Bishop's University

CS 405/CS 505 – Data Mining

Assignment 2: Evaluation and selection of decision models

1. Introduction

The objective of this assignment is to demonstrate the use of cross-validation techniques for the evaluation and comparison of decision models, as well as methods for finding values for hyper-parameters.

2. Performance estimation by cross-validation

To illustrate the use of cross-validation, we consider a ranking problem similar to the one discussed in the previous session. We consider a dataset for which we have monitoring information (class labels) and we partition it into a training set and a test set. We will use Fisher's Irises as the dataset. As a reminder, this consists in identifying to which species a plant belongs among three possibilities (Iris Setosa, Iris Virginica and Iris Versicolor) from the length and width of the sepals and petals.

We use multi-layer perceptrons (MLP) with a single hidden layer of 100 neurons and a value $\alpha = 1$ for the regularization constant (weighting of the forgetting term or weight decay). It is not essential for this session to know in detail how these neural networks work, these decision models will be reviewed later in the course.

Cross-validation will be used to estimate generalization performance from the training set. Then, this estimate will be compared to the estimate obtained on the test set that we had set aside at the start. Explanations of cross-validation and its implementation in Scikit-learn can be found in the documentation.

```
# importing libraries
import numpy as np
import matplotlib.pyplot as plt
import sklearn

# loading the Iris dataset
from sklearn import datasets
iris = datasets.load_iris()
data, labels = iris.data, iris.target

# initial splitting into training data and test data
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)
```

The *train_test_split* function of scikit-learn allows us to randomly split the dataset into two partitions train (learning) and test (evaluation) according to arbitrary proportions.

As usual, it is interesting to visualize the data available to us. We can construct the scatter plots in two dimensions using Matplotlib:

```
fig = plt.figure(figsize=(16, 12))
n_features = data.shape[-1]
n_plots = 6
idx = 1
cmp = np.array(['r', 'g', 'b'])
for dim1 in range(0, n_features):
    for dim2 in range(dim1+1, n_features):
        fig.add_subplot(2, n_plots // 2, idx)
        plt.scatter(X_train[:, dim1], X_train[:, dim2],c=cmp[y_train],
s=50, edgecolors='none')
        plt.scatter(X_test[:, dim1], X_test[:, dim2], c='none', s=50,
edgecolors=cmp[y_test])
        plt.xlabel(iris.feature_names[dim1])
        plt.ylabel(iris.feature_names[dim2])
        idx += 1
plt.show()
```

Question 1: Why do we have six different point clouds? What does each figure represent?

For this classification problem, we will use a MLP with the default values proposed by scikit-learn for the number of layers (1) and neurons (100).

```python
# Usage of MLP
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(solver='lbfgs', alpha=1, tol=5e-3)
```

To estimate generalization error, we will use *K-fold* cross-validation. This will allow us to adjust, if necessary, the hyperparameters of the decision model. scikit-learn implements various cross-validation strategies in the *sklearn.model_selection* module. Let start by experimenting with the *K-fold* approach. The *KFold* object has a *.split()* method that generates lists of observation indices to be used for the training subset and for the validation subset. More details on this object can be found in the *K-Fold* documentation.

```python
# KFold for different values of k
from sklearn.model_selection import KFold

# values of k
n_folds = np.array([2, 3, 5, 7, 10, 13, 16, 20])

# preparing lists to store results
cv_scores = []
cv_scores_std = []

for k in n_folds:    # for each value of k
    kf = KFold(n_splits=k)
    scores = []
    # learning then evaluating a model on each split
    for train_idx, val_idx in kf.split(X_train):
        # learning with .fit()
        clf.fit(X_train[train_idx], y_train[train_idx])
        scores.append(clf.score(X_train[val_idx], y_train[val_idx]))
    # calculation of the mean and standard deviation of the obtained performances
    cv_scores.append(np.mean(scores))
    cv_scores_std.append(np.std(scores))

cv_scores, cv_scores_std = np.array(cv_scores), np.array(cv_scores_std)

# display average performance +- 1 standard deviation for each k
plt.figure(figsize=(8, 6))
plt.plot(n_folds, cv_scores, 'b')
plt.fill_between(n_folds, cv_scores + cv_scores_std, cv_scores - cv_scores_std,
alpha=0.5)
plt.xlabel("Value of $k$ of K-Fold")
plt.ylabel("Average Score")
plt.xlim(2, max(n_folds))
plt.xticks(n_folds)
plt.title("Generalization error estimated as a function of $k$")
plt.show()
```

Question 2: What do you see when looking at this graph? Add values for k (eg 40, 100, be careful it will take longer…) and examine the graph again.

Question 3: For each model learned by k-fold cross-validation, add its evaluation on the test data set aside at the start *X_test*, *y_test*. Display the curves on the same graph. What do you notice?

Question 4: Perform performance estimation using *leave one out (LOO)* cross-validation. What do you see comparing the results of *k-fold* and *leave one out*?

3. Finding the best values for hyperparameters

One of the central issues in decision modeling is to be able to determine the hyperparameters of the model that lead to the best performance. Formally, this amounts to finding the parametric family which obtains the lowest generalization error. Nevertheless, as we have seen, we cannot use the evaluation set to compare hundreds of models: the estimate of the generalization error would become far too optimistic. We will therefore use cross-validation to compare our models, then only once the best values of the hyperparameters have been determined can we estimate the generalization error on the test set.

