Kaggle Version of Bayesian Hyperparameter Optimization of GBM

October 10, 2018

1 Introduction: Automated GBM Hyperparameter Optimization

In this notebook we will walk through automated hyperparameter tuning using Bayesian Optimization. Specifically, we will optimize the hyperparameters of a Gradient Boosting Machine using the Hyperopt library (with the Tree Parzen Estimator algorithm). We will compare the results of random search (implemented manually) for hyperparameter tuning with the Bayesian model-based optimization method to try and understand how the Bayesian method works and what benefits it has over uninformed search methods.

1.1 Hyperopt

Hyperopt is one of several automated hyperparameter tuning libraries using Bayesian optimization. These libraries differ in the algorithm used to both construct the surrogate (probability model) of the objective function and choose the next hyperparameters to evaluate in the objective function. Hyperopt uses the Tree Parzen Estimator (TPE). Other Python libraries include Spearmint, which uses a Gaussian process for the surrogate, and SMAC, which uses a random forest regression.

Hyperopt has a simple syntax for structuring an optimization problem which extends beyond hyperparameter tuning to any problem that involves minimizing a function. Moreover, the structure of a Bayesian Optimization problem is similar across the libraries, with the major differences coming in the syntax (and in the algorithms behind the scenes that we do not have to deal with).

```
In [1]: import pandas as pd
        import numpy as np

        # Modeling
        import lightgbm as lgb

        # Evaluation of the model
        from sklearn.model_selection import KFold
        from sklearn.model_selection import train_test_split

        MAX_EVALS = 2
        N_FOLDS = 2
```

1.2 Data

For this notebook we will work with only the `application` data. The methods developed here can work on any dataset but the run-times can be long! We will also separate the training set into training and testing to evaluate the performance of hyperparameter tuning.

In [2]:
```
# Read in data and separate into training and testing sets
features = pd.read_csv('../input/application_train.csv')

# Extract the labels and format properly
labels = np.array(features['TARGET'].astype(np.int32)).reshape((-1,))

# Drop the unneeded columns
features = features.drop(columns = ['SK_ID_CURR', 'TARGET'])

train_features, test_features, train_labels, test_labels = train_test_split(features, l

print('Train shape: ', train_features.shape)
print('Test shape: ', test_features.shape)

train_features.head()
```

```
Train shape:  (246008, 120)
Test shape:  (61503, 120)
```

Out[2]:

	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALTY	\
44645	Cash loans	F	N	Y	
149709	Cash loans	F	N	Y	
9290	Cash loans	F	N	Y	
279223	Cash loans	M	Y	N	
35335	Cash loans	M	Y	Y	

	CNT_CHILDREN	AMT_INCOME_TOTAL	AMT_CREDIT	AMT_ANNUITY	\
44645	0	135000.0	490495.5	36027.0	
149709	1	99000.0	450000.0	46242.0	
9290	0	148500.0	571446.0	18562.5	
279223	1	157500.0	331920.0	16096.5	
35335	1	225000.0	1125000.0	33025.5	

	AMT_GOODS_PRICE	NAME_TYPE_SUITE	...	\
44645	454500.0	Unaccompanied	...	
149709	450000.0	Spouse, partner	...	
9290	477000.0	Unaccompanied	...	
279223	225000.0	Unaccompanied	...	
35335	1125000.0	Unaccompanied	...	

	FLAG_DOCUMENT_18	FLAG_DOCUMENT_19	FLAG_DOCUMENT_20	FLAG_DOCUMENT_21	\
44645	0	0	0	0	

```
149709                0                0                0                0
9290                  0                0                0                0
279223                0                0                0                0
35335                 0                0                0                0

        AMT_REQ_CREDIT_BUREAU_HOUR  AMT_REQ_CREDIT_BUREAU_DAY  \
44645                          0.0                        0.0
149709                         NaN                        NaN
9290                           0.0                        0.0
279223                         0.0                        0.0
35335                          0.0                        0.0

        AMT_REQ_CREDIT_BUREAU_WEEK  AMT_REQ_CREDIT_BUREAU_MON  \
44645                          0.0                        0.0
149709                         NaN                        NaN
9290                           0.0                        1.0
279223                         0.0                        0.0
35335                          0.0                        0.0

        AMT_REQ_CREDIT_BUREAU_QRT  AMT_REQ_CREDIT_BUREAU_YEAR
44645                         0.0                        1.0
149709                        NaN                        NaN
9290                          0.0                        2.0
279223                        1.0                        0.0
35335                         0.0                        2.0

[5 rows x 120 columns]
```

1.2.1 Gradient Boosting Machine Default Model

We will use the LightGBM implementation of the gradient boosting machine. This is much faster than the Scikit-Learn implementation and achieves results comparable to extreme gradient boosting, XGBoost. For the baseline model, we will use the default hyperparameters as specified in LightGBM.

```
In [3]: model = lgb.LGBMClassifier(random_state=50)
        model

Out[3]: LGBMClassifier(boosting_type='gbdt', class_weight=None, colsample_bytree=1.0,
                learning_rate=0.1, max_depth=-1, min_child_samples=20,
                min_child_weight=0.001, min_split_gain=0.0, n_estimators=100,
                n_jobs=-1, num_leaves=31, objective=None, random_state=50,
                reg_alpha=0.0, reg_lambda=0.0, silent=True, subsample=1.0,
                subsample_for_bin=200000, subsample_freq=1)
```

All we need to do is fit the model on the training data and make predictions on the testing data. For the predictions, because we are measuring ROC AUC and not accuracy, we have the model predict probabilities and not hard binary values.

Encoding Categorical Variables For the default model, we will use one-hot encoding of categorical variables. The gradient boosting machine can handle categorical variables that are integer encoded, which we can look at incorporating as one of our hyperparameters over which to search.

```
In [4]: one_hot_train = pd.get_dummies(train_features)
        one_hot_test = pd.get_dummies(test_features)

        one_hot_train, one_hot_test = one_hot_train.align(one_hot_test, axis = 1, join = 'inner
        one_hot_features = list(one_hot_train.columns)

        print('Number of features: ', one_hot_train.shape[1])
```

```
Number of features:  242
```

```
In [5]: from sklearn.metrics import roc_auc_score
        from sklearn.model_selection import cross_val_score
        from timeit import default_timer as timer

        start = timer()
        cv = cross_val_score(estimator = model, X = one_hot_train,
                             y = train_labels,
                             verbose = 2, n_jobs = -1,
                             cv = N_FOLDS, scoring = 'roc_auc')
        cv_time = timer() - start
```

```
[Parallel(n_jobs=-1)]: Done   2 out of   2 | elapsed:   21.9s finished
```

```
In [6]: print('Baseline model {} fold cv AUC ROC score: {:.5f}'.format(N_FOLDS, np.mean(cv)))
        print('Baseline model eval time: {:.2f} seconds.'.format(cv_time))
```

```
Baseline model 2 fold cv AUC ROC score: 0.75247
Baseline model eval time: 22.52 seconds.
```

That's out metric to beat! Of course, this performance will not necessarily translate to the comptition data, but we can use it for comparison purposes.

