

07__model__interpretation

September 29, 2021

1 Long-Short Strategy, Part 4: How to interpret GBM results

In this section, we'll start designing, implementing, and evaluating a trading strategy for US equities driven by daily return forecasts produced by gradient boosting models.

As in the previous examples, we'll lay out a framework and build a specific example that you can adapt to run your own experiments. There are numerous aspects that you can vary, from the asset class and investment universe to more granular aspects like the features, holding period, or trading rules. See, for example, the **Alpha Factor Library** in the [Appendix](#) for numerous additional features.

We'll keep the trading strategy simple and only use a single ML signal; a real-life application will likely use multiple signals from different sources, such as complementary ML models trained on different datasets or with different lookahead or lookback periods. It would also use sophisticated risk management, from simple stop-loss to value-at-risk analysis.

Six notebooks cover our workflow sequence:

1. [preparing_the_model_data](#): we engineer a few simple features from the Quandl Wiki data
2. [trading_signals_with_lightgbm_and_catboost](#): we tune hyperparameters for LightGBM and CatBoost to select a model, using 2015/16 as our validation period.
3. [evaluate_trading_signals](#): we compare the cross-validation performance using various metrics to select the best model.
4. [model_interpretation](#) (this notebook): we take a closer look at the drivers behind the best model's predictions.
5. [making_out_of_sample_predictions](#): we generate predictions for our out-of-sample test period 2017.
6. [backtesting_with_zipline](#): evaluate the historical performance of a long-short strategy based on our predictive signals using Zipline.

Understanding why a model predicts a certain outcome is very important for several reasons, including trust, actionability, accountability, and debugging.

Insights into the nonlinear relationship between features and the outcome uncovered by the model, as well as interactions among features, are also of value when the goal is to learn more about the underlying drivers of the phenomenon under study.

1.1 Imports & Settings

```
[1]: %matplotlib inline

from pathlib import Path
import warnings
from random import randint
import joblib
from itertools import product

import numpy as np
import pandas as pd

import shap
import lightgbm as lgb
from sklearn.inspection import (plot_partial_dependence,
                                partial_dependence)

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

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

```
[3]: DATA_STORE = Path('../data/assets.h5')
```

1.2 Retrieve best model parameters

The results used below are created by the notebooks [gbm_trading_signals](#).

```
[4]: with pd.HDFStore('data.h5') as store:
      best_params = store['best_params']
```

```
[5]: best_params
```

```
[5]: train_length      1134.00
test_length           63.00
learning_rate          0.01
num_leaves             128.00
feature_fraction        0.95
min_data_in_leaf       250.00
boost_rounds           400.00
Name: 595, dtype: float64
```

1.3 Get Training Data

```
[6]: data = pd.read_hdf('data.h5', 'model_data').sort_index()
data = data.loc[idx[:, '2013':'2018'], :]
```

```
[7]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 1248422 entries, ('A', Timestamp('2013-01-02 00:00:00')) to ('ZION',
Timestamp('2017-12-29 00:00:00'))
Data columns (total 34 columns):
#   Column                Non-Null Count  Dtype
---  -
0   dollar_vol            1248422 non-null float64
1   dollar_vol_rank       1248422 non-null float64
2   rsi                   1248422 non-null float64
3   bb_high               1248422 non-null float64
4   bb_low                1248419 non-null float64
5   NATR                  1248422 non-null float64
6   ATR                   1248422 non-null float64
7   PPO                   1248422 non-null float64
8   MACD                  1248422 non-null float64
9   sector                1248422 non-null int64
10  r01                    1248422 non-null float64
11  r05                    1248422 non-null float64
12  r10                    1248422 non-null float64
13  r21                    1248422 non-null float64
14  r42                    1248422 non-null float64
15  r63                    1248422 non-null float64
16  r01dec                 1248419 non-null float64
17  r05dec                 1248422 non-null float64
18  r10dec                 1248422 non-null float64
19  r21dec                 1248422 non-null float64
20  r42dec                 1248422 non-null float64
21  r63dec                 1248422 non-null float64
22  r01q_sector            1248419 non-null float64
23  r05q_sector            1248422 non-null float64
24  r10q_sector            1248422 non-null float64
25  r21q_sector            1248422 non-null float64
26  r42q_sector            1248422 non-null float64
27  r63q_sector            1248422 non-null float64
28  r01_fwd                1247427 non-null float64
29  r05_fwd                1243447 non-null float64
30  r21_fwd                1227527 non-null float64
31  year                   1248422 non-null int64
32  month                  1248422 non-null int64
33  weekday                1248422 non-null int64
dtypes: float64(30), int64(4)
```

memory usage: 329.4+ MB

1.4 Create Binary Data

