06_model_interpretation

September 29, 2021

1 How to interpret GBM results

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

[8]:
```
%matplotlib inline
import warnings
from random import shuffle, randint
from time import time
import numpy as np
import pandas as pd
from itertools import product
from sklearn.metrics import roc_auc_score
from sklearn.externals import joblib
from sklearn.ensemble.partial_dependence import plot_partial_dependence,␣
 ↪partial_dependence
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
import shap
import xgboost as xgb
from xgboost.callback import reset_learning_rate
```

[9]:
```
warnings.filterwarnings('ignore')
plt.style.use('fivethirtyeight')
idx = pd.IndexSlice
np.random.seed(42)
```

1.2 Helper Functions

[21]:
```python
DATA_STORE = '../data/assets.h5'
```

[22]:
```python
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
```

[23]:
```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
```

[24]:
```
def get_one_hot_data(df, cols=('year', 'month', 'age', 'msize')):
    cols = list(cols)
    df = pd.get_dummies(df,
                        columns=cols + ['sector'],
                        prefix=cols + [''],
                        prefix_sep=['_'] * len(cols) + [''])
    return df.rename(columns={c: c.replace('.0', '').replace(' ', '_').lower()␣
 ↪for c in df.columns})
```

[25]:
```
def get_holdout_set(target, features, period=6):
    idx = pd.IndexSlice
    label = target.name
    dates = np.sort(target.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
```

1.3 Get Data

[26]:
```
y, X = get_data()
X = get_one_hot_data(X)
y_train, X_train, y_test, X_test = get_holdout_set(target=y, features=X)
```

[27]:
```
dtrain = xgb.DMatrix(label=y_train,
                     data=X_train,
                     nthread=-1)
dtest = xgb.DMatrix(label=y_test,
                    data=X_test,
                    nthread=-1)
```

1.4 Retrieve best model parameters

The results used below are created by the notebooks xgboost_lightgbm_catboost_tuning and xgboost_lightgbm_catboost_tuning_results that need to run first.

[28]:
```
with pd.HDFStore('results.h5') as store:
    results = store['xgboost/dummies']
best_model = results.iloc[0]
best_params = best_model.drop(['rounds', 'train', 'valid', 'time'])
```

[32]:
```
params = dict(
    booster='gbtree',
    objective='binary:logistic',
    eval_metric=['logloss', 'auc'],
#     tree_method='gpu_hist',
    max_depth=9,
    learning_rate=0.1,
    gamma=0,
    min_child_weight=1,
    max_delta_step=0,
    subsample=1,
    colsample_bytree=1,
    colsample_bylevel=1,
    reg_alpha=0,
    reg_lambda=1,
    silent=1,
    seed=42,
)
```

[33]:
```
params.update(best_params.to_dict())
```

1.4.1 Train XGBoost Model

[34]:
```
scores = {}
model = xgb.train(params=params,
                  dtrain=dtrain,
                  evals=list(zip([dtrain, dtest], ['train', 'test'])),
                  verbose_eval=False,
                  num_boost_round=int(best_model.rounds * 1.2),
                  early_stopping_rounds=None,
                  evals_result=scores)
```

[35]:
```
y_pred = model.predict(dtest)
roc_auc_score(y_true=y_test, y_score=y_pred)
```

[35]:
```
0.4872499688425391
```

1.5 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.

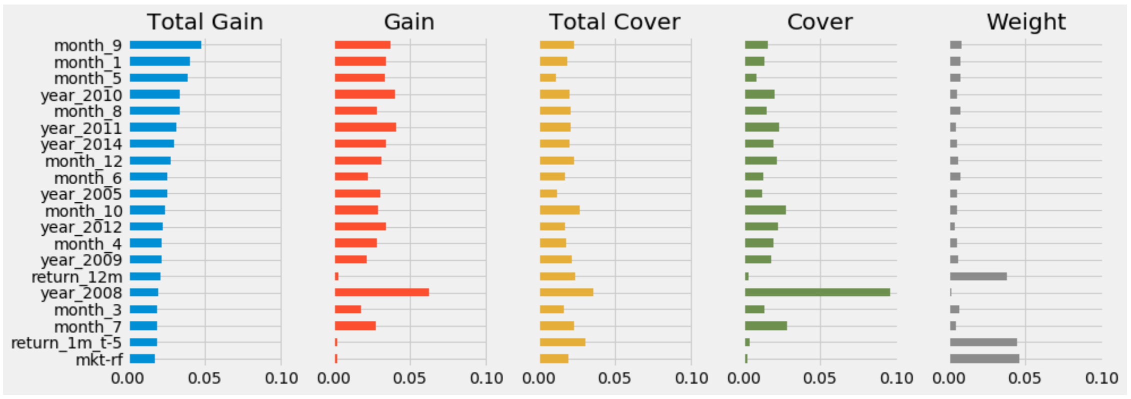
There are three primary ways to compute global feature importance values: - Gain: This classic approach introduced by Leo Breiman in 1984 uses the total reduction of loss or impurity contributed by all splits for a given feature. The motivation is largely heuristic, but it is a commonly used method to 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 XGBoost 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