Label Encoding The other option we have for categorical features is label encoding. This maps each different value of a categorical feature to an integer. After we convert the values, we tell the model the categorical features.

```
In [7]: from sklearn.preprocessing import LabelEncoder

        le = LabelEncoder()

        le_train = train_features.copy()
        le_test = test_features.copy()
```

```
In [8]: cat_features = []

        for i, col in enumerate(le_train):
            if le_train[col].dtype == 'object':
                le_train[col] = le.fit_transform(np.array(le_train[col].astype(str)).reshape((
                le_test[col] = le.transform(np.array(le_test[col].astype(str)).reshape((-1, ))
                cat_features.append(i)

        print('Number of features: ', le_train.shape[1])

Number of features:  120


In [9]: start = timer()
        cv = cross_val_score(estimator = model, X = le_train, y = train_labels,
                             fit_params = {'categorical_feature': cat_features},
                             verbose = 2, n_jobs = -1, cv = N_FOLDS,
                             scoring = 'roc_auc')
        cv_time = timer() - start

[Parallel(n_jobs=-1)]: Done   2 out of   2 | elapsed:   20.4s finished


In [10]: print('Baseline model with label encoding {} fold cv AUC ROC score: {:.5f}'.format(N_
         print('Baseline model with label encoding eval time: {:.2f} seconds.'.format(cv_time)

Baseline model with label encoding 2 fold cv AUC ROC score: 0.75171
Baseline model with label encoding eval time: 20.90 seconds.
```

2 Random Search

First we will implement a common technique for hyperparameter optimization: random search. Each iteration, we choose a random set of model hyperparameters from a search space. Empirically, random search is very effective, returning nearly as good results as grid search with a significant reduction in time spent searching. However, it is still an uninformed method in the sense that it does not use past evaluations of the objective function to inform the choices it makes for the next evaluation.

Random search uses the following four parts, which also are used in Bayesian hyperparameter optimization:

1. Domain: values over which to search
2. Optimization algorithm: pick the next values at random! (yes this qualifies as an algorithm)
3. Objective function to minimize: in this case our metric is cross validation ROC AUC
4. Results history that tracks the hyperparameters tried and the cross validation metric

Random search can be implemented in the Scikit-Learn library using `RandomizedSearchCV`, however, because we are using Early Stopping (to determine the optimal number of estimators),

we will have to implement the method ourselves (more practice!). This is pretty straightforward, and many of the ideas in random search will transfer over to Bayesian hyperparameter optimization.

```
In [11]: import random
```

2.1 Domain for Random Search

Random search and Bayesian optimization both search for hyperparameters from a domain. For random (or grid search) this domain is called a hyperparameter grid and uses discrete values for the hyperparameters.

First, let's look at all of the hyperparamters that need to be tuned.

```
In [12]: lgb.LGBMClassifier()
```

```
Out[12]: LGBMClassifier(boosting_type='gbdt', class_weight=None, colsample_bytree=1.0,
                 learning_rate=0.1, max_depth=-1, min_child_samples=20,
                 min_child_weight=0.001, min_split_gain=0.0, n_estimators=100,
                 n_jobs=-1, num_leaves=31, objective=None, random_state=None,
                 reg_alpha=0.0, reg_lambda=0.0, silent=True, subsample=1.0,
                 subsample_for_bin=200000, subsample_freq=1)
```

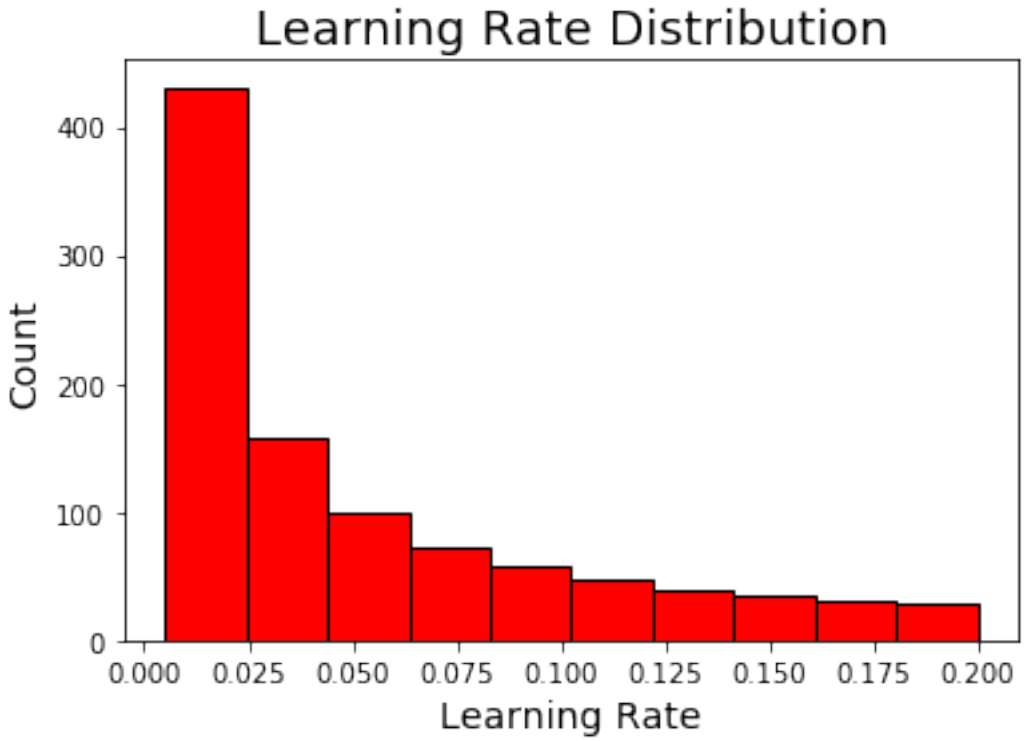
Based on the default values, we can construct the following hyperparameter grid. It's difficult to say ahead of time what choices will work best, so we will use a wide range of values centered around the default for most of the hyperparameters.

