01_boosting_baseline

September 29, 2021

1 Adaptive and Gradient Boosting

In this notebook, we demonstrate the use of AdaBoost and gradient boosting, incuding several state-of-the-art implementations of this very powerful and flexible algorithm that greatly speed up training.

We use the stock return dataset with a few engineered factors created in Chapter 4 on Alpha Factor Research in the notebook feature_engineering.

1.1 Update

This notebook now uses `sklearn.ensemble.HistGradientBoostingClassifier`.

1.2 Imports and Settings

[1]:
```
%matplotlib inline

import sys, os
import warnings
from time import time
from itertools import product
import joblib
from pathlib import Path
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns

from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from catboost import CatBoostClassifier
from sklearn.model_selection import cross_validate
from sklearn.dummy import DummyClassifier
from sklearn.tree import DecisionTreeClassifier
# needed for HistGradientBoostingClassifier
from sklearn.experimental import enable_hist_gradient_boosting
```

```
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier,␣
↪HistGradientBoostingClassifier
from sklearn.inspection import partial_dependence, plot_partial_dependence
from sklearn.metrics import roc_auc_score
```

[2]:
```
sys.path.insert(1, os.path.join(sys.path[0], '..'))
from utils import format_time
```

[3]:
```
results_path = Path('results', 'baseline')
if not results_path.exists():
    results_path.mkdir(exist_ok=True, parents=True)
```

[4]:
```
warnings.filterwarnings('ignore')
sns.set_style("whitegrid")
idx = pd.IndexSlice
np.random.seed(42)
```

1.3 Prepare Data

1.3.1 Get source

We use the `engineered_features` dataset created in Chapter 4, Alpha Factor Research

Set data store location:

[5]:
```
DATA_STORE = '../data/assets.h5'
```

[6]:
```
def get_data(start='2000', end='2018', task='classification', holding_period=1,␣
↪dropna=False):

    idx = pd.IndexSlice
    target = f'target_{holding_period}m'
    with pd.HDFStore(DATA_STORE) as store:
        df = store['engineered_features']

    if start is not None and end is not None:
        df = df.loc[idx[:, start: end], :]
    if dropna:
        df = df.dropna()

    y = (df[target]>0).astype(int)
    X = df.drop([c for c in df.columns if c.startswith('target')], axis=1)
    return y, X
```

1.3.2 Factorize Categories

Define columns with categorical data:

[7]:
```python
cat_cols = ['year', 'month', 'age', 'msize', 'sector']
```

Integer-encode categorical columns:

[8]:
```python
def factorize_cats(df, cats=['sector']):
    cat_cols = ['year', 'month', 'age', 'msize'] + cats
    for cat in cats:
        df[cat] = pd.factorize(df[cat])[0]
    df.loc[:, cat_cols] = df.loc[:, cat_cols].fillna(-1).astype(int)
    return df
```

1.3.3 One-Hot Encoding

Create dummy variables from categorical columns if needed:

[9]:
```python
def get_one_hot_data(df, cols=cat_cols[:-1]):
    df = pd.get_dummies(df,
                        columns=cols + ['sector'],
                        prefix=cols + [''],
                        prefix_sep=['_'] * len(cols) + [''])
    return df.rename(columns={c: c.replace('.0', '') for c in df.columns})
```

1.3.4 Get Holdout Set

Create holdout test set to estimate generalization error after cross-validation:

[10]:
```python
def get_holdout_set(target, features, period=6):
    idx = pd.IndexSlice
    label = target.name
    dates = np.sort(y.index.get_level_values('date').unique())
    cv_start, cv_end = dates[0], dates[-period - 2]
    holdout_start, holdout_end = dates[-period - 1], dates[-1]

    df = features.join(target.to_frame())
    train = df.loc[idx[:, cv_start: cv_end], :]
    y_train, X_train = train[label], train.drop(label, axis=1)

    test = df.loc[idx[:, holdout_start: holdout_end], :]
    y_test, X_test = test[label], test.drop(label, axis=1)
    return y_train, X_train, y_test, X_test
```

1.4 Load Data

The algorithms in this chapter use a dataset generated in Chapter 4 on Alpha Factor Research in the notebook feature-engineering that needs to be executed first.

[11]:
```python
y, features = get_data()
X_dummies = get_one_hot_data(features)
```

```
X_factors = factorize_cats(features)
```

[12]:
```
X_factors.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 358914 entries, ('A', Timestamp('2001-01-31 00:00:00')) to ('ZUMZ',
Timestamp('2018-02-28 00:00:00'))
Data columns (total 28 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   return_1m       358914 non-null  float64
 1   return_2m       358914 non-null  float64
 2   return_3m       358914 non-null  float64
 3   return_6m       358914 non-null  float64
 4   return_9m       358914 non-null  float64
 5   return_12m      358914 non-null  float64
 6   Mkt-RF          358914 non-null  float64
 7   SMB             358914 non-null  float64
 8   HML             358914 non-null  float64
 9   RMW             358914 non-null  float64
 10  CMA             358914 non-null  float64
 11  momentum_2      358914 non-null  float64
 12  momentum_3      358914 non-null  float64
 13  momentum_6      358914 non-null  float64
 14  momentum_9      358914 non-null  float64
 15  momentum_12     358914 non-null  float64
 16  momentum_3_12   358914 non-null  float64
 17  year            358914 non-null  int64
 18  month           358914 non-null  int64
 19  return_1m_t-1   357076 non-null  float64
 20  return_1m_t-2   355238 non-null  float64
 21  return_1m_t-3   353400 non-null  float64
 22  return_1m_t-4   351562 non-null  float64
 23  return_1m_t-5   349724 non-null  float64
 24  return_1m_t-6   347886 non-null  float64
 25  age             358914 non-null  int64
 26  msize           358914 non-null  int64
 27  sector          358914 non-null  int64
dtypes: float64(23), int64(5)
memory usage: 78.1+ MB
```

[13]:
```
y_clean, features_clean = get_data(dropna=True)
X_dummies_clean = get_one_hot_data(features_clean)
X_factors_clean = factorize_cats(features_clean)
```

1.5 Cross-Validation Setup

1.5.1 Custom Time Series KFold Generator

Custom Time Series KFold generator.

[14]:
```python
class OneStepTimeSeriesSplit:
    """Generates tuples of train_idx, test_idx pairs
    Assumes the index contains a level labeled 'date'"""

    def __init__(self, n_splits=3, test_period_length=1, shuffle=False):
        self.n_splits = n_splits
        self.test_period_length = test_period_length
        self.shuffle = shuffle

    @staticmethod
    def chunks(l, n):
        for i in range(0, len(l), n):
            yield l[i:i + n]

    def split(self, X, y=None, groups=None):
        unique_dates = (X.index
                        .get_level_values('date')
                        .unique()
                        .sort_values(ascending=False)
                        [:self.n_splits*self.test_period_length])

        dates = X.reset_index()[['date']]
        for test_date in self.chunks(unique_dates, self.test_period_length):
            train_idx = dates[dates.date < min(test_date)].index
            test_idx = dates[dates.date.isin(test_date)].index
            if self.shuffle:
                np.random.shuffle(list(train_idx))
            yield train_idx, test_idx

    def get_n_splits(self, X, y, groups=None):
        return self.n_splits
```

[15]:
```python
cv = OneStepTimeSeriesSplit(n_splits=12,
                            test_period_length=1,
                            shuffle=False)
```

[16]:
```python
run_time = {}
```

1.5.2 CV Metrics

Define some metrics for use with cross-validation:

```
[17]: metrics = {'balanced_accuracy': 'Accuracy' ,
                 'roc_auc': 'AUC',
                 'neg_log_loss': 'Log Loss',
                 'f1_weighted': 'F1',
                 'precision_weighted': 'Precision',
                 'recall_weighted': 'Recall'
      }
```

Helper function that runs cross-validation for the various algorithms.

```
[18]: def run_cv(clf, X=X_dummies, y=y, metrics=metrics, cv=cv, fit_params=None,␣
       ↪n_jobs=-1):
          start = time()
          scores = cross_validate(estimator=clf,
                                  X=X,
                                  y=y,
                                  scoring=list(metrics.keys()),
                                  cv=cv,
                                  return_train_score=True,
                                  n_jobs=n_jobs,
                                  verbose=1,
                                  fit_params=fit_params)
          duration = time() - start
          return scores, duration
```

