Faculty of Mechanical Engineering and Robotics

Department of Robotics and Mechatronics

Python for Machine Learning and Data Science

Course for Mechatronic Engineering

Instruction 6:

Models configuration & results assessment

You will learn: how to configure and assess learners.

Additional materials:

- Course lecture 5 [*obligatory*] http://galaxy.agh.edu.pl/~zdw/Materials/Python/LectureNotes/

Learning outcomes supported by this instruction:
IMA1A_W07, IMA1A_W12 , IMA1A_U01, IMA1A_U05, IMA1A_U07, IMA1A_U14, IMA1A_K08

Course supervisor:
Ziemowit Dworakowski, zdw@agh.edu.pl

Instruction author:
Adam Machynia, machynia@agh.edu.pl

Introduction

During this laboratory session, you will learn how to configure and assess a learner model performing either a regression or classification task. First, we will begin by learning cross-validation techniques, followed by model configuration, and finally, we will assess the final model.

Cross-validation

You are already familiar with **holdout validation**. In this approach, a portion of the training dataset is left out to create a validation set. The model is then trained on the reduced training set, and its performance is evaluated on the validation set.

In the **k-fold cross-validation** approach, the training set is divided into *k* subsets, called *folds*. The model is trained and assessed *k* times, once for each fold. During each iteration, the model is trained on the data excluding the k^{th} fold, which serves as the validation set. The scores obtained from these iterations are averaged to derive the final score for the model. While this approach requires repeated training, making it computationally expensive, it has the advantage of preserving a larger portion of the data for training.

After applying these approaches, the final assessment is conducted on the test set, which follows training on the entire training set, including the validation set.

Now, let's implement cross-validation using *scikit-learn*. Two similar methods we will use are *cross_val_score* and *cross_validate*. We will begin with the basic *cross_val_score* method. To use this method, we need to pass the following arguments: the model (classifier or regressor), X (data), and y (target). Additionally, we can specify two crucial parameters. The first is *cv*, which when set as an integer determines the number of folds (*k*) in k-fold cross-validation. The *cv* parameter can also specify more sophisticated data splits, but that's beyond our scope for now. Another essential parameter is *scoring*, which sets the metric for model evaluation. Previously, you used accuracy for classification and RMSE for regression. To apply cross-validation with these metrics, you may use the following code. Note that *cross_val_score* (and similarly *cross_validate*) uses the rule "the higher, the better," so instead of RMSE, we use negated RMSE.

```
from sklearn.model_selection import cross_val_score, cross_validate

# in case of classification
clf = svm.SVC()
scores = cross_val_score(clf, x_train, y_train, cv=5, scoring='accuracy')

# in case of regression
scores = cross_val_score(reg, x_train, y_train, cv=5,
                         scoring='neg_root_mean_squared_error')
```

The output of *cross_val_score* is an array of scores for each fold, giving us *k* scores for the k-fold cross-validation.

You should begin by preparing your data, classifier, and regressor as you did in Lab 4.

Task 6.1: Implement cross-validation for a selected classifier and regressor. Use accuracy for the classifier and negated RMSE for the regressor. For regression, convert the negated output back to RMSE. Calculate the mean of the obtained scores.

Task 6.2: Compare the results of 3-fold, 5-fold, and 10-fold cross-validation.

As mentioned earlier, there are a wide range of available metrics for scoring purposes. You can explore these metrics via the following link: https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter. These metrics can be used either through the scoring parameter or as direct functions, similar to how accuracy was used in previous labs.

Additionally, the *cross_validate* method was introduced. It differs from *cross_val_score* in two key ways. First, it tracks the time taken for training and validation, which is important when model efficiency is a factor. Second, it can handle multiple scoring metrics at once by passing a dictionary of strings describing the desired metrics. The output includes arrays for each metric's scores, as well as arrays for training times and scoring times. The keys in this dictionary include 'test_score,' 'train_score,' 'fit_time,' and 'score_time.' The '_score' suffix corresponds to a particular metric, such as 'test_recall,' 'train_f1,' and so forth.

```
scores = cross_validate(clf, x_train, y_train, cv=5,
                        scoring=['accuracy','f1'])
print(scores['test_accuracy'])
print(scores['test_f1'])
print(scores['fit_time'])
print(scores['score_time'])
```

Task 6.3: Utilize different metrics for cross-validation.
For classification, evaluate: recall, precision, and f1.
For regression, assess: R^2 and mean absolute error.

Tuning the hyperparameters of a model

Since hyperparameters are not learned by the model, we need an effective method to set them. In *scikit-learn*, hyperparameters are set as method attributes (e.g., *kernel*, *C*, and *gamma* for SVM, or the number of neurons in a hidden layer for MLP). So far, you've set these manually. Let's explore grid search, which tests all combinations of specified hyperparameters.

Grid search serves as a search method. Before applying this technique, we need to define the model type (e.g., SVM), the scoring function (e.g., accuracy), the evaluation scheme (commonly cross-validation), and the parameter space to search. We can utilize GridSearchCV in the following way.

```
from sklearn.model_selection import GridSearchCV

clf = svm.SVC()
# dictionary of parameters: takes all combinations
parameters_1 = {'kernel':('linear', 'rbf', 'poly'), 'C':[0.1, 1, 10]}
# OR a sequence of dictionaries of parameters; usefull to avoid unnecessary parameters
combination
parameters_2 = [
            {'kernel':('linear', 'rbf',), 'C':[0.1, 1, 10]},
            {'kernel':['poly'], 'C':[0.1, 1, 10], 'degree':[2,3,4]}
            ]

grid_search = GridSearchCV(clf, parameters, cv=5, scoring='accuracy')
grid_search.fit(x_train, y_train)
print(grid_search.best_params_) # to get the best parameters
print(grid_search.best_estimator_) # to get the best estimator
print(grid_search.cv_results_) # to get all results

cv_res = pd.DataFrame(grid_search.cv_results_) # or use DataFrame
display(cv_res) # to display results as a table
```

