12_grid_random_search

December 27, 2019

1 Kaggle Titanic survival - optimising models with grid search and random search

Machine learning models have many hyper-parameters (parameters set before a model is fitted, and which remain constant throughout model fitting). Optimising model hyper-parameters may involve many model runs with alternative hyper-parameters. In SciKit-Learn, this may be performed in an automated fashion using `GridSearchCV` (which explores all combinations of provided hyper-parameters) or `RandomizedSearchCV` (which selects randomly from parameter ranges, which can be useful when there are too many combinations in grid search).

We will go through the following steps:

- Download and save pre-processed data
- Split data into features (X) and label (y)
- Standardise data
- Use grid search to optimise model parameters

`GridSearchCV` used stratified k-fold sampling to perform replicates for each paraemter step. If you are unfamiliar with this method of replication have a look at:

https://github.com/MichaelAllen1966/1804_python_healthcare/blob/master/titanic/03_k_fold.ipynb

1.1 Hide warnings (to keep notebook tidy; do not usually do this)

[1]:
```
# Hide warnings (to keep notebook tidy; do not usually do this)
import warnings
warnings.filterwarnings("ignore")
```

1.2 Load modules

A standard Anaconda install of Python (https://www.anaconda.com/distribution/) contains all the necessary modules.

[2]:
```
import numpy as np
import pandas as pd
# Import machine learning methods
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.preprocessing import StandardScaler
```

1.3 Load data

The section below downloads pre-processed data, and saves it to a subfolder (from where this code is run). If data has already been downloaded that cell may be skipped.

Code that was used to pre-process the data ready for machine learning may be found at: https://github.com/MichaelAllen1966/1804_python_healthcare/blob/master/titanic/01_preprocessing.ipynb

[3]:
```
download_required = True

if download_required:

    # Download processed data:
    address = 'https://raw.githubusercontent.com/MichaelAllen1966/' + \
                '1804_python_healthcare/master/titanic/data/processed_data.csv'

    data = pd.read_csv(address)

    # Create a data subfolder if one does not already exist
    import os
    data_directory ='./data/'
    if not os.path.exists(data_directory):
        os.makedirs(data_directory)

    # Save data
    data.to_csv(data_directory + 'processed_data.csv')
```

[4]:
```
data = pd.read_csv('data/processed_data.csv')
```

The first column is a passenger index number. We will remove this, as this is not part of the original Titanic passenger data.

[5]:
```
# Drop Passengerid (axis=1 indicates we are removing a column rather than a row)
# We drop passenger ID as it is not original data

data.drop('PassengerId', inplace=True, axis=1)
```

1.4 Divide into X (features) and y (lables)

We will separate out our features (the data we use to make a prediction) from our label (what we are truing to predict). By convention our features are called `X` (usually upper case to denote multiple features), and the label (survvive or not) `y`.

[6]:
```
X = data.drop('Survived',axis=1) # X = all 'data' except the 'survived' column
y = data['Survived'] # y = 'survived' column from 'data'
```

1.5 Standardise data

For grid and random serach we will standardise data just once at the beginning. Note - for final model testing you should follow the normal practice of splitting the data into training and test data sets, and standardising both sets of data based on the training data set.

[7]:
```
# Initialise a new scaling object for normalising input data
sc = StandardScaler()

# Set up the scaler just on the training set
sc.fit(X)

# Apply the scaler to the X data
X_std=sc.transform(X)
```

1.6 Grid search

Grid serach is a good method so long as the number of paramater combinations is not too high

1.6.1 Defining paramters to test

We define parameters to test in a dictionary.

NOTE: Grid search can very quickly lead to very many parameters to test. Initially it is best to pick just 2-3 levels of each parameter. You can always narrow the search.

The parameters available for tuning for the logistic regression model are listed on the document page for the model (you can also find this using the `help()` method in Python:

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

We will vary the following in grid search (and will expand the list in random search):

- penalty type (for regularisation)
- Regularisation (C)
- Class weight. 38% of the passengers survive. This can lead to survivors having less influence in the model (as there are fewer of them). We will test weighting non-survivors and survivors in inverse proportion to their number.

[8]:
```
param_grid = {'penalty': ['l1', 'l2'],
              'C': [0.01, 0.1, 1, 10],
              'class_weight': [{0:0.5, 1:0.5},{0:0.38, 1:0.62}]}

# Class weight is defined as a dictionary with class label and weight.
```

In the above paraemter grid we have 2 * 4 * 2 parameter combinations = 16

1.6.2 Run grid search with defined parameters

We run the grid search, defining the number of k-fold replicates to use, and we specificy the accuracy measurement we want to report (in this case we will use 'f1' to balance precision and recall).

