Lesson 3 GridSearchCV的基础及进阶使用方法

1.sklearn中GridSearchCV的使用方法

In [1]:
```
# 科学计算模块
import numpy as np
import pandas as pd

# Scikit-Learn相关模块
# 评估器类
from sklearn.linear_model import LogisticRegression

# 实用函数
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
```

In [2]:
```
from sklearn.model_selection import GridSearchCV
```

In [39]:
```
GridSearchCV?
```

```
Init signature: GridSearchCV(estimator, param_grid, *, scoring=None, n_jobs=None,
refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', error_score=nan, return_t
rain_score=False)
Docstring:
Exhaustive search over specified parameter values for an estimator.

Important members are fit, predict.

GridSearchCV implements a "fit" and a "score" method.
It also implements "score_samples", "predict", "predict_proba",
"decision_function", "transform" and "inverse_transform" if they are
implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized
by cross-validated grid-search over a parameter grid.

Read more in the :ref:`User Guide <grid_search>`.

Parameters
----------
estimator : estimator object.
    This is assumed to implement the scikit-learn estimator interface.
    Either estimator needs to provide a ``score`` function,
    or ``scoring`` must be passed.

param_grid : dict or list of dictionaries
    Dictionary with parameters names (`str`) as keys and lists of
    parameter settings to try as values, or a list of such
    dictionaries, in which case the grids spanned by each dictionary
    in the list are explored. This enables searching over any sequence
    of parameter settings.

scoring : str, callable, list, tuple or dict, default=None
    Strategy to evaluate the performance of the cross-validated model on
    the test set.

    If `scoring` represents a single score, one can use:

    - a single string (see :ref:`scoring_parameter`);
    - a callable (see :ref:`scoring`) that returns a single value.

    If `scoring` represents multiple scores, one can use:

    - a list or tuple of unique strings;
    - a callable returning a dictionary where the keys are the metric
      names and the values are the metric scores;
    - a dictionary with metric names as keys and callables a values.

    See :ref:`multimetric_grid_search` for an example.

n_jobs : int, default=None
    Number of jobs to run in parallel.
    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.

    .. versionchanged:: v0.20
       `n_jobs` default changed from 1 to None

refit : bool, str, or callable, default=True
    Refit an estimator using the best found parameters on the whole
    dataset.

    For multiple metric evaluation, this needs to be a `str` denoting the
    scorer that would be used to find the best parameters for refitting
```

```
        the estimator at the end.

        Where there are considerations other than maximum score in
        choosing a best estimator, ``refit`` can be set to a function which
        returns the selected ``best_index_`` given ``cv_results_``. In that
        case, the ``best_estimator_`` and ``best_params_`` will be set
        according to the returned ``best_index_`` while the ``best_score_``
        attribute will not be available.

        The refitted estimator is made available at the ``best_estimator_``
        attribute and permits using ``predict`` directly on this
        ``GridSearchCV`` instance.

        Also for multiple metric evaluation, the attributes ``best_index_``,
        ``best_score_`` and ``best_params_`` will only be available if
        ``refit`` is set and all of them will be determined w.r.t this specific
        scorer.

        See ``scoring`` parameter to know more about multiple metric
        evaluation.

        .. versionchanged:: 0.20
            Support for callable added.

    cv : int, cross-validation generator or an iterable, default=None
        Determines the cross-validation splitting strategy.
        Possible inputs for cv are:

        - None, to use the default 5-fold cross validation,
        - integer, to specify the number of folds in a `(Stratified)KFold`,
        - :term:`CV splitter`,
        - An iterable yielding (train, test) splits as arrays of indices.

        For integer/None inputs, if the estimator is a classifier and ``y`` is
        either binary or multiclass, :class:`StratifiedKFold` is used. In all
        other cases, :class:`KFold` is used. These splitters are instantiated
        with `shuffle=False` so the splits will be the same across calls.

        Refer :ref:`User Guide <cross_validation>` for the various
        cross-validation strategies that can be used here.

        .. versionchanged:: 0.22
            ``cv`` default value if None changed from 3-fold to 5-fold.

    verbose : int
        Controls the verbosity: the higher, the more messages.

        - >1 : the computation time for each fold and parameter candidate is
          displayed;
        - >2 : the score is also displayed;
        - >3 : the fold and candidate parameter indexes are also displayed
          together with the starting time of the computation.

    pre_dispatch : int, or str, default=n_jobs
        Controls the number of jobs that get dispatched during parallel
        execution. Reducing this number can be useful to avoid an
        explosion of memory consumption when more jobs get dispatched
        than CPUs can process. This parameter can be:

            - None, in which case all the jobs are immediately
              created and spawned. Use this for lightweight and
              fast-running jobs, to avoid delays due to on-demand
              spawning of the jobs

            - An int, giving the exact number of total jobs that are
```

```
                spawned

              - A str, giving an expression as a function of n_jobs,
                as in '2*n_jobs'

    error_score : 'raise' or numeric, default=np.nan
        Value to assign to the score if an error occurs in estimator fitting.
        If set to 'raise', the error is raised. If a numeric value is given,
        FitFailedWarning is raised. This parameter does not affect the refit
        step, which will always raise the error.

    return_train_score : bool, default=False
        If ``False``, the ``cv_results_`` attribute will not include training
        scores.
        Computing training scores is used to get insights on how different
        parameter settings impact the overfitting/underfitting trade-off.
        However computing the scores on the training set can be computationally
