HW4

February 29, 2024

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import sklearn
import lightgbm as lgb
import scipy.optimize as optimize

pd.set_option('use_inf_as_na', True)
from collections import Counter
import _pickle as cPickle

from tqdm import tqdm # to measure progress of for loops
```

```
/var/folders/sp/wlr6xm29791 8vx6kjh2z1dk00000gn/T/ipykernel_81674/4031544042.py:1
2: FutureWarning: use_inf_as_na option is deprecated and will be removed in a
future version. Convert inf values to NaN before operating instead.
  pd.set_option('use_inf_as_na', True)
```

1 Problem 1

1.1 Random Forest

1.1.1 Long term performance with Random Forest Classifier, optimal hyper-parameters, and optimal features

[2]:
```
raw_data = pd.read_pickle(r'dataset.pkl')
data = raw_data[raw_data['market_cap'] > 1000.0]
data = data.copy()
data.fillna(0.0,inplace=True)
```

[3]:
```
def f(x):
    if x > 0.01:
        return 1
    elif x < -0.025:
```

```
        return -1
    else:
        return 0
```

[4]:
```
data['rel_performance'] = data['pred_rel_return'].apply(f)
data.reset_index(inplace=True)
data.set_index('date',inplace=True)
```

[5]:
```
from sklearn.preprocessing import MinMaxScaler,StandardScaler
scaler = StandardScaler()
```

[6]:
```
start_dates = [pd.to_datetime('2001-01-01') + pd.DateOffset(months = 3*i) for i␣
 ↪in range(57)]
end_dates = [d + pd.DateOffset(months = 36) for d in start_dates]
```

[7]:
```
training_frames = [data.loc[d:d+pd.DateOffset(months = 36)] for d in␣
 ↪start_dates]
valid_frames = [data.loc[d:d+pd.DateOffset(months = 3)] for d in end_dates]
test_frames = [data.loc[d+pd.DateOffset(months = 6):d+pd.DateOffset(months =␣
 ↪9)] for d in end_dates]
```

[8]:
```
training_data = [d.reset_index().drop
                                  (['ticker','date',
                                    'next_period_return',
                                    'spy_next_period_return',
                                    'rel_performance','pred_rel_return',
                                   'return', 'cum_ret', 'spy_cum_ret'],axis=1)␣
 ↪for d in training_frames]
```

[9]:
```
valid_data = [d.reset_index().drop
                                  (['ticker','date',
                                    'next_period_return',
                                    'spy_next_period_return',
                                    'rel_performance','pred_rel_return',
                                   'return', 'cum_ret', 'spy_cum_ret'],axis=1)␣
 ↪for d in valid_frames]
```

[10]:
```
test_data = [d.reset_index().drop(['ticker','date',
                                    'next_period_return',
                                    'spy_next_period_return',
                                    'rel_performance','pred_rel_return',
                                   'return', 'cum_ret', 'spy_cum_ret'],axis=1)␣
 ↪for d in test_frames]
```

[11]:
```
training_labels = [d['rel_performance'].values for d in training_frames]
valid_labels = [d['rel_performance'].values for d in valid_frames]
```

[12]:
```
for i in range(len(start_dates)-1):
    float_vars = [x for x in training_data[i].columns if data[x].dtype ==␣
 ↪'float64']

    scaler = StandardScaler()
    training_data[i] = training_data[i].copy()
    valid_data[i] = valid_data[i].copy()
    test_data[i] = test_data[i].copy()
    training_data[i][float_vars] = scaler.
 ↪fit_transform(training_data[i][float_vars])
    valid_data[i][float_vars] = scaler.transform(valid_data[i][float_vars])
    test_data[i][float_vars] = scaler.transform(test_data[i][float_vars])
```

1.1.2 Open and read the shap features over the holding period 2003 - 2018

[13]:
```
with open(r'Random Forest Parameters/shap_features.pkl','rb') as f:
    shap_rf = cPickle.load(f)
with open(r'Random Forest Parameters/optimal_hyperparameters.pkl','rb') as f:
    opt_hyper_params_rf = cPickle.load(f)
```

[14]:
```
# Convert shap_rf to a list of features
shap_rf_list = []
for feats in shap_rf:
    shap_rf_list.extend(list(feats))

