Bayesian Optimization: Hyperparameter Tuning of Light GBM for Flight Departures

by Sonjoy Das, PhD

In addition to the random search and the grid search methods for selecting optimal hyperparameters, we can use Bayesian methods to select the optimal hyperparameters for an algorithm.

In this work, we will use **two packages** to implement the *Bayesian global optimization with Gaussian processes* to perform hyperparmater tuning. We will use the flight departures dataset for both packages.

- One of the packages is `BayesianOptimization` whose documentation can be found here: https://github.com/fmfn/BayesianOptimization.
- The other package is `scikit-optimize` package (re: documentation). We will implement `BayesSearchCV` which is `scikit-learn` hyperparameter search wrapper (re: skopt.BayesSearchCV). For a simple illustration on `scikit-optimize`'s `BayesSearchCV`, watch this YouTube video.

For `BayesianOptimization`, we will optimize the cross-validation (re: lightgbm.cv) performance of *binary* log loss classification through LightGBM (re: LightGBM) which is a gradient boosting framework.

For `BayesSearchCV`, we will again use *binary* log loss classification in `lightgbm.LGBMClassifier` (re: lightgbm.LGBMClassifier) whose hyper parameters will be optimized by employing Bayesian optimization and cross-validation.

Note that in contrast to `GridSearchCV`, not all hyperparameter **values** are tried out in hyperparameter tuning through Bayesian optimization, but rather a fixed number of hyperparameter settings is sampled from specified distributions. The number of hyperparameter settings that are tried is given by `n_iter`.

A **third option** to tune the hyperparameters of a ML model is to use `scikit-learn`'s `gp_minimize` algorithm that relies on Bayesian optimization using Gaussian Processes. The implementation scheme is very similar to the `BayesianOptimization` library we mentioned above. An illustration is provided here. But, we will not discuss this here in this notebook.

Side Note: If `pip install lightgbm` (re: LightGBM Installation) does not work, then you can build from GitHub and then go to `/python-package` under the directory where you built it, and run `python setup.py install`.

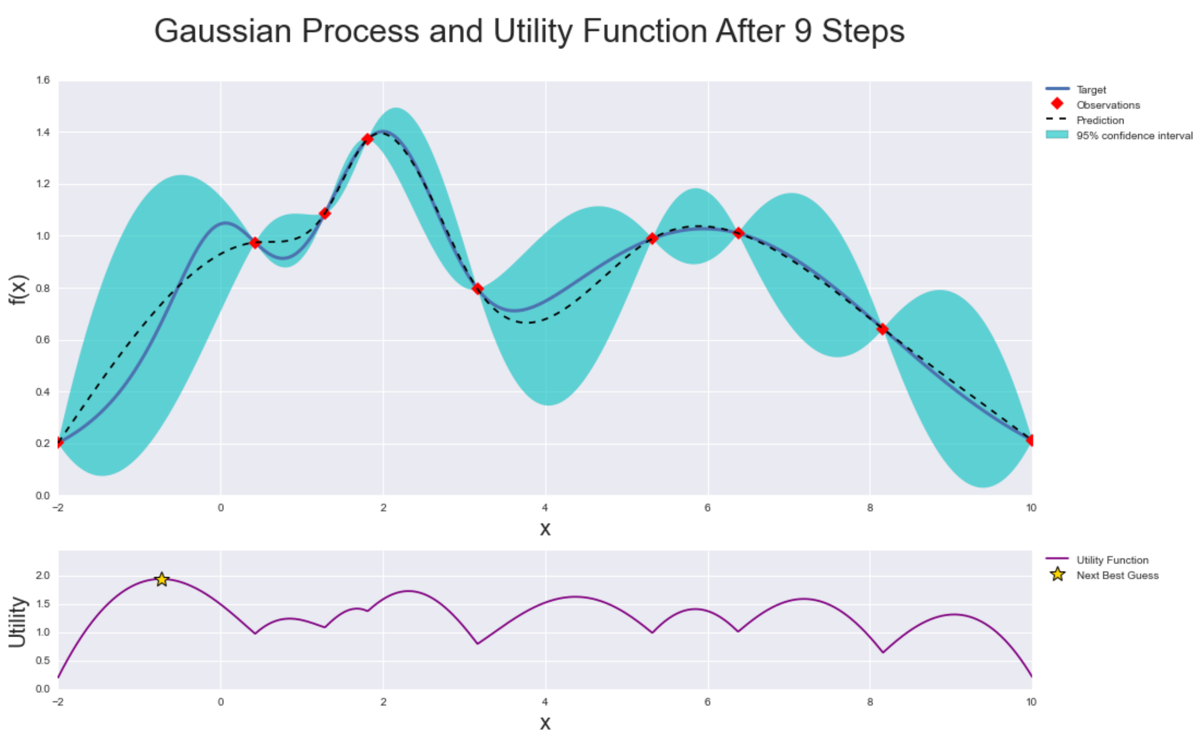
In [1]:

```
from bayes_opt import BayesianOptimization
import pandas as pd

from sklearn.preprocessing import LabelEncoder
```

