classifieur_cancerSein

July 27, 2021

1 Classification de données de cancer du sein avec un perceptron multi-couches

Dans ce notebook nous effectuerons l'apprentissage de données de cancer du sein pour une classification en deux classes. Le package utilisé dans ce TP pour implémenter le perceptron multi-couche est ***scikit-learn***.

Pour le bon fonctionnement de ce notebook, les packages suivants sont nécessaires: 1. *numpy* 2. *sckit-learn*

1.1 Chargement des données et exploration

1.1.1 Chargement des données

Les données que nous utiliserons existent déjà dans le package ***scikit-learn***. Pour les charger, nous utiliserons la fonction `load_breast_cancer`. On aurait très bien pu utiliser un fichier csv et les importer avec le package ***pandas***.

[1]:
```
# Chargement des données pour la classification du cancer du sein
from sklearn.datasets import load_breast_cancer
cancer_data = load_breast_cancer()

print("La base de données chargée est de type : ", type(cancer_data))
```

```
La base de données chargée est de type :  <class 'sklearn.utils.Bunch'>
```

1.1.2 Exploration des données

Cette étape permet de pouvoir analyser les données (répartition des classes, taille, ...) afin d'avoir une idée un peu plus précise de qu'on manipule. Dans cet exemple, nous allons: - Afficher la taille de la base de données - Le nombre de classes - Le nombre d'échantillons dans chaque classe

Pour faire un apprentissage, on a besoin d'avoir les données d'entrée mais également les données de sortie qui sont dans cet exemple les différentes classes. Ces deux données sont accessibles grâce aux clés `data` et `target` dans la base de données chargée.

[2]:
```
# Taille des données de cancer
print("Taille des données chargées: ", cancer_data['data'].shape)
```

```
Taille des données chargées:  (569, 30)
```

[3]:
```
# Charger séparément les entrées et les sorties des données pour préparer␣
↪l'apprentissage
features = cancer_data ['data']
classes = cancer_data ['target']

# Afficher les différentes classes
import numpy as np
v_classes_uniques = np.unique(classes)
print(len(v_classes_uniques), "Différentes classes : ", v_classes_uniques)

# Afficher la taille des données de chacune des classes
for classe in v_classes_uniques:
    print("La classe ", classe, "contient ", np.sum(classes==classe),␣
↪"échantillons.")
```

```
2 Différentes classes :  [0 1]
La classe  0 contient  212 échantillons.
La classe  1 contient  357 échantillons.
```

1.2 Préparation des données pour l'apprentissage

1.2.1 Création des ensembles d'apprentissage et de test

Avant d'effectuer l'apprentissage, il faut diviser notre base de données en 2 : 1. un ensemble d'apprentissage qui servira à entraîner le réseau de neurones C'est sur cet ensemble qu'on optimise la fonction de perte afin de minimier l'erreur de prédiction. 2. un ensemble de test qui servira à évaluer la qualité de notre apprentissage Pour séparer notre ensemble de données dans ce sens, la fonction `train_test_split` est bien adaptée. Il suffit de lui indiquer la répartition qu'on souhaite en terme de ratio entre l'ensemble d'apprentissage et celui de test grâce au paramètre `train_size`.

[4]:
```
from sklearn.model_selection import train_test_split
features_train, features_test, classes_train, classes_test = train_test_split␣
↪(features, classes, train_size=0.75)

# Afficher les tailles des deux ensembles
print("Taille de l'ensemble d'apprentissage : ", features_train.shape)
print("Taille de l'ensemble de test : ", features_test.shape)
```

```
Taille de l'ensemble d'apprentissage :  (426, 30)
Taille de l'ensemble de test :  (143, 30)
```

1.2.2 Normalisation des données

La normalisation des données est bien souvent une étape indispensable pour faciliter l'apprentissage. Cette opération est faite sur l'ensemble d'apprentisage et ensuite appliquer à l'ensemble de test et à chaque échantillon pour lequel on aimerait prédire la classe.

Plusieurs normalisations sont possibles: - la normalisation centrée réduite - l'utilisation de la norme L1 - l'utilisation de la norme L2 - ...

Nous utilisons ici la fonction `StandardScaler` dans `sklearn.prepocesing` pour faire une normalisation centrée réduite. On peut également utiliser la fonction `normalize` en passant en paramètre le type de normalisation qu'on souhaite effectuer.

La normalisation ne change absolument pas la taille des échantillons mais leurs valeurs. Pour le vérifier, nous avons afficher les valeurs minimales et maximales des variances et des moyennes avant et après la normalisation centrée réduite.