The `subsample_dist` will be used for the `subsample` parameter but we can't put it in the param grid because `boosting_type=goss` does not support row subsampling. Therefore we will use an `if` statement when choosing our hyperparameters to choose a subsample ratio if the boosting type is not `goss`.

```
In [13]: # Hyperparameter grid
         param_grid = {
             'class_weight': [None, 'balanced'],
             'boosting_type': ['gbdt', 'goss', 'dart'],
             'num_leaves': list(range(30, 150)),
             'learning_rate': list(np.logspace(np.log(0.005), np.log(0.2), base = np.exp(1), nu
             'subsample_for_bin': list(range(20000, 300000, 20000)),
             'min_child_samples': list(range(20, 500, 5)),
             'reg_alpha': list(np.linspace(0, 1)),
             'reg_lambda': list(np.linspace(0, 1)),
             'colsample_bytree': list(np.linspace(0.6, 1, 10)),
             'encoding': ['one_hot', 'label']
         }

         # Subsampling (only applicable with 'goss')
         subsample_dist = list(np.linspace(0.5, 1, 100))
```

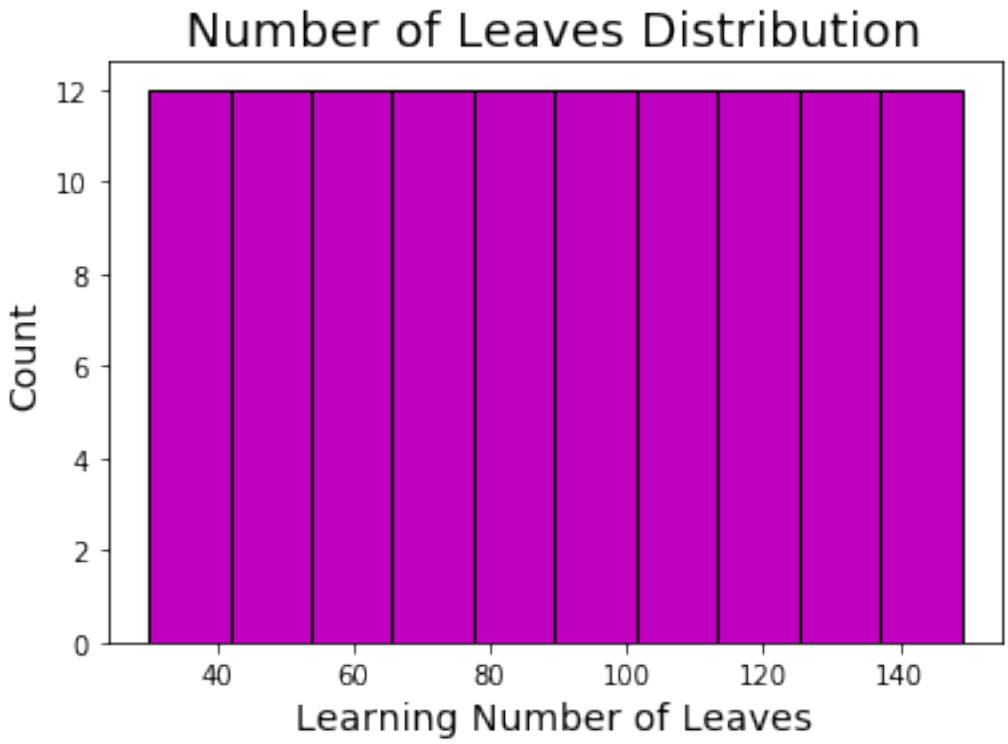
Let's look at two of the distributions, the `learning_rate` and the `num_leaves`. The learning rate is typically represented by a logarithmic distribution because it can vary over several orders of magnitude. `np.logspace` returns values evenly spaced over a log-scale (so if we take the log of the resulting values, the distribution will be uniform.)

```
In [14]: import matplotlib.pyplot as plt
         import seaborn as sns

         %matplotlib inline

         plt.hist(param_grid['learning_rate'], color = 'r', edgecolor = 'k');
         plt.xlabel('Learning Rate', size = 14); plt.ylabel('Count', size = 14); plt.title('Lea
```



Smaller values of the learning rate are more common with the values between 0.005 and 0.2. The width of the domain is fairly large indicating a large amount of uncertainty on our part about the optimal value (which we hope is somewhere in the grid)!

```
In [15]: plt.hist(param_grid['num_leaves'], color = 'm', edgecolor = 'k')
         plt.xlabel('Learning Number of Leaves', size = 14); plt.ylabel('Count', size = 14); pl
```



The number of leaves is a simple uniform domain.

2.1.1 Sampling from Hyperparameter Domain

Let's look at how we sample a set of hyperparameters from our grid using a dictionary comprehension.

```
In [16]: # Randomly sample parameters for gbm
         params = {key: random.sample(value, 1)[0] for key, value in param_grid.items()}
         params
```

```
Out[16]: {'class_weight': 'balanced',
          'boosting_type': 'dart',
          'num_leaves': 104,
          'learning_rate': 0.03168121522242165,
          'subsample_for_bin': 220000,
          'min_child_samples': 185,
          'reg_alpha': 0.9591836734693877,
          'reg_lambda': 0.8571428571428571,
          'colsample_bytree': 0.6444444444444444,
          'encoding': 'one_hot'}
```

To add a `subsample` ratio if the `boosting_type` is not `goss`, we can use an if statement.

```
In [17]: params['subsample'] = random.sample(subsample_dist, 1)[0] if params['boosting_type']
         params
```

```
Out[17]: {'class_weight': 'balanced',
          'boosting_type': 'dart',
          'num_leaves': 104,
          'learning_rate': 0.03168121522242165,
          'subsample_for_bin': 220000,
          'min_child_samples': 185,
          'reg_alpha': 0.9591836734693877,
          'reg_lambda': 0.8571428571428571,
          'colsample_bytree': 0.6444444444444444,
          'encoding': 'one_hot',
          'subsample': 0.7777777777777778}
```

We set the subsample to 1.0 if boosting type is goss which is the same as not using any subsampling. (Subsampling is when we train on a subset of the rows (observations) rather than all of them. This technique is also referred to as bagging for "bootstrap aggregating").

2.2 Cross Validation with Early Stopping in LightGBM

The scikit-learn cross validation api does not include the option for early stopping. Therefore, we will use the LightGBM cross validation function with 100 early stopping rounds. To use this function, we need to create a dataset from our features and labels.

```
In [18]: # Convert to numpy array for splitting in cross validation
         one_hot_features = np.array(one_hot_train)
         one_hot_features_test = np.array(one_hot_test)

         le_features = np.array(le_train)
         le_features_test = np.array(le_test)

         labels = train_labels[:]
         labels_test = test_labels[:]
```

```
In [19]: # Create a lgb dataset
         oh_train_set = lgb.Dataset(one_hot_features, label = train_labels)
         le_train_set = lgb.Dataset(le_features, label = train_labels)
```

```
In [ ]: # Perform cross validation with 10 folds
        r = lgb.cv(params, oh_train_set, num_boost_round = 10000, nfold = 10, metrics = 'auc',
                   early_stopping_rounds = 100, verbose_eval = False, seed = 50)

        # Highest score
        r_best = np.max(r['auc-mean'])

        # Standard deviation of best score
        r_best_std = r['auc-stdv'][np.argmax(r['auc-mean'])]

        print('The maximium ROC AUC in cross validation was {:.5f} with std of {:.5f}.'.format
        print('The ideal number of iterations was {}.'.format(np.argmax(r['auc-mean']) + 1))
```

2.2.1 Results Dataframe

We have our domain and our algorithm which in this case is random selection. The other two parts we need for an optimization problem are an objective function and a data structure to keep track of the results (these four parts will also be required in Bayesian Optimization).