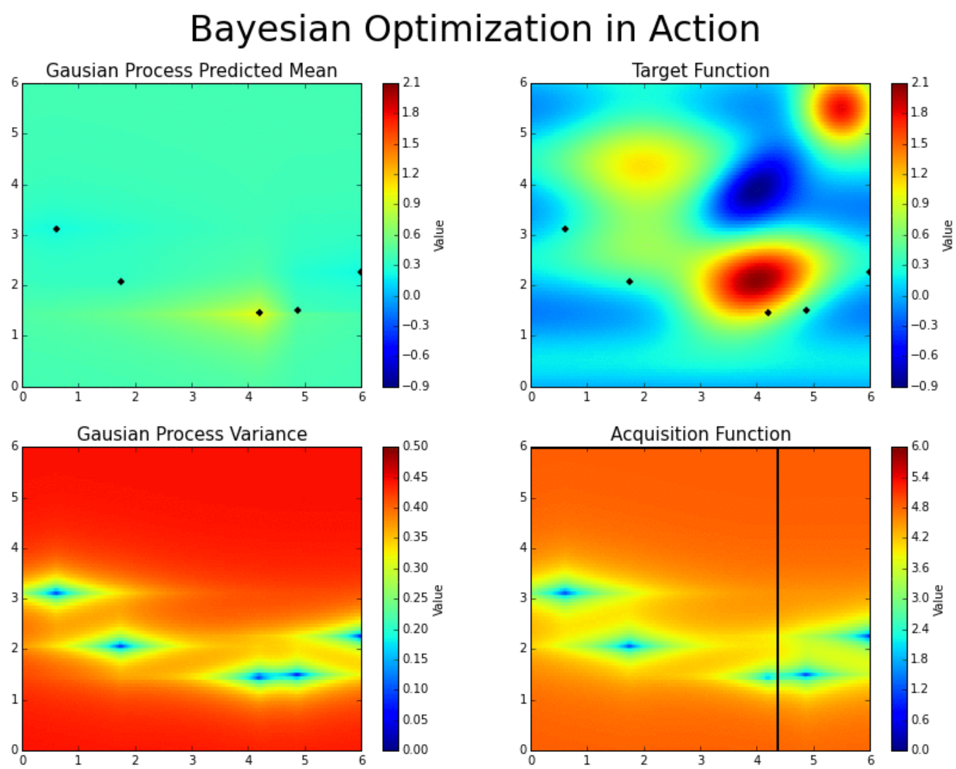
```python
import numpy as np

import lightgbm

from skopt.space import Real, Categorical, Integer
from skopt import BayesSearchCV
```

How does Bayesian optimization work?

Bayesian optimization works by constructing a posterior distribution of functions (Gaussian process) that best describes the function you want to optimize. As the number of observations grows, the posterior distribution improves, and the algorithm becomes more certain of which regions in parameter space are worth exploring and which are not, as seen in the picture below.



As you iterate over and over, the algorithm balances its needs of exploration and exploitation while taking into account what it knows about the target function. At each step, a Gaussian Process is fitted to the known samples (points previously explored), and the posterior distribution, combined with an exploration strategy (such as UCB — aka Upper Confidence Bound), or EI (Expected Improvement). This process is used to determine the next point that should be explored (see the gif below).



This process is designed to minimize the number of steps required to find a combination of parameters that are close to the optimal combination. To do so, this method uses a proxy optimization problem (finding the maximum of the acquisition function) that, albeit still a hard problem, is cheaper (in the computational sense) and common tools can be employed. Therefore, Bayesian Optimization is most adequate for situations where sampling the function to be optimized is a very expensive endeavor. See the References section in the `BayesianOptimization` library documentation for a proper discussion of this method.

Let's look at a simple example

Using `BayesianOptimization` library

The first step is to create an optimizer. It uses two items:

- function to optimize
- bounds of parameters

The function is the procedure that counts metrics of our model quality. The important thing is that our optimization will maximize the value of function. Smaller metrics are best. Hint: don't forget to use negative metric values.

Here we define our simple function we want to optimize:

In [2]:
```python
def simple_func(a, b):
    return a + b
```

Now, we define our bounds of the parameters to optimize, within the Bayesian optimizer.

In [3]:
```python
# Bounded region of parameter space
pbounds = {'a': (1, 3),
           'b': (4, 7)}

optimizer = BayesianOptimization(
                                 f = simple_func,
                                 pbounds = pbounds,
                                 random_state=1
                                )
```

There are many parameters to pass to optimize, nonetheless, the most important ones are:

- `n_iter`: How many steps of bayesian optimization you want to perform. The more steps the more likely to find a good optimal you are.
- `init_points`: How many steps of random exploration you want to perform. Random exploration can help by diversifying the exploration space.

Let's run an example where we use the optimizer to find the best values to *maximize* the target value for `a` and `b` given the inputs of `init_points = 2` and `n_iter = 3`.

In [4]:
```python
# optimizer.maximize(3,2)

optimizer.maximize(
    init_points=2,
    n_iter=3,
)
```

```
|   iter    |  target   |     a     |     b     |
-------------------------------------------------
| 1         | 7.995     | 1.834     | 6.161     |
| 2         | 5.907     | 1.0       | 4.907     |
| 3         | 6.324     | 1.218     | 5.107     |
| 4         | 9.162     | 2.797     | 6.365     |
| 5         | 10.0      | 3.0       | 7.0       |
=================================================
```

Great, now let's print the best parameters and the associated maximized target, which can be accessed via the property `optimizer.max`.

In [5]:
```python
print(optimizer.max)
```

```
{'target': 10.0, 'params': {'a': 3.0, 'b': 7.0}}
```

Or, we can print separately as follows:

In [6]:
```python
print(optimizer.max['params'])
print(optimizer.max['target'])
```

```
{'a': 3.0, 'b': 7.0}
10.0
```

While the list of all parameters probed and their corresponding target values is available via the property `optimizer.res`.

In [7]:
```
for i, res in enumerate(optimizer.res):
    print(f"Iteration {i}: \n\t{res}\n")
```

```
Iteration 0:
	{'target': 7.995017489731622, 'params': {'a': 1.834044009405148, 'b': 6.16
0973480326474}}

Iteration 1:
	{'target': 5.90722646753021, 'params': {'a': 1.0002287496346898, 'b': 4.90
69977178955195}}

Iteration 2:
	{'target': 6.3241901394458235, 'params': {'a': 1.2175526295255183, 'b': 5.
106637509920305}}

Iteration 3:
	{'target': 9.16207290732246, 'params': {'a': 2.7966872435398873, 'b': 6.36
5385663782572}}

Iteration 4:
	{'target': 10.0, 'params': {'a': 3.0, 'b': 7.0}}
```

To produce a tidy output, we can round off the values.