1.5.3 CV Result Handler Functions

The following helper functions manipulate and plot the cross-validation results to produce the outputs below.

```
[19]: def stack_results(scores):
          columns = pd.MultiIndex.from_tuples(
              [tuple(m.split('_', 1)) for m in scores.keys()],
              names=['Dataset', 'Metric'])
          data = np.array(list(scores.values())).T
          df = (pd.DataFrame(data=data,
                             columns=columns)
                .iloc[:, 2:])
          results = pd.melt(df, value_name='Value')
          results.Metric = results.Metric.apply(lambda x: metrics.get(x))
          results.Dataset = results.Dataset.str.capitalize()
          return results
```

```
[20]: def plot_result(df, model=None, fname=None):
          m = list(metrics.values())
          g = sns.catplot(x='Dataset',
                          y='Value',
```

```
                    hue='Dataset',
                    col='Metric',
                    data=df,
                    col_order=m,
                    order=['Train', 'Test'],
                    kind="box",
                    col_wrap=3,
                    sharey=False,
                    height=4, aspect=1.2)
    df = df.groupby(['Metric', 'Dataset']).Value.mean().unstack().loc[m]
    for i, ax in enumerate(g.axes.flat):
        s = f"Train: {df.loc[m[i], 'Train']:>7.4f}\nTest:  {df.loc[m[i],␣
↪'Test'] :>7.4f}"
        ax.text(0.05, 0.85, s, fontsize=10, transform=ax.transAxes,
                bbox=dict(facecolor='white', edgecolor='grey',␣
↪boxstyle='round,pad=0.5'))
    g.fig.suptitle(model, fontsize=16)
    g.fig.subplots_adjust(top=.9)
    if fname:
        g.savefig(fname, dpi=300);
```

1.6 Baseline Classifier

`sklearn` provides the DummyClassifier that makes predictions using simple rule and is useful as a simple baseline to compare with the other (real) classifiers we use below.

The `stratified` rule generates predictions based on the training set's class distribution, i.e. always predicts the most frequent class.

[21]:
```
dummy_clf = DummyClassifier(strategy='stratified',
                            random_state=42)
```

[22]:
```
algo = 'dummy_clf'
```

[23]:
```
fname = results_path / f'{algo}.joblib'
if not Path(fname).exists():
    dummy_cv_result, run_time[algo] = run_cv(dummy_clf)
    joblib.dump(dummy_cv_result, fname)
else:
    dummy_cv_result = joblib.load(fname)
```

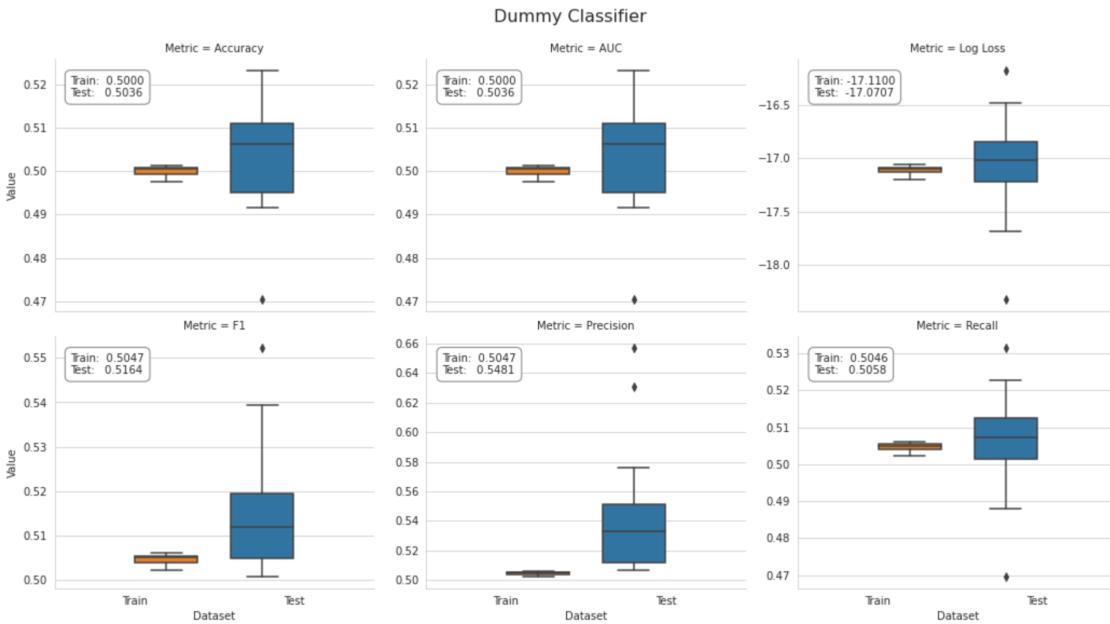
Unsurprisingly, it produces results near the AUC threshold for arbitrary predictions of 0.5:

[24]:
```
dummy_result = stack_results(dummy_cv_result)
dummy_result.groupby(['Metric', 'Dataset']).Value.mean().unstack()
```

[24]:
```
Dataset          Test      Train
Metric
```

```
AUC          0.503582   0.500008
Accuracy     0.503582   0.500008
F1           0.516424   0.504660
Log Loss   -17.070718 -17.109954
Precision    0.548116   0.504698
Recall       0.505758   0.504622
```

[25]:
```
plot_result(dummy_result, model='Dummy Classifier')
```



1.7 RandomForest

For comparison, we train a `RandomForestClassifier` as presented in Chapter 11 on Decision Trees and Random Forests.

1.7.1 Configure

[26]:
```
rf_clf = RandomForestClassifier(n_estimators=100,
                                criterion='gini',
                                max_depth=None,
                                min_samples_split=2,
                                min_samples_leaf=1,
                                min_weight_fraction_leaf=0.0,
                                max_features='auto',
                                max_leaf_nodes=None,
                                min_impurity_decrease=0.0,
                                min_impurity_split=None,
                                bootstrap=True,
```

```
                                  oob_score=True,
                                  n_jobs=-1,
                                  random_state=42,
                                  verbose=1)
```

1.7.2 Cross-validate

[27]:
```
algo = 'random_forest'
```

[28]:
```
fname = results_path / f'{algo}.joblib'
if not Path(fname).exists():
    rf_cv_result, run_time[algo] = run_cv(rf_clf, y=y_clean, X=X_dummies_clean)
    joblib.dump(rf_cv_result, fname)
else:
    rf_cv_result = joblib.load(fname)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 out of  12 | elapsed:  8.1min remaining:  1.6min
[Parallel(n_jobs=-1)]: Done  12 out of  12 | elapsed:  8.1min finished
```

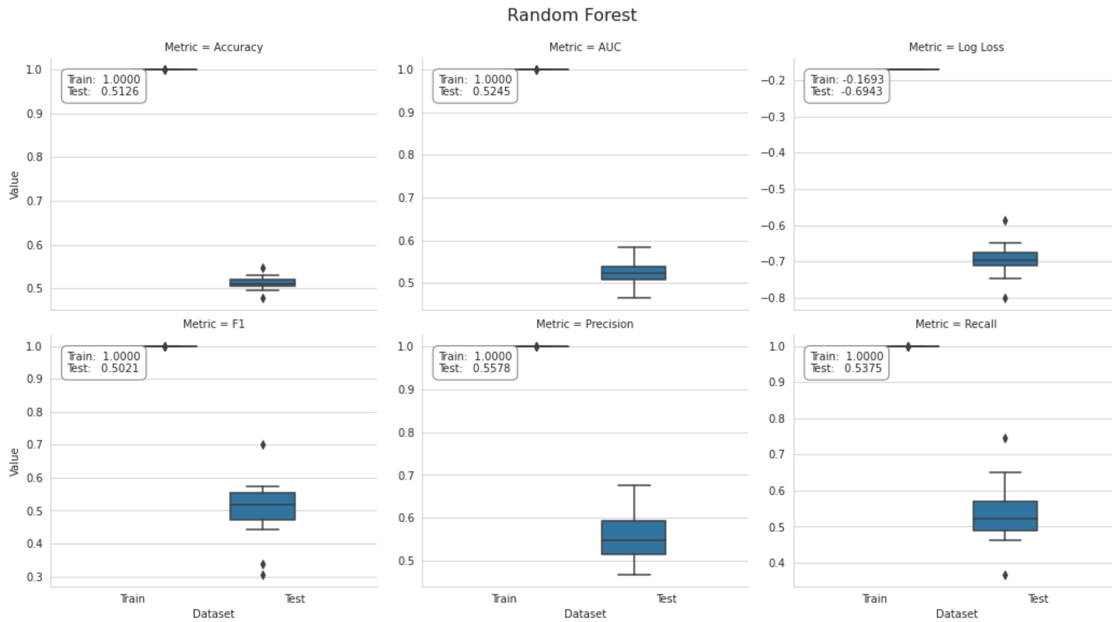
1.7.3 Plot Results

[29]:
```
rf_result = stack_results(rf_cv_result)
rf_result.groupby(['Metric', 'Dataset']).Value.mean().unstack()
```