        expensive and is not strictly required to select the parameters that
        yield the best generalization performance.

        .. versionadded:: 0.19

        .. versionchanged:: 0.21
            Default value was changed from ``True`` to ``False``


    Examples
    --------
    >>> 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)
    GridSearchCV(estimator=SVC(),
                 param_grid={'C': [1, 10], 'kernel': ('linear', 'rbf')})
    >>> sorted(clf.cv_results_.keys())
    ['mean_fit_time', 'mean_score_time', 'mean_test_score',...
     'param_C', 'param_kernel', 'params',...
     'rank_test_score', 'split0_test_score',...
     'split2_test_score', ...
     'std_fit_time', 'std_score_time', 'std_test_score']

    Attributes
    ----------
    cv_results_ : dict of numpy (masked) ndarrays
        A dict with keys as column headers and values as columns, that can be
        imported into a pandas ``DataFrame``.

        For instance the below given table

        +------------+-----------+------------+-----------------+---+---------+
        |param_kernel|param_gamma|param_degree|split0_test_score|...|rank_t...|
        +============+===========+============+=================+===+=========+
        |  'poly'    |     --    |      2     |       0.80      |...|    2    |
        +------------+-----------+------------+-----------------+---+---------+
        |  'poly'    |     --    |      3     |       0.70      |...|    4    |
        +------------+-----------+------------+-----------------+---+---------+
        |  'rbf'     |     0.1   |     --     |       0.80      |...|    3    |
        +------------+-----------+------------+-----------------+---+---------+
        |  'rbf'     |     0.2   |     --     |       0.93      |...|    1    |
        +------------+-----------+------------+-----------------+---+---------+

        will be represented by a ``cv_results_`` dict of::
```

```
        {
        'param_kernel': masked_array(data = ['poly', 'poly', 'rbf', 'rbf'],
                                     mask = [False False False False]...)
        'param_gamma': masked_array(data = [-- -- 0.1 0.2],
                                    mask = [ True  True False False]...),
        'param_degree': masked_array(data = [2.0 3.0 -- --],
                                     mask = [False False  True  True]...),
        'split0_test_score'  : [0.80, 0.70, 0.80, 0.93],
        'split1_test_score'  : [0.82, 0.50, 0.70, 0.78],
        'mean_test_score'    : [0.81, 0.60, 0.75, 0.85],
        'std_test_score'     : [0.01, 0.10, 0.05, 0.08],
        'rank_test_score'    : [2, 4, 3, 1],
        'split0_train_score' : [0.80, 0.92, 0.70, 0.93],
        'split1_train_score' : [0.82, 0.55, 0.70, 0.87],
        'mean_train_score'   : [0.81, 0.74, 0.70, 0.90],
        'std_train_score'    : [0.01, 0.19, 0.00, 0.03],
        'mean_fit_time'      : [0.73, 0.63, 0.43, 0.49],
        'std_fit_time'       : [0.01, 0.02, 0.01, 0.01],
        'mean_score_time'    : [0.01, 0.06, 0.04, 0.04],
        'std_score_time'     : [0.00, 0.00, 0.00, 0.01],
        'params'             : [{'kernel': 'poly', 'degree': 2}, ...],
        }

    NOTE

    The key ``'params'`` is used to store a list of parameter
    settings dicts for all the parameter candidates.

    The ``mean_fit_time``, ``std_fit_time``, ``mean_score_time`` and
    ``std_score_time`` are all in seconds.

    For multi-metric evaluation, the scores for all the scorers are
    available in the ``cv_results_`` dict at the keys ending with that
    scorer's name (``'_<scorer_name>'``) instead of ``'_score'`` shown
    above. ('split0_test_precision', 'mean_train_precision' etc.)

best_estimator_ : estimator
    Estimator that was chosen by the search, i.e. estimator
    which gave highest score (or smallest loss if specified)
    on the left out data. Not available if ``refit=False``.

    See ``refit`` parameter for more information on allowed values.

best_score_ : float
    Mean cross-validated score of the best_estimator

    For multi-metric evaluation, this is present only if ``refit`` is
    specified.

    This attribute is not available if ``refit`` is a function.

best_params_ : dict
    Parameter setting that gave the best results on the hold out data.

    For multi-metric evaluation, this is present only if ``refit`` is
    specified.

best_index_ : int
    The index (of the ``cv_results_`` arrays) which corresponds to the best
    candidate parameter setting.

    The dict at ``search.cv_results_['params'][search.best_index_]`` gives
    the parameter setting for the best model, that gives the highest
    mean score (``search.best_score_``).
```

```
        For multi-metric evaluation, this is present only if ``refit`` is
        specified.

    scorer_ : function or a dict
        Scorer function used on the held out data to choose the best
        parameters for the model.

        For multi-metric evaluation, this attribute holds the validated
        ``scoring`` dict which maps the scorer key to the scorer callable.

    n_splits_ : int
        The number of cross-validation splits (folds/iterations).

    refit_time_ : float
        Seconds used for refitting the best model on the whole dataset.

        This is present only if ``refit`` is not False.

        .. versionadded:: 0.20

    multimetric_ : bool
        Whether or not the scorers compute several metrics.

    Notes
    -----
    The parameters selected are those that maximize the score of the left out
    data, unless an explicit score is passed in which case it is used instead.

    If `n_jobs` was set to a value higher than one, the data is copied for each
    point in the grid (and not `n_jobs` times). This is done for efficiency
    reasons if individual jobs take very little time, but may raise errors if
    the dataset is large and not enough memory is available.  A workaround in
    this case is to set `pre_dispatch`. Then, the memory is copied only
    `pre_dispatch` many times. A reasonable value for `pre_dispatch` is `2 *
    n_jobs`.

    See Also
    ---------
    ParameterGrid : Generates all the combinations of a hyperparameter grid.
    train_test_split : Utility function to split the data into a development
        set usable for fitting a GridSearchCV instance and an evaluation set
        for its final evaluation.
    sklearn.metrics.make_scorer : Make a scorer from a performance metric or
        loss function.
File:           c:\users\user\anaconda3\lib\site-packages\sklearn\model_selection\_se
arch.py
Type:           ABCMeta
```