[9]:
```
# Import GridSearch
from sklearn.model_selection import GridSearchCV

# Define model
model = LogisticRegression()

# Define grid search to use 5 k-fold validation, and use 'f1' for accuracy
grid_search = GridSearchCV(model, param_grid, cv=5, scoring='f1')

# Run grid search
grid_search.fit(X_std, y); #';' suppresses printed output
```

1.6.3 Show grid search performance

[10]:
```
# show best performance and parameters
# If best parameters are at the extremes of the searches then extend the range

print ('Best performance (f1):')
print (grid_search.best_score_)
print ('Best parameters:')
print (grid_search.best_params_)
```

```
Best performance (f1):
0.7355099445362381
Best parameters:
{'C': 0.1, 'class_weight': {0: 0.38, 1: 0.62}, 'penalty': 'l2'}
```

Or, show full description (which may be copied in to a model)

[11]:
```
grid_search.best_estimator_
```

[11]:
```
LogisticRegression(C=0.1, class_weight={0: 0.38, 1: 0.62}, dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=100, multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

Full results are stored in a dictionary `cv_results_`. Below we display them by passing them to a Pandas DataFrame (and we limit the columns to those of most interest to us).

[12]:
```
results = pd.DataFrame(grid_search.cv_results_)
cols_to_show = ['param_penalty','param_C', 'param_class_weight',
                'mean_test_score','rank_test_score' ]
print(results[cols_to_show])
```

```
   param_penalty param_C  param_class_weight  mean_test_score  rank_test_score
0             l1    0.01    {0: 0.5, 1: 0.5}         0.000000               15
1             l2    0.01    {0: 0.5, 1: 0.5}         0.714713               13
2             l1    0.01  {0: 0.38, 1: 0.62}         0.000000               15
3             l2    0.01  {0: 0.38, 1: 0.62}         0.730961                2
4             l1     0.1    {0: 0.5, 1: 0.5}         0.709479               14
5             l2     0.1    {0: 0.5, 1: 0.5}         0.722657                8
6             l1     0.1  {0: 0.38, 1: 0.62}         0.724712                6
7             l2     0.1  {0: 0.38, 1: 0.62}         0.735510                1
8             l1       1    {0: 0.5, 1: 0.5}         0.721540                9
9             l2       1    {0: 0.5, 1: 0.5}         0.720458               11
10            l1       1  {0: 0.38, 1: 0.62}         0.723316                7
11            l2       1  {0: 0.38, 1: 0.62}         0.729697                3
12            l1      10    {0: 0.5, 1: 0.5}         0.720458               11
13            l2      10    {0: 0.5, 1: 0.5}         0.720487               10
14            l1      10  {0: 0.38, 1: 0.62}         0.727679                4
15            l2      10  {0: 0.38, 1: 0.62}         0.727679                4
```

When looking at the results, it is worth noting the range of results. Tou may then consider whether it is worth refining the grid search to focus on a narrower area.

1.7 Random search

Random search is very similar to grid search, but randomly selects combinations of parameters to test, with the maximum number of tests given by the `n_iter` argument.

As we've been through the process with grid search, we'll put all our code together here, but note the larger number of parameters defined.

[13]:
```
# Import GridSearch
from sklearn.model_selection import RandomizedSearchCV

# Define paraemter grid and maximum number of tests

param_grid = {'penalty': ['l1', 'l2'],
              'C': [0.01, 0.03, 0.1, 0.3, 1, 3, 10],
              'class_weight': [{0:0.5, 1:0.5},
                               {0:0.38, 1:0.62},
                               {0:0.62, 1:0.38}],
              'max_iter': [30, 100, 300, 1000]}

n_iter_search = 50
```

```
# Define model
model = LogisticRegression()

# Set up random search
random_search = RandomizedSearchCV(model, param_grid, cv=5,
                           n_iter=n_iter_search, scoring='f1')

# Run grid search
random_search.fit(X_std, y); #';' suppresses printed output

# Get and print output
print ('Best performance (f1):')
print (random_search.best_score_)
print ('Best parameters:')
print (random_search.best_params_)
```

```
Best performance (f1):
0.7355099445362381
Best parameters:
{'penalty': 'l2', 'max_iter': 100, 'class_weight': {0: 0.38, 1: 0.62}, 'C': 0.1}
```