In [8]:
```
# Ref: https://stackoverflow.com/questions/32434112/round-off-floating-point-values

for i, res in enumerate(optimizer.res):

    for key, value in res.items():
        if key == 'target':
            res[key] = round(value, 3)
        elif key == 'params':
            for key_para, value_para in value.items():
                value[key_para] = round(value_para, 3)

    print(f"Iteration {i}: \n\t{res}\n")
```

```
Iteration 0:
	{'target': 7.995, 'params': {'a': 1.834, 'b': 6.161}}

Iteration 1:
	{'target': 5.907, 'params': {'a': 1.0, 'b': 4.907}}

Iteration 2:
	{'target': 6.324, 'params': {'a': 1.218, 'b': 5.107}}

Iteration 3:
	{'target': 9.162, 'params': {'a': 2.797, 'b': 6.365}}

Iteration 4:
	{'target': 10.0, 'params': {'a': 3.0, 'b': 7.0}}
```

Test it on real data using the Light GBM

The dataset we will now use is the famous flight departures dataset. Our modeling goal will be **to predict if a flight departure is going to be delayed by 15 minutes or more** based on the other attributes in our dataset. As part of this modeling exercise, we will use Bayesian hyperparameter optimization to identify the best parameters for our model.

You can load the zipped csv files just as you would regular csv files using `Pandas`' `read_csv`. In the next cell load the train and test data into two seperate dataframes.

In [9]:
```
train_df = pd.read_csv('flight_delays_train.csv.zip')
test_df = pd.read_csv('flight_delays_test.csv.zip')
```

In [10]:
```
train_df.shape
```

Out[10]:
```
(100000, 9)
```

In [11]:
```
test_df.shape
```

Out[11]:
```
(100000, 8)
```

Print the top five rows of the train dataframe and review the columns in the data.

In [12]:
```
train_df.head()
```

Out[12]:

	Month	DayofMonth	DayOfWeek	DepTime	UniqueCarrier	Origin	Dest	Distance	dep_delayed_15
0	c-8	c-21	c-7	1934	AA	ATL	DFW	732	
1	c-4	c-20	c-3	1548	US	PIT	MCO	834	
2	c-9	c-2	c-5	1422	XE	RDU	CLE	416	
3	c-11	c-25	c-6	1015	OO	DEN	MEM	872	
4	c-10	c-7	c-6	1828	WN	MDW	OMA	423	

Use the describe function to review the numeric columns in the train and test dataframes.

In [13]:
```
train_df.describe()
```

Out[13]:

	DepTime	Distance
count	100000.000000	100000.00000
mean	1341.523880	729.39716
std	476.378445	574.61686
min	1.000000	30.00000
25%	931.000000	317.00000
50%	1330.000000	575.00000
75%	1733.000000	957.00000
max	2534.000000	4962.00000

In [14]:
```
test_df.describe()
```

Out[14]:

	DepTime	Distance
count	100000.000000	100000.00000
mean	1338.936600	723.13011
std	480.554102	563.22322
min	1.000000	31.00000
25%	928.000000	321.00000
50%	1329.000000	574.00000
75%	1733.000000	948.00000
max	2400.000000	4962.00000

Notice, `DepTime` is the departure time in a numeric representation in 2400 hours. But, we see that `train_df` has `DepTime` more than 2400 hours.

In [15]:
```
idx = (train_df.DepTime > 2400)
```

In [16]:
```
idx.sum()
```

Out[16]:
```
17
```

In [17]:
```
train_df[idx]
```

Out[17]:

	Month	DayofMonth	DayOfWeek	DepTime	UniqueCarrier	Origin	Dest	Distance	dep_delay
8189	c-6	c-14	c-2	2435	EV	CVG	AVL	275	
20766	c-5	c-31	c-2	2534	EV	ATL	HSV	151	
27391	c-3	c-23	c-4	2505	EV	ATL	AGS	143	
44332	c-7	c-15	c-5	2440	EV	ATL	SHV	552	
45796	c-8	c-18	c-4	2447	EV	ATL	JAN	341	
47218	c-1	c-2	c-1	2500	EV	ATL	ILM	377	
48180	c-2	c-27	c-7	2514	EV	ATL	CAE	191	
55909	c-8	c-9	c-3	2417	EV	ATL	SYR	793	
60639	c-1	c-8	c-7	2459	EV	ATL	JAN	341	
62669	c-3	c-20	c-1	2412	EV	ATL	GSP	153	
73435	c-8	c-14	c-7	2418	EV	CHA	ATL	106	
76370	c-8	c-11	c-5	2401	EV	ATL	CLE	554	
77163	c-2	c-27	c-7	2522	EV	CVG	SHV	686	
93122	c-7	c-31	c-7	2435	EV	ATL	AVL	164	
93784	c-9	c-16	c-5	2450	EV	GNV	ATL	300	
98924	c-7	c-5	c-2	2530	EV	ATL	GNV	300	
99072	c-4	c-30	c-6	2415	EV	CAE	ATL	191	

Let's replace these values by the closeset `DepTime` of those flights whose `UniqueCarrier`, `Origin`, and `Dest` features are same.