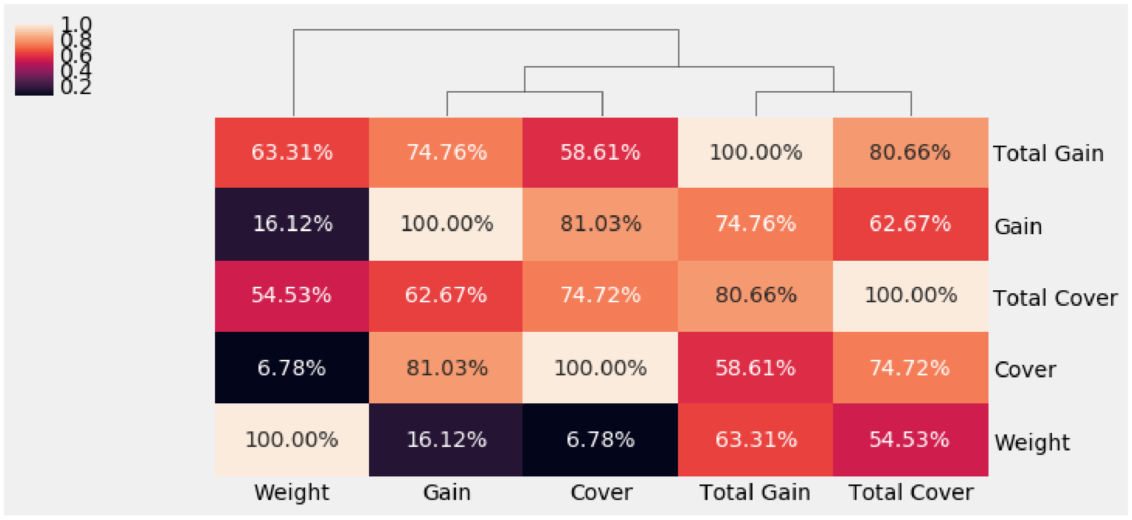
These values are available using the trained model's .get_score() method with the corresponding importance_type parameter. For the best performing XGBoost model, the results are as follows (the total measures have a correlation of 0.8, as do cover and total_cover):

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).

[38]:
```
fi_types = ['weight', 'gain', 'cover', 'total_gain', 'total_cover']
fi =pd.concat([pd.Series(model.get_score(importance_type=f)).to_frame(f) for f␣
 ↪in fi_types], axis=1)
fi = fi.div(fi.sum()).sort_values('total_gain', ascending=False)
fi = fi[['total_gain', 'gain','total_cover', 'cover', 'weight']]
fi.columns = [' '.join([c.capitalize() for c in t.split('_')]) for t in fi.
 ↪columns]
fi.head(20).sort_values('Total Gain').plot.barh(figsize=(14, 5), subplots=True,␣
 ↪layout=(1,5), sharey=True, legend=False)
plt.tight_layout();
```



```
[39]: corr = fi.corr('spearman')
      sns.clustermap(corr, annot=True, fmt='.2%', figsize=(10,5), row_cluster=False);
```



1.6 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 depends 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.