Tracking the results will be done via a `dataframe` where each row will hold one evaluation of the objective function.

```
In [ ]: # Dataframe to hold cv results
        random_results = pd.DataFrame(columns = ['loss', 'params', 'iteration', 'estimators',
                                      index = list(range(1, MAX_EVALS)))
```

2.2.2 Objective Function

The objective function will take in the hyperparameters and return the validation loss (along with some other information to track the search progress). We already choose our metric as ROC AUC and now we need to figure out how to measure it. We can't evaluate the ROC AUC on the test set because that would be cheating. Instead we must use a validation set to tune the model and hope that the results translate to the test set.

A better approach than drawing the validation set from the training data (thereby limiting the amount of training data we have) is KFold cross validation. In addition to not limiting the training data, this method should also give us a better estimate of generalization error on the test set because we will be using K validations rather than only one. For this example we will use 10-fold cross validation which means testing and training each set of model hyperparameters 10 times, each time using a different subset of the training data as the validation set. The objective function will return a list of information, including the validation AUC ROC. We also want to make sure to save the hyperparameters used so we know which ones are optimal (or the best out of those we tried).

In the case of random search, the next values selected are not based on the past evaluation results, but we clearly should keep track so we know what values worked the best! This will also allow us to contrast random search with informed Bayesian optimization.

```
In [ ]: def random_objective(params, iteration, n_folds = N_FOLDS):
            """Random search objective function. Takes in hyperparameters
               and returns a list of results to be saved."""

            encoding_type = params['encoding']

            # Handle the encoding
            if encoding_type == 'one_hot':
                train_set = oh_train_set
            elif encoding_type == 'label':
                train_set = le_train_set

            del params['encoding']

            start = timer()

            # Perform n_folds cross validation
```

```
        cv_results = lgb.cv(params, train_set, num_boost_round = 10000, nfold = n_folds,
                            early_stopping_rounds = 100, metrics = 'auc', seed = 50)
        end = timer()
        best_score = np.max(cv_results['auc-mean'])

        # Loss must be minimized
        loss = 1 - best_score

        # Boosting rounds that returned the highest cv score
        n_estimators = int(np.argmax(cv_results['auc-mean']) + 1)

        params['encoding'] = encoding_type

        # Return list of results
        return [loss, params, iteration, n_estimators, end - start, best_score]
```

2.3 Random Search Implementation

Now we can write a loop to iterate through the number of evals, each time choosing a different set of hyperparameters to evaluate. Each time through the function, the results are saved to the dataframe.

The %%capture magic captures any outputs from running a cell in a Jupyter Notebook. This is useful because the output from a LightGBM training run cannot be suppressed. However, if we still want to see progress (but not all the progress that LightGBM shows) we can set the option --no-display which does not capture IPython display calls. Instead of printing the information each iteration, we just call display with the same format. This is because print goes to stdout which is being captured. Sorry if this is confusing, but there really does not appear to be any other method for suppressing the LightGBM output!

```
In [ ]: from IPython.display import display
```

```
In [ ]: %%capture --no-display

        # Iterate through the specified number of evaluations
        for i in range(1, MAX_EVALS + 1):

            # Randomly sample parameters for gbm
            params = {key: random.sample(value, 1)[0] for key, value in param_grid.items()}

            # Handle the boosting type
            if params['boosting_type'] == 'goss':

                # Cannot subsample with goss
                params['subsample'] = 1.0
            else:

                # Subsample supported for gdbt and dart
                params['subsample'] = random.sample(subsample_dist, 1)[0]
```

```
        # Evaluate the objective function
        results_list = random_objective(params, i)

        if i % 50 == 0:
            # Display the information
            display('Iteration {}: {} Fold CV AUC ROC {:.5f}'.format(i, N_FOLDS, results_li

        # Add results to next row in dataframe
        random_results.loc[i, :] = results_list
```

In [ ]:
```
# Sort results by best validation score
random_results.sort_values('loss', ascending = True, inplace = True)
random_results.reset_index(inplace = True, drop = True)
random_results.head()
```

In [ ]:
```
random_results.to_csv('results/random_results_kaggle.csv', index = False)
```

2.3.1 Random Search Performance

As a reminder, the baseline gradient boosting model achieved a score of 0.71 on the training set. We can use the best parameters from random search and evaluate them on the testing set.

What were the hyperparameters that returned the highest score on the objective function?

In [ ]:
```
random_results.loc[0, 'params']
```

The `estimators` key holds the average number of estimators trained with early stopping (averaged over 10 folds). We can use this as the optimal number of estimators in the gradient boosting model.

In [ ]:
```
# Find the best parameters and number of estimators
best_random_params = random_results.loc[0, 'params'].copy()
best_random_estimators = int(random_results.loc[0, 'estimators'])
best_random_model = lgb.LGBMClassifier(n_estimators=best_random_estimators, n_jobs = -
                                       objective = 'binary', **best_random_params, rand
```

In [ ]:
```
# Fit on the training data
best_random_model.fit(one_hot_features, labels)

# Make test predictions
predictions = best_random_model.predict_proba(one_hot_features_test)[:, 1]

print('The best model from random search scores {:.4f} on the test data.'.format(roc_au
print('This was achieved using {} search iterations.'.format(random_results.loc[0, 'ite
```

A modest gain over the baseline. Using more evaluations might increase the score, but at the cost of more optimization time. We also have to remember that the hyperparameters are optimized on the validation data whigh may not translate to the testing data.

Now, we can move on to Bayesian methods and see if they are able to achieve better results.

3 Bayesian Hyperparameter Optimization using Hyperopt

For Bayesian optimization in Hyperopt, we need the following four parts:

1. Objective function
2. Domain space
3. Hyperparameter optimization algorithm
4. History of results

We already used all of these in random search, but for Hyperopt we will have to make a few changes.

3.1 Objective Function

This objective function will still take in the hyperparameters but it will return not a list but a dictionary. The only requirement for an objective function in Hyperopt is that it has a key in the return dictionary called `"loss"` to minimize and a key called `"status"` indicating if the evaluation was successful.