```
[8]: dates = sorted(data.index.get_level_values('date').unique())
```

```
[9]: train_dates = dates[-int(best_params.train_length):]
```

```
[10]: data = data.loc[idx[:, train_dates], :]
```

```
[11]: labels = sorted(data.filter(like='_fwd').columns)
features = data.columns.difference(labels).tolist()
```

```
[12]: lookahead = 1
label = f'r{lookahead:02}_fwd'
```

```
[13]: categoricals = ['year', 'month', 'sector', 'weekday']
```

```
[14]: lgb_train = lgb.Dataset(data=data[features],
                             label=data[label],
                             categorical_feature=categoricals,
                             free_raw_data=False)
```

1.5 Train LightGBM Model

```
[15]: params = dict(boosting='gbdt', objective='regression', verbose=-1)
```

```
[16]: train_params = ['learning_rate', 'num_leaves', 'feature_fraction',
                      ↪ 'min_data_in_leaf']
```

```
[17]: params.update(best_params.loc[train_params].to_dict())
for p in ['min_data_in_leaf', 'num_leaves']:
    params[p] = int(params[p])
```

```
[18]: params
```

```
[18]: {'boosting': 'gbdt',
      'objective': 'regression',
      'verbose': -1,
      'learning_rate': 0.01,
      'num_leaves': 128,
      'feature_fraction': 0.95,
      'min_data_in_leaf': 250}
```

```
[19]: lgb_model = lgb.train(params=params,
                             train_set=lgb_train,
                             num_boost_round=int(best_params.boost_rounds))
```

1.6 Compute Feature Importance

A common approach to gaining insights into the predictions made by tree ensemble methods, such as gradient boosting or random forest models, is to attribute feature importance values to each input variable. These feature importance values can be computed on an individual basis for a single prediction or globally for an entire dataset (that is, for all samples) to gain a higher-level perspective on how the model makes predictions.

There are three primary ways to compute global feature importance values: - Gain: This classic approach introduced by Leo Breiman in 1984 uses the total reduction of loss or impurity contributed by all splits for a given feature. The motivation is largely heuristic, but it is a commonly used method to select features. - Split count: This is an alternative approach that counts how often a feature is used to make a split decision, based on the selection of features for this purpose based on the resultant information gain. - Permutation: This approach randomly permutes the feature values in a test set and measures how much the model's error changes, assuming that an important feature should create a large increase in the prediction error. Different permutation choices lead to alternative implementations of this basic approach.

All gradient boosting implementations provide feature-importance scores after training as a model attribute. The `lightGBM` library provides five versions, as shown in the following list: - `total_gain` and `gain` as its average per split - `total_cover` as the number of samples per split when a feature was used - `weight` as the split count from preceding values

These values are available using the trained model's `.feature_importance()` method with the corresponding `importance_type` parameter. For the best performing XGBoost model, the results are as follows (the total measures have a correlation of 0.8, as do `cover` and `total_cover`):

While the indicators for different months and years dominate, the most recent 1m return is the second-most important feature from a `total_gain` perspective, and is used frequently according to the weight measure, but produces low average gains as it is applied to relatively few instances on average).