[29]:

Dataset Metric	Test	Train
AUC	0.524482	1.000000
Accuracy	0.512583	0.999999
F1	0.502093	0.999999
Log Loss	-0.694309	-0.169309
Precision	0.557773	0.999999
Recall	0.537495	0.999999

[30]:
```
plot_result(rf_result, model='Random Forest')
```



1.8 scikit-learn: AdaBoost

As part of its ensemble module, sklearn provides an AdaBoostClassifier implementation that supports two or more classes. The code examples for this section are in the notebook gbm_baseline that compares the performance of various algorithms with a dummy classifier that always predicts the most frequent class.

1.8.1 Base Estimator

We need to first define a base_estimator as a template for all ensemble members and then configure the ensemble itself. We'll use the default DecisionTreeClassifier with max_depth=1—that is, a stump with a single split. The complexity of the base_estimator is a key tuning parameter because it depends on the nature of the data.

As demonstrated in the previous chapter, changes to `max_depth` should be combined with appropriate regularization constraints using adjustments to, for example, `min_samples_split`:

[31]:
```
base_estimator = DecisionTreeClassifier(criterion='gini',
                                        splitter='best',
                                        max_depth=1,
                                        min_samples_split=2,
                                        min_samples_leaf=20,
                                        min_weight_fraction_leaf=0.0,
                                        max_features=None,
                                        random_state=None,
                                        max_leaf_nodes=None,
                                        min_impurity_decrease=0.0,
                                        min_impurity_split=None,
```

```
                              class_weight=None)
```

1.8.2 AdaBoost Configuration

In the second step, we'll design the ensemble. The n_estimators parameter controls the number of weak learners and the learning_rate determines the contribution of each weak learner, as shown in the following code. By default, weak learners are decision tree stumps:

[32]:
```
ada_clf = AdaBoostClassifier(base_estimator=base_estimator,
                             n_estimators=100,
                             learning_rate=1.0,
                             algorithm='SAMME.R',
                             random_state=42)
```

The main tuning parameters that are responsible for good results are `n_estimators` and the base estimator complexity because the depth of the tree controls the extent of the interaction among the features.

1.8.3 Cross-validate

We will cross-validate the AdaBoost ensemble using a custom 12-fold rolling time-series split to predict 1 month ahead for the last 12 months in the sample, using all available prior data for training, as shown in the following code:

[33]:
```
algo = 'adaboost'
```

[34]:
```
fname = results_path / f'{algo}.joblib'
if not Path(fname).exists():
    ada_cv_result, run_time[algo] = run_cv(ada_clf, y=y_clean,␣
 ↪X=X_dummies_clean)
    joblib.dump(ada_cv_result, fname)
else:
    ada_cv_result = joblib.load(fname)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 out of  12 | elapsed:  6.4min remaining:  1.3min
[Parallel(n_jobs=-1)]: Done  12 out of  12 | elapsed:  6.4min finished
```

1.8.4 Plot Result

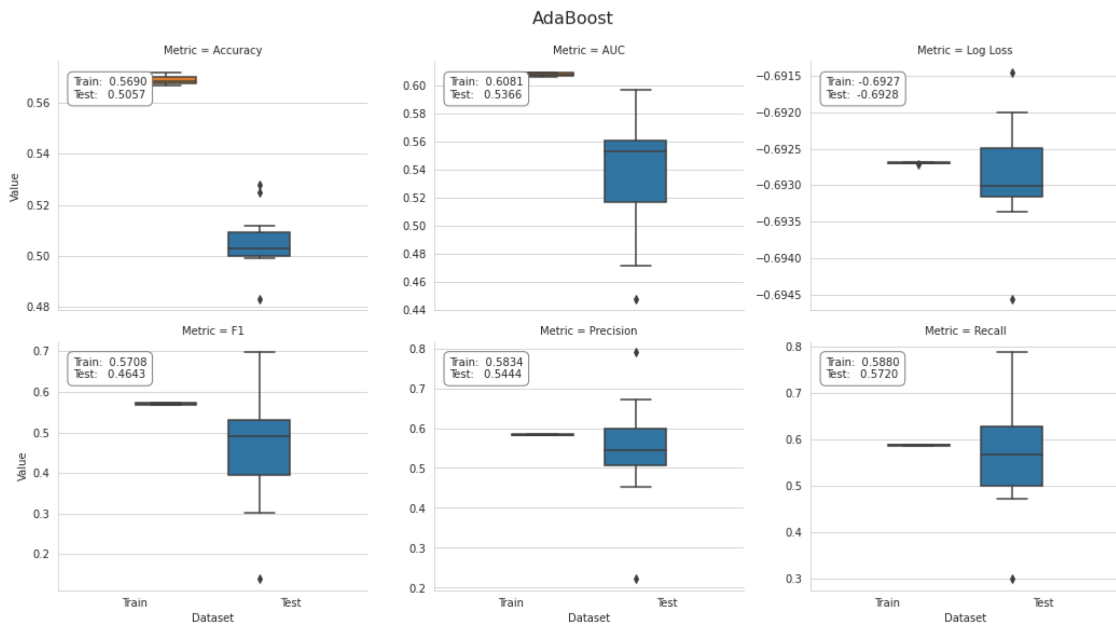
[35]:
```
ada_result = stack_results(ada_cv_result)
ada_result.groupby(['Metric', 'Dataset']).Value.mean().unstack()
```

[35]:

Dataset	Test	Train
Metric		
AUC	0.536567	0.608133
Accuracy	0.505709	0.569019
F1	0.464293	0.570756

```
Log Loss   -0.692850 -0.692697
Precision   0.544433  0.583411
Recall      0.571999  0.588008
```

[36]:
```python
plot_result(ada_result, model='AdaBoost')
```



1.9 scikit-learn: HistGradientBoostingClassifier

The ensemble module of sklearn contains an implementation of gradient boosting trees for regression and classification, both binary and multiclass.

1.9.1 Configure

The following HistGradientBoostingClassifier initialization code illustrates the key tuning parameters that we previously introduced, in addition to those that we are familiar with from looking at standalone decision tree models.

This estimator is much faster than GradientBoostingClassifier for big datasets (n_samples >= 10 000).

This estimator has native support for missing values (NaNs). During training, the tree grower learns at each split point whether samples with missing values should go to the left or right child, based on the potential gain. When predicting, samples with missing values are assigned to the left or right child consequently. If no missing values were encountered for a given feature during training, then samples with missing values are mapped to whichever child has the most samples.