In [18]:

```
idx_lst = list(train_df[idx].index)

for i in range(len(idx_lst)):

    df_i = train_df[['DepTime', 'UniqueCarrier', 'Origin', 'Dest']].loc[[idx_lst[i]

    df1 = train_df[['DepTime', 'UniqueCarrier',
                    'Origin', 'Dest']][(train_df.Origin == df_i.Origin.values[0])
                                      & (train_df.Dest == df_i.Dest.values[0])
                                      & (train_df.UniqueCarrier == df_i.UniqueCarr
                                      & (train_df.DepTime <= 2400)
                                     ]
    df1['DiffTime'] = abs(df1.DepTime - df_i.loc[idx_lst[i],'DepTime'])

    closest_DepTime_idx = df1.DiffTime.idxmin()

    closest_DepTime = train_df.DepTime.loc[[closest_DepTime_idx]].values[0]

    train_df.loc[idx_lst[i], 'DepTime'] = closest_DepTime
```

In [19]:

```
(train_df.DepTime > 2400).sum()
```

Out[19]: 0

In [20]:
```
train_df[idx]
```

Out[20]:

	Month	DayofMonth	DayOfWeek	DepTime	UniqueCarrier	Origin	Dest	Distance	dep_delay
8189	c-6	c-14	c-2	2300	EV	CVG	AVL	275	
20766	c-5	c-31	c-2	2355	EV	ATL	HSV	151	
27391	c-3	c-23	c-4	2240	EV	ATL	AGS	143	
44332	c-7	c-15	c-5	2356	EV	ATL	SHV	552	
45796	c-8	c-18	c-4	2317	EV	ATL	JAN	341	
47218	c-1	c-2	c-1	2249	EV	ATL	ILM	377	
48180	c-2	c-27	c-7	2304	EV	ATL	CAE	191	
55909	c-8	c-9	c-3	2240	EV	ATL	SYR	793	
60639	c-1	c-8	c-7	2317	EV	ATL	JAN	341	
62669	c-3	c-20	c-1	2300	EV	ATL	GSP	153	
73435	c-8	c-14	c-7	2125	EV	CHA	ATL	106	
76370	c-8	c-11	c-5	2210	EV	ATL	CLE	554	
77163	c-2	c-27	c-7	2055	EV	CVG	SHV	686	
93122	c-7	c-31	c-7	2305	EV	ATL	AVL	164	
93784	c-9	c-16	c-5	2021	EV	GNV	ATL	300	
98924	c-7	c-5	c-2	2201	EV	ATL	GNV	300	
99072	c-4	c-30	c-6	2024	EV	CAE	ATL	191	

In [21]:
```
train_df.describe()
```

Out[21]:

	DepTime	Distance
count	100000.000000	100000.00000
mean	1341.484730	729.39716
std	476.297622	574.61686
min	1.000000	30.00000
25%	931.000000	317.00000
50%	1330.000000	575.00000
75%	1733.000000	957.00000
max	2400.000000	4962.00000

In [22]:
```
train_df.isnull().sum()
```

Out[22]:
```
Month                0
DayofMonth           0
DayOfWeek            0
DepTime              0
UniqueCarrier        0
Origin               0
Dest                 0
Distance             0
dep_delayed_15min    0
dtype: int64
```

In [23]:
```
test_df.isnull().sum()
```

Out[23]:
```
Month            0
DayofMonth       0
DayOfWeek        0
DepTime          0
UniqueCarrier    0
Origin           0
Dest             0
Distance         0
dtype: int64
```

The response variable is `dep_delayed_15min` which is a categorical column, so we need to map the `Y` to `1` and `N` to `0`.

In [24]:
```
train_df.dep_delayed_15min.value_counts(dropna = False)
```

Out[24]:
```
N    80956
Y    19044
Name: dep_delayed_15min, dtype: int64
```

In [25]:
```
#train_df = train_df[train_df.DepTime <= 2400].copy()
y_train = train_df['dep_delayed_15min'].map({'Y': 1, 'N': 0})

print(y_train.value_counts(dropna = False))
```

```
0    80956
1    19044
Name: dep_delayed_15min, dtype: int64
```

In [26]:
```
# y_train = y_train.values
```

Feature Engineering

Use these defined functions to create additional features for the model. Run the cell to add the functions to your workspace.

In [27]:
```
def label_enc(df_column):
    df_column = LabelEncoder().fit_transform(df_column)
    return df_column

def make_harmonic_features_sin(value, period=2400):
    value *= 2 * np.pi / period
    return np.sin(value)

def make_harmonic_features_cos(value, period=2400):
    value *= 2 * np.pi / period
    return np.cos(value)
```

```
def feature_eng(df):
    df['flight'] = df['Origin']+df['Dest']
    df['Month'] = df.Month.map(lambda x: x.split('-')[-1]).astype('int32')
    df['DayofMonth'] = df.DayofMonth.map(lambda x: x.split('-')[-1]).astype('uint8'
    df['begin_of_month'] = (df['DayofMonth'] < 10).astype('uint8')
    df['midddle_of_month'] = ((df['DayofMonth'] >= 10)&(df['DayofMonth'] < 20)).ast
    df['end_of_month'] = (df['DayofMonth'] >= 20).astype('uint8')
    df['DayOfWeek'] = df.DayOfWeek.map(lambda x: x.split('-')[-1]).astype('uint8')
    df['hour'] = df.DepTime.map(lambda x: x/100).astype('int32')
    df['morning'] = df['hour'].map(lambda x: 1 if (x <= 11)& (x >= 7) else 0).astyp
    df['day'] = df['hour'].map(lambda x: 1 if (x >= 12) & (x <= 18) else 0).astype(
    df['evening'] = df['hour'].map(lambda x: 1 if (x >= 19) & (x <= 23) else 0).ast
    df['night'] = df['hour'].map(lambda x: 1 if (x >= 0) & (x <= 6) else 0).astype(
    df['winter'] = df['Month'].map(lambda x: x in [12, 1, 2]).astype('int32')
    df['spring'] = df['Month'].map(lambda x: x in [3, 4, 5]).astype('int32')
    df['summer'] = df['Month'].map(lambda x: x in [6, 7, 8]).astype('int32')
    df['autumn'] = df['Month'].map(lambda x: x in [9, 10, 11]).astype('int32')
    df['holiday'] = (df['DayOfWeek'] >= 5).astype(int)
    df['weekday'] = (df['DayOfWeek'] < 5).astype(int)
    df['airport_dest_per_month'] = df.groupby(['Dest', 'Month'])['Dest'].transform(
    df['airport_origin_per_month'] = df.groupby(['Origin', 'Month'])['Origin'].trar
    df['airport_dest_count'] = df.groupby(['Dest'])['Dest'].transform('count')
    df['airport_origin_count'] = df.groupby(['Origin'])['Origin'].transform('count'
    df['carrier_count'] = df.groupby(['UniqueCarrier'])['Dest'].transform('count')
    df['carrier_count_per month'] = df.groupby(['UniqueCarrier', 'Month'])['Dest'].
    df['deptime_cos'] = df['DepTime'].map(make_harmonic_features_cos)
    df['deptime_sin'] = df['DepTime'].map(make_harmonic_features_sin)
    df['flightUC'] = df['flight']+df['UniqueCarrier']
    df['DestUC'] = df['Dest']+df['UniqueCarrier']
    df['OriginUC'] = df['Origin']+df['UniqueCarrier']
    return df.drop('DepTime', axis=1)
```