Name	Description
estimator	调参对象，某评估器
param_grid	参数空间，可以是字典或者字典构成的列表，稍后介绍参数空间的创建方法
scoring	评估指标，支持同时输出多个参数
n_jobs	设置工作时参与计算的CPU逻辑核数
refit	挑选评估指标和最佳参数，在完整数据集上进行训练
cv	交叉验证的折数
verbose	输出工作日志形式
pre_dispatch	多任务并行时任务划分数量

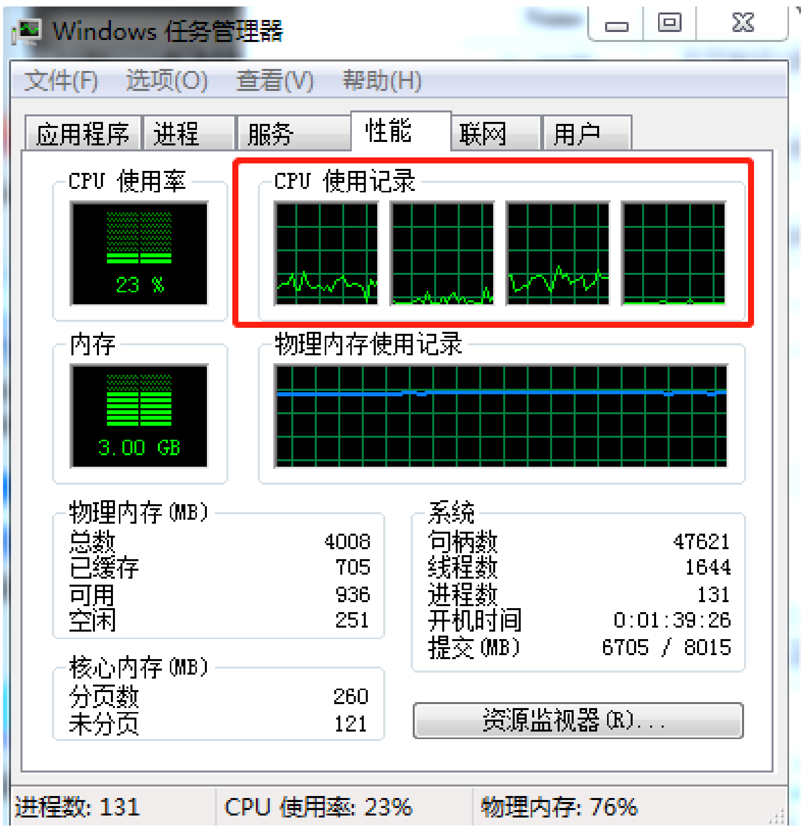
Name	Description
error_score	当网格搜索报错时返回结果，选择'raise'时将直接报错并中断训练过程，其他情况会显示警告信息后继续完成训练
return_train_score	在交叉验证中是否显示训练集中参数得分