```
[20]: def get_feature_importance(model, importance_type='split'):
        fi = pd.Series(model.feature_importance(importance_type=importance_type),
                        index=model.feature_name())
        return fi/fi.sum()
```

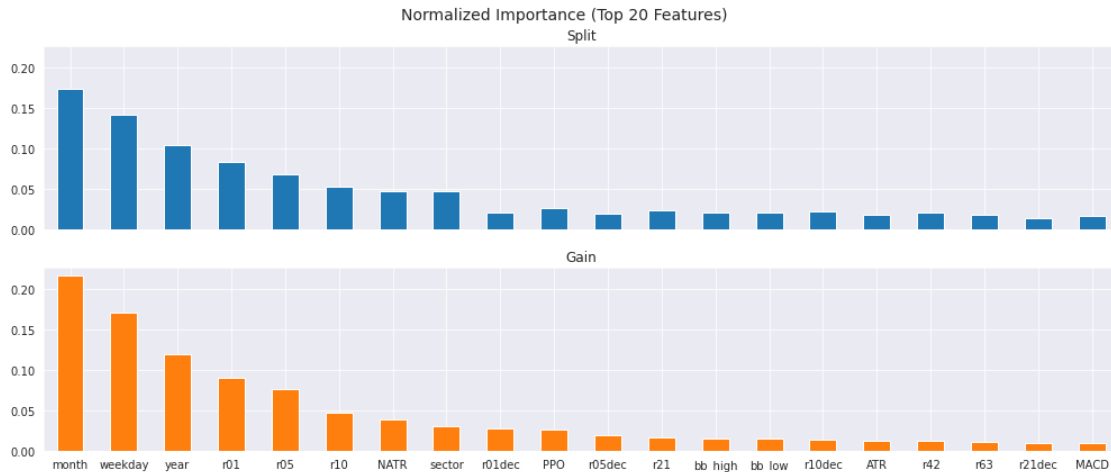
```
[21]: feature_importance = (get_feature_importance(lgb_model).to_frame('Split').
                            join(get_feature_importance(lgb_model, 'gain').
                                ↪to_frame('Gain')))
```

```
[22]: (feature_importance
        .nlargest(20, columns='Gain')
        .sort_values('Gain', ascending=False)
        .plot
        .bar(subplots=True,
              layout=(2, 1),
              figsize=(14, 6),
              legend=False,
              sharey=True,
```

```

    rot=0))
plt.suptitle('Normalized Importance (Top 20 Features)', fontsize=14)
plt.tight_layout()
plt.subplots_adjust(top=.9);

```



1.7 Partial Dependence Plots

In addition to the summary contribution of individual features to the model's prediction, partial dependence plots visualize the relationship between the target variable and a set of features. The nonlinear nature of gradient boosting trees causes this relationship to depend on the values of all other features.

Hence, we will marginalize these features out. By doing so, we can interpret the partial dependence as the expected target response. We can visualize partial dependence only for individual features or feature pairs. The latter results in contour plots that show how combinations of feature values produce different predicted probabilities, as shown in the following code:

Requires running cross-validation in [sklearn_gbm_tuning](#).

```

[23]: class OneStepTimeSeriesSplit:
      pass

```

```

[24]: gb_clf = joblib.load('results/baseline/sklearn_gbm_model.joblib')

```

```

[25]: def get_data(start='2000', end='2018', 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

```

```

[26]: 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)
    return df

```

```

[27]: y_clean, features_clean = get_data(dropna=True)
X = factorize_cats(features_clean).drop(['year', 'month'], axis=1)

```

1.7.1 2D Partial Dependency

The below plot shows the dependence of the probability of a positive return over the next month given the range of values for lagged 1-month and 3-month returns after eliminating outliers at the [5%, 95%] percentiles. The month_9 variable is a dummy variable, hence the step-function-like plot.

```

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

X_ = features_clean
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].axis.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')

