parameter_tuning_randomized_search

May 29, 2021

1 Hyperparameter tuning by randomized-search

In the previous notebook, we showed how to use a grid-search approach to search for the best hyperparameters maximizing the statistical performance of a predictive model.

However, a grid-search approach has limitations. It does not scale when the number of parameters to tune is increasing. Also, the grid will imposed a regularity during the search which might be problematic.

In this notebook, we will present the another method to tune hyperparameters called randomized search.

1.1 Our predictive model

Let us reload the dataset as we did previously:

[1]:
```
from sklearn import set_config

set_config(display="diagram")
```

[2]:
```
import pandas as pd

adult_census = pd.read_csv("../datasets/adult-census.csv")
```

We extract the column containing the target.

[3]:
```
target_name = "class"
target = adult_census[target_name]
target
```

[3]:
```
0         <=50K
1         <=50K
2          >50K
3          >50K
4         <=50K
          ...
48837     <=50K
48838      >50K
48839     <=50K
48840     <=50K
```

```
48841      >50K
Name: class, Length: 48842, dtype: object
```

We drop from our data the target and the `"education-num"` column which duplicates the information with `"education"` columns.

[4]:
```
data = adult_census.drop(columns=[target_name, "education-num"])
data.head()
```

[4]:

	age	workclass	education	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country
0	25	Private	11th	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States
1	38	Private	HS-grad	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States
2	28	Local-gov	Assoc-acdm	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States
3	44	Private	Some-college	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States
4	18	?	Some-college	Never-married	?	Own-child	White	Female	0	0	30	United-States

Once the dataset is loaded, we split it into a training and testing sets.

[5]:
```
from sklearn.model_selection import train_test_split

data_train, data_test, target_train, target_test = train_test_split(
    data, target, random_state=42)
```

We will create the same predictive pipeline as seen in the grid-search section.

[6]:
```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OrdinalEncoder
from sklearn.compose import make_column_selector as selector

categorical_columns_selector = selector(dtype_include=object)
categorical_columns = categorical_columns_selector(data)

categorical_preprocessor = OrdinalEncoder(handle_unknown="use_encoded_value",
                                          unknown_value=-1)
```

```
preprocessor = ColumnTransformer([
    ('cat-preprocessor', categorical_preprocessor, categorical_columns)],
    remainder='passthrough', sparse_threshold=0)
```

[7]:
```
# for the moment this line is required to import HistGradientBoostingClassifier
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.pipeline import Pipeline

model = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier",
     HistGradientBoostingClassifier(random_state=42, max_leaf_nodes=4))])
model
```

[7]:
```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(remainder='passthrough', sparse_threshold=0,
                                   transformers=[('cat-preprocessor',
OrdinalEncoder(handle_unknown='use_encoded_value',
unknown_value=-1),
                                                  ['workclass', 'education',
                                                   'marital-status',
                                                   'occupation', 'relationship',
                                                   'race', 'sex',
                                                   'native-country'])])),
                ('classifier',
                 HistGradientBoostingClassifier(max_leaf_nodes=4,
                                                random_state=42))])
```

1.2 Tuning using a randomized-search

With the `GridSearchCV` estimator, the parameters need to be specified explicitly. We already mentioned that exploring a large number of values for different parameters will be quickly untractable.

Instead, we can randomly generate the parameter candidates. Indeed, such approach avoids the regularity of the grid. Hence, adding more evaluations can increase the resolution in each direction. This is the case in the frequent situation where the choice of some hyperparameters is not very important, as for hyperparameter 2 in the figure below.

Indeed, the number of evaluation points need to be divided across the two different hyperparameters. With a grid, the danger is that the region of good hyperparameters fall between the line of the grid: this region is aligned with the grid given that hyperparameter 2 has a weak influence. Rather, stochastic search will sample hyperparameter 1 independently from hyperparameter 2 and find the optimal region.

The `RandomizedSearchCV` class allows for such stochastic search. It is used similarly to the `GridSearchCV` but the sampling distributions need to be specified instead of the parameter values. For instance, we will draw candidates using a log-uniform distribution because the parameters we are interested in take positive values with a natural log scaling (.1 is as close to 1 as 10 is).