The naivest method for finding hyperparameters is systematic search, or grid search. It consists in exploring all the possible combinations of values in each set. Let use grid search to find the best values of two hyperparameters for MLPs in the same ranking task as before. These hyperparameters are the number of neurons in the single hidden layer of the PMC and the value of the regularization constant (by weight decay), α.

```
# importing libraries
import numpy as np
import matplotlib.pyplot as plt

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

# initial splitting into training data and test data
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)

# to use MLP
from sklearn.neural_network import MLPClassifier
```

To use grid search and cross-validation to compare the resulting models with all combinations of values for the hyperparameters, it is necessary to employ *GridSearchCV*. This scikit-learn estimator automatically combines a grid search with a *K-fold* cross-validation.

```
from sklearn.model_selection import GridSearchCV
```

It is necessary to indicate in a "dictionary" which are the hyperparameters whose values we wish to explore, and which are the different values to be evaluated. Each dictionary entry consists of a character string that contains the name of the hyperparameter as defined in the estimator used. We will be using *MLPClassifier* here, so the names of the parameters can be found in the overview of this class. We consider here only two parameters, *hidden_layer_sizes* (number of neurons in the unique hidden layer) and *alpha* (the constant α of regularization by weight decay), α.

```
tuned_parameters = {'hidden_layer_sizes':[(5,), (20,), (50,), (100,), (150,), 
(200,)], 'alpha':[0.001, 0.01, 1, 2]}
```

In the call to *GridSearchCV* we then indicate for *MLPClassifier* the solver to be used systematically (which is not the default one), then the dictionary with the values of the (hyper)parameters to explore and finally the fact that it is the validation *k-fold* cross with $k = 5$ which is used to compare the different models.

```
clf=GridSearchCV(MLPClassifier(solver='lbfgs',tol=5e-3),tuned_parameters, cv=5)
```

```
# grid search execution
clf.fit(X_train, y_train)
```

Scikit-learn then runs the following program:

- from the lists of values for the different (hyper)parameters all the combinations of values are generated,
- for each combination, the performances of the corresponding models are evaluated by 5-fold cross-validation (applied only on the training data *X_train*, *y_train*),
- the values of the (hyper)parameters corresponding to the best cross-validation performance are selected,
- with these values for the (hyper)parameters a new learning is performed with all the data of *X_train*, *y_train* (and not only $\frac{k}{k-1}$ folds).

The following lines are used to display the results: the parameters of the best model with *clf.best_params_*, as well as the cross-validation results obtained for all the combinations of values for the (hyper)parameters (*clf.cv_results_* gives access to this information and to many others).

```
print(clf.best_params_)
```

We can also do a more comprehensive display, plotting the surface of the average score of the different models as a function of the combination of the two hyperparameters:

```
n_hidden=np.array([s[0]for s in tuned_parameters['hidden_layer_sizes']])
alphas = np.array(tuned_parameters['alpha'])

# creating the hyperparameter grid
xx, yy = np.meshgrid(n_hidden, alphas)
scores = clf.cv_results_['mean_test_score'].reshape(xx.shape)

# wireframe display of the results of the evaluated models
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(8, 6))
ax = plt.axes(projection='3d')
ax.set_xlabel("Hidden neurons")
ax.set_ylabel("Regularization $\\alpha$")
ax.set_zlabel("Good classification rate ")
ax.plot_wireframe(xx, yy, scores)
plt.show()
```

We have used *plot_wireframe* here because the readability is better than with *plot_surface*.

Question 5: How many MLPs are learned in total in this example?

Question 6: What is the meaning of the *refit* parameter of *GridSearchCV*?

Question 7: Take a more complete look at the contents of *clf.cv_results_*.

Question 8: Evaluate the selected model on the test data (*X_test*, *y_test*).

Question 9: Do the appearance of the results encourage you to refine the grid? Edit the grid and review the new results.

Question 10: Use randomized search with *RandomizedSearchCV*. The "cost" (total number of evaluated combinations) can be set with *n_iter*. Explain the choice of the laws used for drawing the values of the two (hyper)parameters *hidden_layer_sizes* and *alpha*.

4. For further

Other more advanced hyperparameter selection strategies are possible. Indeed, the search by grid, like the random search, are relatively rudimentary and rather resource-intensive methods. This can particularly pose a problem when the decision models considered belong to a computationally expensive parametric family, such as deep neural networks.

scikit-learn thus offers an experimental implementation of a search by dichotomy, allowing resources to be concentrated on the most promising hyperparameter combinations.

Question 11: Experiment with *HalvingGridSearchCV*. Compare the final score and the calculation time with that obtained with *GridSearchCV*.

Outside of scikit-learn, there are also other libraries that implement advanced selection strategies using various heuristics to choose which hyperparameter combinations to explore. Some of these libraries are compatible with the scikit-learn interface, for example tune-sklearn or sklearn-genetic-opt.

5. Submission

You must submit the *pdf* file of your report. In fact, it must contain the responses for the eleven questions. In addition, you have to submit your *.ipynb*. ***Please, do not submit your assignment in .zip or .rar files.***

6. Useful external references

- NumPy documentation
- SciPy documentation
- MatPlotLib documentation
- scikit-learn
- Python programming language