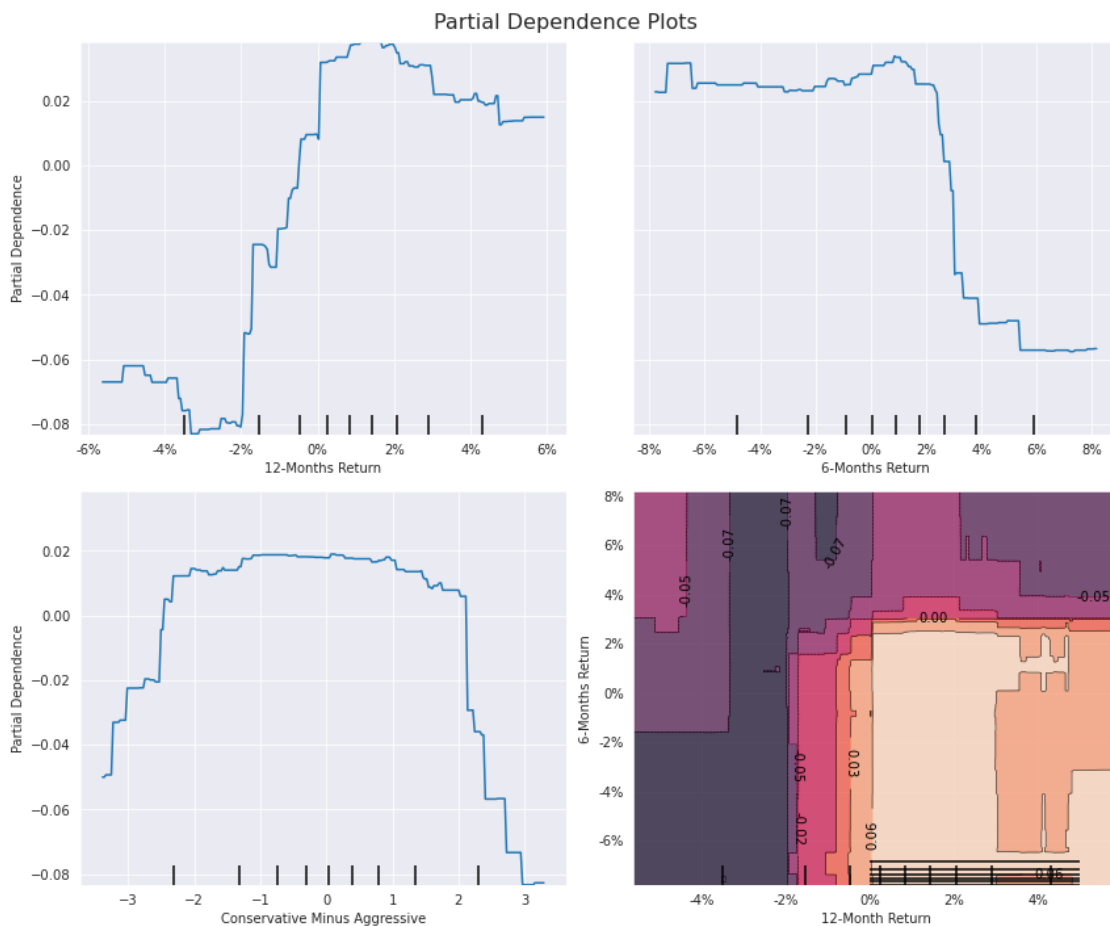
```

```

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);

```



1.7.2 3D Partial Dependency

We can also visualize the dependency in 3D, as shown in the following code that produces the below 3D plot of the partial dependence of the 1-month return direction on lagged 1-month and 3-months returns:


```