Note

Random search (with RandomizedSearchCV) is typically beneficial compared to grid search (with GridSearchCV) to optimize 3 or more hyperparameters.

We will optimize 3 other parameters in addition to the ones we optimized above:

- `max_iter`: it corresponds to the number of trees in the ensemble;
- `min_samples_leaf`: it corresponds to the minimum number of samples required in a leaf;
- `max_bins`: it corresponds to the maximum number of bins to construct the histograms.

Note

The loguniform function from SciPy returns a floating number. Since we want to us this distribution to create integer, we will create a class that will cast the floating number into an integer.

[8]:
```
from scipy.stats import loguniform


class loguniform_int:
    """Integer valued version of the log-uniform distribution"""
    def __init__(self, a, b):
        self._distribution = loguniform(a, b)

    def rvs(self, *args, **kwargs):
        """Random variable sample"""
        return self._distribution.rvs(*args, **kwargs).astype(int)
```

Now, we can define the randomized search using the different distributions.

[9]:
```
from sklearn.model_selection import RandomizedSearchCV

param_distributions = {
    'classifier__l2_regularization': loguniform(1e-6, 1e3),
    'classifier__learning_rate': loguniform(0.001, 10),
    'classifier__max_leaf_nodes': loguniform_int(2, 256),
    'classifier__min_samples_leaf': loguniform_int(1, 100),
    'classifier__max_bins': loguniform_int(2, 255)}

model_random_search = RandomizedSearchCV(
    model, param_distributions=param_distributions, n_iter=10,
    n_jobs=4, cv=5)
model_random_search.fit(data_train, target_train)
```

[9]:
```
RandomizedSearchCV(cv=5,
                   estimator=Pipeline(steps=[('preprocessor',
ColumnTransformer(remainder='passthrough',
sparse_threshold=0,
transformers=[('cat-preprocessor',
OrdinalEncoder(handle_unknown='use_encoded_value',
```

```
                unknown_value=-1),
['workclass',
'education',
'marital-status',
'occupation',
'relationship',
'race',
'sex',
'native-country'])])),
                                                  ('classifier',
                                                   Hi…
                    param_distributions={'classifier__l2_regularization':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x7f3eda2c74c0>,
                                                  'classifier__learning_rate':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x7f3ed96ce790>,
                                                  'classifier__max_bins':
<__main__.loguniform_int object at 0x7f3ed98fed60>,
                                                  'classifier__max_leaf_nodes':
<__main__.loguniform_int object at 0x7f3edb012ee0>,
                                                  'classifier__min_samples_leaf':
<__main__.loguniform_int object at 0x7f3ed96ceb20>})
```

Then, we can compute the accuracy score on the test set.

[10]:
```
accuracy = model_random_search.score(data_test, target_test)

print(f"The test accuracy score of the best model is "
      f"{accuracy:.2f}")
```

```
The test accuracy score of the best model is 0.88
```

[11]:
```
from pprint import pprint

print("The best parameters are:")
pprint(model_random_search.best_params_)
```

```
The best parameters are:
{'classifier__l2_regularization': 0.005801123693646886,
 'classifier__learning_rate': 0.1755210042480209,
 'classifier__max_bins': 123,
 'classifier__max_leaf_nodes': 48,
 'classifier__min_samples_leaf': 39}
```

We can inspect the results using the attributes `cv_results` as we previously did.