# Find the 10 most common features
c = Counter(shap_rf_list)
c.most_common(10)
```

[14]:
```
[('fcf_yield', 42),
 ('cf_yield', 41),
 ('oancfy', 41),
 ('oancfy_q', 39),
 ('fcf_csfhdq', 37),
 ('lt_ppentq', 37),
 ('evmq', 36),
 ('dprq', 36),
 ('oepsxy', 35),
 ('dvpspq', 35)]
```

[15]:
```
# Choose 10 most common shap features to be the optimal features
opt_rf_feats = [val[0] for val in c.most_common(10)]
```

[16]:
```
# Initialize the list of classifiers with the optimal hyperparameters
rf_classifiers = []

for hyp_par in opt_hyper_params_rf:
```

```
    rf_clf = RandomForestClassifier(**hyp_par)
    rf_classifiers.append(rf_clf)
```

[17]:
```
start_dates = [pd.to_datetime('2001-01-01') + pd.DateOffset(months = 3*i) for i␣
 ↪in range(57)]
end_dates = [d + pd.DateOffset(months = 39) for d in start_dates]
```

[18]:
```
training_frames = [data.loc[d:d+pd.DateOffset(months = 39)] for d in␣
 ↪start_dates]
test_frames = [data.loc[d+pd.DateOffset(months = 3):d+pd.DateOffset(months =␣
 ↪6)] for d in end_dates]
```

[19]:
```
training_labels = [d['rel_performance'].values for d in training_frames]
```

[20]:
```
scalers = [StandardScaler() for i in range(len(start_dates)-1)]

opt_training_data = [pd.DataFrame(scalers[i].
 ↪fit_transform(training_frames[i][opt_rf_feats].values),columns=opt_rf_feats)␣
 ↪for i in range(len(start_dates)-1)]
opt_test_data = [pd.DataFrame(scalers[i].transform(test_frames[i][opt_rf_feats].
 ↪values),columns=opt_rf_feats) for i in range(len(start_dates)-1)]
```

[21]:
```
opt_test_data[1]
```

[21]:
```
      fcf_yield  cf_yield    oancfy  oancfy_q  fcf_csfhdq  lt_ppentq  \
0     -0.206398 -0.645950 -0.171431 -0.142220   -0.222443  -0.138036
1     -0.420684  2.792597  0.852188  0.210787   -0.359797  -0.137840
2     -0.020692 -0.102749 -0.051488 -0.110094   -0.118421  -0.143237
3      0.049269 -0.199209 -0.200669 -0.110826    0.041014  -0.126797
4      0.006049 -0.332625  1.494588  1.340345    0.089421  -0.136787
…           …         …         …         …           …          …
1131  -0.062845 -0.275955 -0.011859  0.030989   -0.150327  -0.125497
1132  -0.036429 -0.386222 -0.195607 -0.156680    0.068950  -0.122519
1133  -0.242139  0.414694 -0.090863  0.021878   -0.311895  -0.146362
1134  -0.032373 -0.453909 -0.213380 -0.163768    0.006329  -0.099544
1135  -0.081537 -0.517354 -0.191139 -0.168606   -0.252757  -0.135636

          evmq      dprq    oepsxy    dvpspq
0     0.080339 -0.042204 -0.093053 -0.595682
1    -1.060025 -0.112661 -0.578317 -0.408904
2     0.013442 -0.006715  0.215752 -0.198778
3     0.011170 -0.030968  0.107470 -0.595682
4     0.018215  0.007137  0.307992  0.478292
…          …         …         …         …
1131 -0.007114  0.091678 -0.133157  0.618376
1132  0.014597 -0.023044  0.099449 -0.408904
1133  0.002207 -0.042204 -0.297586 -0.595682
```

```
1134  0.015523 -0.042204  0.051323 -0.595682
1135  0.044254 -0.042204 -0.301596 -0.595682

[1136 rows x 10 columns]
```

[22]:
```
x_rf = [1]
ret_rf = []

for i in tqdm(range(len(start_dates)-1)):
        rf_classifiers[i].fit(opt_training_data[i],training_labels[i])
        preds = rf_classifiers[i].predict(opt_test_data[i])
        profit_i = (preds*test_frames[i]['next_period_return']).sum()
        ret_rf.append(profit_i)
        num_names = len(opt_test_data[i])
        x_rf.append(x_rf[i] + (x_rf[i]/num_names)*profit_i)
```

