University Paris-Est Creteil, Creteil, France

Master 1 OIVM Optique Image Vision MultiMedia

Apprentissage supervisé

Auteur:
Nassim Battache
Iskander Djouad

Superviseur :
Amir NAKIB

March 26, 2022

Contents

Introduction

Pour réduire la variance de l'estimation du risque espéré obtenue sur un échantillon test, plusieurs partitionnement différents sont réalisés. A chaque fois un modèle appris sur des données, son erreur est calculée. Nous obtenions ainsi un estimateur de variance plus faible, out en utilisant mieux les données disponibles.

Chapter 1

Estimation des performances par validation croisée :

Pour répondre à la question de nombre d'échantillent on peut exécuter simplement la commande suivante :

```
In [4]: len(data)
Out[4]: 800
```

Figure 1.1: nombre d'échantillents

puis on visualise nos données .

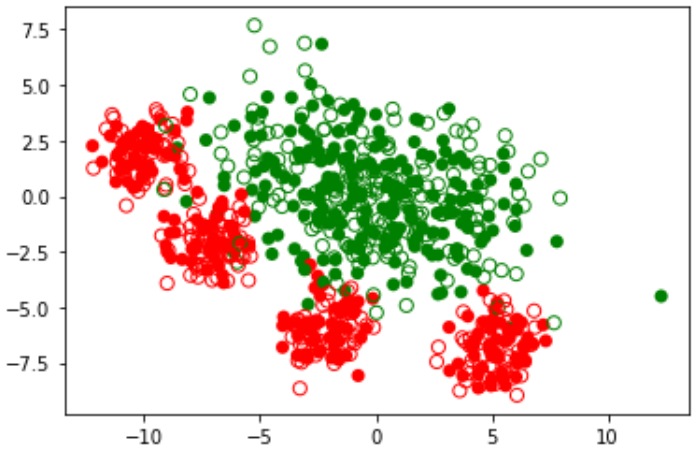

Figure 1.2: Données

la variable "kcvf" dans la boucle correspond au nombre de flods ou de partition. S'il vaut 2, alors chaque PMC est appris sur la moitié du jeu de données, soit 400 pour la partie entrainement (train) et 400 pour la partie test (test).

Ces exemples sont tirés au hasard uniformement.

Nous allons entraîner un perceptron multicouche (PMC) sur ce jeu d'apprentissage. La classe KFold de scikit-learn permet de générer automatiquement les partitions du jeu d'apprentissage pour la validation croisée.

Une fois que nous avons obtenu les scores de validation croisée pour différentes valeurs de k qui sont sur la figure1.3.

D'après cette visualisation on constate qu'à chaque fois que on augment le k la variance des enchantions augment aussi .

On peut voir facilement que la variance augment au fil des itération jusqu'à k=100 Soit 400/100 = 4 échantillons par classe . [figure 1.4]

affichage des courbes sur le méme graphique [figure1.5]