[29]: sns.set_style('whitegrid')
targets = ['return_12m', 'return_6m']
pdp, axes = partial_dependence(estimator=gb_clf,
                               features=targets,
                               X=X_,
                               grid_resolution=100)

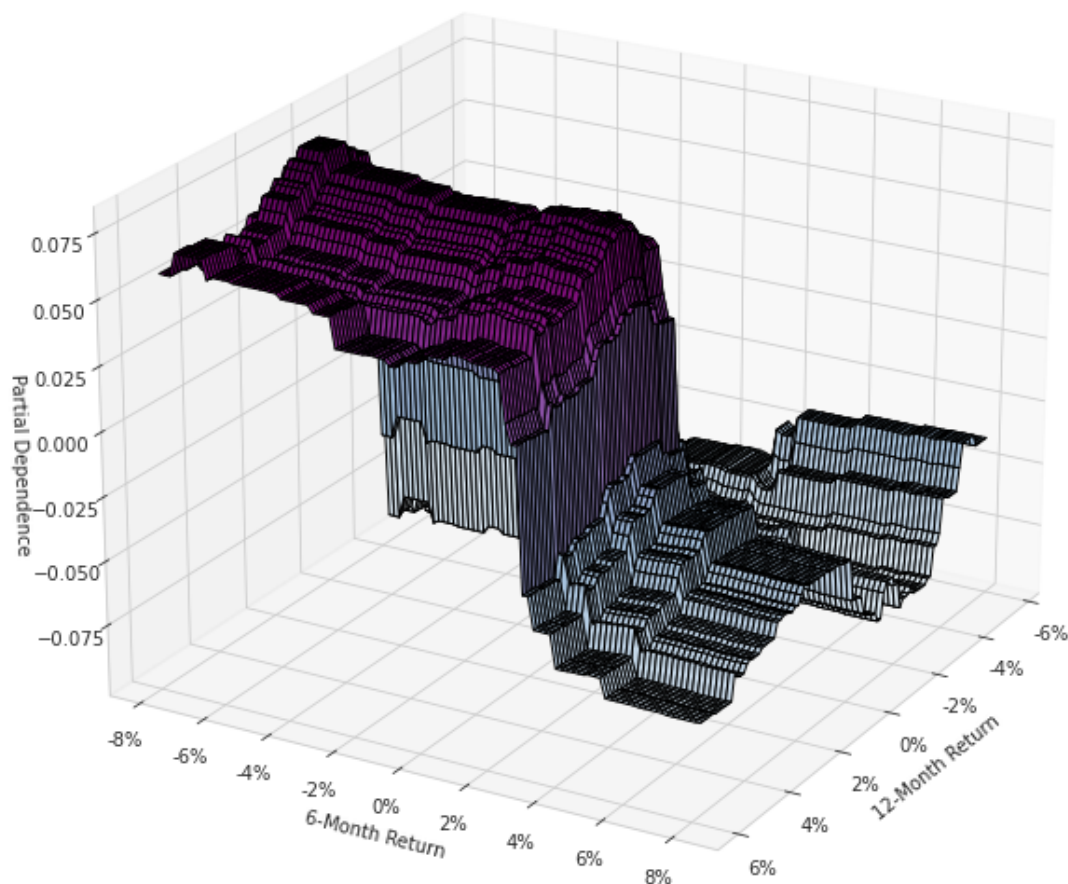
XX, YY = np.meshgrid(axes[0], axes[1])
Z = pdp[0].reshape(list(map(np.size, axes))).T