[37]:
```python
gb_clf = HistGradientBoostingClassifier(loss='binary_crossentropy',
```

```
                                        learning_rate=0.1,          # regulates␣
 ↪the contribution of each tree
                                        max_iter=100,               # number of␣
 ↪boosting stages
                                        min_samples_leaf=20,
                                        max_depth=None,
                                        random_state=None,
                                        max_leaf_nodes=31,           # opt␣
 ↪value depends on feature interaction
                                        warm_start=False,
#                                         early_stopping=True,
#                                         scoring='loss',
#                                         validation_fraction=0.1,
#                                         n_iter_no_change=None,
                                        verbose=0,
                                        tol=0.0001)
```

1.9.2 Cross-validate

[38]:
```
algo = 'sklearn_gbm'
```

[39]:
```
fname = results_path / f'{algo}.joblib'
if not Path(fname).exists():
    gb_cv_result, run_time[algo] = run_cv(gb_clf, y=y_clean, X=X_dummies_clean)
    joblib.dump(gb_cv_result, fname)
else:
    gb_cv_result = joblib.load(fname)
```

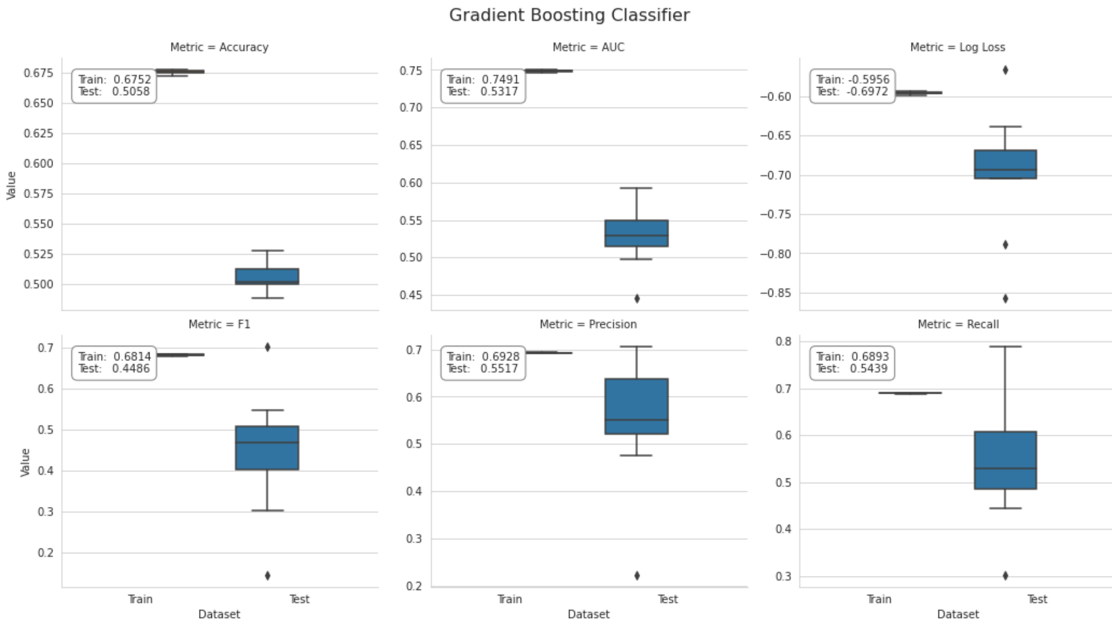
```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 out of  12 | elapsed:   52.5s remaining:   10.5s
[Parallel(n_jobs=-1)]: Done  12 out of  12 | elapsed:   53.5s finished
```

1.9.3 Plot Results

[40]:
```
gb_result = stack_results(gb_cv_result)
gb_result.groupby(['Metric', 'Dataset']).Value.mean().unstack()
```

[40]:
```
Dataset        Test     Train
Metric
AUC        0.531743  0.749078
Accuracy   0.505791  0.675196
F1         0.448612  0.681376
Log Loss  -0.697194 -0.595612
Precision  0.551708  0.692833
Recall     0.543934  0.689301
```

[41]:
```
plot_result(gb_result, model='Gradient Boosting Classifier')
```



1.9.4 Partial Dependence Plots

Drop time periods to avoid over-reliance for in-sample fit.

[42]:
```
X_ = X_factors_clean.drop(['year', 'month'], axis=1)
```

[43]:
```
fname = results_path / f'{algo}_model.joblib'
if not Path(fname).exists():
    gb_clf.fit(y=y_clean, X=X_)
    joblib.dump(gb_clf, fname)
else:
    gb_clf = joblib.load(fname)
```

[44]:
```
# mean accuracy
gb_clf.score(X=X_, y=y_clean)
```

[44]: 0.5826965098819537

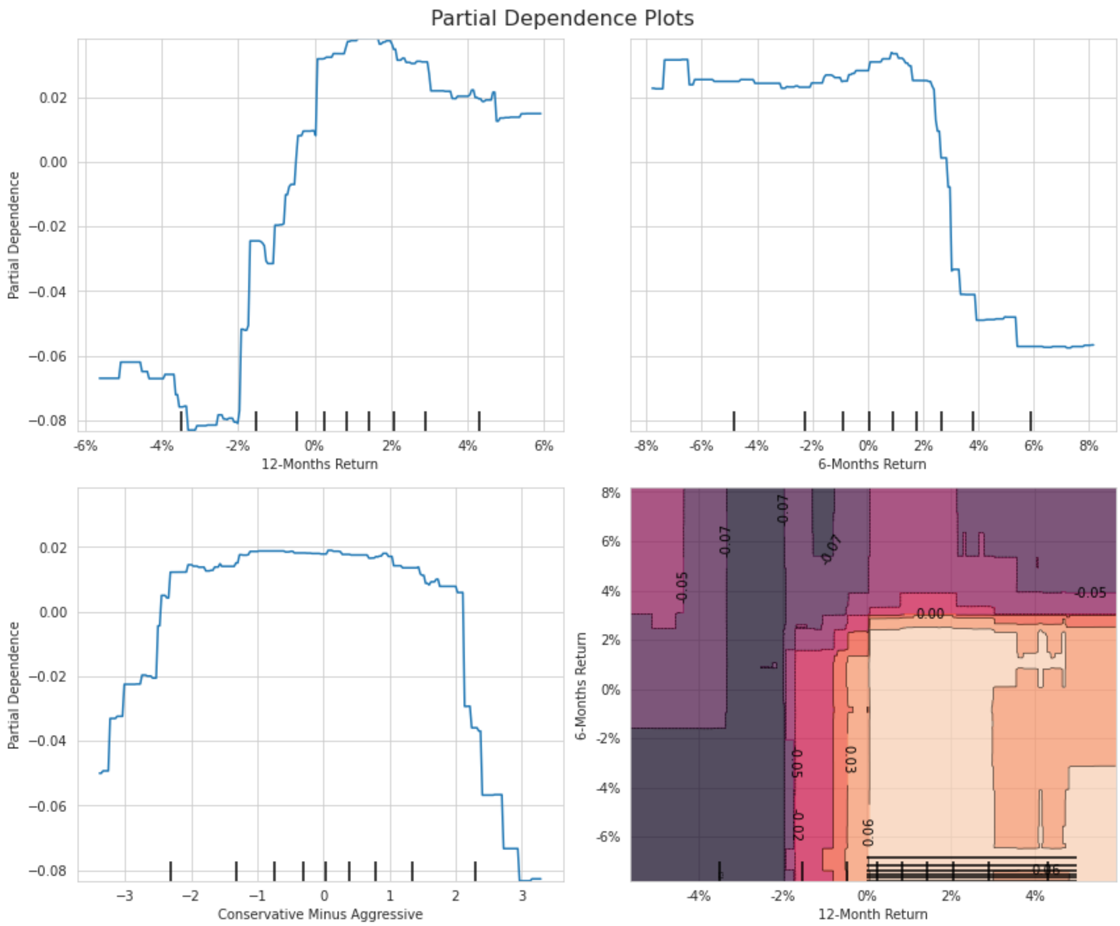
[45]:
```
y_score = gb_clf.predict_proba(X_)[:, 1]
roc_auc_score(y_score=y_score, y_true=y_clean)
```