If we want to keep track of the number of iterations, we can declare a global variables called `ITERATION` that is incremented every time the function is called. In addition to returning comprehensive results, every time the function is evaluated, we will write the results to a new line of a csv file. This can be useful for extremely long evaluations if we want to check on the progress (this might not be the most elegant solution, but it's better than printing to the console because our results will be saved!)

The most important part of this function is that now we need to return a **value to minimize** and not the raw ROC AUC. We are trying to find the best value of the objective function, and even though a higher ROC AUC is better, Hyperopt works to minimize a function. Therefore, a simple solution is to return 1 - ROC (we did this for random search as well for practice).

```
In [ ]: import csv
        from hyperopt import STATUS_OK
        from timeit import default_timer as timer

        def objective(params, n_folds = N_FOLDS):
            """Objective function for Gradient Boosting Machine Hyperparameter Optimization"""

            # Keep track of evals
            global ITERATION

            ITERATION += 1

            encoding_type = params['encoding']

            # Handle the encoding
            if encoding_type == 'one_hot':
                train_set = oh_train_set
            elif encoding_type == 'label':
                train_set = le_train_set
```

```
        del params['encoding']

        # Retrieve the subsample
        subsample = params['boosting_type'].get('subsample', 1.0)

        # Extract the boosting type and subsample to top level keys
        params['boosting_type'] = params['boosting_type']['boosting_type']
        params['subsample'] = subsample

        # Make sure parameters that need to be integers are integers
        for parameter_name in ['num_leaves', 'subsample_for_bin', 'min_child_samples']:
            params[parameter_name] = int(params[parameter_name])

        start = timer()

        # Perform n_folds cross validation
        cv_results = lgb.cv(params, train_set, num_boost_round = 10000, nfold = n_folds,
                            early_stopping_rounds = 100, metrics = 'auc', seed = 50)

        run_time = timer() - start

        # Extract the best score
        best_score = np.max(cv_results['auc-mean'])

        # Loss must be minimized
        loss = 1 - best_score

        # Boosting rounds that returned the highest cv score
        n_estimators = int(np.argmax(cv_results['auc-mean']) + 1)

        params['encoding'] = encoding_type

        if ITERATION % 100 == 0:
            # Display the information
            display('Iteration {}: {} Fold CV AUC ROC {:.5f}'.format(ITERATION, N_FOLDS, be

        # Write to the csv file ('a' means append)
        of_connection = open(out_file, 'a')
        writer = csv.writer(of_connection)
        writer.writerow([loss, params, ITERATION, n_estimators, run_time, best_score])
        of_connection.close()

        # Dictionary with information for evaluation
        return {'loss': loss, 'params': params, 'iteration': ITERATION,
                'estimators': n_estimators,
                'train_time': run_time, 'status': STATUS_OK}
```

Although Hyperopt only needs the `loss`, it's a good idea to track other metrics so we can

inspect the results. Later we can compare the sequence of searches to that from random search which will help us understand how the method works.

3.2 Domain Space

Specifying the domain (called the `space` in Hyperopt) is a little trickier than in grid search. In Hyperopt, and other Bayesian optimization frameworks, the domian is not a discrete grid but instead has probability distributions for each hyperparameter. For each hyperparameter, we will use the same limits as with the grid, but instead of being defined at each point, the domain represents probabilities for each hyperparameter. This will probably become clearer in the code and the images!

```
In [ ]: from hyperopt import hp
        from hyperopt.pyll.stochastic import sample
```

First we will go through an example of the learning rate. Again, we are using a log-uniform space for the learning rate defined from 0.005 to 0.2 (same as with the grid.) This time, when we graph the domain, it's more accurate to see a kernel density estimate plot than a histogram (although both show distributions).

```
In [ ]: # Create the learning rate
        learning_rate = {'learning_rate': hp.loguniform('learning_rate', np.log(0.005), np.log
```

We can visualize the learning rate by sampling from the space using a Hyperopt utility. Here we plot 10000 samples.

```
In [ ]: learning_rate_dist = []

        # Draw 10000 samples from the learning rate domain
        for _ in range(10000):
            learning_rate_dist.append(sample(learning_rate)['learning_rate'])

        plt.figure(figsize = (8, 6))
        sns.kdeplot(learning_rate_dist, color = 'red', linewidth = 2, shade = True);
        plt.title('Learning Rate Distribution', size = 18); plt.xlabel('Learning Rate', size =
```

The number of leaves is again a uniform distribution. Here we used `quniform` which means a discrete uniform (as opposed to continuous).

```
In [ ]: # Discrete uniform distribution
        num_leaves = {'num_leaves': hp.quniform('num_leaves', 30, 150, 1)}
        num_leaves_dist = []

        # Sample 10000 times from the number of leaves distribution
        for _ in range(10000):
            num_leaves_dist.append(sample(num_leaves)['num_leaves'])

        # kdeplot
        plt.figure(figsize = (8, 6))
        sns.kdeplot(num_leaves_dist, linewidth = 2, shade = True);
        plt.title('Number of Leaves Distribution', size = 18); plt.xlabel('Number of Leaves',
```

3.2.1 Conditional Domain

In Hyperopt, we can use nested conditional statements to indicate hyperparameters that depend on other hyperparameters. For example, we know that `goss` boosting type cannot use subsample, so when we set up the `boosting_type` categorical variable, we have to se the `subsample` to 1.0 while for the other boosting types it's a float between 0.5 and 1.0 Let's see this with an example:

```
In [ ]: # boosting type domain
        boosting_type = {'boosting_type': hp.choice('boosting_type',
                                                    [{'boosting_type': 'gbdt', 'subsample': hp
                                                     {'boosting_type': 'dart', 'subsample': hp
                                                     {'boosting_type': 'goss', 'subsample': 1.0

        # Draw a sample
        params = sample(boosting_type)
        params
```

We need to set both the `boosting_type` and `subsample` as top-level keys in the parameter dictionary. We can use the Python `dict.get` method with a default value of 1.0. This means that if the key is not present in the dictionary, the value returned will be the default (1.0).

```
In [ ]: # Retrieve the subsample if present otherwise set to 1.0
        subsample = params['boosting_type'].get('subsample', 1.0)

        # Extract the boosting type
        params['boosting_type'] = params['boosting_type']['boosting_type']
        params['subsample'] = subsample

        params
```

This is because the gbm cannot use the nested dictionary so we need to set the `boosting_type` and `subsample` as top level keys. Nested conditionals allow us to use a different set of hyperparameters depending on other hyperparameters. For example, we can explore different models with completely different sets of hyperparameters by using nested conditionals. The only requirement is that the first nested statement must be based on a `choice` hyperparameter (the choice could be the type of model).