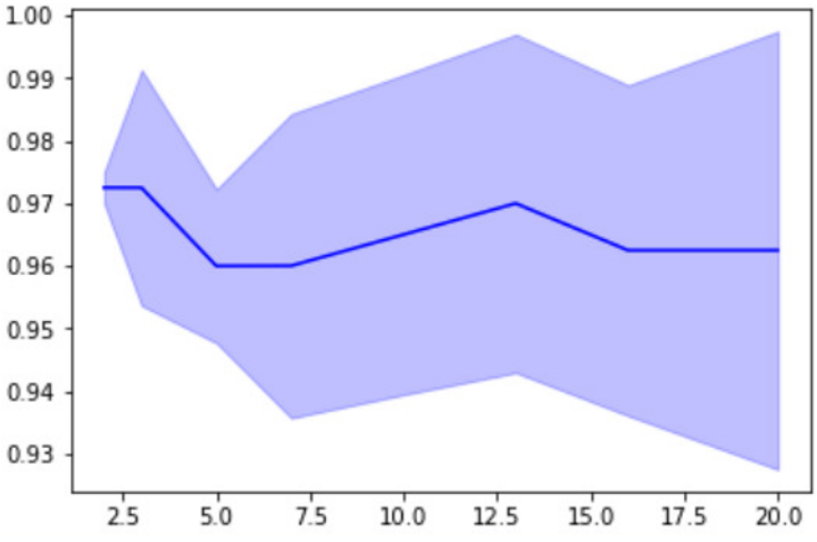

Figure 1.3: courbe score kfold

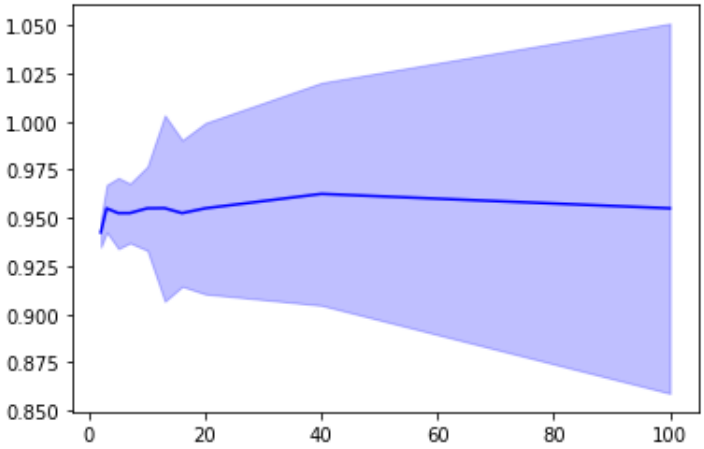

Figure 1.4: courbe score kfold aprés ajout de la valeur 40,100

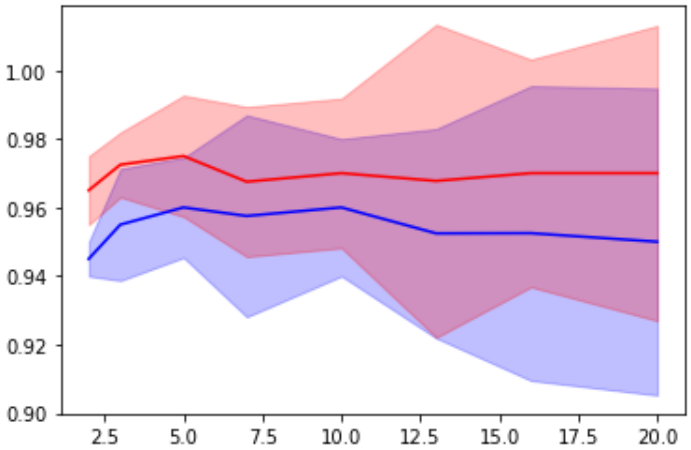

Figure 1.5: courbe score kfold aprés ajout de la valeur 40,100

On ajoute des listes pour stocker ces résultats avec le code suivant. [figure1.6]

En constate qu'à chaque fois que on augment le nombre de k on obtient une variance plus fort entre les échantillons mais le temps l'exécution augment aussi .

Remarque : augmenter le nombre de k n'est pas une bonne méthode pour des datasets volumineux il prend trop de temps .

```
kcvscores = list()
kcvscores_std = list()
testscores = list()
testscores_std = list()

for kcvf in kcvfs:    # pour chaque valeur de k
  kf = KFold(n_splits=kcvf)
  these_scores = list()
  these_test_scores = list()
  # apprentissage puis évaluation d'un modèle sur chaque split
  for train_idx, test_idx in kf.split(X_train):
    clf.fit(X_train[train_idx], y_train[train_idx])
    these_scores.append(clf.score(X_train[test_idx], y_train[test_idx]))
    these_test_scores.append(clf.score(X_test, y_test))
  # calcul de la moyenne et de l'écart-type des performances obtenues
  kcvscores.append(np.mean(these_scores))
  kcvscores_std.append(np.std(these_scores))
  testscores.append(np.mean(these_test_scores))
  testscores_std.append(np.std(these_test_scores))

# création de np.array à partir des listes
kcvscores, kcvscores_std = np.array(kcvscores), np.array(kcvscores_std)
testscores, testscores_std = np.array(testscores), np.array(testscores_std)

# affichage performance moyenne +- 1 écart-type pour chaque k
plt.figure()
plt.plot(kcvfs, kcvscores, 'b')
plt.plot(kcvfs, kcvscores+kcvscores_std, 'b--')
plt.plot(kcvfs, kcvscores-kcvscores_std, 'b--')
plt.plot(kcvfs, testscores, 'g')
plt.plot(kcvfs, testscores+testscores_std, 'g--')
plt.plot(kcvfs, testscores-testscores_std, 'g--')
```

Figure 1.6: ajout des lignes de code

Les résultats montrent que l'estimation de l'erreur de généralisation par validation croisée sur les données d'apprentissage (courbes en bleu) reste en général optimiste par rapport à l'estimation sur des données de test supplémentaires (courbes en rouge). Aussi, la variance des estimations sur les données de test est comparativement faible car ces données sont ici aussi volumineuses que les données d'apprentissage.

leave one out (LOO) fournit des indices d'entraînement/test pour diviser les données en ensembles d'entraînement/test. Chaque échantillon est utilisé une fois comme ensemble de test (singleton) tandis que les échantillons restants forment l'ensemble d'apprentissage.

