ST1511 AI & 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

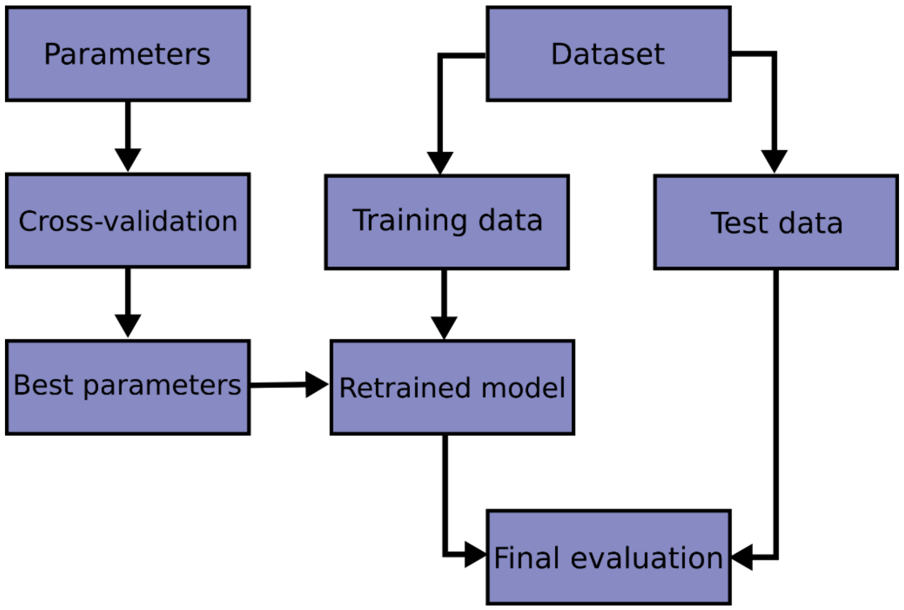
1. OVERVIEW

In this practical we will be exploring what is cross-validation and how do 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 would just 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's load the iris data set to fit a linear support vector machine on it:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn import svm

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

We can now quickly sample a training set while holding out 40% of the data for testing (evaluating) our classifier:

```
X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.4, random_state=0)

clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
clf.score(X_test, y_test)
```

When evaluating different settings ("hyperparameters") for estimators, such as the C setting that must be manually set for an SVM, there is still a risk of overfitting on the test set because the parameters can be tweaked until the estimator performs optimally. This way, **knowledge about the test set can "leak" into the model** and evaluation metrics no longer report on generalization performance.

To solve this problem, yet another part of the dataset can be held out as a so-called **"validation set"**: training proceeds on the training set, after which evaluation 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_validation, y_train, y_validation = train_test_split(
...     X_train, y_train, test_size=0.4, random_state=0)

clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
clf.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**.