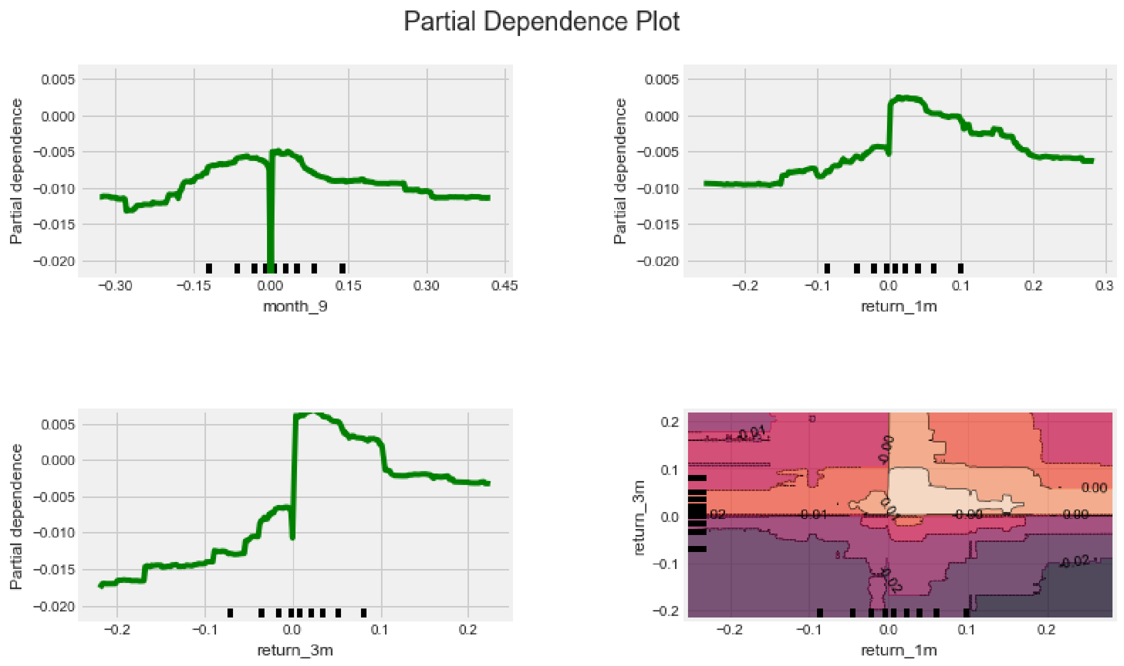
```
[120]: gb_clf = joblib.load('results/gb_gridsearch.joblib')
       best_model = gb_clf.best_estimator_
```

```
[116]: n_splits = 12
       y, features = get_data()
       X = get_one_hot_data(features).dropna()

       y, X, y_test, X_test = get_holdout_set(target=y,
                                              features=X)
```

1.6.1 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 [1%, 99%] percentiles. The month_9 variable is a dummy variable, hence the step-function-like plot.

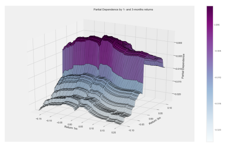
```
[121]: fig, axes = plot_partial_dependence(gbrt=best_model,
                                         X=X,
                                         features=['month_9', 'return_1m',␣
        ↪'return_3m', ('return_1m', 'return_3m')],
                                         feature_names=['month_9','return_1m',␣
        ↪'return_3m'],
                                         percentiles=(0.01, 0.99),
                                         n_jobs=-1,
                                         n_cols=2,
                                         grid_resolution=250)
       fig.suptitle('Partial Dependence Plot', fontsize=18)
       fig.tight_layout()
       fig.set_size_inches(12, 7)
       fig.savefig('partial_dependence', dpi=300);
```



1.6.2 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:

```
[133]: targets = ['return_1m', 'return_3m']
target_feature = [X.columns.get_loc(t) for t in targets]
pdp, axes = partial_dependence(best_model,
                               target_feature,
                               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)
surf = ax.plot_surface(XX, YY, Z,
                       rstride=1,
                       cstride=1,
                       cmap=plt.cm.BuPu,
                       edgecolor='k')
ax.set_xlabel(' '.join(targets[0].split('_')).capitalize())
ax.set_ylabel(' '.join(targets[1].split('_')).capitalize())
ax.set_zlabel('Partial Dependence')
ax.view_init(elev=22, azim=300)

fig.colorbar(surf)
fig.suptitle('Partial Dependence by 1- and 3-months returns')
fig.tight_layout()
fig.savefig('partial_dependence_3D', dpi=300);
```



1.7 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.