[5]:
```
# Normalisation des données avant l'apprentissage
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

# La normalisation se fait uniquement sur les données d'apprentissage
scaler.fit(features_train)
StandardScaler(copy=True , with_mean=True , with_std=True)

# La normalisation ne modifie pas la taille des données mais leurs valeurs
# Afficher les valeurs minimales et maximales de la variance et de la moyenne␣
 ↪des données après la normalisations
variances_train = np.std(features_train, 0)
moyennes_train = np.mean(features_train, 0)

print("Avant l'opération de normalisation")
print("Taille de l'ensemble d'apprentissage : ", features_train.shape)

print("Variances des données d'apprentissage : ", [np.round(np.
 ↪min(variances_train), 3),
                                                    np.round(np.
 ↪max(variances_train), 3)] )
print("Moyennes des données d'apprentissage : ", [np.round(np.
 ↪min(moyennes_train)),
                                                   np.round(np.
 ↪max(moyennes_train), 3)])

# Application du modèle de normalisation aux données de test et d'apprentissage
features_train = scaler.transform (features_train)
features_test = scaler.transform (features_test)

# La normalisation ne modifie pas la taille des données mais leurs valeurs
variances_train = np.std(features_train, 0)
moyennes_train = np.mean(features_train, 0)

print("\nAprès l'opération de normalisation")
print("Taille de l'ensemble d'apprentissage : ", features_train.shape)

# Afficher les valeurs minimales et maximales de la variance et de la moyenne␣
 ↪des données après la normalisations
```

```
print("Variances des données d'apprentissage : ", [np.round(np.
 ↪min(variances_train), 3),
                                                    np.round(np.
 ↪max(variances_train), 3)] )
print("Moyennes des données d'apprentissage : ", [np.round(np.
 ↪min(moyennes_train)),
                                                   np.round(np.
 ↪max(moyennes_train), 3)])
print(variances_train.shape)
```

```
Avant l'opération de normalisation
Taille de l'ensemble d'apprentissage :  (426, 30)
Variances des données d'apprentissage :  [0.002, 600.277]
Moyennes des données d'apprentissage :  [0.0, 908.308]

Après l'opération de normalisation
Taille de l'ensemble d'apprentissage :  (426, 30)
Variances des données d'apprentissage :  [1.0, 1.0]
Moyennes des données d'apprentissage :  [-0.0, 0.0]
(30,)
```

1.3 Entraînement et évaluation du réseau de neurones multi-perceptron

Pour implémenter notre réseau de neurones, nous utilisons la fonction `MLPClassifier` du module `neural_network` du package ***scikit-learn***. Cette fonction permet de paramétrer les couches cachées (nombre et taille), la fonction d'activation des couches cachées, la fonction d'optimisation, le taux d'apprentissage, le nombre maximum d'itérations, ...

1.3.1 Entraînement du réseau de neurones

[6]:
```
# Utilisation d'un réseau de neurones multi-perceptron pour la classification
from sklearn.neural_network import MLPClassifier

# Taille des différentes couches cachés
taille_couches_cachees = (30, 10)
print("Nombre de couches cachées : ", len(taille_couches_cachees))

# Définition du réseau de neurones
mlp = MLPClassifier(hidden_layer_sizes=taille_couches_cachees, max_iter=2000)

# Entrainement du réseau défini sur les données d'apprentissage
mlp.fit(features_train, classes_train)
```

```
Nombre de couches cachées :  2
```

[6]:
```
MLPClassifier(hidden_layer_sizes=(30, 10), max_iter=2000)
```

1.3.2 Evaluation de l'apprentissage sur les données de test

Pour évaluer la qualité de l'apprentissage, on peut utiliser plusieurs métriques. Dans cet exemple, nous utiliserons : - la matrice de confusion - la précision - le recall - le score F1

Ces différentes métriques sont déjà implémentées dans le module `metrics` du package ***scikit-learn***.

[7]:
```
predictions_test = mlp.predict( features_test )
from sklearn.metrics import classification_report, confusion_matrix
print("Matrice de confusion")
print(confusion_matrix (classes_test, predictions_test))
```

```
Matrice de confusion
[[52  3]
 [ 3 85]]
```

[8]:
```
print(classification_report(classes_test, predictions_test))
```

```
              precision    recall  f1-score   support

           0       0.95      0.95      0.95        55
           1       0.97      0.97      0.97        88

    accuracy                           0.96       143
   macro avg       0.96      0.96      0.96       143
weighted avg       0.96      0.96      0.96       143
```

1.3.3 Validation croisée

La validation croissée est une approche permettant d'entraîner le modèle plusieurs fois sur des sous-échantillons de la base d'apprentissage. Elle permet de s'assurer que le modèle ne fonctionne pas uniquement dans une seule configuration de l'ensemble d'apprentissage. On peut par la suite sélectionner le modèle qui offre la meilleure précision, le meilleur recall, ...

Nous avons effecutée une validation croissée avec la fonction `cross_val_score` du module `model_selction` du package ***scikit-learn***. Il faut indiquer à cette fonction : - le nombre de validations croisées qu'on souhaite effectuer avec le paramètre `cv` - la métrique d'évaluation de la validation croisée avec le paramètre `scoring`

[9]:
```
from sklearn.model_selection import cross_val_score
cross_val_score (mlp , features_train, classes_train, cv=5, scoring="accuracy")
```

[9]:
```
array([0.98837209, 0.95294118, 0.97647059, 0.98823529, 0.91764706])
```