3.2. Tuning the hyper-parameters of an estimator

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.

Some models allow for specialized, efficient parameter search strategies, outlined below. 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. After describing these tools we detail best practice applicable to both approaches.

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.

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.

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.
- See Nested versus non-nested cross-validation for an example of Grid Search within a cross validation loop on the iris dataset. This is the best practice for evaluating the performance of a model with grid search.
- See Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV for an example of **`GridSearchCV`** being used to evaluate multiple metrics simultaneously.
- See Balance model complexity and cross-validated score for an example of using `refit=callable` interface in **`GridSearchCV`**. The example shows how this interface adds certain amount of flexibility in identifying the “best” estimator. This interface can also be used in multiple metrics evaluation.

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':['balanced', 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` prior to version scipy 0.16 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`. However, beginning scikit-learn 0.18, the `sklearn.model_selection` module sets the random state provided by the user if scipy >= 0.16 is also available.

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.

A continuous log-uniform random variable is available through `loguniform`. This is a continuous version of log-spaced parameters. For example to specify `C` above, `loguniform(1, 100)` can be used instead of `[1, 10, 100]` or `np.logspace(0, 2, num=1000)`. This is an alias to SciPy's stats.reciprocal.

Mirroring the example above in grid search, we can specify a continuous random variable that is log-uniformly distributed between `1e0` and `1e3`:

```
from sklearn.utils.fixes import loguniform
{'C': loguniform(1e0, 1e3),
 'gamma': loguniform(1e-4, 1e-3),
 'kernel': ['rbf'],
 'class_weight':['balanced', None]}
```

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. Tips for parameter search

3.2.3.1. Specifying an objective metric

By default, parameter search 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`, `RandomizedSearchCV` and many of the specialized cross-validation tools described below. See The scoring parameter: defining model evaluation rules for more details.

3.2.3.2. Specifying multiple metrics for evaluation

`GridSearchCV` and `RandomizedSearchCV` allow specifying multiple metrics for the `scoring` parameter.

Multimetric scoring can either be specified as a list of strings of predefined scores names or a dict mapping the scorer name to the scorer function and/or the predefined scorer name(s). See Using multiple metric evaluation for more details.

When specifying multiple metrics, the `refit` parameter must be set to the metric (string) for which the `best_params_` will be found and used to build the `best_estimator_` on the whole dataset. If the search should not be refit, set `refit=False`. Leaving refit to the default value `None` will result in an error when using multiple metrics.

See Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV for an example usage.

3.2.3.3. Composite estimators and parameter spaces

`GridSearchCV` and `RandomizedSearchCV` allow searching over parameters of composite or nested estimators such as `Pipeline`, `ColumnTransformer`, `VotingClassifier` or `CalibratedClassifierCV` using a dedicated `<estimator>__<parameter>` syntax:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.calibration import CalibratedClassifierCV
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons()
>>> calibrated_forest = CalibratedClassifierCV(
...     base_estimator=RandomForestClassifier(n_estimators=10))
>>> param_grid = {
...     'base_estimator__max_depth': [2, 4, 6, 8]}
>>> search = GridSearchCV(calibrated_forest, param_grid, cv=5)
>>> search.fit(X, y)
GridSearchCV(cv=5,
             estimator=CalibratedClassifierCV(...),
             param_grid={'base_estimator__max_depth': [2, 4, 6, 8]})
```

Here, `<estimator>` is the parameter name of the nested estimator, in this case `base_estimator`. If the meta-estimator is constructed as a collection of estimators as in `pipeline.Pipeline`, then `<estimator>` refers to the name of the estimator, see Nested parameters. In practice, there can be several levels of nesting:

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.feature_selection import SelectKBest
>>> pipe = Pipeline([
...     ('select', SelectKBest()),
...     ('model', calibrated_forest)])
>>> param_grid = {
...     'select__k': [1, 2],
...     'model__base_estimator__max_depth': [2, 4, 6, 8]}
>>> search = GridSearchCV(pipe, param_grid, cv=5).fit(X, y)
```

3.2.3.4. Model selection: development and evaluation

Model selection by evaluating various parameter settings 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 `train_test_split` utility function.

3.2.3.5. Parallelism

`GridSearchCV` and `RandomizedSearchCV` evaluate each parameter setting independently. Computations can be run in parallel if your OS supports it, by using the keyword `n_jobs=-1`. See function signature for more details.

3.2.3.6. Robustness to failure

Some parameter settings may result in a failure to `fit` one or more folds of the data. By default, this will cause the entire search to fail, even if some parameter settings could be fully evaluated. Setting `error_score=0` (or `=np.NaN`) will make the procedure robust to such failure, issuing a warning and setting the score for that fold to 0 (or `NaN`), but completing the search.

3.2.4. Alternatives to brute force parameter search

3.2.4.1. Model specific cross-validation

Some models can fit data for a range of values 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.ElasticNetCV`([l1_ratio, eps, …])	Elastic Net model with iterative fitting along a regularization path.
`linear_model.LarsCV`([fit_intercept, …])	Cross-validated Least Angle Regression model.
`linear_model.LassoCV`([eps, n_alphas, …])	Lasso linear model with iterative fitting along a regularization path.
`linear_model.LassoLarsCV`([fit_intercept, …])	Cross-validated Lasso, using the LARS algorithm.
`linear_model.LogisticRegressionCV`([Cs, …])	Logistic Regression CV (aka logit, MaxEnt) classifier.
`linear_model.MultiTaskElasticNetCV`([…])	Multi-task L1/L2 ElasticNet with built-in cross-validation.
`linear_model.MultiTaskLassoCV`([eps, …])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.
`linear_model.OrthogonalMatchingPursuitCV`([…])	Cross-validated Orthogonal Matching Pursuit model (OMP).
`linear_model.RidgeCV`([alphas, …])	Ridge regression with built-in cross-validation.
`linear_model.RidgeClassifierCV`([alphas, …])	Ridge classifier with built-in cross-validation.

3.2.4.2. 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 benefiting from the Akaike 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

3.2.4.3. 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.