Concatenate the training and testing dataframes.

In [28]:
```
full_df = pd.concat([train_df.drop('dep_delayed_15min', axis=1), test_df])
full_df = feature_eng(full_df)
```

Apply the earlier defined feature engineering functions to the full dataframe.

In [29]:
```
for column in ['UniqueCarrier', 'Origin', 'Dest','flight',  'flightUC', 'DestUC', '
    full_df[column] = label_enc(full_df[column])
```

Split the new full dataframe into X_train and X_test.

In [30]:
```
X_train = full_df[:train_df.shape[0]]
X_test = full_df[train_df.shape[0]:]
```

Create a list of the categorical features.

In [31]:
```
categorical_features = ['Month',  'DayOfWeek', 'UniqueCarrier', 'Origin', 'Dest','f
```

Let's build a light GBM model to test the bayesian optimizer.

Using `BayesianOptimization` library

LightGBM (re: documentation) is a gradient boosting framework that uses tree-based learning algorithms. It is designed to be distributed and efficient with the following

advantages:

- Faster training speed and higher efficiency.
- Lower memory usage.
- Better accuracy.
- Support of parallel and GPU learning.
- Capable of handling large-scale data.

First, we define the function we want to maximize and that will count cross-validation metrics of `lightGBM` for our parameters.

Some params such as `num_leaves`, `max_depth`, `min_child_samples`, and `min_data_in_leaf` should be integers.

In [32]:
```
print(lightgbm.__version__)
```

```
3.3.3.99
```

Note that `early_stopping_rounds` parameter is now deprecated in `lightgbm.cv`. So, we will need to pass `.early_stopping()` (re: lightgbm.early_stopping) callback via `callbacks` argument instead.

In [33]:
```
def lgb_eval(num_leaves, max_depth,
             lambda_l2, lambda_l1,
             min_child_samples, min_data_in_leaf):

    params = {
        "objective" : "binary",
        "metric" : "auc",
        'is_unbalance': True,
        "num_leaves" : int(num_leaves),
        "max_depth" : int(max_depth),
        "lambda_l2" : lambda_l2,
        "lambda_l1" : lambda_l1,
        "num_threads" : 20,
        "min_child_samples" : int(min_child_samples),
        'min_data_in_leaf': int(min_data_in_leaf),
        "learning_rate" : 0.03,
        "subsample_freq" : 5,
        "bagging_seed" : 42,
        "verbosity" : -1
    }

    lgtrain = lightgbm.Dataset(data = X_train,
                               label = y_train,
                               categorical_feature=categorical_features)

    callbacks_list = [lightgbm.early_stopping(stopping_rounds = 100)]

    cv_result = lightgbm.cv(params = params,
                            train_set = lgtrain,
                            num_boost_round = 1000,
                            # early_stopping_rounds=100,
                            callbacks = callbacks_list,
                            stratified=True,
                            nfold=3,
                            seed = 0
                           )
```

```python
    # return cv_result['auc-mean'][-1]
    # return cv_result['valid auc-mean'][-1]
    return cv_result
```

In [34]:
```python
cv_result = lgb_eval(num_leaves = 25, max_depth = 5,
             lambda_l2 = 0.05, lambda_l1 = 0.05,
             min_child_samples = 50, min_data_in_leaf = 100)
```

```
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[89]    cv_agg's valid auc: 0.719896 + 0.00465946
```

In [35]:
```python
cv_result
```

```
Out[35]: {'valid auc-mean': [0.6929202501986076,
  0.7004373284428901,
  0.7044168279777426,
  0.7062722474806749,
  0.707461487890057,
  0.708336063026542,
  0.7092492955179744,
  0.7100503344991246,
  0.7104914452692426,
  0.7110266802194968,
  0.711251028534698,
  0.7119665958384988,
  0.7124862253233114,
  0.7128016216421983,
  0.7129819648235981,
  0.7134900508735692,
  0.7136614419098768,
  0.7138113356560423,
  0.7141437656673156,
  0.714191112689942 6,
  0.7146255450817164,
  0.7148423569791748,
  0.7149063190179962,
  0.7151688463754189,
  0.7153159671451911,
  0.7153477532278304,
  0.7156227542619259,
  0.7157065434223361,
  0.7158535530346765,
  0.7159019494818867,
  0.716048742641024,
  0.7160988827018498,
  0.7162444117387681,
  0.716474729752290 9,
  0.716424707950015,
  0.7165515905277329,
  0.7166391281839392,
  0.7166454279781913,
  0.7168430008429257,
  0.7169838985977983,
  0.7169947544975508,
  0.7171104591866403,
  0.7172160464371494,
  0.717365004477136,
  0.7174210278362662,
  0.7174953234243598,
  0.717547122639956,
  0.7176211215333214,
  0.7176922361491872,
  0.7177661409379491,
  0.7178291143044774,
  0.7179096453921402,
  0.7179613588520001,
  0.7179849728208083,
  0.718067881360828,
  0.7181062466991701,
  0.7181186495491149,
  0.718204919146677 5,
  0.7182581382939633,
  0.7183856399846138,
  0.7184485119782034,
  0.7185637847733496,
```