[45]: 0.6056119291581973

One-way and two-way partial depende plots

[46]:
```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 10))
```

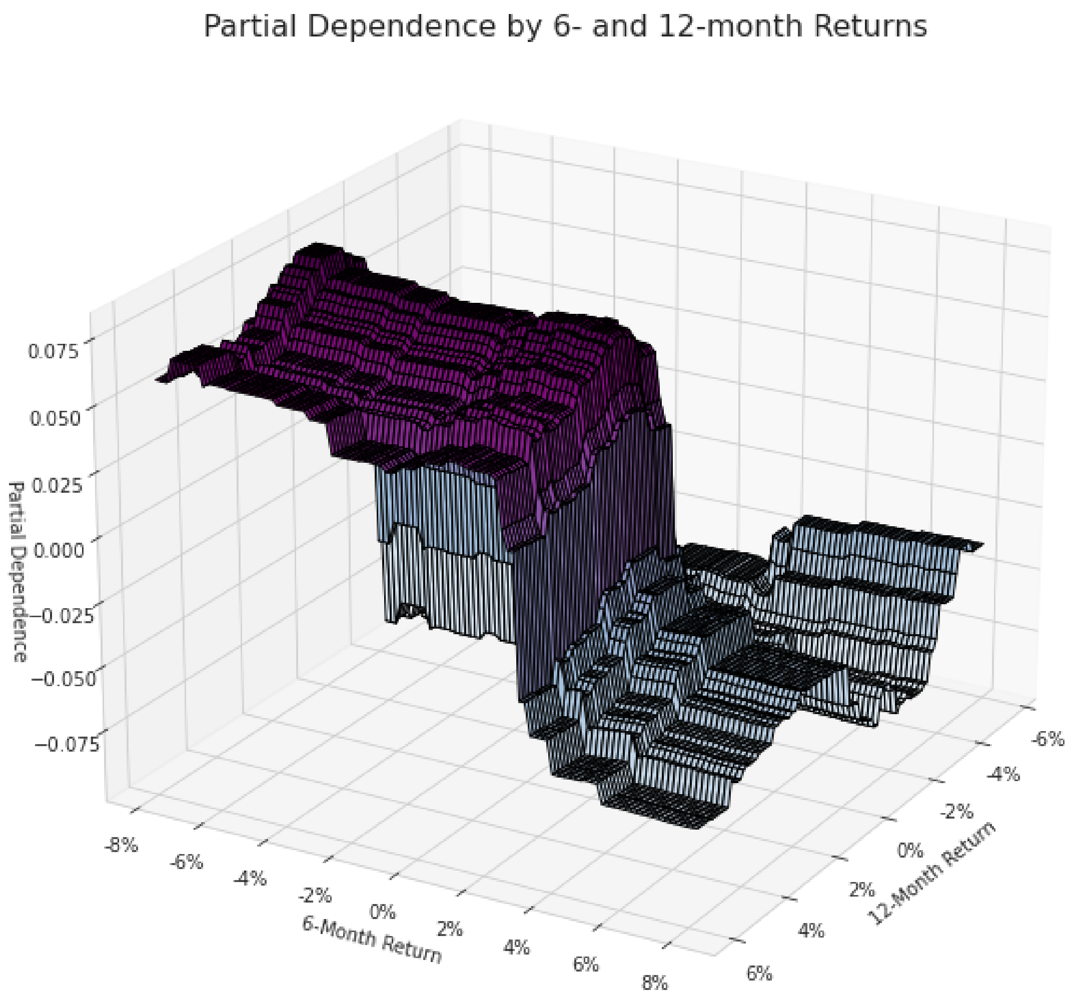
```
plot_partial_dependence(
    estimator=gb_clf,
    X=X_,
    features=['return_12m', 'return_6m', 'CMA', ('return_12m', 'return_6m')],
    percentiles=(0.05, 0.95),
    n_jobs=-1,
    n_cols=2,
    response_method='decision_function',
    grid_resolution=250,
    ax=axes)

for i, j in product([0, 1], repeat=2):
    if i!=1 or j!= 0:
        axes[i][j].xaxis.set_major_formatter(FuncFormatter(lambda y, _: '{:.
↪0%}'.format(y)))

axes[1][1].yaxis.set_major_formatter(FuncFormatter(lambda y, _: '{:.0%}'.
↪format(y)))

axes[0][0].set_ylabel('Partial Dependence')
axes[1][0].set_ylabel('Partial Dependence')
axes[0][0].set_xlabel('12-Months Return')
axes[0][1].set_xlabel('6-Months Return')
axes[1][0].set_xlabel('Conservative Minus Aggressive')

axes[1][1].set_xlabel('12-Month Return')
axes[1][1].set_ylabel('6-Months Return')
fig.suptitle('Partial Dependence Plots', fontsize=16)
fig.tight_layout()
fig.subplots_adjust(top=.95)
```



Two-way partial dependence as 3D plot

```
[47]: targets = ['return_12m', 'return_6m']
pdp, axes = partial_dependence(estimator=gb_clf,
                               features=targets,
                               X=X_,
                               grid_resolution=100)

XX, YY = np.meshgrid(axes[0], axes[1])
Z = pdp[0].reshape(list(map(np.size, axes))).T

fig = plt.figure(figsize=(14, 8))
ax = Axes3D(fig)
surface = ax.plot_surface(XX, YY, Z,
                          rstride=1,
                          cstride=1,
                          cmap=plt.cm.BuPu,
                          edgecolor='k')
```

```python
ax.set_xlabel('12-Month Return')
ax.set_ylabel('6-Month Return')
ax.set_zlabel('Partial Dependence')
ax.view_init(elev=22, azim=30)
ax.yaxis.set_major_formatter(FuncFormatter(lambda y, _: '{:.0%}'.format(y)))
ax.xaxis.set_major_formatter(FuncFormatter(lambda y, _: '{:.0%}'.format(y)))

# fig.colorbar(surface)
fig.suptitle('Partial Dependence by 6- and 12-month Returns', fontsize=16)
fig.tight_layout()
```



1.10 XGBoost

See XGBoost docs for details on parameters and usage.

1.10.1 Configure

[48]:
```
xgb_clf = XGBClassifier(max_depth=3,                    # Maximum tree depth for␣
 ↪base learners.
                        learning_rate=0.1,              # Boosting learning rate␣
 ↪(xgb's "eta")
                        n_estimators=100,               # Number of boosted trees␣
 ↪to fit.
                        silent=True,                    # Whether to print␣
 ↪messages while running
                        objective='binary:logistic',    # Task and objective or␣
 ↪custom objective function
                        booster='gbtree',               # Select booster: gbtree,␣
 ↪gblinear or dart
#                         tree_method='gpu_hist',
                        n_jobs=-1,                      # Number of parallel␣
 ↪threads
                        gamma=0,                        # Min loss reduction for␣
 ↪further splits
                        min_child_weight=1,             # Min sum of sample␣
 ↪weight(hessian) needed
                        max_delta_step=0,               # Max delta step for each␣
 ↪tree's weight estimation
                        subsample=1,                    # Subsample ratio of␣
 ↪training samples
                        colsample_bytree=1,             # Subsample ratio of cols␣
 ↪for each tree
                        colsample_bylevel=1,            # Subsample ratio of cols␣
 ↪for each split
                        reg_alpha=0,                    # L1 regularization term␣
 ↪on weights
                        reg_lambda=1,                   # L2 regularization term␣
 ↪on weights
                        scale_pos_weight=1,             # Balancing class weights
                        base_score=0.5,                 # Initial prediction␣
 ↪score; global bias
                        random_state=42)                # random seed
```

1.10.2 Cross-validate

[49]:
```
algo = 'xgboost'
```

[50]:
```
fname = results_path / f'{algo}.joblib'
if not Path(fname).exists():
    xgb_cv_result, run_time[algo] = run_cv(xgb_clf)
    joblib.dump(xgb_cv_result, fname)
```

```
else:
    xgb_cv_result = joblib.load(fname)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 out of  12 | elapsed:  7.9min remaining:  1.6min
[Parallel(n_jobs=-1)]: Done  12 out of  12 | elapsed:  8.0min finished
```

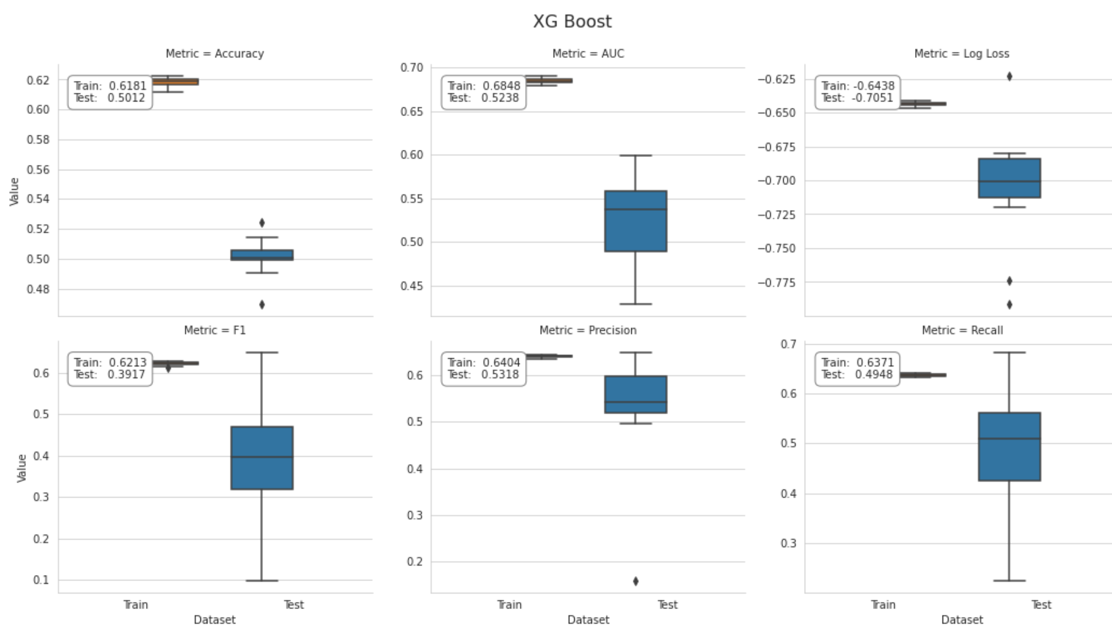
1.10.3 Plot Results

[51]:
```
xbg_result = stack_results(xgb_cv_result)
xbg_result.groupby(['Metric', 'Dataset']).Value.mean().unstack()
```