3.3 Complete Bayesian Domain

Now we can define the entire domain. Each variable needs to have a label and a few parameters specifying the type and extent of the distribution. For the variables such as boosting type that are categorical, we use the `choice` variable. Other variables types include `quniform`, `loguniform`, and `uniform`. For the complete list, check out the documentation for Hyperopt.

```
In [ ]: # Define the search space
        space = {
            'encoding': hp.choice('encoding', ['one_hot', 'label']),
            'class_weight': hp.choice('class_weight', [None, 'balanced']),
            'boosting_type': hp.choice('boosting_type',
```

```
                                                    [{'boosting_type': 'gbdt', 'subsample': hp
                                                     {'boosting_type': 'dart', 'subsample': hp
                                                     {'boosting_type': 'goss', 'subsample': 1.0
        'num_leaves': hp.quniform('num_leaves', 30, 150, 1),
        'learning_rate': hp.loguniform('learning_rate', np.log(0.01), np.log(0.2)),
        'subsample_for_bin': hp.quniform('subsample_for_bin', 20000, 300000, 20000),
        'min_child_samples': hp.quniform('min_child_samples', 20, 500, 5),
        'reg_alpha': hp.uniform('reg_alpha', 0.0, 1.0),
        'reg_lambda': hp.uniform('reg_lambda', 0.0, 1.0),
        'colsample_bytree': hp.uniform('colsample_by_tree', 0.6, 1.0)
    }
```

3.3.1 Example of Sampling from the Domain

Let's sample from the domain (using the conditional logic) to see the result of each draw. Every time we run this code, the results will change. (Again notice that we need to assign the top level keys to the keywords understood by the GBM).

```
In [ ]: # Sample from the full space
        x = sample(space)

        # Conditional logic to assign top-level keys
        subsample = x['boosting_type'].get('subsample', 1.0)
        x['boosting_type'] = x['boosting_type']['boosting_type']
        x['subsample'] = subsample

        x
```

```
In [ ]: x = sample(space)
        subsample = x['boosting_type'].get('subsample', 1.0)
        x['boosting_type'] = x['boosting_type']['boosting_type']
        x['subsample'] = subsample
        x
```

3.4 Optimization Algorithm

Although this is the most technical part of Bayesian optimization, defining the algorithm to use in Hyperopt is simple. We will use the Tree Parzen Estimator (read about it in this paper) which is one method for constructing the surrogate function and choosing the next hyperparameters to evaluate.

```
In [ ]: from hyperopt import tpe

        tpe_algorithm = tpe.suggest
```

3.5 Results History

The final part is the result history. Here, we are using two methods to make sure we capture all the results:

1. A `Trials` object that stores the dictionary returned from the objective function
2. Writing to a csv file every iteration

The csv file option also lets us monitor the results of an on-going experiment. (Although do not use Excel to open the file while training is on-going. Instead check the results using `tail results/gbm_trials.csv` from bash or another command line.

```
In [ ]: from hyperopt import Trials

        trials = Trials()
```

The `Trials` object will hold everything returned from the objective function in the `.results` attribute. It also holds other information from the search, but we return everything we need from the objective.

```
In [ ]: # File to save first results
        out_file = 'results/gbm_results_kaggle.csv'
        of_connection = open(out_file, 'w')
        writer = csv.writer(of_connection)

        # Write the headers to the file
        writer.writerow(['loss', 'params', 'iteration', 'estimators', 'time', 'ROC AUC'])
        of_connection.close()
```

Every time the objective function is called, it will write one line to this file. Running the cell above does clear the file though.

3.6 Bayesian Optimization

We have everything in place needed to run the optimization. First we declare the global variable that will be used to keep track of the number of iterations. Then, we call `fmin` passing in everything we defined above and the maximum number of iterations to run.

```
In [ ]: from hyperopt import fmin

In [ ]: %%capture

        # Global variable
        global  ITERATION

        ITERATION = 0

        # Run optimization
        best = fmin(fn = objective, space = space, algo = tpe.suggest,
                    max_evals = MAX_EVALS, trials = trials, verbose = 1)
```

The `.results` attribute of the `Trials` object has all information from the objective function. If we sort this by the lowest loss, we can see the hyperparameters that performed the best in terms of validation loss.

```
In [ ]: # Sort the trials with lowest loss (highest AUC) first
        trials_results = sorted(trials.results, key = lambda x: x['loss'])
        trials_results[:2]
```

We can also access the results from the csv file (which might be easier since it's already a dataframe).

```
In [ ]: results = pd.read_csv('results/gbm_results_kaggle.csv')

        # Sort with best scores on top and reset index for slicing
        results.sort_values('loss', ascending = True, inplace = True)
        results.reset_index(inplace = True, drop = True)
        results.head()
```

For some reason, when we save to a file and then read back in, the dictionary of hyperparameters is represented as a string. To convert from a string back to a dictionary we can use the `ast` library and the `literal_eval` function.

```
In [ ]: import ast

        # Convert from a string to a dictionary
        ast.literal_eval(results.loc[0, 'params'])
```

3.7 Evaluate Best Results

Now for the moment of truth: did the optimization pay off? For this problem with a relatively small dataset, the benefits of hyperparameter optimization compared to random search are probably minor (if there are any). Random search might turn up a better result in fewer iterations simply becuase of randomness!

```
In [ ]: # Extract the ideal number of estimators and hyperparameters
        best_bayes_estimators = int(results.loc[0, 'estimators'])
        best_bayes_params = ast.literal_eval(results.loc[0, 'params']).copy()

        # Re-create the best model and train on the training data
        best_bayes_model = lgb.LGBMClassifier(n_estimators=best_bayes_estimators,
                                              n_jobs = -1,
                                              objective = 'binary',
                                              random_state = 50,
                                              **best_bayes_params)
        best_bayes_model.fit(one_hot_features, labels)
```

```
In [ ]: # Evaluate on the testing data
        preds = best_bayes_model.predict_proba(one_hot_features_test)[:, 1]
        print('The best model from Bayes optimization scores {:.5f} AUC ROC on the test set.'.
        print('This was achieved after {} search iterations'.format(results.loc[0, 'iteration'
```

If we want to save the results, we can use the json file format. Saving the trials results will allow us to continue the hyperparameter search where we left off.

```
In [ ]: import json

        # Save the trial results
        with open('trials_kaggle.json', 'w') as f:
            f.write(json.dumps(trials_results))
```

4 Comparison to Random Search

Comparing the results to random seach both in numbers and figures can help us understand how Bayesian Optimization searches work. First, we can look at the best hyperparameters (as determined from the validation error) from both searches.

