04_xgboost_lightgbm_catboost_tuning

September 29, 2021

1 XGBoost, LightGBM and CatBoost Parameter Tuning

1.1 Imports & Settings

```
[ ]: import warnings
     from pathlib import Path
     from random import shuffle
     from time import time
     import numpy as np
     import pandas as pd
     import xgboost as xgb
     from xgboost.callback import reset_learning_rate
     import lightgbm as lgb
     from catboost import Pool, CatBoostClassifier
     from itertools import product
     from sklearn.metrics import roc_auc_score
     from math import ceil
```

```
[ ]: pd.set_option('display.expand_frame_repr', False)
     warnings.filterwarnings('ignore')
     idx = pd.IndexSlice
     np.random.seed(42)
```

Change path to data store in `gbm_utils.py` if needed

If you choose to compile any of the libraries with GPU support, amend the parameters in `gbm_params.py` accordingly.

```
[ ]: results_path = Path('results')
     if not results_path.exists():
         results_path.mkdir(exist_ok=True)
```

```
[ ]: from gbm_utils import format_time, get_data, get_one_hot_data, factorize_cats,␣
      ↪get_holdout_set, OneStepTimeSeriesSplit
     from gbm_params import get_params
```

1.2 Learning Rate Schedule

Define declining learning rate schedule:

```
[ ]: def learning_rate(n, ntot):
         start_eta = 0.1
         k = 8 / ntot
         x0 = ntot / 1.8
         return start_eta * (1 - 1 / (1 + np.exp(-k * (n - x0))))
```

1.2.1 Visualize Learning Rate Schedule

```
[ ]: ntot = 10000
     x = np.asarray(range(ntot))
     pd.Series(learning_rate(x, ntot)).plot();
```

1.3 Cross-Validate GBM Model

1.3.1 CV Settings

```
[ ]: GBM = 'lightgbm'
     HOLDOUT = True
     FACTORS = True
     n_splits = 12

     result_key = f"/{GBM}/{'factors' if FACTORS else 'dummies'}/results/2"
```

1.3.2 Create Binary Datasets

All libraries have their own data format to precompute feature statistics to accelerate the search for split points, as described previously. These can also be persisted to accelerate the start of subsequent training.

The following code constructs binary train and validation datasets for each model to be used with the OneStepTimeSeriesSplit.

The available options vary slightly: : - xgboost allows the use of all available threads - lightgbm explicitly aligns the quantiles that are created for the validation set with the training set - The catboost implementation needs feature columns identified using indices rather than labels

```
[ ]: def get_datasets(features, target, kfold, model='xgboost'):
         cat_cols = ['year', 'month', 'age', 'msize', 'sector']
         data = {}
         for fold, (train_idx, test_idx) in enumerate(kfold.split(features)):
             print(fold, end=' ', flush=True)
             if model == 'xgboost':
                 data[fold] = {'train': xgb.DMatrix(label=target.iloc[train_idx],
                                                    data=features.iloc[train_idx],
                                                    nthread=-1),                   ␣
      ↪# use avail. threads
                               'valid': xgb.DMatrix(label=target.iloc[test_idx],
                                                    data=features.iloc[test_idx],
                                                    nthread=-1)}
             elif model == 'lightgbm':
                 train = lgb.Dataset(label=target.iloc[train_idx],
                                     data=features.iloc[train_idx],
                                     categorical_feature=cat_cols,
                                     free_raw_data=False)

                 # align validation set histograms with training set
                 valid = train.create_valid(label=target.iloc[test_idx],
                                            data=features.iloc[test_idx])

                 data[fold] = {'train': train.construct(),
                               'valid': valid.construct()}

             elif model == 'catboost':
                 # get categorical feature indices
                 cat_cols_idx = [features.columns.get_loc(c) for c in cat_cols]
                 data[fold] = {'train': Pool(label=target.iloc[train_idx],
                                             data=features.iloc[train_idx],
                                             cat_features=cat_cols_idx),

                               'valid': Pool(label=target.iloc[test_idx],
                                             data=features.iloc[test_idx],
                                             cat_features=cat_cols_idx)}
         return data
```

1.3.3 Get Data

[ ]:
```
y, features = get_data()
if FACTORS:
    X = factorize_cats(features)
else:
    X = get_one_hot_data(features)

if HOLDOUT:
    y, X, y_test, X_test = get_holdout_set(target=y,
                                           features=X)

    with pd.HDFStore('model_tuning.h5') as store:
        key = f'{GBM}/holdout/'
        if not any([k for k in store.keys() if k[1:].startswith(key)]):
            store.put(key + 'features', X_test, format='t' if FACTORS else 'f')
            store.put(key + 'target', y_test)

cv = OneStepTimeSeriesSplit(n_splits=n_splits)

datasets = get_datasets(features=X, target=y, kfold=cv, model=GBM)
```

1.3.4 Define Parameter Grid

The numerous hyperparameters are listed in `gbm_params.py`. Each library has parameter settings to: - specify the overall objectives and learning algorithm - design the base learners - apply various regularization techniques - handle early stopping during training - enabling the use of GPU or parallelization on CPU

The documentation for each library details the various parameters that may refer to the same concept, but which have different names across libraries. This site highlights the corresponding parameters for xgboost and lightgbm.