0.7186999319766785,
0.7187748960023866,
0.7189861361079579,
0.7190467640065256,
0.7190906076259211,
0.7191280886992676,
0.7190701705543976,
0.7191488561790916,
0.7191122388654376,
0.7191855480654259,
0.7193478387528972,
0.7193409836716601,
0.7194188560514427,
0.7194832489736919,
0.7195728616851156,
0.7195250825387386,
0.7195586869226854,
0.7196710393729885,
0.71973554009343,
0.7197629626792029,
0.7197892347356579,
0.7198446359842715,
0.7198294631138391,
0.7198343088170241,
0.7198896620830942,
0.7198657558585723,
0.719896186597358],
'valid auc-stdv': [0.005335712530694081,
0.004951626863911197,
0.005269853536785308,
0.004893728663251103,
0.004265449585639365,
0.0044501002357770734,
0.004108270684326549,
0.0036283377586810795,
0.003696565579825946,
0.003777635960997283,
0.003539654707075172,
0.0039469729500791,
0.003820124197991052,
0.003864605635353278,
0.003953586810559165,
0.00403313487584218,
0.003904817880813542,
0.003785026286651578,
0.0035469684825931457,
0.003704702051967724,
0.0038857005357057902,
0.003940127732524427,
0.003924803887869135,
0.004147417020706626,
0.004025062024963819,
0.0040240062403109535,
0.0038852194883877487,
0.003946737380852556,
0.0038543047872557443,
0.0039476424657766,
0.004195962589079066,
0.004178775744039299,
0.004322291124899235,
0.004410228975044078,
0.004351434448231003,

```
0.004362671518065668,
0.004503355871702706,
0.0045193211929612535,
0.004582379193514865,
0.004779774645286601,
0.004786981235127323,
0.004843596052435584,
0.0048733344449233915,
0.004951883204528863,
0.004929389468584705,
0.005056935229084309,
0.005048281503263609,
0.005041008096454098,
0.005001176541588234,
0.005001899293392679,
0.005000299299586746,
0.005077611531507379,
0.005053437006429143,
0.005040696700569245,
0.004986780117465001,
0.005017444077045633,
0.005005305980644856,
0.004898341245749118,
0.004967734949896259,
0.004971218720597151,
0.004900549911507634,
0.004941031032961393,
0.0050405889194350935,
0.005106767330417018,
0.005035436192167761,
0.0049635241424914964,
0.004974094629993617,
0.005001151199686021,
0.004916091162666147,
0.004988244920999459,
0.004995121009245433,
0.004956746253636251,
0.004929006209068483,
0.004825712110779603,
0.004832839480282809,
0.0048764892257422655,
0.004854628288025386,
0.004805747226619253,
0.004721424330467936,
0.0047473829078575475,
0.0047602716525936084,
0.0047497744498123115,
0.004705771141818384,
0.004692983412629555,
0.004705926398133292,
0.004579915113398914,
0.004626397599339903,
0.0046285894971449994,
0.004659461990973263]}
```

Apply the Bayesian optimizer to the function we created in the previous step to identify the best hyperparameters. We will run 10 iterations (`n_iter=10`) and set `init_points = 2` .

In [36]:
```python
def f_to_maximize(num_leaves, max_depth,
                  lambda_l2, lambda_l1,
                  min_child_samples, min_data_in_leaf):
```

```
    cv_result = lgb_eval(num_leaves, max_depth,
                         lambda_l2, lambda_l1,
                         min_child_samples, min_data_in_leaf)

    return cv_result['valid auc-mean'][-1]
```

In [37]:
```
lgbBO = BayesianOptimization(
                             f = f_to_maximize,
                             pbounds = {'num_leaves': (25, 4000),
                                        'max_depth': (5, 63),
                                        'lambda_l2': (0.0, 0.05),
                                        'lambda_l1': (0.0, 0.05),
                                        'min_child_samples': (50, 10000),
                                        'min_data_in_leaf': (100, 2000)
                                       },
                             random_state=1
                            )

lgbBO.maximize(n_iter=10, init_points=2)
```

```
|   iter    |  target   | lambda_l1 | lambda_l2 | max_depth | min_ch... | min_d
a... | num_le... |
-------------------------------------------------------------------------------------
---------------
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[95]	cv_agg's valid auc: 0.719205 + 0.00412265
| 1         | 0.7192    | 0.02085   | 0.03602   | 5.007     | 3.058e+03 | 378.8
| 392.0     |
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[229]	cv_agg's valid auc: 0.728491 + 0.0044553
| 2         | 0.7285    | 0.009313  | 0.01728   | 28.01     | 5.411e+03 | 896.5
| 2.749e+03 |
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[23]	cv_agg's valid auc: 0.719487 + 0.0036219
| 3         | 0.7195    | 0.02807   | 0.01877   | 32.78     | 5.634e+03 | 361.7
| 2.587e+03 |
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[28]	cv_agg's valid auc: 0.718921 + 0.00386974
| 4         | 0.7189    | 0.02989   | 0.01592   | 43.46     | 4.511e+03 | 427.2
| 1.229e+03 |
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[228]	cv_agg's valid auc: 0.724459 + 0.00423982
| 5         | 0.7245    | 0.04486   | 0.02648   | 8.422     | 8.381e+03 | 531.1
| 760.9     |
Training until validation scores don't improve for 100 rounds
Did not meet early stopping. Best iteration is:
[904]	cv_agg's valid auc: 0.742554 + 0.00400268
| 6         | 0.7426    | 0.02048   | 0.006115  | 47.34     | 8.687e+03 | 1.359e+0
3 | 1.595e+03 |
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[29]	cv_agg's valid auc: 0.72238 + 0.00378389
| 7         | 0.7224    | 0.04584   | 0.004731  | 62.23     | 9.473e+03 | 125.3
| 1.83e+03  |
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[807]	cv_agg's valid auc: 0.743677 + 0.00467297
| 8         | 0.7437    | 0.0347    | 0.02868   | 24.09     | 7.113e+03 | 1.615e+0
3 | 145.7     |
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[724]	cv_agg's valid auc: 0.743217 + 0.0048058
| 9         | 0.7432    | 0.01009   | 0.01468   | 14.24     | 2.839e+03 | 1.642e+0
3 | 2.833e+03 |
Training until validation scores don't improve for 100 rounds
Early stopping, best iteration is:
[780]	cv_agg's valid auc: 0.743074 + 0.00435219
| 10        | 0.7431    | 0.0002309 | 0.04105   | 34.72     | 5.39e+03  | 1.503e+0
3 | 3.782e+03 |
Training until validation scores don't improve for 100 rounds
Did not meet early stopping. Best iteration is:
[914]	cv_agg's valid auc: 0.743272 + 0.0040963
| 11        | 0.7433    | 0.001704  | 0.03593   | 55.09     | 3.039e+03 | 1.709e+0
3 | 1.702e+03 |
Training until validation scores don't improve for 100 rounds
Did not meet early stopping. Best iteration is:
[1000]	cv_agg's valid auc: 0.74312 + 0.00398806
```

```
|  12        |  0.7431    |  0.03704   |  0.03347   |  6.426     |  7.676e+03 |  1.943e+0
3 |  1.317e+03 |
=================================================================================
===============
```