4.0.1 Optimal Hyperparameters

We can compare the "best" hyperparameters found from both search methods. It's interesting to compare the results because they suggest there may be multiple configurations that yield roughly the same validation error.

```
In [ ]: best_random_params['method'] = 'random search'
        best_random_params['iteration'] = random_results.loc[0, 'iteration']

        best_bayes_params['method'] = 'Bayesian optimization'
        best_bayes_params['iteration'] = random_results.loc[0, 'iteration']

        best_params = pd.DataFrame(best_bayes_params, index = [0]).append(pd.DataFrame(best_ra
                                                                          ignore_index = True,
        best_params
```

4.1 Visualizing Hyperparameters

One interesting thing we can do with the results is to see the different hyperparameters tried by both random search and the Tree Parzen Estimator. Since random search is choosing without regards to the previous results, we would expect that the distribution of samples should be close to the domain space we defined (it won't be exact since we are using a fairly small number of iterations). On the other hand, the Bayes Optimization, if given enough time, should concetrate on the "more promising" hyperparameters.

In addition to a more concentrated search, we expect that the average validation loss of the Bayesian Optimization should be lower than that on the random search because it chooses values likely (according to the probability model) to yield lower losses on the objective function. The validation loss should also decrease over time with the Bayesian method.

First we will need to extract the hyperparameters from both search methods. We will store these in separate dataframes.

```
In [ ]: # Create a new dataframe for storing parameters
        random_params = pd.DataFrame(columns = list(random_results.loc[0, 'params'].keys()),
                                     index = list(range(len(random_results))))

        # Add the results with each parameter a different column
```

```
        for i, params in enumerate(random_results['params']):
            random_params.loc[i, :] = list(params.values())

        random_params['loss'] = random_results['loss']
        random_params['iteration'] = random_results['iteration']
        random_params.head()
```

```
In [ ]: # Create a new dataframe for storing parameters
        bayes_params = pd.DataFrame(columns = list(ast.literal_eval(results.loc[0, 'params']).
                                    index = list(range(len(results))))

        # Add the results with each parameter a different column
        for i, params in enumerate(results['params']):
            bayes_params.loc[i, :] = list(ast.literal_eval(params).values())

        bayes_params['loss'] = results['loss']
        bayes_params['iteration'] = results['iteration']

        bayes_params.head()
```

Learning Rates The first plot shows the sampling distribution, random search, and Bayesian optimization learning rate distributions.

```
In [ ]: plt.figure(figsize = (20, 8))
        plt.rcParams['font.size'] = 18

        # Density plots of the learning rate distributions
        sns.kdeplot(learning_rate_dist, label = 'Sampling Distribution', linewidth = 4)
        sns.kdeplot(random_params['learning_rate'], label = 'Random Search', linewidth = 4)
        sns.kdeplot(bayes_params['learning_rate'], label = 'Bayes Optimization', linewidth = 4
        plt.vlines([best_random_params['learning_rate'], best_bayes_params['learning_rate']],
                   ymin = 0.0, ymax = 50.0, linestyles = '--', linewidth = 4, colors = ['orang
        plt.legend()
        plt.xlabel('Learning Rate'); plt.ylabel('Density'); plt.title('Learning Rate Distributi
```

Boosting Type Random search should use the boosting types with the same frequency. However, Bayesian Optimization might have decided (modeled) that one boosting type is better than another for this problem.

```
In [ ]: fig, axs = plt.subplots(1, 2, sharey = True, sharex = True)

        # Bar plots of boosting type
        random_params['boosting_type'].value_counts().plot.bar(ax = axs[0], figsize = (14, 6),
        bayes_params['boosting_type'].value_counts().plot.bar(ax = axs[1], figsize = (14, 6), c
```

```
In [ ]: print('Random Search boosting type percentages')
        100 * random_params['boosting_type'].value_counts() / len(random_params)
```

```
In [ ]: print('Bayes Optimization boosting type percentages')
        100 * bayes_params['boosting_type'].value_counts() / len(bayes_params)
```

4.1.1 Plots of All Numeric Hyperparameters

```
In [ ]: # Iterate through each hyperparameter
        for i, hyper in enumerate(random_params.columns):
            if hyper not in ['class_weight', 'boosting_type', 'iteration', 'subsample', 'metri
                plt.figure(figsize = (14, 6))
                # Plot the random search distribution and the bayes search distribution
                if hyper != 'loss':
                    sns.kdeplot([sample(space[hyper]) for _ in range(1000)], label = 'Sampling
                sns.kdeplot(random_params[hyper], label = 'Random Search', linewidth = 4)
                sns.kdeplot(bayes_params[hyper], label = 'Bayes Optimization', linewidth = 4)
                plt.vlines([best_random_params[hyper], best_bayes_params[hyper]],
                           ymin = 0.0, ymax = 10.0, linestyles = '--', linewidth = 4, colors
                plt.legend(loc = 1)
                plt.title('{} Distribution'.format(hyper))
                plt.xlabel('{}'.format(hyper)); plt.ylabel('Density');
                plt.show();
```

4.2 Evolution of Hyperparameters

We can also plot the hyperparameters over time (against the number of iterations) to see how they change for the Bayes Optimization. First we will map the `boosting_type` to an integer for plotting.

```
In [ ]: # Map boosting type to integer (essentially label encoding)
        bayes_params['boosting_int'] = bayes_params['boosting_type'].replace({'gbdt': 1, 'goss

        # Plot the boosting type over the search
        plt.plot(bayes_params['iteration'], bayes_params['boosting_int'], 'ro')
        plt.yticks([1, 2, 3], ['gdbt', 'goss', 'dart']);
        plt.xlabel('Iteration'); plt.title('Boosting Type over Search');
```

```
In [ ]: fig, axs = plt.subplots(1, 4, figsize = (24, 6))
        i = 0

        # Plot of four hyperparameters
        for i, hyper in enumerate(['colsample_bytree', 'learning_rate', 'min_child_samples', '

                # Scatterplot
                sns.regplot('iteration', hyper, data = bayes_params, ax = axs[i])
                axs[i].scatter(best_bayes_params['iteration'], best_bayes_params[hyper], marke
                axs[i].set(xlabel = 'Iteration', ylabel = '{}'.format(hyper), title = '{} over

        plt.tight_layout()
```

```
In [ ]: best_bayes_params
```

```
In [ ]: fig, axs = plt.subplots(1, 3, figsize = (18, 6))
        i = 0
```

```
        # Scatterplot of next three hyperparameters
        for i, hyper in enumerate(['reg_alpha', 'reg_lambda', 'subsample_for_bin']):
                sns.regplot('iteration', hyper, data = bayes_params, ax = axs[i])
                axs[i].scatter(best_bayes_params['iteration'], best_bayes_params[hyper], marke
                axs[i].set(xlabel = 'Iteration', ylabel = '{}'.format(hyper), title = '{} over

        plt.tight_layout()
```

