IT8302 APPLIED MACHINE LEARNING

Practical 5
Model Improvement

What you will learn / do in this lab

1. *Explore Cross-Validation*
2. *Explore GridSearchCV for hyperparameter tuning*

TABLE OF CONTENTS

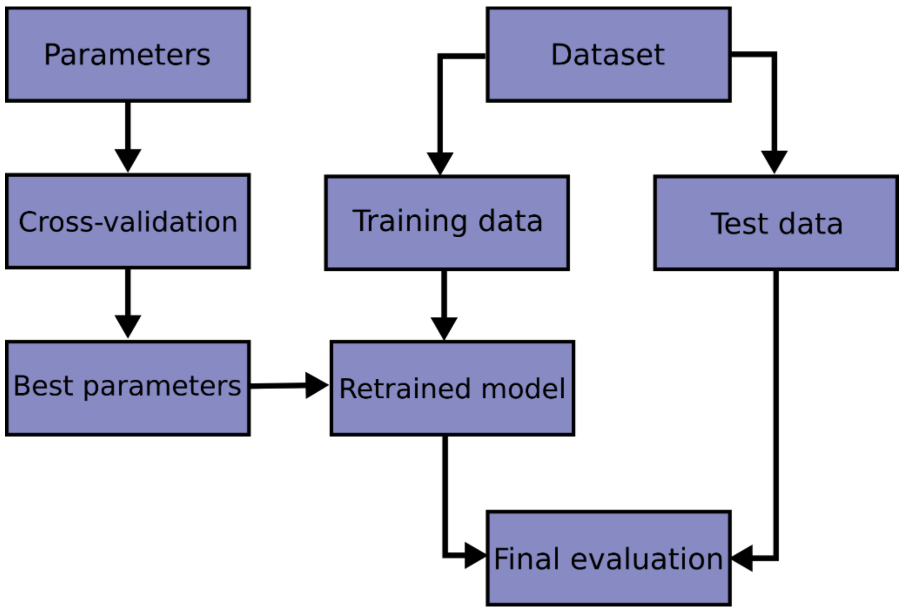
1. OVERVIEW

In this practical we will be exploring what cross-validation is and how we use it for hyperparameter tuning.

CROSS-VALIDATION

Learning the parameters of a prediction function and testing it on the **same data** is a methodological mistake: a model that learns to repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet unseen data. This situation is called **overfitting**.

To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set X_test, y_test.



Here is a flowchart of typical cross validation workflow in model training. The best parameters can be determined by grid search techniques.

GRIDSEARCHCV

Hyperparameter tuning involves search through different possible combinations of the hyperparameters and evaluating the scores from the cross-validation. The best parameters that result in the best score is used as the hyperparameters for the retrained model (see diagram above).

2. CROSS-VALIDATION

In the section, we will explore two types of validation.

SPLIT VALIDATION

This performs a simple validation i.e. randomly splits up the dataset into a training set and test set and evaluates the model. This operation performs a split validation in order to estimate the performance of a learning algorithm (usually on unseen data sets). It is mainly used to estimate how accurately a model (learnt by a particular learning algorithm) will perform in practice.

In scikit-learn this could be done using the train_test_split. Let us load the wine dataset to fit a support vector machine and logistic regression on it:

```
from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

X, y = datasets.load_wine(return_X_y=True)
```

We can now quickly sample a training set while holding out 20% of the data for testing (evaluating) our classifier:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf = SVC().fit(X_train, y_train)
clf.score(X_test, y_test)
```

When evaluating different models, such as SVM and Logistic Regression, another part of the dataset should be held out as a so-called **"validation set"**: training proceeds on the training set, after which comparison is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

```
# Further split the train set into a new train and validation sets.

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2)

clf1 = SVC().fit(X_train, y_train)
clf2 = LogisticRegression().fit(X_train, y_train)

print('Support Vector Accuracy on validation: ', clf1.score(X_val, y_val))
print('Logistic Regression Accuracy on validation: ', clf2.score(X_val, y_val))
```

Based on the above results, we will use the testing data to evaluate the best model.