1.7.1 Summary Plot

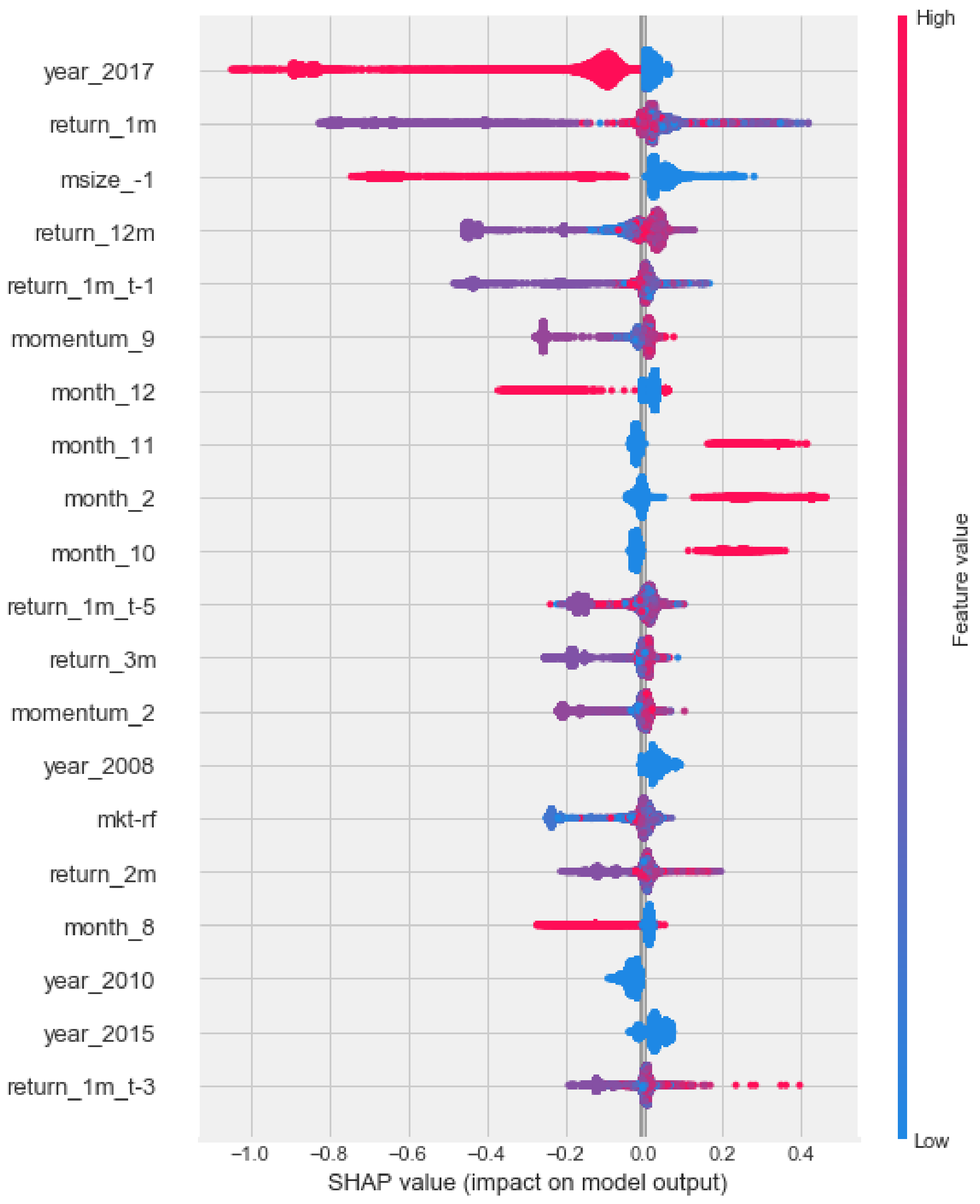
To get a high-level overview of the feature importance across a number of samples, there are two ways to plot the SHAP values: a simple average across all samples that resembles the global feature-importance measures computed previously (as shown in the second plot), or a scatter graph to display the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