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')
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();

```

Partial Dependence by 6- and 12-month Returns



1.8 SHAP Values

At the 2017 NIPS conference, Scott Lundberg and Su-In Lee from the University of Washington presented a new and more accurate approach to explaining the contribution of individual features to the output of tree ensemble models called [SHapley Additive exPlanations](#), or SHAP values.

This new algorithm departs from the observation that feature-attribution methods for tree ensembles, such as the ones we looked at earlier, are inconsistent—that is, a change in a model that increases the impact of a feature on the output can lower the importance values for this feature.

SHAP values unify ideas from collaborative game theory and local explanations, and have been shown to be theoretically optimal, consistent, and locally accurate based on expectations. Most importantly, Lundberg and Lee have developed an algorithm that manages to reduce the complexity of computing these model-agnostic, additive feature-attribution methods from $O(TLDM)$ to $O(TLD2)$, where T and M are the number of trees and features, respectively, and D and L are the

maximum depth and number of leaves across the trees.

This important innovation permits the explanation of predictions from previously intractable models with thousands of trees and features in a fraction of a second. An open source implementation became available in late 2017 and is compatible with XGBoost, LightGBM, CatBoost, and sklearn tree models.

Shapley values originated in game theory as a technique for assigning a value to each player in a collaborative game that reflects their contribution to the team's success. SHAP values are an adaptation of the game theory concept to tree-based models and are calculated for each feature and each sample. They measure how a feature contributes to the model output for a given observation. For this reason, SHAP values provide differentiated insights into how the impact of a feature varies across samples, which is important given the role of interaction effects in these nonlinear models.

1.8.1 Summary Plot

To get a high-level overview of the feature importance across a number of samples, there are two ways to plot the SHAP values: a simple average across all samples that resembles the global feature-importance measures computed previously (as shown in the second plot), or a scatter graph to display the impact of every feature for every sample (as shown in first plot).

The scatter plot sorts features by their total SHAP values across all samples, and then shows how each feature impacts the model output as measured by the SHAP value as a function of the feature's value, represented by its color, where red represents high and blue represents low values relative to the feature's range.

They are very straightforward to produce using a trained model of a compatible library and matching input data, as shown in the following code:

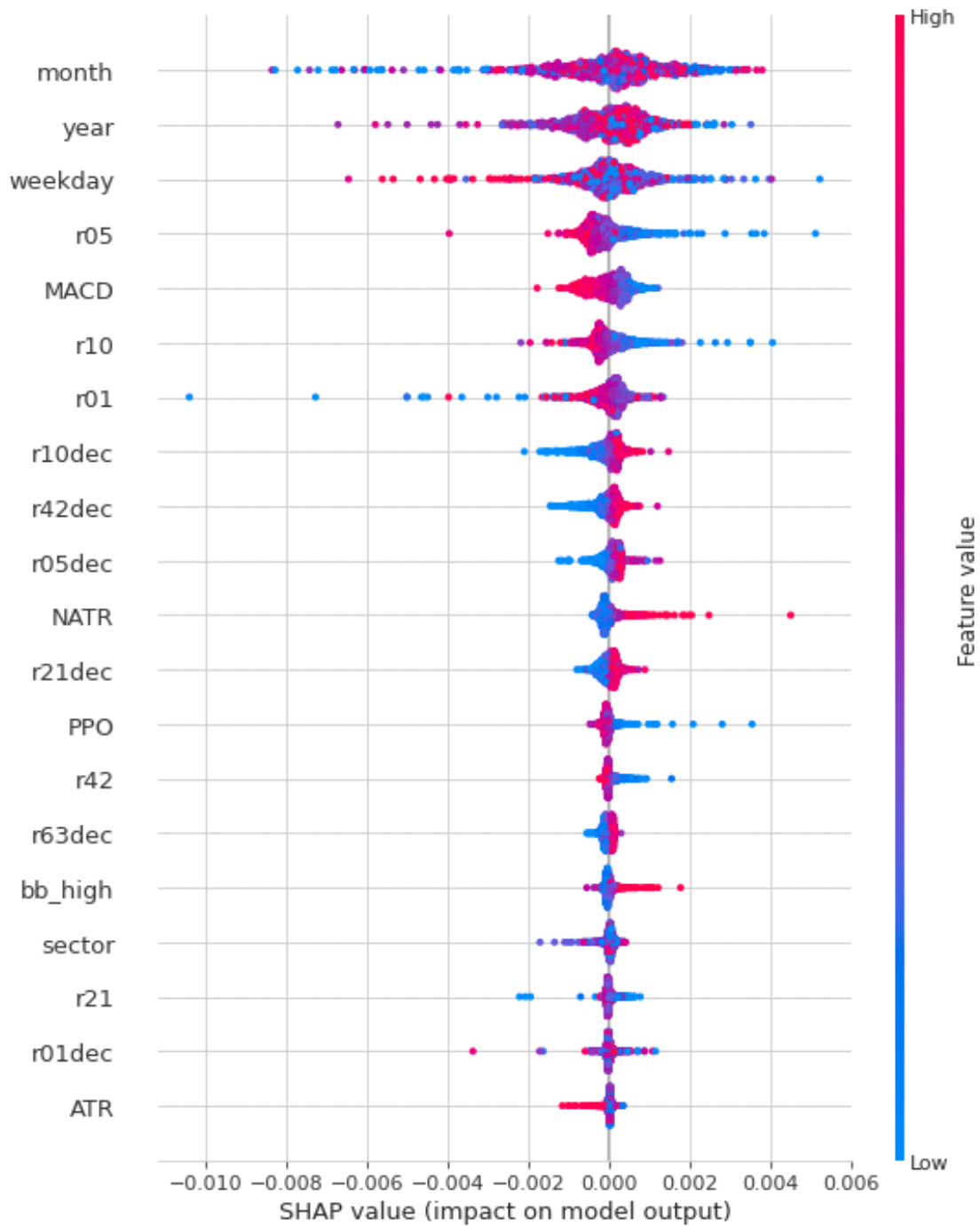
```
[30]: X = data[features].sample(n=1000)