Print the best result by using the `.max` function.

In [38]:
```python
lgbBO.max
```

Out[38]:
```
{'target': 0.7436766033692709,
 'params': {'lambda_l1': 0.03469976418304162,
  'lambda_l2': 0.02868054749024032,
  'max_depth': 24.086990454131087,
  'min_child_samples': 7112.606300057905,
  'min_data_in_leaf': 1615.1169568627445,
  'num_leaves': 145.731287667976}}
```

Review the process at each step by using the `.res` function.

In [39]:
```python
lgbBO.res[0]
```

Out[39]:
```
{'target': 0.7192046561987094,
 'params': {'lambda_l1': 0.020851100235128702,
  'lambda_l2': 0.036016224672107904,
  'max_depth': 5.006633739406004,
  'min_child_samples': 3058.2090976868058,
  'min_data_in_leaf': 378.8361925525148,
  'num_leaves': 392.04591420597126}}
```

In [40]:
```python
for i, res in enumerate(lgbBO.res):
    print(f"Iteration {i}: \n\t{res}\n")
```

```
Iteration 0:
        {'target': 0.7192046561987094, 'params': {'lambda_l1': 0.02085110023512870
2, 'lambda_l2': 0.036016224672107904, 'max_depth': 5.006633739406004, 'min_child_s
amples': 3058.2090976868058, 'min_data_in_leaf': 378.8361925525148, 'num_leaves':
392.04591420597126}}

Iteration 1:
        {'target': 0.7284909691129068, 'params': {'lambda_l1': 0.00931301056888354
6, 'lambda_l2': 0.017278036352152387, 'max_depth': 28.012513505378855, 'min_child_
samples': 5411.2265033334015, 'min_data_in_leaf': 896.4695773662602, 'num_leaves':
2748.747514077119}}

Iteration 2:
        {'target': 0.7194867165879869, 'params': {'lambda_l1': 0.0280714440278675
5, 'lambda_l2': 0.018768698805027365, 'max_depth': 32.776027838297395, 'min_child_
samples': 5634.398463157358, 'min_data_in_leaf': 361.74531561040993, 'num_leaves':
2586.9523143420997}}

Iteration 3:
        {'target': 0.7189211257797788, 'params': {'lambda_l1': 0.0298855399702403
2, 'lambda_l2': 0.015920641365851813, 'max_depth': 43.457408005967025, 'min_child_
samples': 4511.479070472726, 'min_data_in_leaf': 427.1898899167602, 'num_leaves':
1229.1640671419732}}

Iteration 4:
        {'target': 0.7244590780581482, 'params': {'lambda_l1': 0.04485907103440249
5, 'lambda_l2': 0.026481551575717174, 'max_depth': 8.421792068340979, 'min_child_s
amples': 8380.938792623298, 'min_data_in_leaf': 531.0767301432701, 'num_leaves': 7
60.9037338702151}}

Iteration 5:
        {'target': 0.7425535314221031, 'params': {'lambda_l1': 0.0204759241176022
2, 'lambda_l2': 0.0061148610969832436, 'max_depth': 47.338933657789326, 'min_child
_samples': 8687.455713973215, 'min_data_in_leaf': 1359.487526530498, 'num_leaves':
1595.1065395531155}}

Iteration 6:
        {'target': 0.722379865090064, 'params': {'lambda_l1': 0.04583991321660958,
'lambda_l2': 0.00473059880053558, 'max_depth': 62.225317117758415, 'min_child_samp
les': 9472.526739930845, 'min_data_in_leaf': 125.32170264605843, 'num_leaves': 182
9.9255125413588}}

Iteration 7:
        {'target': 0.7436766033692709, 'params': {'lambda_l1': 0.0346997641830416
2, 'lambda_l2': 0.02868054749024032, 'max_depth': 24.086990454131087, 'min_child_s
amples': 7112.606300057905, 'min_data_in_leaf': 1615.1169568627445, 'num_leaves':
145.731287667976}}

Iteration 8:
        {'target': 0.7432169068035505, 'params': {'lambda_l1': 0.01008976968333178
5, 'lambda_l2': 0.014684694101171526, 'max_depth': 14.235401676748285, 'min_child_
samples': 2838.525277631191, 'min_data_in_leaf': 1641.894826296873, 'num_leaves':
2832.7023352287306}}

Iteration 9:
        {'target': 0.7430742786644228, 'params': {'lambda_l1': 0.00023085970103171
595, 'lambda_l2': 0.041050660030163715, 'max_depth': 34.722236668136574, 'min_chil
d_samples': 5389.929485099175, 'min_data_in_leaf': 1502.9546377234371, 'num_leave
s': 3782.395611363518}}

Iteration 10:
        {'target': 0.743272287761489, 'params': {'lambda_l1': 0.001704411313760617
```

```
6, 'lambda_l2': 0.0359267032524404, 'max_depth': 55.08798969860291, 'min_child_sam
ples': 3039.223822953911, 'min_data_in_leaf': 1708.7214215465383, 'num_leaves': 17
02.0707131062459}}

Iteration 11:
        {'target': 0.7431204223885794, 'params': {'lambda_l1': 0.0370446620885838
3, 'lambda_l2': 0.03347077965786851, 'max_depth': 6.426212256907103, 'min_child_sa
mples': 7675.527921957597, 'min_data_in_leaf': 1943.3361144231287, 'num_leaves': 1
317.2885989797571}}
```