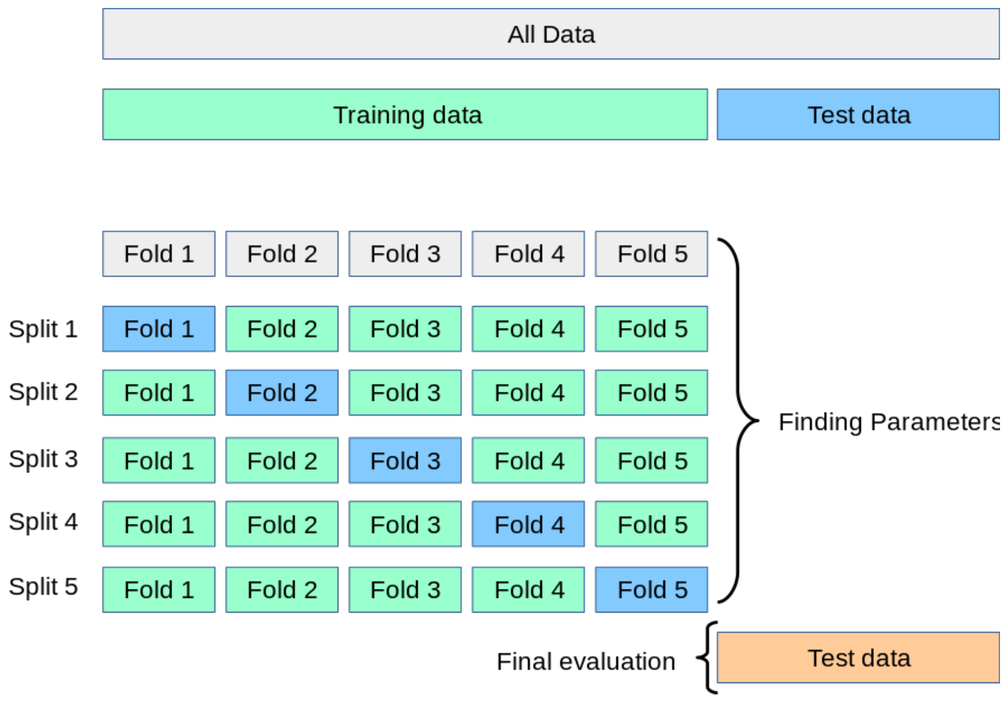
K-FOLD CROSS-VALIDATION

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 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), which is a major advantage in problems such as inverse inference where the number of samples is very small.



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 linear kernel support vector machine on the iris 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
clf = svm.SVC(kernel='linear', C=1, random_state=42)
scores = cross_val_score(clf, X, y, cv=5)
scores
```

The mean score and the standard deviation are hence given by:

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), sc
ores.std()))
```

When the cv argument is an integer, cross_val_score uses the KFold or StratifiedKFold strategies by default, the latter being used if the estimator derives from ClassifierMixin.

It is also possible to use other cross validation strategies by passing a cross validation iterator instead, for instance

```
from sklearn.model_selection import ShuffleSplit
n_samples = X.shape[0]
cv = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)
cross_val_score(clf, X, y, cv=cv)
```

3. GRIDSEARCHCV

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

Hyper-parameters 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 hyper-parameter space for the best cross validation score.

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

```
estimator.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. For instance, the following param_grid:

```
param_grid = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},
 ]
```

specifies that two grids should be explored: one with a linear kernel and C values in [1, 10, 100, 1000], and the second one with an RBF kernel, and the cross-product of C values ranging in [1, 10, 100, 1000] and gamma values in [0.001, 0.0001].

The GridSearchCV instance implements the usual estimator API: when “fitting” it on a dataset all the possible combinations of parameter values are evaluated and the best combination is retained.

```
from sklearn import svm, datasets
from sklearn.model_selection import GridSearchCV
iris = datasets.load_iris()
parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
svc = svm.SVC()
clf = GridSearchCV(svc, parameters)
clf.fit(iris.data, iris.target)
```

RANDOMIZEDSEARCHCV

While using a grid of parameter settings is currently the most widely used method for parameter optimization, other search methods have more favourable properties. RandomizedSearchCV implements a randomized search over parameters, where each setting is sampled from a distribution over possible parameter values. This has two main benefits over an exhaustive search:

A budget can be chosen independent of the number of parameters and possible values.

Adding parameters that do not influence the performance does not decrease efficiency.

Specifying how parameters should be sampled is done using a dictionary, very similar to specifying parameters for GridSearchCV. Additionally, a computation budget, being the number of sampled candidates or sampling iterations, is specified using the n_iter parameter. For each parameter, either a distribution over possible values or a list of discrete choices (which will be sampled uniformly) can be specified:

```
{'C': scipy.stats.expon(scale=100), 'gamma': scipy.stats.expon(scale=.1),
  'kernel': ['rbf'], 'class_weight':['balanced', None]}
```

Example of using RandomizedSearchCV

```
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform
iris = load_iris()
logistic = LogisticRegression(solver='saga', tol=1e-2, max_iter=200,
...                                   random_state=0)
distributions = dict(C=uniform(loc=0, scale=4),
...                         penalty=['l2', 'l1'])
clf = RandomizedSearchCV(logistic, distributions, random_state=0)
search = clf.fit(iris.data, iris.target)
search.best_params_
```