[31]: # load JS visualization code to notebook
      shap.initjs()

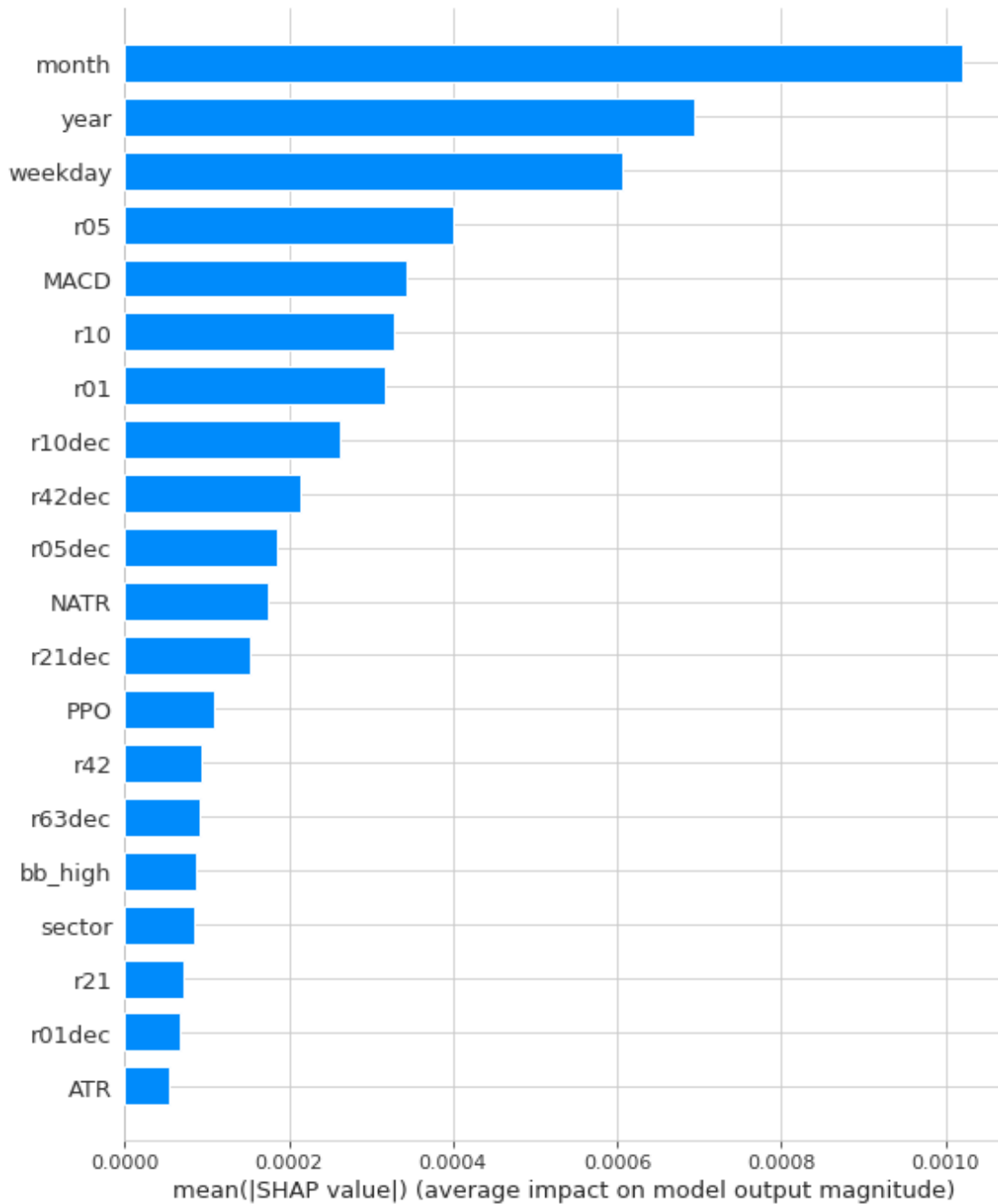
      # explain the model's predictions using SHAP values
      explainer = shap.TreeExplainer(lgb_model)
      shap_values = explainer.shap_values(X=X)

      shap.summary_plot(shap_values, X, show=False)
      plt.tight_layout();
```

<IPython.core.display.HTML object>



```
[32]: shap.summary_plot(shap_values, X, plot_type="bar", show=False)
plt.tight_layout();
```



1.8.2 Feature Interaction

1.8.3 Force Plots

The following force plot shows the cumulative impact of various features and their values on the model output, which in this case was 0.6, quite a bit higher than the base value of 0.13 (the average model output over the provided dataset).

Features highlighted in red increase the output. The month being October is the most important feature and increases the output from 0.338 to 0.537, whereas the year being 2017 reduces the output.

Hence, we obtain a detailed breakdown of how the model arrived at a specific prediction:

```
[33]: i = randint(0, len(X))
      # visualize the first prediction's explanation
      shap.force_plot(explainer.expected_value, shap_values[i,:], X.iloc[i,:])
```