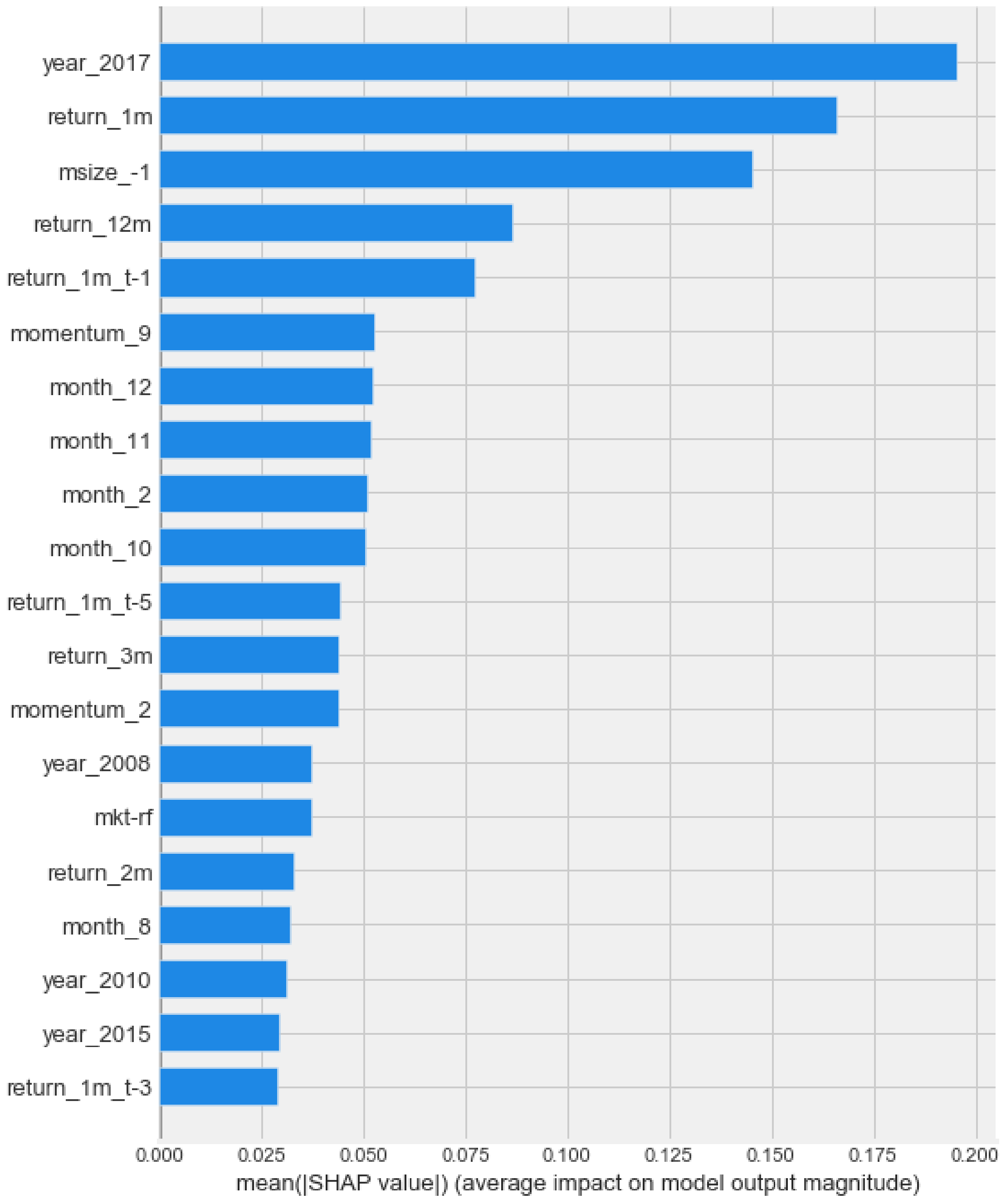
They are very straightforward to produce using a trained model of a compatible library and matching input data, as shown in the following code:

[143]:
```
# load JS visualization code to notebook
shap.initjs()

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

shap.summary_plot(shap_values, X_test, show=False)
plt.tight_layout()
plt.savefig('shap_dots', dpi=300)
```

```
<IPython.core.display.HTML object>
```



[144]:
```
shap.summary_plot(shap_values, X_test, plot_type="bar",show=False)
plt.tight_layout()
plt.savefig('shap_bar', dpi=300)
```



