des formes abordées en cours sont illustrées dans ce TD à travers la librairie python `sklearn`. L'annexe 1 est un programme permettant de comparer les performances de plusieurs classifieurs.

Le but de ce TD est d'illustrer le fonctionnement de trois classifieurs étudiés en cours :
- La classification Bayesienne,
- Les K plus proches voisins
- Le classifier Adaboost

En fonction du temps restant, vous pourrez tester d'autres classifieurs disponibles dans la bibliothèque (SVM, Arbre de décision, Réseau de neurones).

1 Compréhension du programme exemple

Le programme utilisé dans ce Tp est un des exemples de la librairie `sklearn`. son but est de générer un jeux de données de synthèse (2 classes), puis de comparer les performances de plusieurs classifieurs, sur les données générées.

Le tableau `classifiers` contient les différents types de classifiers à tester. Les trois classifiers que nous allons comarer sont `AdaBoostClassifier` pour la méthode Adaboost, `KNeighborsClassifier` pour la méthode des K plus proches voisins et `GaussianNB` pour la classification Bayesienne.

Le tableau `datasets` contient les bases de données des différents scénarios à tester. Les trois générateurs de données utilisées ici sont : `make_classification`, `make_moons`, `make_circles`.

Les tableaux `classifiers` et `datasets` peuvent être modifiés en fonction du nombre de classifieurs à étudier et des datasets que l'on souhaite générer.

On affiche, en bas à droite de chaque résultat, les performances du classifieur sur la base d'apprentissage et sur la base de test.

1. Executer le programme et analyser l'affichage généré. Vous noterez les frontières de décision et le taux de bonne classification obtenue par chaque méthode
2. Modifier le programme pour afficher uniquement les performances de classification pour le dataset généré par `make_classification`
3. Modifier le programme pour afficher uniquement les performances du classifier de type `Adaboost` sur les trois datasets.

2 La classification Bayesienne naïve

Ce type de classification consiste à considérer que le problème peut être formalisé sous la forme de lois de probabilités. Connaissant une observation, on cherche à déterminer la

classe associée en fonction des probabilités à priori et d'une fonction de vraisemblance (*likelihood*). Ce type de classifier s'appelle avec la fonction `GaussianNB`. Cette fonction prend en paramètre les *a priori* de chaque classe ($priors = [0.5, 0.5]$ pour une equi-probabilité par exemple).

- Soit $\{\omega_1, \omega_2, ...\omega_c\}$ un ensemble de c classes et $\mathbf{x}$ un vecteur de caractéristiques.
- Pour chaque classe ω_i on suppose connaître :
  - $P(\omega_i)$: la probabilité a priori de cette classe,
  - $p(\mathbf{x}|\omega_i)$: la densité de probabilité de $\mathbf{x}$ conditionnée par cette classe

La règle de Bayes permet de calculer la probabilité d'une classe a posteriori, c'est-à-dire conditionnée par l'observation de x, soit

$$P(\omega_i|\mathbf{x}) = \frac{p(\mathbf{x}|\omega_i)P(\omega_i)}{p(\mathbf{x})}$$

avec :

$$p(\mathbf{x}) = \sum_i (p(\mathbf{x}|\omega_i).P(\omega_i))$$

Dans une approche dite naïve, ce type de classification s'applique en modélisant les distributions de probabilités par des lois normales : $N(\boldsymbol{\mu}, \Sigma)$:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}\,|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})'\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right]$$

avec $\boldsymbol{\mu}$: vecteur moyen

$$\boldsymbol{\mu} = E[\mathbf{x}] = \int \mathbf{x} p(\mathbf{x}) d\mathbf{x}$$

$$\mu_i = E[x_i]$$

et Σ : matrice de covariance

$$\Sigma = E[(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})']$$

$$\Sigma = \int (\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})' p(\mathbf{x}) dx$$

$$\sigma_{ij} = E[(x_i - \mu_i)(x_j - \mu_j)]$$

Dans le cas discret :

$$\sigma_{ij} = \frac{1}{N}\sum_x (x_i - \mu_i)(x_j - \mu_j)$$

La création d'un classifier Bayesien passe par la fonction `GaussianNB` de la librairie sklearn.

1. Pour une équi-probabilité ($priors = [0.5, 0.5]$), Observer, pour les datasets `make_moons` et `make_circles`, les performances de classification lorsque le bruit augmente (variable `noise`).
2. Observer l'évolution de la frontière de décision pour les jeux de probabilités à priori suivants : $priors = [0.5, 0.5]$, $priors = [0.1, 0.9]$, $priors = [0.9, 0.1]$.