```
# Test the best model performance using testing dataset
print('Logistic Regression Model Final Performance: ', clf2.score(X_test, y_test))
```

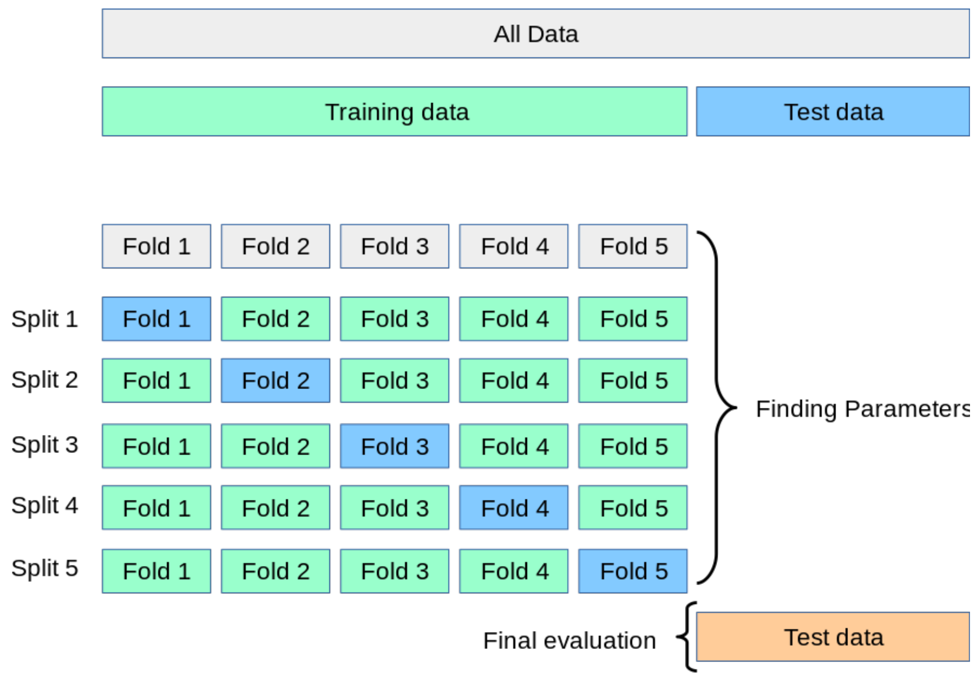
However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and **the results can depend on a particular random choice for the pair of (train, validation) sets**.

K-FOLD CROSS-VALIDATION

A solution to this problem is a procedure called cross-validation (CV for short). A **test set** should still be held out for final evaluation, but the **validation set is no longer needed** when doing CV. In the basic approach, called k-fold CV, the **training set is split into k smaller sets** (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the k "folds":

A model is trained using k-1 of the folds as training data. The resulting model is validated on the remaining part of the data (i.e., it is used as a "test" set to compute a performance measure such as accuracy).

The performance measure reported by k-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive but does not waste too much data (as is the case when fixing an arbitrary validation set).



The simplest way to use cross-validation is to call the cross_val_score helper function on the estimator and the dataset.

The following example demonstrates how to estimate the accuracy of a support vector machine and logistic regression on the wine dataset by splitting the data, fitting a model and computing the score 5 consecutive times (with different splits each time):

```
from sklearn.model_selection import cross_val_score

X, y = datasets.load_wine(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
clf1 = SVC()
clf2 = LogisticRegression()
scores1 = cross_val_score(clf1, X_train, y_train, cv=5)
scores2 = cross_val_score(clf2, X_train, y_train, cv=5)
print('SVC:', scores1)
print('Logistics Regression', scores2)
```

The mean score is hence given by:

```
print('SVC Average Score: {:.2f}'.format(scores1.mean()))
print('Logistics Regression Average Score: {:.2f}'.format(scores2.mean()))
```

3. GRIDSEARCHCV

In this section we will be using different techniques for hyperparameter tuning.

Hyperparameters are parameters that are not directly learnt within estimators. In scikit-learn they are passed as arguments to the constructor of the estimator classes. Typical examples include C, kernel and gamma for Support Vector Classifier, alpha for Lasso, etc.

It is possible and recommended to search the hyperparameter space for the best cross-validation score.

Any parameter provided when constructing an estimator may be optimised in this manner. Specifically, to find the names and current values for all parameters for a given estimator, use:

```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier
model.get_params()
```

A search consists of:

- *an estimator (regressor or classifier such as sklearn.svm.SVC());*
- *a parameter space;*
- *a method for searching or sampling candidates;*
- *a cross-validation scheme; and*
- *a score function.*

Two generic approaches to parameter search are provided in scikit-learn: for given values, GridSearchCV exhaustively considers all parameter combinations, while RandomizedSearchCV can sample a given number of candidates from a parameter space with a specified distribution. Both these tools have successive halving counterparts HalvingGridSearchCV and HalvingRandomSearchCV, which can be much faster at finding a good parameter combination.

Note that it is common that a small subset of those parameters can have a large impact on the predictive or computation performance of the model while others can be left to their default values. It is recommended to read the docstring of the estimator class to get a finer understanding of their expected behavior, possibly by reading the enclosed reference to the literature.

GRIDSEARCHCV

The grid search provided by GridSearchCV exhaustively generates candidates from a grid of parameter values specified with the param_grid parameter.

Let us load the necessary libraries and prepare our dataset:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_percentage_error
from sklearn.metrics import r2_score

# load the california datasets and split
X, y = fetch_california_housing(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

The next step is to define the model and take a look at what hyperparameters we can tune and optimise:

```
# define model
model = KNeighborsRegressor()

# take a look at what are the hyperparameters that we can tune
model.get_params()
```

Then, let us define the hyperparameter space (combination of different hyperparameters):

```
# define hyperparameters space
hyperparameter_space = {'n_neighbors': list(range(1, 101)),
                        'p': [1, 2, 3, 4, 5]}

print(hyperparameter_space)
```

Now, we can use grid search to search for the best hyperparameter combination:

```
# define grid search
grid = GridSearchCV(estimator=model, param_grid=hyperparameter_space, cv=10)
grid.fit(X_train, y_train)
print('The best n_neighbors:', grid.best_estimator_.n_neighbors)
print('The best p:', grid.best_estimator_.p)
```

Since we find out the best hyperparameters already, let us apply the best model to test the data and evaluate it.

```
# evaluate the best model
best_model = grid.best_estimator_
y_pred = best_model.predict(X_test)
rmse = mean_squared_error(y_test, y_pred) ** 0.5
mape = mean_absolute_percentage_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print('For the best model, RMSE is {:.2f}'.format(rmse))
print('For the best model, MAPE is {:.2f}%'.format(mape*100))
print('For the best model, r2 is {:.2f}'.format(r2))
```