```
/home/michael/anaconda3/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
```

Print all tests

[14]:
```
results = pd.DataFrame(random_search.cv_results_)
cols_to_show = ['param_penalty','param_C', 'param_class_weight',
                'mean_test_score','rank_test_score' ]
print(results[cols_to_show])
```

```
   param_penalty param_C  param_class_weight  mean_test_score  rank_test_score
0             l1       3    {0: 0.5, 1: 0.5}         0.721539               13
1             l2    0.03    {0: 0.5, 1: 0.5}         0.731564                2
2             l2     0.3  {0: 0.62, 1: 0.38}         0.714191               21
3             l1       3  {0: 0.62, 1: 0.38}         0.704302               33
4             l1      10    {0: 0.5, 1: 0.5}         0.720458               16
5             l1     0.3    {0: 0.5, 1: 0.5}         0.721433               14
6             l1     0.1  {0: 0.38, 1: 0.62}         0.724712                7
7             l1     0.1  {0: 0.38, 1: 0.62}         0.724712                7
8             l2      10  {0: 0.62, 1: 0.38}         0.703537               36
9             l2    0.01  {0: 0.62, 1: 0.38}         0.650990               43
10            l2      10  {0: 0.38, 1: 0.62}         0.727679                6
11            l2      10    {0: 0.5, 1: 0.5}         0.720487               15
12            l1       1  {0: 0.62, 1: 0.38}         0.704254               35
```

13	l2	0.03	{0: 0.62, 1: 0.38}	0.692330	41
14	l1	0.01	{0: 0.5, 1: 0.5}	0.000000	46
15	l1	0.1	{0: 0.5, 1: 0.5}	0.709479	25
16	l1	0.01	{0: 0.62, 1: 0.38}	0.000000	46
17	l2	0.3	{0: 0.62, 1: 0.38}	0.714191	21
18	l1	0.01	{0: 0.38, 1: 0.62}	0.000000	46
19	l1	1	{0: 0.5, 1: 0.5}	0.721540	10
20	l1	0.3	{0: 0.38, 1: 0.62}	0.722912	9
21	l2	0.3	{0: 0.38, 1: 0.62}	0.730898	4
22	l1	0.01	{0: 0.38, 1: 0.62}	0.000000	46
23	l1	0.01	{0: 0.62, 1: 0.38}	0.000000	46
24	l1	1	{0: 0.5, 1: 0.5}	0.721540	10
25	l1	10	{0: 0.5, 1: 0.5}	0.720458	16
26	l2	0.1	{0: 0.62, 1: 0.38}	0.711792	23
27	l1	0.1	{0: 0.5, 1: 0.5}	0.709479	25
28	l2	0.01	{0: 0.38, 1: 0.62}	0.730961	3
29	l1	0.3	{0: 0.62, 1: 0.38}	0.695724	40
30	l1	0.03	{0: 0.38, 1: 0.62}	0.709217	27
31	l2	1	{0: 0.38, 1: 0.62}	0.729697	5
32	l2	3	{0: 0.5, 1: 0.5}	0.720458	16
33	l1	0.03	{0: 0.38, 1: 0.62}	0.709217	27
34	l2	0.03	{0: 0.62, 1: 0.38}	0.692330	41
35	l1	3	{0: 0.62, 1: 0.38}	0.704302	33
36	l2	0.1	{0: 0.38, 1: 0.62}	0.735510	1
37	l1	0.03	{0: 0.5, 1: 0.5}	0.709217	27
38	l1	10	{0: 0.62, 1: 0.38}	0.703537	36
39	l1	0.03	{0: 0.38, 1: 0.62}	0.709217	27
40	l2	1	{0: 0.5, 1: 0.5}	0.720458	16
41	l1	10	{0: 0.62, 1: 0.38}	0.703537	36
42	l2	3	{0: 0.62, 1: 0.38}	0.702510	39
43	l2	0.01	{0: 0.62, 1: 0.38}	0.650990	43
44	l2	0.01	{0: 0.62, 1: 0.38}	0.650990	43
45	l2	1	{0: 0.62, 1: 0.38}	0.706706	31
46	l2	1	{0: 0.5, 1: 0.5}	0.720458	16
47	l1	1	{0: 0.5, 1: 0.5}	0.721540	10
48	l2	1	{0: 0.62, 1: 0.38}	0.706706	31
49	l2	0.1	{0: 0.62, 1: 0.38}	0.711792	23

1.8 Summary

In this small example we have found grid search and random search both identified a solution with the same accuracy, and in good time. In larger models you may find it best to run a random search initially (which helps to show which paraemters are most influential), and then use a grid search once you have narrowed down the area of search.