```
100%|                         | 56/56 [00:12<00:00,  4.45it/s]
```

1.2 Gradient Boosting

1.2.1 Long term performance with Gradient Boosting Classifier, and optimal shapley features

[23]:
```
def f(x):
    if x > 0.01:
        return 1
    elif x < -0.01:
        return -1
    else:
        return 0
```

[24]:
```
data['rel_performance'] = data['pred_rel_return'].apply(f)
data.reset_index(inplace=True,)
data.set_index('date',inplace=True)
```

[25]:
```
from sklearn.preprocessing import MinMaxScaler,StandardScaler
scaler = StandardScaler()
```

1.2.2 Open and read the shap features for the Gradient Boosting classifiers

[26]:
```
with open(r'Gradient Boosting Parameters/shap_features_gb_clf_01.pkl','rb') as␣
↪f:
    shap_gb = cPickle.load(f)
```

[27]:
```
# Convert shap_gb to a list of features
shap_gb_list = []
for feats in shap_gb:
    shap_gb_list.extend(list(feats))
```

```python
# Find the 10 most common features
c = Counter(shap_gb_list)
c.most_common(10)
```

[27]:
```
[('fcf_yield', 47),
 ('cf_yield', 47),
 ('evmq', 42),
 ('lt_ppentq', 42),
 ('oancfy', 39),
 ('opmbdq', 38),
 ('rect_turnq', 38),
 ('rectq', 37),
 ('fcf_csfhdq', 37),
 ('oepsxy', 36)]
```

[28]:
```python
# Choose 10 most common shap features to be the optimal features
opt_gb_feats = [val[0] for val in c.most_common(10)]
```

1.2.3 Open and read gradient boosting classifiers with optimal hyperparameters

[29]:
```python
with open(r'Gradient Boosting Parameters/classifiers_gb_clf_01.pkl','rb') as f:
    gb_clfs = cPickle.load(f)

# Add a parameter to surpress warnings
for i in range(len(gb_clfs)):
    gb_clfs[i].set_params(verbose=-1)
```

```
/Users/shri/miniconda3/lib/python3.11/site-packages/sklearn/base.py:348:
InconsistentVersionWarning: Trying to unpickle estimator LabelEncoder from
version 1.3.0 when using version 1.3.1. This might lead to breaking code or
invalid results. Use at your own risk. For more info please refer to:
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-
limitations
  warnings.warn(
```

[30]:
```python
gb_clfs[0]
```

[30]:
```
LGBMClassifier(learning_rate=0.18000000000000002, min_data_in_leaf=1200,
               n_estimators=80, num_leaves=21,
               reg_lambda=4.1926760590611446e-05, verbose=-1)
```

[31]:
```python
start_dates = [pd.to_datetime('2001-01-01') + pd.DateOffset(months = 3 * i) for␣
↪i in range(57)]
end_dates = [d + pd.DateOffset(months = 39) for d in start_dates]
```

[32]:
```python
training_data = [d.reset_index().drop
                                 (['ticker','date',
```

```
                                    'next_period_return',
                                    'spy_next_period_return',
                                    'rel_performance','pred_rel_return',
                                   'return', 'cum_ret', 'spy_cum_ret'],axis=1)␣
  ↪for d in training_frames]

test_data = [d.reset_index().drop(['ticker','date',
                                    'next_period_return',
                                    'spy_next_period_return',
                                    'rel_performance','pred_rel_return',
                                   'return', 'cum_ret', 'spy_cum_ret'],axis=1)␣
  ↪for d in test_frames]
```

[33]:
```
training_labels = [d['rel_performance'].values for d in training_frames]
```

[34]:
```
opt_training_data = [t[opt_gb_feats] for t in training_data]
opt_test_data = [v[opt_gb_feats] for v in test_data]
```

[35]:
```
training_frames = [data.loc[d:d+pd.DateOffset(months = 39)] for d in␣
  ↪start_dates]
test_frames = [data.loc[d+pd.DateOffset(months = 3):d+pd.DateOffset(months =␣
  ↪6)] for d in end_dates]
```

[36]:
```
training_labels = [d['rel_performance'].values for d in training_frames]
```

[37]:
```
scalers = [StandardScaler() for i in range(len(start_dates)-1)]