Using `scikit-optimize`'s `BayesSearchCV` class

Instead of `BayesianOptimization` library, we can use `scikit-optimize`'s `BayesSearchCV` which is `scikit-learn` hyperparameter search wrapper (re: skopt.BayesSearchCV). Refer to this YouTube video for a simple illustration.

Note that the default `base_estimator` parameter of `optimizer_kwargs` (re: Dict of arguments passed to Optimizer) of `BayesSearchCV` is Gaussian Process. We will use its default value to make it consistent with the `BayesianOptimization` library for comparison purpose.

The `estimator` parameter of `BayesSearchCV` will be set as `lightgbm.LGBMClassifier` (re: lightgbm.LGBMClassifier) whose `objective` argument will be set as `objective = "binary"` to consider *binary* log loss classification. This will be consistent with what we did in `BayesianOptimization`.

In [41]:
```
opt = BayesSearchCV(
                    estimator = lightgbm.LGBMClassifier(objective = "binary",
                                                        metric = "auc",
                                                        is_unbalance = True,
                                                        num_threads = 20,
                                                        learning_rate = 0.03,
                                                        subsample_freq = 5,
                                                        bagging_seed = 42,
                                                        verbosity = -1
                                                       ),
                    search_spaces = {
                        "num_leaves" : Integer(25, 4000),
                        "max_depth" : Integer(5, 63),
                        "reg_lambda" : Real(0.0, 0.05),
                        "reg_alpha" : Real(0.0, 0.05),
                        "min_child_samples" : Integer(50, 10000),
                        'min_data_in_leaf': Integer(100, 2000)
                    },
                    n_iter=10,
                    cv = 3,
                    random_state=1,
                    verbose = 1,
                    n_jobs = -1
                    )
```

In [42]:
```
opt
```

Out[42]:

```
BayesSearchCV
estimator: LGBMClassifier
LGBMClassifier
```

In [43]:

```
# executes bayesian optimization
opt.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
```

Out[43]:

```
BayesSearchCV
estimator: LGBMClassifier
LGBMClassifier
```

In [44]:

```
# model can be saved, used for predictions or scoring
opt.score(X_train, y_train)
```

Out[44]:

```
0.78944
```

In [45]:

```
# estimator which gave highest score (or smallest loss if specified) on the left ou
opt.best_estimator_
```

Out[45]:

```
LGBMClassifier
LGBMClassifier(bagging_seed=42, is_unbalance=True, learning_rate=0.03,
                max_depth=56, metric='auc', min_child_samples=3627,
                min_data_in_leaf=149, num_leaves=2294, num_threads=20,
                objective='binary', reg_alpha=0.023531218502545023,
                reg_lambda=0.022648700827027503, subsample_freq=5, verbosi
ty=-1)
```

In [46]:

```
# Score of best_estimator on the left out data NOT on X_train.
opt.best_score_
```

Out[46]:

```
0.7116199874424495
```

In [47]:

```
# Score on the training data
opt.best_estimator_.score(X_train, y_train)
```

Out[47]:

```
0.78944
```

In [48]:

```
# Parameter setting that gave the best results on the hold out data.
opt.best_params_
```

```
Out[48]: OrderedDict([('max_depth', 56),
                      ('min_child_samples', 3627),
                      ('min_data_in_leaf', 149),
                      ('num_leaves', 2294),
                      ('reg_alpha', 0.023531218502545023),
                      ('reg_lambda', 0.022648700827027503)])
```

```
In [49]: lgbBO.max
```

```
Out[49]: {'target': 0.7436766033692709,
          'params': {'lambda_l1': 0.03469976418304162,
           'lambda_l2': 0.02868054749024032,
           'max_depth': 24.086990454131087,
           'min_child_samples': 7112.606300057905,
           'min_data_in_leaf': 1615.1169568627445,
           'num_leaves': 145.731287667976}}
```

Let us present a comparison table of performance of the two schemes we reported above.

	BayesianOptimization	BayesSearchCV
Accuracy	0.7437	0.7894
lambda_l1	0.0347	0.0235
lambda_l2	0.0287	0.0226
max_depth	24.0870	56
min_child_samples	7112.6063	3627
min_data_in_leaf	1615.1170	149
num_leaves	145.7313	2294

We see that `BayesSearchCV` provided a better performance.

As mentioned earlier, `scikit-learn`'s `gp_minimize` algorithm, that also relies on Bayesian optimization using Gaussian Processes, can be used to tune the hyperparameters of a ML model. Its implementation scheme is very similar to the `BayesianOptimization` library (see an illustration), but, we have not discussed it here in this notebook.