[51]:

Dataset	Test	Train
Metric		
AUC	0.523803	0.684837
Accuracy	0.501167	0.618149
F1	0.391726	0.621304
Log Loss	-0.705113	-0.643818
Precision	0.531815	0.640388
Recall	0.494786	0.637127

[52]:
```
plot_result(xbg_result, model='XG Boost', fname=f'figures/{algo}_cv_result')
```



1.10.4 Feature Importance

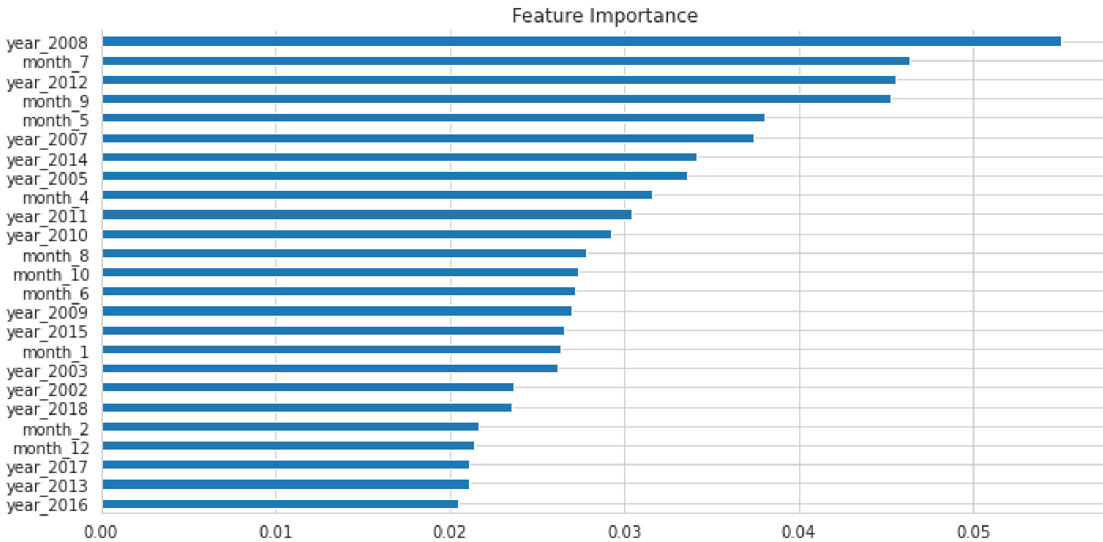
[53]:
```
xgb_clf.fit(X=X_dummies, y=y)
```

```
[20:19:06] WARNING: ../src/learner.cc:541:
Parameters: { silent } might not be used.

  This may not be accurate due to some parameters are only used in language
bindings but
  passed down to XGBoost core.  Or some parameters are not used but slip through
this
  verification. Please open an issue if you find above cases.


[20:19:06] WARNING: ../src/learner.cc:1061: Starting in XGBoost 1.3.0, the
default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore
the old behavior.
```

[53]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.1, max_delta_step=0, max_depth=3,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=100, n_jobs=-1, num_parallel_tree=1, random_state=42,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, silent=True,
              subsample=1, tree_method='exact', validate_parameters=1,
              verbosity=None)
```

[54]:
```
fi = pd.Series(xgb_clf.feature_importances_,
               index=X_dummies.columns)
```

[55]:
```
fi.nlargest(25).sort_values().plot.barh(figsize=(10, 5),
                                        title='Feature Importance')
sns.despine()
plt.tight_layout();
```



1.11 LightGBM

See LightGBM docs for details on parameters and usage.

1.11.1 Configure

[56]:
```
lgb_clf = LGBMClassifier(boosting_type='gbdt',
#                            device='gpu',
                         objective='binary',           # learning task
                         metric='auc',
                         num_leaves=31,                # Maximum tree leaves for␣
 ↪base learners.
                         max_depth=-1,                 # Maximum tree depth for␣
 ↪base learners, -1 means no limit.
                         learning_rate=0.1,           # Adaptive lr via callback␣
 ↪override in .fit() method
                         n_estimators=100,             # Number of boosted trees␣
 ↪to fit
                         subsample_for_bin=200000,     # Number of samples for␣
 ↪constructing bins.
                         class_weight=None,            # dict, 'balanced' or None
                         min_split_gain=0.0,           # Minimum loss reduction␣
 ↪for further split
                         min_child_weight=0.001,       # Minimum sum of instance␣
 ↪weight(hessian)
                         min_child_samples=20,         # Minimum number of data␣
 ↪need in a child(leaf)
```

```
                              subsample=1.0,              # Subsample ratio of␣
↪training samples
                              subsample_freq=0,           # Frequency of␣
↪subsampling, <=0: disabled
                              colsample_bytree=1.0,       # Subsampling ratio of␣
↪features
                              reg_alpha=0.0,              # L1 regularization term␣
↪on weights
                              reg_lambda=0.0,             # L2 regularization term␣
↪on weights
                              random_state=42,            # Random number seed;␣
↪default: C++ seed
                              n_jobs=-1,                  # Number of parallel␣
↪threads.
                              silent=False,
                              importance_type='gain',     # default: 'split' or␣
↪'gain'
                             )
```

1.11.2 Cross-Validate

Using categorical features

[57]:
```
algo = 'lgb_factors'
```

[58]:
```
fname = results_path / f'{algo}.joblib'
if not Path(fname).exists():
    lgb_factor_cv_result, run_time[algo] = run_cv(lgb_clf, X=X_factors,␣
↪fit_params={'categorical_feature': cat_cols})
    joblib.dump(lgb_factor_cv_result, fname)
else:
    lgb_factor_cv_result = joblib.load(fname)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 out of  12 | elapsed: 30.7min remaining:  6.1min
[Parallel(n_jobs=-1)]: Done  12 out of  12 | elapsed: 30.8min finished
```

Plot Results

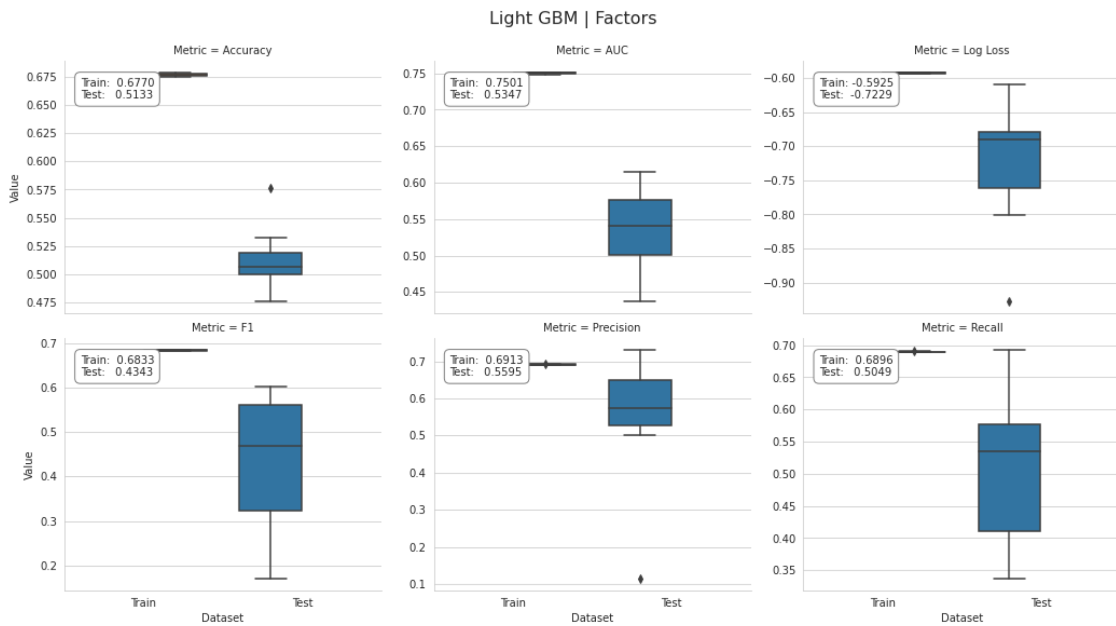
[59]:
```
lgb_factor_result = stack_results(lgb_factor_cv_result)
lgb_factor_result.groupby(['Metric', 'Dataset']).Value.mean().unstack()
```

[59]:

Dataset Metric	Test	Train
AUC	0.534674	0.750110
Accuracy	0.513278	0.676953
F1	0.434291	0.683308
Log Loss	-0.722875	-0.592547