opt_training_data = [pd.DataFrame(scalers[i].
  ↪fit_transform(training_frames[i][opt_gb_feats].values),columns=opt_gb_feats)␣
  ↪for i in range(len(start_dates)-1)]
opt_test_data = [pd.DataFrame(scalers[i].transform(test_frames[i][opt_gb_feats].
  ↪values),columns=opt_gb_feats) for i in range(len(start_dates)-1)]
```

[38]:
```
opt_test_data[0].head()
```

[38]:
```
   fcf_yield  cf_yield      evmq  lt_ppentq    oancfy    opmbdq  rect_turnq  \
0   0.215735  0.006661  0.024102  -0.100428 -0.239974  0.079469   -0.034211
1   0.065261 -0.108632  0.009218  -0.121176  0.433740  0.038632   -0.021523
2   0.383129  0.263649  0.010836  -0.142092 -0.189594  0.045681   -0.035822
3  -0.059195  0.862655  0.001162  -0.147453  0.470706  0.056214   -0.020043
4  -0.245033 -0.713570 -0.005808  -0.156643 -0.241345 -0.011360   -0.039469

      rectq  fcf_csfhdq    oepsxy
0 -0.187885    0.022842 -0.403834
1 -0.156410    0.161560  0.706011
2 -0.081546    0.405495  0.253565
3 -0.046906   -0.104874  0.075681
```

```
4  2.572147   -0.310581  0.241964
```

[39]:
```
P_L_gb = []
x_gb = [1]
ret_gb = []

