3.2. Grid Search: Searching for estimator parameters

Parameters that are not directly learnt within estimators can be set by searching a parameter space for the best *Cross-validation: evaluating estimator performance* score. Typical examples include `C`, `kernel` and `gamma` for Support Vector Classifier, `alpha` for Lasso, etc.

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()
```

Such parameters are often referred to as *hyperparameters* (particularly in Bayesian learning), distinguishing them from the parameters optimised in a machine learning procedure.

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 sampling search candidates 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.

3.2.1. Exhaustive Grid Search

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.

Model selection: development and evaluation

Model selection with `GridSearchCV` can be seen as a way to use the labeled data to “train” the parameters of the grid.

When evaluating the resulting model it is important to do it on held-out samples that were not seen during the grid search process: it is recommended to split the data into a **development set** (to be fed to the `GridSearchCV` instance) and an **evaluation set** to compute performance metrics.

This can be done by using the **`cross_validation.train_test_split`** utility function.

3.2.1.1. Scoring functions for parameter search

By default, **`GridSearchCV`** uses the `score` function of the estimator to evaluate a parameter setting. These are the **`sklearn.metrics.accuracy_score`** for classification and **`sklearn.metrics.r2_score`** for regression. For some applications, other scoring functions are better suited (for example in unbalanced classification, the accuracy score is often uninformative). An alternative scoring function can be specified via the `scoring` parameter to **`GridSearchCV`**. See *The scoring parameter: defining model evaluation rules* for more details.

Examples:

- See *Parameter estimation using grid search with cross-validation* for an example of Grid Search computation on the digits dataset.
- See *Sample pipeline for text feature extraction and evaluation* for an example of Grid Search coupling parameters from a text documents feature extractor (n-gram count vectorizer and TF-IDF transformer) with a classifier (here a linear SVM trained with SGD with either elastic net or L2 penalty) using a **`pipeline.Pipeline`** instance.

Note: Computations can be run in parallel if your OS supports it, by using the keyword `n_jobs=-1`, see function signature for more details.

3.2.2. Randomized Parameter Optimization

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':['auto', None]}]
```

This example uses the `scipy.stats` module, which contains many useful distributions for sampling parameters, such as `expon`, `gamma`, `uniform` or `randint`. In principle, any function can be passed that provides a `rvs` (random variate sample) method to sample a value. A call to the `rvs` function should provide independent random samples from possible parameter values on consecutive calls.

> **Warning:** The distributions in `scipy.stats` do not allow specifying a random state. Instead, they use the global numpy random state, that can be seeded via `np.random.seed` or set using `np.random.set_state`.

For continuous parameters, such as `C` above, it is important to specify a continuous distribution to take full advantage of the randomization. This way, increasing `n_iter` will always lead to a finer search.

Examples:

- *Comparing randomized search and grid search for hyperparameter estimation* compares the usage and efficiency of randomized search and grid search.

References:

- Bergstra, J. and Bengio, Y., Random search for hyper-parameter optimization, The Journal of Machine Learning Research (2012)

3.2.3. Alternatives to brute force parameter search

3.2.3.1. Model specific cross-validation

Some models can fit data for a range of value of some parameter almost as efficiently as fitting the estimator for a single value of the parameter. This feature can be leveraged to perform a more efficient cross-validation used for model selection of this parameter.

The most common parameter amenable to this strategy is the parameter encoding the strength of the regularizer. In this case we say that we compute the **regularization path** of the estimator.

Here is the list of such models:

`linear_model.RidgeCV`([alphas, ...])	Ridge regression with built-in cross-validation.
`linear_model.RidgeClassifierCV`([alphas, ...])	Ridge classifier with built-in cross-validation.
`linear_model.LarsCV`([fit_intercept, ...])	Cross-validated Least Angle Regression model
`linear_model.LassoLarsCV`([fit_intercept, ...])	Cross-validated Lasso, using the LARS algorithm
`linear_model.LassoCV`([eps, n_alphas, ...])	Lasso linear model with iterative fitting along a regularization path

`linear_model.ElasticNetCV`([l1_ratio, eps, ...])	Elastic Net model with iterative fitting along a regularization path

3.2.3.2. Information Criterion

Some models can offer an information-theoretic closed-form formula of the optimal estimate of the regularization parameter by computing a single regularization path (instead of several when using cross-validation).

Here is the list of models benefitting from the Aikike Information Criterion (AIC) or the Bayesian Information Criterion (BIC) for automated model selection:

`linear_model.LassoLarsIC`([criterion, ...])	Lasso model fit with Lars using BIC or AIC for model selection

3.2.3.3. Out of Bag Estimates

When using ensemble methods base upon bagging, i.e. generating new training sets using sampling with replacement, part of the training set remains unused. For each classifier in the ensemble, a different part of the training set is left out.

This left out portion can be used to estimate the generalization error without having to rely on a separate validation set. This estimate comes “for free” as no additional data is needed and can be used for model selection.

This is currently implemented in the following classes:

`ensemble.RandomForestClassifier`([...])	A random forest classifier.
`ensemble.RandomForestRegressor`([...])	A random forest regressor.
`ensemble.ExtraTreesClassifier`([...])	An extra-trees classifier.
`ensemble.ExtraTreesRegressor`([n_estimators, ...])	An extra-trees regressor.
`ensemble.GradientBoostingClassifier`([loss, ...])	Gradient Boosting for classification.
`ensemble.GradientBoostingRegressor`([loss, ...])	Gradient Boosting for regression.