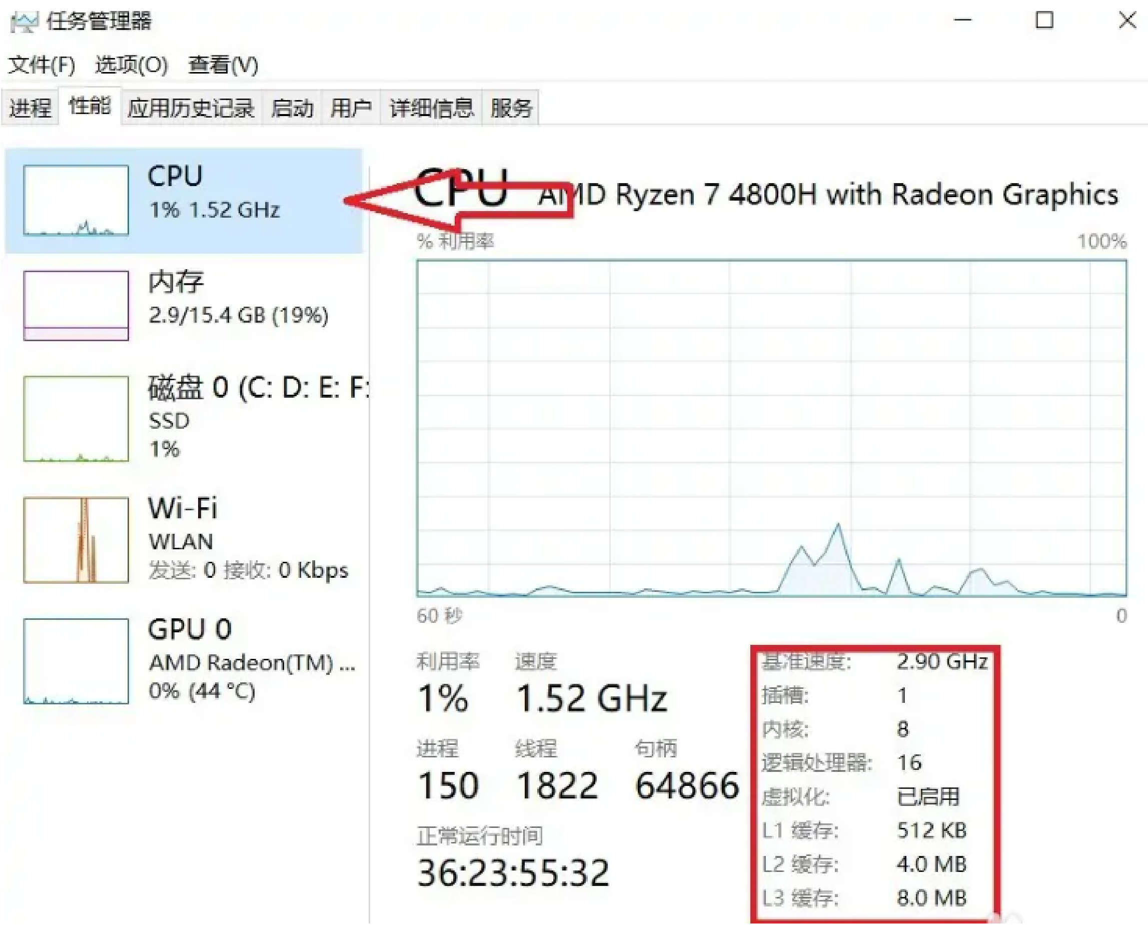
- n_jobs：设置工作时参与计算的CPU逻辑核数。值为-1表示使用全部CPU资源进行并行计算。