```
Precision  0.559479  0.691272
Recall     0.504942  0.689605
```

[60]:
```
plot_result(lgb_factor_result, model='Light GBM | Factors', fname=f'figures/
↪{algo}_cv_result')
```



Using dummy variables

[61]:
```
algo = 'lgb_dummies'
```

[62]:
```
fname = results_path / f'{algo}.joblib'
if not Path(fname).exists():
    lgb_dummy_cv_result, run_time[algo] = run_cv(lgb_clf)
    joblib.dump(lgb_dummy_cv_result, fname)
else:
    lgb_dummy_cv_result = joblib.load(fname)
```

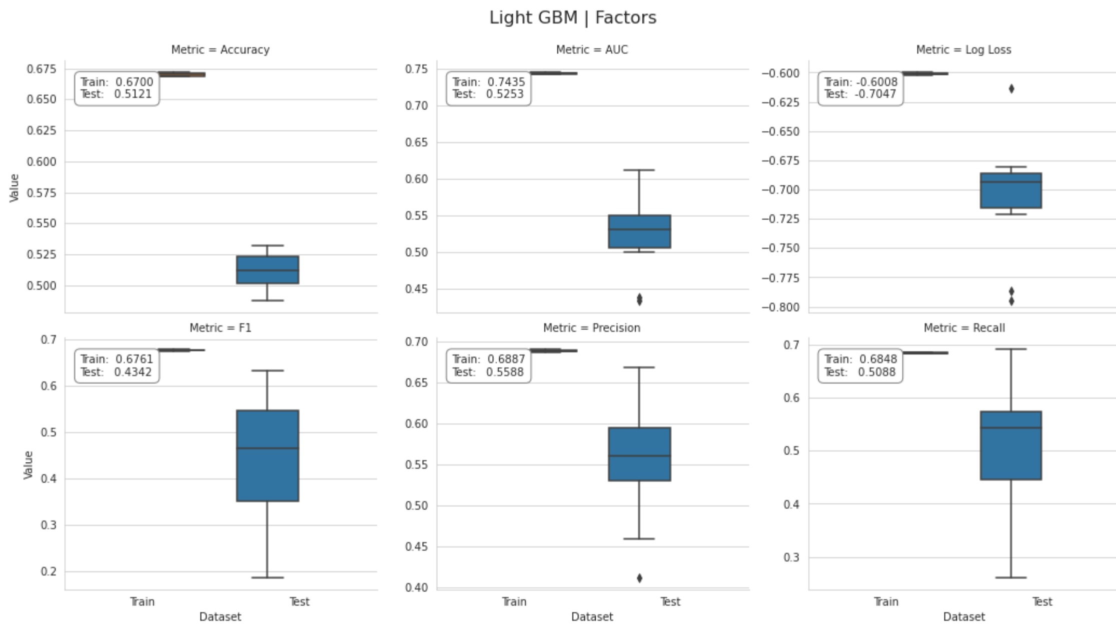
```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 out of  12 | elapsed: 10.2min remaining:  2.0min
[Parallel(n_jobs=-1)]: Done  12 out of  12 | elapsed: 10.2min finished
```

Plot results

[63]:
```
lgb_dummy_result = stack_results(lgb_dummy_cv_result)
lgb_dummy_result.groupby(['Metric', 'Dataset']).Value.mean().unstack()
```

[63]:
```
Dataset        Test     Train
Metric
AUC         0.525324  0.743517
Accuracy    0.512141  0.670014
F1          0.434198  0.676106
Log Loss   -0.704733 -0.600785
Precision   0.558771  0.688686
Recall      0.508796  0.684816
```

[64]:
```
plot_result(lgb_dummy_result, model='Light GBM | Factors', fname=f'figures/
 ↪{algo}_cv_result')
```



1.12 Catboost

See CatBoost docs for details on parameters and usage.

1.12.1 CPU

Configure

[65]:
```
cat_clf = CatBoostClassifier()
```

Cross-Validate

[66]:
```
s = pd.Series(X_factors.columns.tolist())
cat_cols_idx = s[s.isin(cat_cols)].index.tolist()
```

Catboost requires integer values for categorical variables.

[67]:
```python
algo = 'catboost'
```

[68]:
```python
fname = results_path / f'{algo}.joblib'
if not Path(fname).exists():
    fit_params = {'cat_features': cat_cols_idx}
    cat_cv_result, run_time[algo] = run_cv(cat_clf,
                                           X=X_factors,
                                           fit_params=fit_params,
                                           n_jobs=-1)
    joblib.dump(cat_cv_result, fname)
else:
    cat_cv_result = joblib.load(fname)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 out of  12 | elapsed: 30.3min remaining:  6.1min
[Parallel(n_jobs=-1)]: Done  12 out of  12 | elapsed: 30.3min finished
```

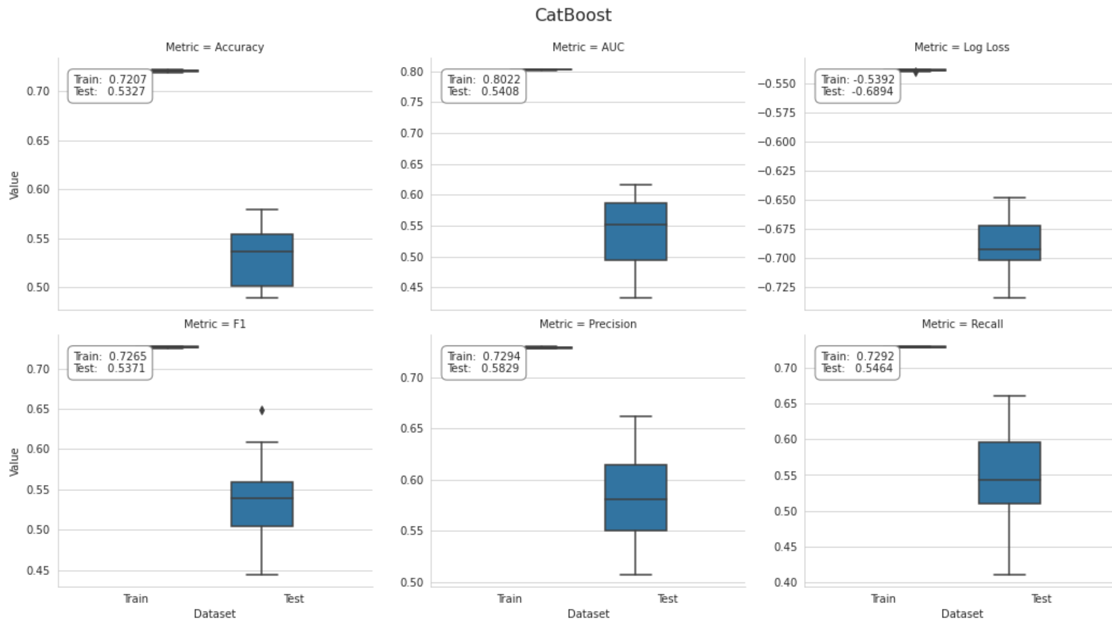
Plot Results

[69]:
```python
cat_result = stack_results(cat_cv_result)
cat_result.groupby(['Metric', 'Dataset']).Value.mean().unstack()
```

[69]:

Dataset Metric	Test	Train
AUC	0.540770	0.802151
Accuracy	0.532725	0.720658
F1	0.537121	0.726497
Log Loss	-0.689399	-0.539176
Precision	0.582897	0.729352
Recall	0.546427	0.729151

[70]:
```python
plot_result(cat_result, model='CatBoost', fname=f'figures/{algo}_cv_result')
```



1.12.2 GPU

Naturally, the following requires that you have a GPU.

Configure

[71]:
```
cat_clf_gpu = CatBoostClassifier(task_type='GPU')
```

Cross-Validate

[72]:
```
s = pd.Series(X_factors.columns.tolist())
cat_cols_idx = s[s.isin(cat_cols)].index.tolist()
```

[73]:
```
algo = 'catboost_gpu'
```

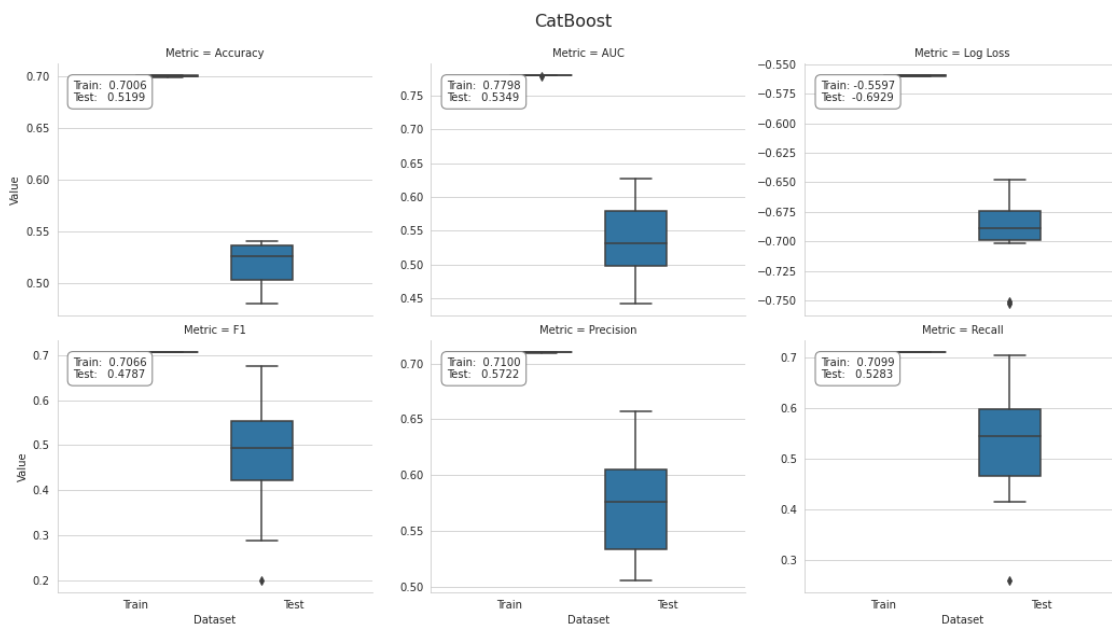
[74]:
```
fname = results_path / f'{algo}.joblib'
if not Path(fname).exists():
    fit_params = {'cat_features': cat_cols_idx}
    cat_gpu_cv_result, run_time[algo] = run_cv(cat_clf_gpu,
                                               y=y,
                                               X=X_factors,
                                               fit_params=fit_params,
                                               n_jobs=1)
    joblib.dump(cat_gpu_cv_result, fname)
else:
    cat_gpu_cv_result = joblib.load(fname)
```

Plot Results

[75]:
```
cat_gpu_result = stack_results(cat_gpu_cv_result)
cat_gpu_result.groupby(['Metric', 'Dataset']).Value.mean().unstack()
```

[75]:
```
Dataset        Test     Train
Metric
AUC         0.534941  0.779761
Accuracy    0.519893  0.700589
F1          0.478687  0.706628
Log Loss   -0.692898 -0.559732
Precision   0.572199  0.709989
Recall      0.528337  0.709944
```

[76]:
```
plot_result(cat_gpu_result, model='CatBoost', fname=f'figures/{algo}_cv_result')
```



1.13 Compare Results

[77]:
```
results = {'Baseline': dummy_result,
           'Random Forest': rf_result,
           'AdaBoost': ada_result,
           'Gradient Booster': gb_result,
           'XGBoost': xbg_result,
           'LightGBM Dummies': lgb_dummy_result,
           'LightGBM Factors': lgb_factor_result,
           'CatBoost': cat_result,
           'CatBoost GPU': cat_gpu_result}
```