1.7.2 Feature Interaction

```
[154]: X_test.columns
```

```
[154]: Index(['return_1m', 'return_2m', 'return_3m', 'return_6m', 'return_9m',
              'return_12m', 'mkt-rf', 'smb', 'hml', 'rmw', 'cma', 'momentum_2',
              'momentum_3', 'momentum_6', 'momentum_9', 'momentum_12',
              'momentum_3_12', 'return_1m_t-1', 'return_1m_t-2', 'return_1m_t-3',
              'return_1m_t-4', 'return_1m_t-5', 'return_1m_t-6', 'year_2001',
              'year_2002', 'year_2003', 'year_2004', 'year_2005', 'year_2006',
              'year_2007', 'year_2008', 'year_2009', 'year_2010', 'year_2011',
              'year_2012', 'year_2013', 'year_2014', 'year_2015', 'year_2016',
              'year_2017', 'year_2018', 'month_1', 'month_2', 'month_3', 'month_4',
              'month_5', 'month_6', 'month_7', 'month_8', 'month_9', 'month_10',
              'month_11', 'month_12', 'age_-1', 'age_0', 'age_1', 'age_2', 'age_3',
              'age_4', 'age_5', 'msize_-1', 'msize_1', 'msize_2', 'msize_3',
              'msize_4', 'msize_5', 'msize_6', 'msize_7', 'msize_8', 'msize_9',
              'msize_10', 'basic_industries', 'capital_goods', 'consumer_durables',
              'consumer_non-durables', 'consumer_services', 'energy', 'finance',
              'health_care', 'miscellaneous', 'public_utilities', 'technology',
              'transportation', 'unknown'],
             dtype='object')
```

1.7.3 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:

[152]:
```
i = randint(0, len(X_test))
# visualize the first prediction's explanation
shap.force_plot(explainer.expected_value, shap_values[i,:], X_test.iloc[i,:])
```

[152]:
```
<IPython.core.display.HTML object>
```

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:

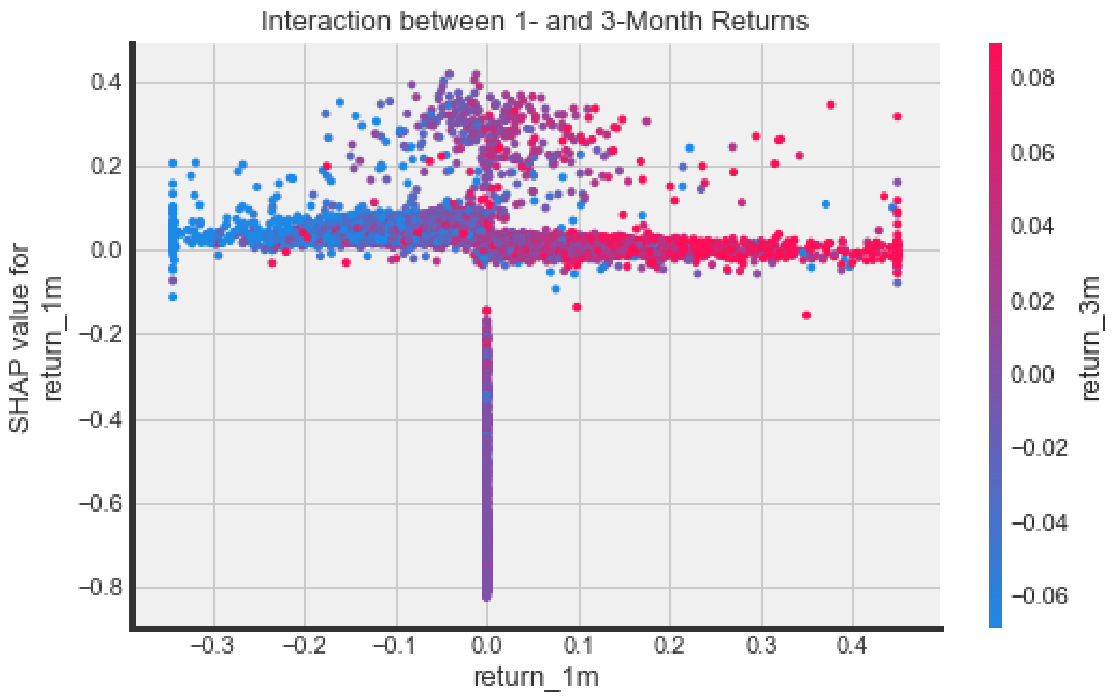
[135]:
```
shap.force_plot(explainer.expected_value, shap_values[:1000,:], X_test.iloc[:
 ↪1000])
```

[135]:
```
<IPython.core.display.HTML object>
```

1.7.4 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:

[157]:
```
shap.dependence_plot("return_1m", shap_values, X_test, interaction_index=2,␣
↪title='Interaction between 1- and 3-Month 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.