> CPU个数、核数、逻辑核数的概念辨析
>
> （1）CPU个数： 电脑插槽上的CPU个数, 物理cpu数量
>
> （2）CPU核数：一个物理CPU上面能处理数据的芯片组的数量
>
> （3）CPU逻辑核数：一般情况，我们认为一颗cpu可以有多核，加上intel的超线程技术(HT), 可以在逻辑上把一个物理线程模拟出两个线程来使用，使得单个核心用起来像两个核一样，以充分发挥CPU的性能
>
> -> 总核数 = 物理CPU个数 X 每个物理CPU的核数
>
> -> 总逻辑核数= 物理CPU个数 X 每个物理CPU的核数 X 超线程数

- 关于CPU逻辑核数的查看方法：

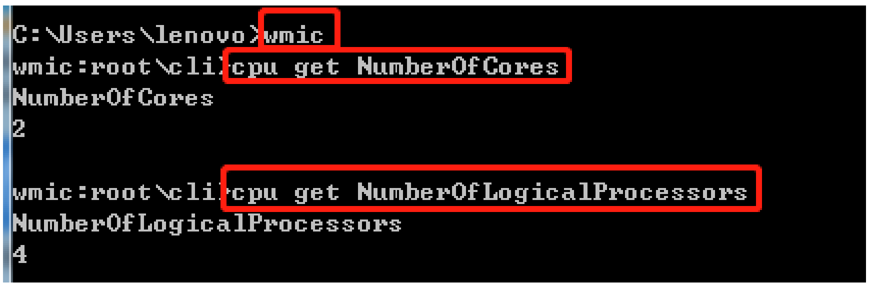
> (1) 可以打开"任务管理器"-"性能"窗口中，在"cpu使用记录"中看到四个窗口，表示有四个逻辑核数。win10系统可在"性能"中看到"逻辑处理器"数量为16，则表示有16个逻辑核数。





> (2) 查看CPU核数：在cmd命令中输入wmic，然后在出现的新窗口中输入cpu get NumberOfCores

查看CPU逻辑核数：在cmd命令中输入wmic，然后在出现的新窗口中输入cpu get NumberOfLogicalProcessors



1.1 GridSearchCV评估器训练过程

- Step 1.创建评估器

首先我们还是需要实例化一个评估器，这里可以是一个模型、也可以是一个机器学习流，网格搜索都可以对其进行调参。此处我们先从简单入手，尝试实例化逻辑回归模型并对其进行调参。

In [40]:
```
# 数据导入
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=24)
```

In [3]:
```
clf = LogisticRegression(max_iter=int(1e6), solver='saga')
```

> 此处将solver设置成saga，也是为了方便后续同时比较l1正则化和l2正则化时无需更换求解器。

In [42]:
```
clf.get_params()
```

Out[42]:
```
{'C': 1.0,
 'class_weight': None,
 'dual': False,
 'fit_intercept': True,
 'intercept_scaling': 1,
 'l1_ratio': None,
 'max_iter': 1000000,
 'multi_class': 'auto',
 'n_jobs': None,
 'penalty': 'l2',
 'random_state': None,
 'solver': 'saga',
 'tol': 0.0001,
 'verbose': 0,
 'warm_start': False}
```

- Step 2.创建参数空间

此处我们挑选penalty和C这两个参数来进行参数空间的构造。参数空间首先可以是一个字典：

In [43]:
```
param_grid_simple = {'penalty': ['l1', 'l2'],
                     'C': [1, 0.5, 0.1, 0.05, 0.01]}
```

其中，字典的Key用参数的字符串来代表不同的参数，对应的Value则用列表来表示对应参数不同的取值范围。也就是字典的Key是参数空间的维度，而Value则是不同纬度上可选的取值。而后续的网格搜索则是在上述参数的不同组合中挑选出一组最优的参数取值。

我们可以创造多个参数空间（字典），然后将其封装在一个列表中，而该列表则表示多个参数空间的集成。

In [44]:
```
param_grid_multi = [
    {'penalty': ['l1', 'l2'], 'C': [1, 0.5, 0.1, 0.05, 0.01]},
    {'penalty': ['elasticnet'], 'C': [1, 0.5, 0.1, 0.05, 0.01], 'l1_ratio': [0.3, 0.6,
]
```

- Step 3.实例化网格搜索评估器

In [45]:
```
search = GridSearchCV(estimator=clf,
                      param_grid=param_grid_simple)
```

- Step 4.训练网格搜索评估器

In [46]:
```
search.fit(X_train, y_train)
```

Out[46]:
```
GridSearchCV(estimator=LogisticRegression(max_iter=1000000, solver='saga'),
             param_grid={'C': [1, 0.5, 0.1, 0.05, 0.01],
                         'penalty': ['l1', 'l2']})
```