In this example, we're using an SVM model with 5-fold cross-validation and accuracy as the scoring metric, similar to previous tasks. The key part is the 'parameters' variable, which can be defined in two ways. First, it can be a dictionary containing the hyperparameters to explore, where the keys correspond to specific parameters/attributes of the model (e.g., *'kernel'* or *'C'*), and the values represent the respective options, as shown in the *parameters_1* example. In this approach, all parameter combinations are explored. Alternatively, you can use a sequence of dictionaries, as in the *parameters_2* variable. This method evaluates all parameter combinations within each dictionary, allowing you to avoid unnecessary configurations – such as setting the degree for kernels other than polynomial, as shown in the example.

To obtain the results, you can use one of the following attributes:

- *best_params_* – returns the configuration with the highest score,
- *best_estimator_* – returns the model with the highest score,
- *cv_results_* – returns a dictionary containing all the results.

The *cv_results_* attribute can be easily used to display the results as a table, as demonstrated in the example above.

Task: 6.4: Implement a grid search for the SVM classifier. Compare different kernels and different values for C parameter. Use 5-fold cross-validation.

Task 6.5: Implement a grid search for another classifier. Choose an appropriate parameter space and scoring metric.

Task 6.6: Implement a grid search for the selected regressor.

Grid search is typically computationally expensive and time-consuming because the model needs to be trained and evaluated for each parameter combination *k* times, where *k* represents the number of folds in cross-validation. An alternative approach is the random search method, which specifies a set number of tests and explores the parameter space randomly. This can be easily executed using *RandomizedSearchCV*, which functions similarly to *GridSearchCV*.

Task 6.7: Implement a random search by following the documentation: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html.

Task 6.8: Implement a grid search for the selected classifier, but using pre-processed data. Perform operations similar to those in Lab 3, such as feature scaling and outlier removal.

Final assessment of the model

After tuning the hyperparameters, the final assessment of the model should be conducted. Once the model has been trained on the full training set, its performance on the test set is evaluated. As mentioned earlier, *scikit-learn* provides various built-in metrics that can be used directly. However, let's also explore calculating some metrics from scratch, starting with predictions and labels. Here's an example of how to calculate true positives, true negatives, false positives, and false negatives:

```
tp = ((y_pred == y_test) & y_test).sum() # true positive
tn = ((y_pred == y_test) & ~y_test).sum() # true negative
fp = ((y_pred != y_test) & ~y_test).sum() # false positive
fn = ((y_pred != y_test) & y_test).sum() # false negative

# or convenient way using confusion matrix
from sklearn.metrics import confusion_matrix
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
```

Task 6.9: Train the selected model with the tuned hyperparameters using the full training set. Evaluate its performance.

- For classification, assess: accuracy, recall, precision, and F1 score.
- For regression, assess: R^2 and mean absolute error.

Task 6.10: For the selected classifier, calculate the true positive rate, true negative rate, false positive rate, and false negative rate.

Additionally, here are three examples for classifier assessment: plotting the receiver operating characteristic (ROC) curve, the precision-recall graph, and the confusion matrix.

```
from sklearn.metrics import RocCurveDisplay
RocCurveDisplay.from_estimator(clf, x_test, y_test)
```

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
cm = confusion_matrix(y_test, y_pred)
cm_display = ConfusionMatrixDisplay(cm).plot(cmap=plt.cm.Blues)
```

The **ROC** curve is a graphical representation of a classifier's performance at various threshold settings. It plots the true positive rate (sensitivity) against the false positive rate (1 - specificity). A model with an area under the ROC curve (AUC) close to 1 indicates a good classifier. ROC curves are useful for comparing the performance of different classifiers.

The **precision-recall graph** is particularly useful when dealing with imbalanced datasets. It plots precision (the ratio of true positives to all predicted positives) against recall (the ratio of true positives to all actual positives). A high area under the precision-recall curve indicates both high precision and recall, meaning the model makes fewer false positives and false negatives.

The **confusion matrix** provides a detailed breakdown of classification performance by showing the number of true positives, true negatives, false positives, and false negatives. It allows you to visualize how well the classifier distinguishes between different classes and is especially useful for multi-class classification problems.

Task 6.11: For the selected classifier, plot the ROC curve.

Task 6.12: For the selected classifier, plot the precision-recall graph.

Task 6.13: For the selected classifier, plot the confusion matrix.

For regressors, there's the prediction error display, which enables a visual comparison between predicted values and true ones. You can generate this plot either from the estimator or from predictions. In both cases, you can specify whether to plot actual values or residuals (the difference between actual and predicted values).

```
PredictionErrorDisplay.from_estimator(reg, x_test, y_test,
                                      kind="residual_vs_predicted")

PredictionErrorDisplay.from_predictions(y_true=y_test, y_pred=y_pred,
                                        kind="actual_vs_predicted")
```

Task 6.14: For the selected regressor, plot the prediction error.

Task 6.15: Repeat Tasks 6.9-6.13 using pre-processed data like in Task 6.8. Compare the results.

Task 6.16: Repeat Task 6.14 using pre-processed data like in Task 6.8. Compare the results.

Additional tasks

Task 6.17: Implement grid-search for a model selected by the teacher. Consult the parameter space to be searched.

Task 6.18: Repeat Tasks 6.9-6.13 for model selected by the teacher.