[12]:
```
def shorten_param(param_name):
    if "__" in param_name:
        return param_name.rsplit("__", 1)[1]
```

```
    return param_name
```

[13]:
```
# get the parameter names
column_results = [
    f"param_{name}" for name in param_distributions.keys()]
column_results += [
    "mean_test_score", "std_test_score", "rank_test_score"]

cv_results = pd.DataFrame(model_random_search.cv_results_)
cv_results = cv_results[column_results].sort_values(
    "mean_test_score", ascending=False)
cv_results = cv_results.rename(shorten_param, axis=1)
cv_results
```

[13]:

	l2_regularization	learning_rate	max_leaf_nodes	min_samples_leaf	max_bins	\
8	0.005801	0.175521	48	39	123	
5	410.069231	0.097489	34	4	21	
6	1.451583	0.055881	11	24	38	
4	98.221268	0.474348	16	2	7	
7	0.00805	0.308162	3	1	2	
3	0.00012	0.008263	13	32	17	
1	0.000043	0.002362	23	27	166	
2	4.009738	0.005228	4	2	45	
0	0.036002	2.559237	101	11	78	
9	0.000014	7.229585	105	60	7	

	mean_test_score	std_test_score	rank_test_score
8	0.868308	0.002565	1
5	0.855887	0.002353	2
6	0.854276	0.002645	3
4	0.841064	0.002830	4
7	0.801561	0.002362	5
3	0.795911	0.002346	6
1	0.758947	0.000013	7
2	0.758947	0.000013	7
0	0.738308	0.028132	9
9	0.672901	0.082198	10

In practice, a randomized hyperparameter search is usually run with a large number of iterations. In order to avoid the computation cost and still make a decent analysis, we load the results obtained from a similar search with 200 iterations.

[14]:
```
# model_random_search = RandomizedSearchCV(
#     model, param_distributions=param_distributions, n_iter=500,
#     n_jobs=4, cv=5)
# model_random_search.fit(df_train, target_train)
# cv_results =  pd.DataFrame(model_random_search.cv_results_)
```

```
# cv_results.to_csv("../figures/randomized_search_results.csv")
```

[15]:
```
cv_results = pd.read_csv("../figures/randomized_search_results.csv",
                         index_col=0)
```

As we have more than 2 paramters in our grid-search, we cannot visualize the results using a heatmap. However, we can us a parallel coordinates plot.

[16]:
```
(cv_results[column_results].rename(
    shorten_param, axis=1).sort_values("mean_test_score"))
```

[16]:

	l2_regularization	learning_rate	max_leaf_nodes	min_samples_leaf	max_bins	mean_test_score	std_test_score	rank_test_score
357	0.000026	3.075318	3	68	31	0.241053	0.000013	500
200	0.000444	6.236325	2	2	30	0.344629	0.207156	499
413	0.000001	8.828574	64	1	144	0.448205	0.253714	497
344	0.000003	7.091079	5	1	95	0.448205	0.253714	497
232	0.000097	9.976823	28	5	3	0.448205	0.253714	496
..	…	…	…	…	…	…	…	…
327	4.733808	0.036786	61	5	241	0.869673	0.002417	5
328	2.036232	0.224702	28	49	236	0.869837	0.000808	4
21	4.994918	0.077047	53	7	192	0.870793	0.001993	3
343	0.000404	0.244503	15	15	229	0.871339	0.002741	2
208	0.011775	0.076653	24	2	155	0.871393	0.001588	1

[500 rows x 8 columns]

[17]:
```
import numpy as np
import plotly.express as px

fig = px.parallel_coordinates(
    cv_results.rename(shorten_param, axis=1).apply({
        "learning_rate": np.log10,
        "max_leaf_nodes": np.log2,
        "max_bins": np.log2,
```

```
        "min_samples_leaf": np.log10,
        "l2_regularization": np.log10,
        "mean_test_score": lambda x: x}),
    color="mean_test_score",
    color_continuous_scale=px.colors.sequential.Viridis,
)
fig.show()
```

The parallel coordinates plot will display the values of the hyperparameters on different columns while the performance metric is color coded. Thus, we are able to quickly inspect if there is a range of hyperparameters which is working or not.

Note

We transformed most axis values by taking a log10 or log2 to spread the active ranges and improve the readability of the plot.

It is possible to **select a range of results by clicking and holding on any axis** of the parallel coordinate plot. You can then slide (move) the range selection and cross two selections to see the intersections.

In this notebook, we have seen how randomized search offer a valuable alternative to grid-search when the number of hyperparameters to tune is more than two. It also alleviates the regularity imposed by the grid that might be problematic sometimes.