1.2 GridSearchCV评估器结果查看

Name	Description
cv*results*	交叉验证过程中的重要结果
best*estimator*	最终挑选出的最优
best*score*	在最优参数情况下，训练集的交叉验证的平均得分
best*params*	最优参数组合
best*index*	CV过程会对所有参数组合标号，该参数表示最优参数组合的标号
scorer	在最优参数下，计算模型得分的方法
n*splits*	交叉验证的折数

In [47]:
```
#best_estimator_: 训练完成后的最佳评估器
search.best_estimator_
```
Out[47]:
```
LogisticRegression(C=1, max_iter=1000000, penalty='l1', solver='saga')
```

In [48]:
```
#best_score_: 最优参数时交叉验证时验证集准确率的平均值，而不是所有数据的准确率
search.best_score_
```
Out[48]:
```
0.9644268774703558
```

In [49]:
```
search.best_params_  #最优参数组合
```
Out[49]:
```
{'C': 1, 'penalty': 'l1'}
```

In [50]:
```
search.best_index_ ##最优参数组合的索引
```
Out[50]:
```
0
```

In [51]:
```
# 查看参数
search.best_estimator_.coef_
```
Out[51]:
```
array([[ 0.        ,  0.        , -3.4734417 ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ],
       [-0.5550736 , -0.34229207,  3.03236645,  4.12146594]])
```

In [52]:
```
# 查看训练误差、测试误差
search.best_estimator_.score(X_train,y_train), search.best_estimator_.score(X_test,y_t
```
Out[52]:
```
(0.9732142857142857, 0.9736842105263158)
```

In [53]:
```
# 等价于search.best_estimator_.score
search.score(X_train,y_train), search.score(X_test,y_test)
```
Out[53]:
```
(0.9732142857142857, 0.9736842105263158)
```

In [54]:
```
search.n_splits_
```
Out[54]:
```
5
```

In [55]:
```
search.refit_time_ #在整个训练集上训练最佳模型的用时
```
Out[55]:
```
0.10850024223327637
```

In [56]:
```
search.cv_results_
```

Out[56]:
```
{'mean_fit_time': array([0.08030992, 0.04200077, 0.05280018, 0.02789931, 0.01040025,
        0.01521077, 0.00429974, 0.01109977, 0.00080037, 0.00449982]),
 'std_fit_time': array([8.12094464e-03, 1.24215189e-02, 2.15884069e-03, 7.34262965e-0
4,
        6.63032876e-04, 3.94517573e-04, 2.44776570e-04, 1.99462722e-04,
        2.44697944e-04, 3.56832255e-07]),
 'mean_score_time': array([3.99827957e-04, 3.99780273e-04, 3.99780273e-04, 2.00319290e
-04,
        2.99882889e-04, 3.00693512e-04, 9.98973846e-05, 3.00407410e-04,
        1.99747086e-04, 1.99937820e-04]),
 'std_score_time': array([0.00019993, 0.00019989, 0.00019989, 0.00024534, 0.00024485,
        0.00024552, 0.00019979, 0.00024528, 0.00024464, 0.00024487]),
 'param_C': masked_array(data=[1, 1, 0.5, 0.5, 0.1, 0.1, 0.05, 0.05, 0.01, 0.01],
              mask=[False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_penalty': masked_array(data=['l1', 'l2', 'l1', 'l2', 'l1', 'l2', 'l1', 'l2',
'l1',
                    'l2'],
              mask=[False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'C': 1, 'penalty': 'l1'},
  {'C': 1, 'penalty': 'l2'},
  {'C': 0.5, 'penalty': 'l1'},
  {'C': 0.5, 'penalty': 'l2'},
  {'C': 0.1, 'penalty': 'l1'},
  {'C': 0.1, 'penalty': 'l2'},
  {'C': 0.05, 'penalty': 'l1'},
  {'C': 0.05, 'penalty': 'l2'},
  {'C': 0.01, 'penalty': 'l1'},
  {'C': 0.01, 'penalty': 'l2'}],
 'split0_test_score': array([1.        , 1.        , 1.        , 1.        , 1.
,
        1.        , 0.82608696, 1.        , 0.30434783, 0.91304348]),
 'split1_test_score': array([0.91304348, 0.91304348, 0.82608696, 0.86956522, 0.8260869
6,
        0.73913043, 0.69565217, 0.73913043, 0.39130435, 0.69565217]),
 'split2_test_score': array([1.        , 1.        , 1.        , 1.        , 0.9545454
5,
        0.95454545, 0.86363636, 0.90909091, 0.36363636, 0.86363636]),
 'split3_test_score': array([0.95454545, 0.95454545, 0.95454545, 0.90909091, 0.9545454
5,
        0.95454545, 0.86363636, 0.90909091, 0.36363636, 0.90909091]),
 'split4_test_score': array([0.95454545, 0.95454545, 0.95454545, 0.95454545, 0.9545454
5,
        0.90909091, 0.86363636, 0.95454545, 0.36363636, 0.90909091]),
 'mean_test_score': array([0.96442688, 0.96442688, 0.94703557, 0.94664032, 0.93794466,
        0.91146245, 0.82252964, 0.90237154, 0.35731225, 0.85810277]),
 'std_test_score': array([0.03276105, 0.03276105, 0.06379941, 0.05120065, 0.05863407,
        0.09083516, 0.06508431, 0.08830786, 0.02856808, 0.08323326]),
 'rank_test_score': array([ 1,  1,  3,  4,  5,  6,  9,  7, 10,  8])}
```