Validation Losses Finally, we can look at the losses recorded by both random search and Bayes Optimization. We would expect the average loss recorded by Bayes Optimization to be lower because this method should spend more time in promising regions of the search space.

```
In [ ]: # Dataframe of just scores
        scores = pd.DataFrame({'ROC AUC': 1 - random_params['loss'], 'iteration': random_params
        scores = scores.append(pd.DataFrame({'ROC AUC': 1 - bayes_params['loss'], 'iteration':

        scores['ROC AUC'] = scores['ROC AUC'].astype(np.float32)
        scores['iteration'] = scores['iteration'].astype(np.int32)

        scores.head()
```

We can make histograms of the scores (not taking in account the iteration) on the same x-axis scale to see if there is a difference in scores.

```
In [ ]: plt.figure(figsize = (18, 6))

        # Random search scores
        plt.subplot(1, 2, 1)
        plt.hist(1 - random_results['loss'].astype(np.float64), label = 'Random Search', edgeco
        plt.xlabel("Validation ROC AUC"); plt.ylabel("Count"); plt.title("Random Search Validat
        plt.xlim(0.75, 0.78)

        # Bayes optimization scores
        plt.subplot(1, 2, 2)
        plt.hist(1 - bayes_params['loss'], label = 'Bayes Optimization', edgecolor = 'k');
        plt.xlabel("Validation ROC AUC"); plt.ylabel("Count"); plt.title("Bayes Optimization Va
        plt.xlim(0.75, 0.78);
```

It does appear that the validation ROC AUC for the Bayesian optimization is higher than that for Random Search. However, as we have seen, this does not necessarily translate to a better testing score!

Bayesian optimization should get better over time. Let's plot the scores against the iteration to see if there was improvement.

```
In [ ]: # Plot of scores over the course of searching
        sns.lmplot('iteration', 'ROC AUC', hue = 'search', data = scores, size = 8);
        plt.scatter(best_bayes_params['iteration'], 1 - best_bayes_params['loss'], marker = '*
        plt.scatter(best_random_params['iteration'], 1 - best_random_params['loss'], marker =
        plt.xlabel('Iteration'); plt.ylabel('ROC AUC'); plt.title("Validation ROC AUC versus It
```

It's reassuring to see that the validation ROC AUC scores of Bayesian optimization increase over time. What this shows is that the model is exploring hyperparameters that are better according to the cross validation metric! It would be interesting to continue searching and see if there is a plateau in the validation scores (there would have to be eventually). Moreover, even if validation scores continue to increase, that does not mean a better model for the testing data!

```
In [ ]: # Save dataframes of parameters
        bayes_params.to_csv('results/bayes_params_kaggle.csv', index = False)
        random_params.to_csv('results/random_params_kaggle.csv', index = False)
```

4.3 Continue Searching

We can keep running the Bayesian hyperparameter search for more iterations to try for better results. Hyperopt will continue searching where it left off if we pass it a trials object that already has information on previous runs. This raises a good point: always save your previous results, because you never know when they will be useful!

Another interesting point to not is that Bayesian Optimization methods do not have any internal state which means all they need are the results: previous inputs to the objective function and the resulting loss. Based only on these results, these methods can construct a surrogate function and suggest the next set of hyperparameters to evaluate. The internals of the objective function have no effect on the Bayesian Optimization method hence the naming of this as a black box optimization method.

```
In [ ]: # Continue training
        ITERATION = MAX_EVALS + 1

        # Set more evaluations
        MAX_EVALS = 1000
```

```
In [ ]: %%capture

        # Use the same trials object to keep training
        best = fmin(fn = objective, space = space, algo = tpe.suggest,
                    max_evals = MAX_EVALS, trials = bayes_trials, verbose = 1,
                    rstate = np.random.RandomState(50))
```

```
In [ ]: # Sort the trials with lowest loss (highest AUC) first
        bayes_trials_results = sorted(bayes_trials.results, key = lambda x: x['loss'])
        bayes_trials_results[:2]
```

```
In [ ]: results = pd.read_csv('results/gbm_trials_kaggle.csv')

        # Sort values with best on top and reset index for slicing
        results.sort_values('loss', ascending = True, inplace = True)
        results.reset_index(inplace = True, drop = True)
        results.head()
```

```
In [ ]: # Extract the ideal number of estimators and hyperparameters
        best_bayes_estimators = int(results.loc[0, 'estimators'])
```

```
        best_bayes_params = ast.literal_eval(results.loc[0, 'params']).copy()

        # Re-create the best model and train on the training data
        best_bayes_model = lgb.LGBMClassifier(n_estimators=best_bayes_estimators, n_jobs = -1,
                                              objective = 'binary', random_state = 50, **best_
        best_bayes_model.fit(features, labels)
```

```
In [ ]: # Evaluate on the testing data
        preds = best_bayes_model.predict_proba(test_features)[:, 1]
        print('The best model from Bayes optimization scores {:.5f} AUC ROC on the test set.'.
        print('This was achieved after {} search iterations'.format(results.loc[0, 'iteration'
```

The continuation of the search did slightly improve the validation score (again depending on training run). Instead of training more, we might want to restart the search so the algorithm can spend more time exploring the domain space. As searching continues, the algorithm shifts from exploring (trying new values) to exploiting (trying those values that worked best in the past). This is generally what we want unless the model gets stuck in a local minimum at which point we would want to restart the search in a different region of the hyperparameter space. Bayesian Optimization of hyperparameters is still prone to overfitting, even when using cross-validation because it can get settle into a local minimum of the objective function. It is very difficult to tell when this occurs for a high-dimensional problem!

5 Conclusions

In this notebook, we saw how to implement automated hyperparameter tuning with Bayesian Optimization methods. We used the open-source Python library Hyperopt with the Tree Parzen Estimator to optimize the hyperparameters of a gradient boosting machine.

Bayesian model-based optimization can be more efficient than random search, finding a better set of model hyperparameters in fewer search iterations (although not in every case). However, just because the model hyperparameters are better on the validation set does not mean they are better for the testing set! For this training run, Bayesian Optimization found a better set of hyperparamters according to the validation and the test data although the testing score was much lower than the validation ROC AUC. This is a useful lesson that even when using cross-validation, overfitting is still one of the top problems in machine learning.

Bayesian optimization is a powerful technique that we can use to tune any machine learning model, so long as we can define an objective function that returns a value to minimize and a domain space over which to search. This can extend to any function that we want to minimize (not just hyperparameter tuning). Bayesian optimization can be a significant upgrade over uninformed methods such as random search and because of the ease of use in Python are now a good option to use for hyperparameter tuning. As with most subjects in machine learning, there is no single best answer for hyperparameter tuning, but Bayesian optimization methods should be a tool that helps data scientists with the tedious but necessary task of model tuning!