```
[33]: <shap.plots._force.AdditiveForceVisualizer at 0x7f2e678d7d60>
```

We can also compute force plots for numerous data points or predictions at a time and use a clustered visualization to gain insights into how prevalent certain influence patterns are across the dataset.

The following plot shows the force plots for the first 1,000 observations rotated by 90 degrees, stacked horizontally, and ordered by the impact of different features on the outcome for the given observation.

The implementation uses hierarchical agglomerative clustering of data points on the feature SHAP values to identify these patterns, and displays the result interactively for exploratory analysis, as shown in the following code:

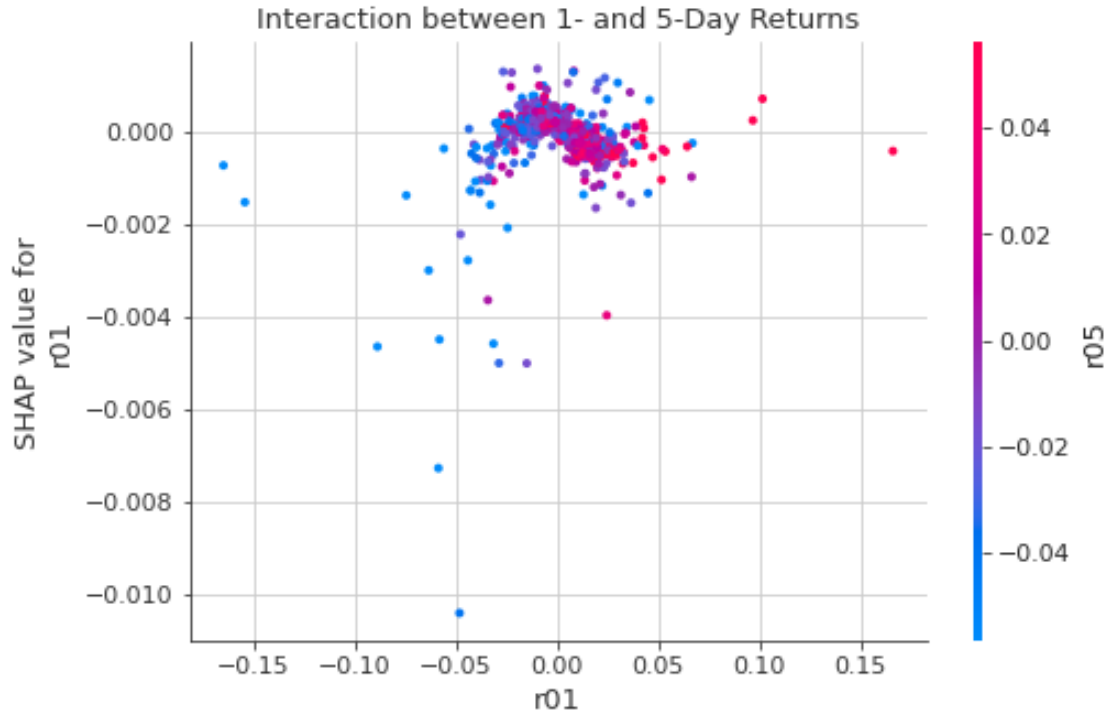
```
[34]: shap.force_plot(explainer.expected_value, shap_values[:1000,:], X.iloc[:1000])
```

```
[34]: <shap.plots._force.AdditiveForceArrayVisualizer at 0x7f2e6f776580>
```

1.8.4 Interaction Plot

Lastly, SHAP values allow us to gain additional insights into the interaction effects between different features by separating these interactions from the main effects. The `shap.dependence_plot` can be defined as follows. It displays how different values for 1-month returns (on the x axis) affect the outcome (SHAP value on the y axis), differentiated by 3-month returns:

```
[35]: shap.dependence_plot(ind='r01',
                          shap_values=shap_values,
                          features=X,
                          interaction_index='r05',
                          title='Interaction between 1- and 5-Day Returns')
```



SHAP values provide granular feature attribution at the level of each individual prediction, and enable much richer inspection of complex models through (interactive) visualization. The SHAP summary scatterplot displayed at the beginning of this section offers much more differentiated insights than a global feature-importance bar chart. Force plots of individual clustered predictions allow for more detailed analysis, while SHAP dependence plots capture interaction effects and, as a result, provide more accurate and detailed results than partial dependence plots.

The limitations of SHAP values, as with any current feature-importance measure, concern the attribution of the influence of variables that are highly correlated because their similar impact could be broken down in arbitrary ways.