In [57]:
```
pd.DataFrame(search.cv_results_) #可以pandas的DataFrame形式展示，rank_test_score为验证
```

Out[57]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_penalty	params
0	0.080310	8.120945e-03	0.000400	0.000200	1	l1	{'C': 1, 'penalty': 'l1'}
1	0.042001	1.242152e-02	0.000400	0.000200	1	l2	{'C': 1, 'penalty': 'l2'}
2	0.052800	2.158841e-03	0.000400	0.000200	0.5	l1	{'C': 0.5, 'penalty': 'l1'}
3	0.027899	7.342630e-04	0.000200	0.000245	0.5	l2	{'C': 0.5, 'penalty': 'l2'}
4	0.010400	6.630329e-04	0.000300	0.000245	0.1	l1	{'C': 0.1, 'penalty': 'l1'}
5	0.015211	3.945176e-04	0.000301	0.000246	0.1	l2	{'C': 0.1, 'penalty': 'l2'}
6	0.004300	2.447766e-04	0.000100	0.000200	0.05	l1	{'C': 0.05, 'penalty': 'l1'}
7	0.011100	1.994627e-04	0.000300	0.000245	0.05	l2	{'C': 0.05, 'penalty': 'l2'}
8	0.000800	2.446979e-04	0.000200	0.000245	0.01	l1	{'C': 0.01, 'penalty': 'l1'}
9	0.004500	3.568323e-07	0.000200	0.000245	0.01	l2	{'C': 0.01, 'penalty': 'l2'}

2.借助机器学习流的全域参数搜索方法

In [58]:

```
# 科学计算模块
import numpy as np
import pandas as pd

# 绘图模块
import matplotlib as mpl
import matplotlib.pyplot as plt

# Scikit-Learn相关模块
# 评估器类
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV

from sklearn.metrics import roc_auc_score
# 实用函数
from sklearn.model_selection import train_test_split
```

```
# 数据准备
from sklearn.datasets import load_iris
```

In [59]:
```
# 构造机器学习流
pipe = make_pipeline(PolynomialFeatures(),
                     StandardScaler(),
                     LogisticRegression(max_iter=int(1e6)))
```

In [60]:
```
# 构造参数空间
param_grid = [
    {'polynomialfeatures__degree': np.arange(1, 5).tolist(), 'logisticregression__pena
    {'polynomialfeatures__degree': np.arange(1, 5).tolist(), 'logisticregression__pena
    {'polynomialfeatures__degree': np.arange(1, 5).tolist(), 'logisticregression__pena
]
```

In [61]:
```
import sklearn
sorted(sklearn.metrics.SCORERS.keys())
```

Out[61]:
```
['accuracy',
 'adjusted_mutual_info_score',
 'adjusted_rand_score',
 'average_precision',
 'balanced_accuracy',
 'completeness_score',
 'explained_variance',
 'f1',
 'f1_macro',
 'f1_micro',
 'f1_samples',
 'f1_weighted',
 'fowlkes_mallows_score',
 'homogeneity_score',
 'jaccard',
 'jaccard_macro',
 'jaccard_micro',
 'jaccard_samples',
 'jaccard_weighted',
 'max_error',
 'mutual_info_score',
 'neg_brier_score',
 'neg_log_loss',
 'neg_mean_absolute_error',
 'neg_mean_absolute_percentage_error',
 'neg_mean_gamma_deviance',
 'neg_mean_poisson_deviance',
 'neg_mean_squared_error',
 'neg_mean_squared_log_error',
 'neg_median_absolute_error',
 'neg_root_mean_squared_error',
 'normalized_mutual_info_score',
 'precision',
 'precision_macro',
 'precision_micro',
 'precision_samples',
 'precision_weighted',
 'r2',
 'rand_score',
 'recall',
 'recall_macro',
 'recall_micro',
 'recall_samples',
 'recall_weighted',
 'roc_auc',
 'roc_auc_ovo',
 'roc_auc_ovo_weighted',
 'roc_auc_ovr',
 'roc_auc_ovr_weighted',
 'top_k_accuracy',
 'v_measure_score']
```