Réalisation de l'estimation des performances en utilisant la validation croisée leave one out (LOO) :

```
for train_idx, test_idx in loo.split(X_train):
# clf = MLPClassifier(solver='lbfgs',alpha=1)
    clf = MLPClassifier(solver='lbfgs', alpha=1, random_state=1)
    clf.fit(X_train[train_idx], y_train[train_idx])
    scores_kf.append(clf.score(X_train[test_idx], y_train[test_idx]))
score_ts=clf.score(X_test, y_test)
print(score_ts)
# calcul de la moyenne et de l'écart-type des performances obtenues
scores.append(np.mean(scores_kf))
scores_std.append(np.std(scores_kf))
# création de np.array à partir des listes
scores, scores_std = np.array(scores), np.array(scores_std)

print(scores_kf)
print(scores_lo)
```

Figure 1.7: boucle LOO

Le nombre d'exemples d'apprentissage est de N-1 dans notre cas ici c'est 400-1=399

On a utilisé l'optimiseur 'lbfgs' avec alpha =1 et de score 97.25

Pour l'écart type de k-fold qui est égal à 95.37

en comparant les résultats de k-fold et de leave one out en constate que le score obtenu avec Loo =0.9725 soit 97.25 est plus fort que le résultat de k-fold qui est de 0.95337 soit 95.337

mais le temps d'excution de k-fold est plus inferieur que celui de LOO

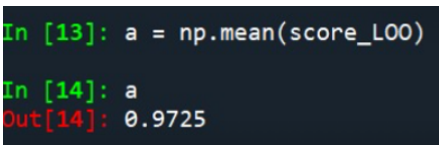

Figure 1.8: score LOO

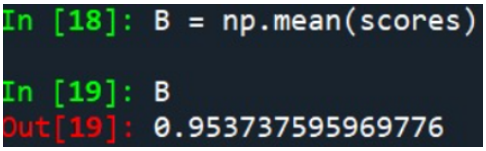

Figure 1.9: score k-fold

au final on constate que l'écart-type est bien plus élevé pour l'estimation leave one out que pour les estimations k-fold (pour toutes les valeurs considérées ici pour k).

Chapter 2

Recherche des meilleures valeurs pour les hyerparamètres :

on execute le programme suivant:

```
# importations
import numpy as np
import matplotlib.pyplot as plt

# chargement des données iris
from sklearn import datasets
data, labels = datasets.load_iris(return_X_y=True)

# découpage initial en données d'apprentissage et données de test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.5)

# emploi de PMC
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
tuned_parameters = {'hidden_layer_sizes':[(5,), (20,), (50,), (100,), (150,), (200,)],
                    'alpha':   [0.001, 0.01, 1, 2]}
clf = GridSearchCV(MLPClassifier(solver='lbfgs', tol=5e-3), tuned_parameters, cv=5)

# exécution de grid search
clf.fit(X_train, y_train)
print(clf.best_params_)
n_hidden = np.array([s[0] for s in tuned_parameters['hidden_layer_sizes']])
alphas = np.array(tuned_parameters['alpha'])

# création de la grille des hyperparamètres
xx, yy = np.meshgrid(n_hidden, alphas)
scores = clf.cv_results_['mean_test_score'].reshape(xx.shape)

# affichage sous forme de fil de fer de la surface des résultats des modèles évalués
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(8, 6))
ax = plt.axes(projection='3d')
ax.set_xlabel("Neurones cachés")
ax.set_ylabel("Régularisation $\\alpha$")
ax.set_zlabel("Taux de bon classement")
ax.plot_wireframe(xx, yy, scores)
plt.show()
```

Figure 2.1: programme 2

Le nombre de combinaisons de (hyper)paramétres explorées est “len(tuned-parameters[’hidden-layer-sizes’]) * len(tuned-parameters[’alpha’])“ = 24. Pour chacune de ces combinaisons, “cv=5“ indique que :‘k = 5‘ PMC différents sont appris. Donc un total de 24 * 5 = 120.

Il faut définir une nouvelle grille plus fine autour de ce point et appeler de nouveau “GridSearchCV“.

la signification du paramètre refit de GridSearchCV :si la valeur de ce paramètre est True (valeur par défaut) alors, une fois trouvées les meilleures valeurs pour les hyperparamètres, un nouveau modèle est appris avec ces valeurs-là sur la totalité des N données d’apprentissage X-train, y-train (sans en exclure N/k). Ce modèle est directement accessible dans l’attribut .best-estimator- et l’appel à .predict() sur l’instance de GridSearchCV (ici clf) permet de l’utiliser.

Evaluez le modèle sélectionné sur les données de test (X-test, y-test)?

Le paramètres et par défaut True, le modèle entrainner avec les meilleures valeurs pour les hyperparamètres est directement accessible via l'instance de GridSearchCV (ici clf), donc pour l'évaluer sur les données de test il sufit de coplier la commande suivante

```
plt.show()
clf.score(X_test, y_test)
```

Figure 2.2: instance de GridSearchCV

L'appel à RandomizedSerchCV aura la forme

```
rlf = RandomizedSerchCV(MLPClassifier(solver='lbfgs'), param_distributions=param_distrib, n_iter=50, cv=5)
```

Figure 2.3: instance de GridSearchCV

où param-distrib est le dictionnaire qui précise les distributions employées pour les différents (hyper)paramètres et n-iter=50 indique un « budget » de 50 essais.

Bibliography