3 Les K plus proches voisins

Il s'agit d'une méthode de classification non paramètrique. Le principe de base est le suivant : on cherche à estimer $p(\mathbf{x}|\omega_i)$ ou $p(\omega_i|\mathbf{x})$. Soit $\mathcal{D}$ un domaine inclu dans l'espace des attributs ($\mathcal{D} \subset I\!R^d$) et pouvant être considéré comme un voisinage du vecteur de paramétres $\mathbf{x}$ pour lequel on cherche $p(\mathbf{x}|\omega_i)$. Si on fait l'Hypothèse que $p(\mathbf{x}|\omega_i) \approx$ const. sur $\mathcal{D}$ alors :

$$p(\mathbf{x} \in \mathcal{D}) = \int_{\mathcal{D}} p(\mathbf{x}'|\omega_i) d\mathbf{x}' \approx p(\mathbf{x}|\omega_i) \int_{\mathcal{D}} d\mathbf{x}'$$

Si $V(\mathcal{D})$ est l'hypervolume de $\mathcal{D}$:

$$p(\mathbf{x} \in \mathcal{D}) \approx p(\mathbf{x}|\omega_i) V(\mathcal{D})$$

Finalement :

$$\forall \mathbf{x} \in \mathcal{D} p(\mathbf{x}|\omega_i) \approx \frac{p(\mathbf{x} \in \mathcal{D})}{V(\mathcal{D})}$$

Si on dispose de t échantillons, sur n au total, se trouvant dans le domaine $\mathcal{D}$, alors :

$$p(\mathbf{x} \in \mathcal{D}) \approx \frac{t}{n}$$

Il est aussi possible de montrer que la probabilité *a posteriori* peut être estimée par :

1. Observer l'évolution de la frontière de décision et des performances du classifieur de type KPPV pour 1, 2, 3, 5 voisins.
2. Observer le comportement du classifier (performances sur la base d'apprentissage et sur la base de test)lorsque l'on bruite les données générées (variable `noise` sur la base `make_circles`). On testera $noise = 0.2, 0.5, 0.9$.

4 Adaboost

Les méthodes de boosting sont des algorithmes d'apprentissage supervisé, i.e. qui utilisent une base d'apprentissage labellisée. Notons $\mathcal{S} \doteq \{(\mathbf{x}_i, y_i)\}_{i=1,\dots,N}$ cette base où $\mathbf{x}_i \in I\!R^n$ est un vecteur de données représentant un exemple d'apprentissage et $y_i \in \{-1, 1\}$ est le label associé à $\mathbf{x}_i$ (généralement, le label 1 représente la classe des positifs et le label -1 la classe des négatifs). Le but de ces méthodes est de construire un classifieur $H(\mathbf{x}) : \mathbb{R}^n \to \{-1, 1\}$ permettant d'associer un label à un exemple inconnu $\mathbf{x}$. Les méthodes issues du boosting se basent sur l'observation suivante : il est rare d'avoir à sa disposition un expert omniscient permettant de prendre la meilleure décision et par conséquent, on a plutôt recours à un comité d'experts plus ou moins compétents pour ensuite combiner leurs avis et prendre une décision. De manière étonnante, des recherches en apprentissage artificiel datant du début des années 90 montrent qu'il est possible d'atteindre une décision aussi précise que souhaitée par une combinaison judicieuse d'experts imparfaits mais correctement entraînés. Plusieurs algorithmes d'apprentissage ont été développés à la suite de ces travaux. Les méthodes issues du boosting sont des méthodes capables de générer des règles de décision précises en utilisant des règles de décision produites par des classifieurs faibles, i.e. des classifieurs ayant un taux de réussite un peu meilleur que le hasard.

Le premier algorithme de boosting a été proposé par Schapire en 1990 et permet d'obtenir un classifieur après avoir entraîné un classifieur faible sur trois sous-ensembles d'apprentissage.

En 1997, Freund et Schapire proposent une amélioration de l'algorithme proposé par Schapire en 90 qui est aujourd'hui très utilisé : l'algorithme *Adaboost*. Nous présentons ici l'algorithme Adaboost dit *discret*. L'algorithme, présenté dans l'algorithme, se base toujours sur l'idée d'utiliser un comité d'experts pour prendre une décision et rajoute deux autres idées :

1. La pondération adaptative des votes par une technique de mise à jour multiplicative ;
2. La modification de la distribution des exemples disponibles pour entraîner chaque expert, en surpondérant au fur et à mesure les exemples mal classés.

Algorithme 1: L'algorithme Adaboost Discret

Entrées : $\mathcal{S} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ une base d'apprentissage.
Sorties : $\text{sign}(H(\mathbf{x}))$ où $H(\mathbf{x})$ est un classifieur fort entraîné sur $\mathcal{S}$.

$\omega_1(i) = 1/N, \quad \forall i = 1, \ldots, N$;
pour $t = 1$ **a** T **faire**
  Normaliser les poids $\omega_t(i) = \dfrac{\omega_t(i)}{\sum_{i=1}^{N} \omega_t(i)}$;
  Estimer un classifieur faible $h_t(\mathbf{x}) : \mathbb{R}^n \to \{-1, 1\}$ sur $\mathcal{S}$ en utilisant les poids ω_t ;
  Calculer l'erreur d'apprentissage $\epsilon_t = \sum_{i=1}^{N} \omega_t(i) \mathbb{1}_{\{y_i \neq h_t(\mathbf{x}_i)\}}$;
  Calculer $\alpha_t = \ln\left(\dfrac{1-\epsilon_t}{\epsilon_t}\right)$;
  Mise à jour des poids : $w_{t+1}(i) = \omega_t(i) e^{\alpha_t \mathbb{1}_{\{y_i \neq h_t(\mathbf{x}_i)\}}}$;
fin pour
Le classifieur final est donné par : $\text{sign}(H(\mathbf{x})) = \text{sign}\Big(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})\Big)$;

La création d'un classifieur de type Adaboost passe par la fonction `AdaBoostClassifier` de la librairie sklearn.