In [62]:
```
search1 = GridSearchCV(estimator=pipe,
             param_grid=param_grid,
                      n_jobs=2,
                      verbose=1,
             scoring='roc_auc_ovr')
```

In [63]:
```
search1.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 1064 candidates, totalling 5320 fits
```

Out[63]:
```
GridSearchCV(estimator=Pipeline(steps=[('polynomialfeatures',
                                        PolynomialFeatures()),
                                       ('standardscaler', StandardScaler()),
                                       ('logisticregression',
                                        LogisticRegression(max_iter=1000000))]),
             n_jobs=2,
             param_grid=[{'logisticregression__C': [0.1, 0.2,
                                                    0.30000000000000004, 0.4,
                                                    0.5, 0.6,
                                                    0.7000000000000001, 0.8,
                                                    0.9, 1.0, 1.1,
                                                    1.2000000000000002,
                                                    1.3000000000000003,
                                                    1.4000000...
                                                    1.4000000000000001,
                                                    1.5000000000000002, 1.6,
                                                    1.7000000000000002,
                                                    1.8000000000000003,
                                                    1.9000000000000001],
                          'logisticregression__l1_ratio': [0.1, 0.2,
                                                           0.30000000000000004,
                                                           0.4, 0.5, 0.6,
                                                           0.7000000000000001,
                                                           0.8, 0.9],
                          'logisticregression__penalty': ['elasticnet'],
                          'logisticregression__solver': ['saga'],
                          'polynomialfeatures__degree': [1, 2, 3, 4]}],
             scoring='roc_auc_ovr', verbose=True)
```

In [64]:
```
search1.best_score_
```

Out[64]:
```
0.9987103174603174
```

In [65]:
```
search1.best_params_
```

Out[65]:
```
{'logisticregression__C': 1.5000000000000002,
 'logisticregression__penalty': 'l1',
 'logisticregression__solver': 'saga',
 'polynomialfeatures__degree': 4}
```

In [66]:
```
search1.best_estimator_.score(X_train,y_train)
```

Out[66]:
```
0.9821428571428571
```

In [67]:
```
search1.best_estimator_.score(X_test,y_test)
```

Out[67]:
```
0.9736842105263158
```

In [68]:
```
search2 = GridSearchCV(estimator=pipe,
             param_grid=param_grid,
                      n_jobs=2,
                      verbose=1,
             scoring='accuracy')
search2.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 1064 candidates, totalling 5320 fits
```

Out[68]:
```
GridSearchCV(estimator=Pipeline(steps=[('polynomialfeatures',
                                        PolynomialFeatures()),
                                       ('standardscaler', StandardScaler()),
                                       ('logisticregression',
                                        LogisticRegression(max_iter=1000000))]),
             n_jobs=2,
             param_grid=[{'logisticregression__C': [0.1, 0.2,
                                                    0.30000000000000004, 0.4,
                                                    0.5, 0.6,
                                                    0.7000000000000001, 0.8,
                                                    0.9, 1.0, 1.1,
                                                    1.2000000000000002,
                                                    1.3000000000000003,
                                                    1.4000000...
                                                    1.4000000000000001,
                                                    1.5000000000000002, 1.6,
                                                    1.7000000000000002,
                                                    1.8000000000000003,
                                                    1.9000000000000001],
                          'logisticregression__l1_ratio': [0.1, 0.2,
                                                           0.30000000000000004,
                                                           0.4, 0.5, 0.6,
                                                           0.7000000000000001,
                                                           0.8, 0.9],
                          'logisticregression__penalty': ['elasticnet'],
                          'logisticregression__solver': ['saga'],
                          'polynomialfeatures__degree': [1, 2, 3, 4]}],
             scoring='accuracy', verbose=True)
```

In [69]:
```
search2.best_score_
```

Out[69]:
```
0.9731225296442687
```

In [70]:
```
search2.best_params_
```

Out[70]:
```
{'logisticregression__C': 0.6,
 'logisticregression__penalty': 'l2',
 'logisticregression__solver': 'sag',
 'polynomialfeatures__degree': 3}
```

In [71]:
```
search2.best_estimator_.score(X_train,y_train)
```

Out[71]:
```
0.9821428571428571
```

In [72]:
```
search2.best_estimator_.score(X_test,y_test)
```

Out[72]:
```
0.9473684210526315
```