```
df = pd.DataFrame()
for model, result in results.items():
    df = pd.concat([df, result.groupby(['Metric', 'Dataset']
                                       ).Value.mean().unstack()['Test'].
 ↪to_frame(model)], axis=1)

df.T.sort_values('AUC', ascending=False)
```

[77]:

Metric	AUC	Accuracy	F1	Log Loss	Precision	Recall
CatBoost	0.540770	0.532725	0.537121	-0.689399	0.582897	0.546427
AdaBoost	0.536567	0.505709	0.464293	-0.692850	0.544433	0.571999
CatBoost GPU	0.534941	0.519893	0.478687	-0.692898	0.572199	0.528337
LightGBM Factors	0.534674	0.513278	0.434291	-0.722875	0.559479	0.504942
Gradient Booster	0.531743	0.505791	0.448612	-0.697194	0.551708	0.543934
LightGBM Dummies	0.525324	0.512141	0.434198	-0.704733	0.558771	0.508796
Random Forest	0.524482	0.512583	0.502093	-0.694309	0.557773	0.537495
XGBoost	0.523803	0.501167	0.391726	-0.705113	0.531815	0.494786
Baseline	0.503582	0.503582	0.516424	-17.070718	0.548116	0.505758

[78]:
```
algo_dict = dict(zip(['dummy_clf', 'random_forest', 'adaboost', 'sklearn_gbm',
                      'xgboost', 'lgb_factors', 'lgb_dummies', 'catboost',␣
 ↪'catboost_gpu'],
                     ['Baseline', 'Random Forest', 'AdaBoost', 'Gradient␣
 ↪Booster',
                      'XGBoost', 'LightGBM Dummies', 'LightGBM Factors',␣
 ↪'CatBoost', 'CatBoost GPU']))
```

[79]:
```
print(run_time)
```

{'dummy_clf': 3.446434736251831, 'random_forest': 486.9282796382904, 'adaboost': 385.62260723114014, 'sklearn_gbm': 53.61000990867615, 'xgboost': 477.78596901893616, 'lgb_factors': 1847.2539386749268, 'lgb_dummies': 613.5608298778534, 'catboost': 1819.1900961399078}

[80]:
```
r = pd.Series(run_time).to_frame('t')
r.index = r.index.to_series().map(algo_dict)
r.to_csv(results_path / 'runtime.csv')
```

[81]:
```
# r = pd.read_csv(results_path / 'runtime.csv', index_col=0)
```

[82]:
```
auc = pd.concat([v.loc[(v.Dataset=='Test') & (v.Metric=='AUC'), 'Value'].
 ↪to_frame('AUC').assign(Model=k)
                 for k, v in results.items()])
# auc = auc[auc.Model != 'Baseline']
```

[83]:
```
fig, axes = plt.subplots(figsize=(15, 5), ncols=2)
idx = df.T.drop('Baseline')['AUC'].sort_values(ascending=False).index
```

```
sns.barplot(x='Model', y='AUC',
            data=auc,
            order=idx, ax=axes[0])
axes[0].set_xticklabels([c.replace(' ', '\n') for c in idx])
axes[0].set_ylim(.49, .58)
axes[0].set_title('Predictive Accuracy')

(r.drop('Baseline').sort_values('t').rename(index=lambda x: x.replace(' ',␣
 ↪'\n'))
 .plot.barh(title='Runtime', ax=axes[1], logx=True, legend=False))
axes[1].set_xlabel('Seconds (log scale)')
sns.despine()
fig.tight_layout()
```