1. Comparer le comportement et les performaces d'un classifieur Adaboost pour 1, 2, 4, 10, 50, 500 classifieurs faibles.
2. Mettre en évidence le phénomène de sur-apprentissage en augmentant le nombre de classifieurs faibles et en observant l'évolution du taux de bonnes classifications sur la base d'apprentissage et sur la base de test.

5 Annexe : programme Python (exemple issu de sklearn) permettant de comparer le comportement de plusieurs types de classifeurs

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""
=====================
Classifier comparison
=====================

A comparison of a several classifiers in scikit-learn on synthetic datasets.
The point of this example is to illustrate the nature of decision boundaries
of different classifiers.
This should be taken with a grain of salt, as the intuition conveyed by
these examples does not necessarily carry over to real datasets.

Particularly in high-dimensional spaces, data can more easily be separated
linearly and the simplicity of classifiers such as naive Bayes and linear SVMs
might lead to better generalization than is achieved by other classifiers.

The plots show training points in solid colors and testing points
semi-transparent. The lower right shows the classification accuracy on the test
set.
"""
print(__doc__)


# Code source: Gaël Varoquaux
#              Andreas Müller
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.kernel_ridge import KernelRidge
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
```

```
h␣=␣.02␣␣#␣step␣size␣in␣the␣mesh
"""
names␣=␣["Nearest␣Neighbors",␣"Linear␣SVM",␣"RBF␣SVM",␣"Gaussian␣Process",
"Decision␣Tree",␣"Random␣Forest",␣"Neural␣Net",␣"AdaBoost",
"Naive␣Bayes",␣"QDA"]
"""
names␣=␣[␣"Adaboost","Nearest␣Neighbors:␣3","Gaussian␣NB","Gaussian␣NB2"]
"""
classifiers␣=␣[
KNeighborsClassifier(3),
SVC(kernel="linear",␣C=0.025),
SVC(gamma=2,␣C=1),
GaussianProcessClassifier(1.0␣*␣RBF(1.0),␣warm_start=True),
DecisionTreeClassifier(max_depth=5),
RandomForestClassifier(max_depth=5,␣n_estimators=10,␣max_features=1),
MLPClassifier(alpha=1),
AdaBoostClassifier(),
GaussianNB(),
QuadraticDiscriminantAnalysis()]
"""
classifiers␣=␣[
AdaBoostClassifier(n_estimators=2),
KNeighborsClassifier(3),
GaussianNB(priors=[0.1,0.9]),
GaussianNB(priors=[0.9,0.1]),
]

X,␣y␣=␣make_classification(n_features=2,␣n_redundant=0,␣n_informative=2,
random_state=1,␣n_clusters_per_class=1)
rng␣=␣np.random.RandomState(2)
X␣+=␣2␣*␣rng.uniform(size=X.shape)
linearly_separable␣=␣(X,␣y)

datasets␣=␣[make_moons(noise=0.3,␣random_state=0),
make_circles(noise=0.2,␣factor=0.5,␣random_state=1),
linearly_separable
]

figure␣=␣plt.figure(figsize=(27,␣9))
i␣=␣1
#␣iterate␣over␣datasets
for␣ds_cnt,␣ds␣in␣enumerate(datasets):
#␣preprocess␣dataset,␣split␣into␣training␣and␣test␣part
X,␣y␣=␣ds
X␣=␣StandardScaler().fit_transform(X)
X_train,␣X_test,␣y_train,␣y_test␣=␣\
train_test_split(X,␣y,␣test_size=.4,␣random_state=42)
```

```
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
np.arange(y_min, y_max, h))

# just plot the dataset first
cm = plt.cm.RdBu
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
if ds_cnt == 0:
ax.set_title("Input data")
# Plot the training points
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
edgecolors='k')
# and testing points
ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6,
edgecolors='k')
ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_xticks(())
ax.set_yticks(())
i += 1

# iterate over classifiers
for name, clf in zip(names, classifiers):
ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
clf.fit(X_train, y_train)
score = clf.score(X_test, y_test)

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
if hasattr(clf, "decision_function"):
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
else:
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

# Put the result into a color plot
Z = Z.reshape(xx.shape)
ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

# Plot also the training points
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
edgecolors='k')
# and testing points
ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
edgecolors='k', alpha=0.6)
```

```
ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_xticks(())
ax.set_yticks(())
if ds_cnt == 0:
ax.set_title(name)
ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
size=15, horizontalalignment='right')
i += 1

plt.tight_layout()
plt.show()
```