To explore the hyperparameter space, we specify values for key parameters that we would like to test in combination. The sklearn library supports RandomizedSearchCV to cross-validate a subset of parameter combinations that are sampled randomly from specified distributions.

We will implement a custom version that allows us to leverage early stopping while monitoring the current best-performing combinations so we can abort the search process once satisfied with the result rather than specifying a set number of iterations beforehand.

To this end, we specify a parameter grid according to each library's parameters as before, generate all combinations using the built-in Cartesian product generator provided by the itertools library, and randomly shuffle the result.

In the case of LightGBM, we automatically set `max_depth` as a function of the current num_leaves value, as shown in the following code:

[ ]:
```
param_grid = dict(
        # common options
        learning_rate=[.01, .1, .3],
        # max_depth=list(range(3, 14, 2)),
        colsample_bytree=[.8, 1],  # except catboost

        # lightgbm
        # max_bin=[32, 128],
        num_leaves=[2 ** i for i in range(9, 14)],
        boosting=['gbdt', 'dart'],
        min_gain_to_split=[0, 1, 5],  # not supported on GPU

        # xgboost
        # booster=['gbtree', 'dart'],
        # gamma=[0, 1, 5],

        # catboost
        # one_hot_max_size=[None, 2],
        # max_ctr_complexity=[1, 2, 3],
        # random_strength=[None, 1],
        # colsample_bylevel=[.6, .8, 1]
)
```

```
[ ]: all_params = list(product(*param_grid.values()))
     n_models = len(all_params)
     shuffle(all_params)

     print('\n# Models:', n_models)
```

1.3.5 Run Cross Validation

The following function `run_cv()` implements cross-validation using the library-specific commands. The `train()` method also produces validation scores that are stored in the `scores` dictionary.

When early stopping takes effect, the last iteration is also the best score.

```
[ ]: def run_cv(test_params, data, n_splits=10, gb_machine='xgboost'):
         """Train-Validate with early stopping"""
         result = []
         cols = ['rounds', 'train', 'valid']
         for fold in range(n_splits):
             train = data[fold]['train']
             valid = data[fold]['valid']

             scores = {}
             if gb_machine == 'xgboost':
                 model = xgb.train(params=test_params,
                                   dtrain=train,
                                   evals=list(zip([train, valid], ['train',␣
      ↪'valid'])),
                                   verbose_eval=50,
                                   num_boost_round=250,
                                   early_stopping_rounds=25,
                                   evals_result=scores)

                 result.append([model.best_iteration,
                                scores['train']['auc'][-1],
                                scores['valid']['auc'][-1]])
             elif gb_machine == 'lightgbm':
                 model = lgb.train(params=test_params,
                                   train_set=train,
                                   valid_sets=[train, valid],
                                   valid_names=['train', 'valid'],
                                   num_boost_round=250,
                                   early_stopping_rounds=25,
                                   verbose_eval=50,
                                   evals_result=scores)

                 result.append([model.current_iteration(),
                                scores['train']['auc'][-1],
                                scores['valid']['auc'][-1]])
```

```
        elif gb_machine == 'catboost':
            model = CatBoostClassifier(**test_params)
            model.fit(X=train,
                      eval_set=[valid],
                      logging_level='Silent')

            train_score = model.predict_proba(train)[:, 1]
            valid_score = model.predict_proba(valid)[:, 1]
            result.append([
                model.tree_count_,
                roc_auc_score(y_score=train_score, y_true=train.get_label()),
                roc_auc_score(y_score=valid_score, y_true=valid.get_label())
            ])

    df = pd.DataFrame(result, columns=cols)
    return (df
            .mean()
            .append(df.std().rename({c: c + '_std' for c in cols}))
            .append(pd.Series(test_params)))
```

The following code executes and exhaustive search over the parameter grid. The algorithms are already multithreaded so GridSearchCV does not add parallelization benefits. The below 'manual' implementation allows for more transparency during execution.

[ ]:
```
results = pd.DataFrame()

start = time()
for n, test_param in enumerate(all_params, 1):
    iteration = time()

    cv_params = get_params(GBM)
    cv_params.update(dict(zip(param_grid.keys(), test_param)))
    if GBM == 'lightgbm':
        cv_params['max_depth'] = int(ceil(np.log2(cv_params['num_leaves'])))

    results[n] = run_cv(test_params=cv_params,
                        data=datasets,
                        n_splits=n_splits,
                        gb_machine=GBM)
    results.loc['time', n] = time() - iteration

    if n > 1:
        df = results[~results.eq(results.iloc[:, 0], axis=0).all(1)].T
        if 'valid' in df.columns:
            df.valid = pd.to_numeric(df.valid)
            print('\n')
            print(df.sort_values('valid', ascending=False).head(5).
↪reset_index(drop=True))

    out = f'\n\tModel: {n} of {n_models} | '
    out += f'{format_time(time() - iteration)} | '
    out += f'Total: {format_time(time() - start)} | '
    print(out + f'Remaining: {format_time((time() - start)/n*(n_models-n))}\n')

    with pd.HDFStore('model_tuning.h5') as store:
        store.put(result_key, results.T.apply(pd.to_numeric, errors='ignore'))
```