for i in tqdm(range(len(start_dates)-1)):
        gb_clfs[i].fit(opt_training_data[i],training_labels[i])
        pred_i = gb_clfs[i].predict(opt_test_data[i])
        profit_i = (pred_i * test_frames[i]['next_period_return']).sum()
        P_L_gb.append(profit_i)
        num_positions = len(pred_i)
        ret_gb.append((1.0/num_positions) * profit_i)
        x_gb.append(x_gb[i] + (x_gb[i]/num_positions) * profit_i)
```

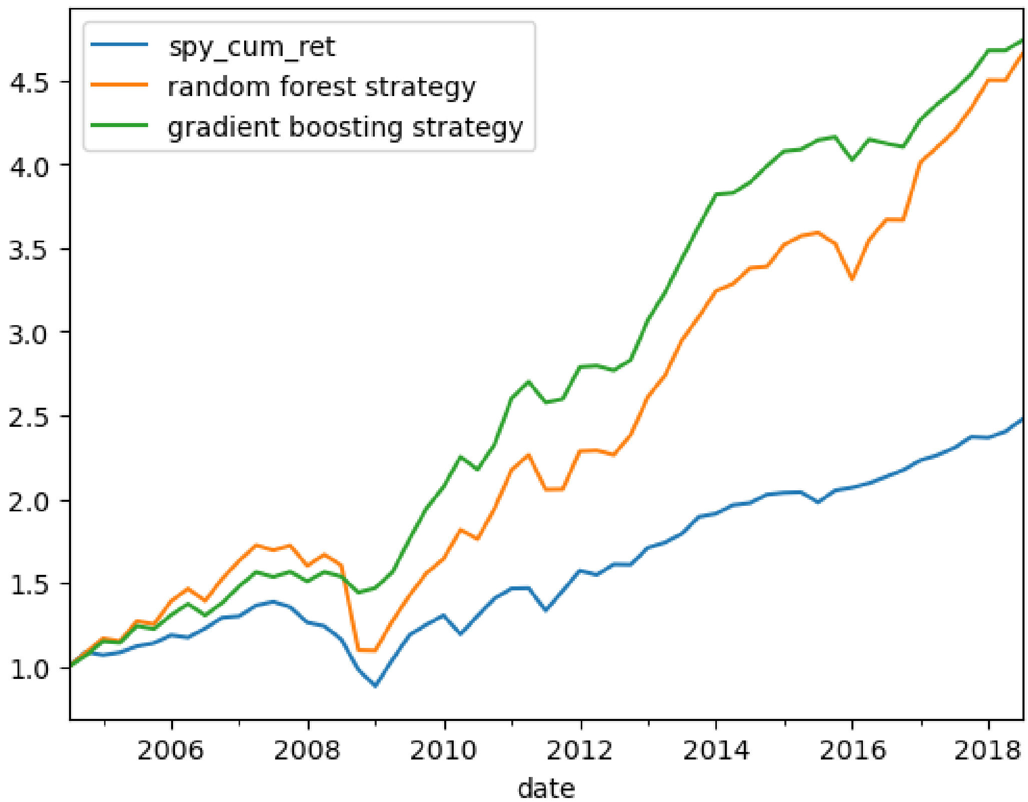
```
100%|                        | 56/56 [00:08<00:00,  6.98it/s]
```

1.3 Plot the P/L curves and compute the Sharpe Ratios for all strategies

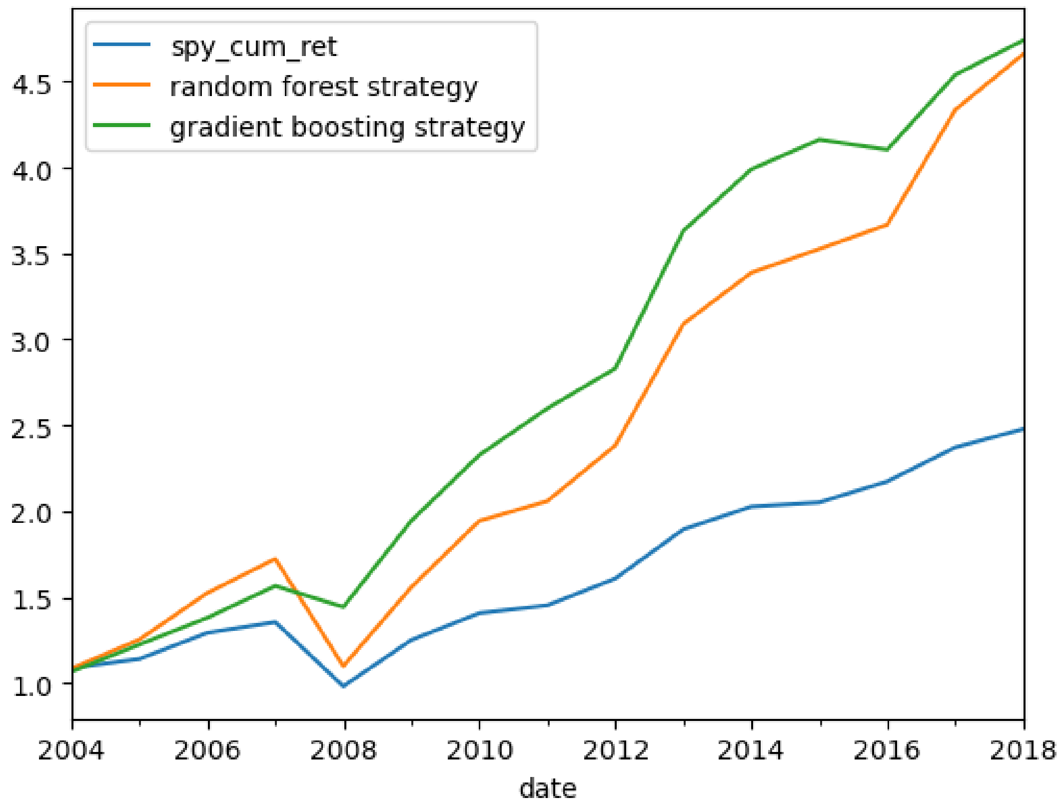
[40]:
```
SPY = pd.read_pickle(r'SPY_cum_ret.pkl')
SPY = SPY.loc['2004-07-01':'2018-09-30']
SPY = SPY.resample('Q').ffill()
SPY['spy_cum_ret'] = (SPY['spy_cum_ret'] - SPY['spy_cum_ret'][0] + 1)

SPY['random forest strategy'] = x_rf
SPY['gradient boosting strategy'] = x_gb
```

```
/var/folders/sp/wlr6xm297918vx6kjh2z1dk00000gn/T/ipykernel_81674/3282873245.py:4
: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In
a future version, integer keys will always be treated as labels (consistent with
DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  SPY['spy_cum_ret'] = (SPY['spy_cum_ret'] - SPY['spy_cum_ret'][0] + 1)
```

[41]:
```
SPY.plot();
```



[42]:
```
SPY = SPY.resample('Y').ffill()
SPY.plot();
```



[43]:

```
spy_mean_ret = (SPY['spy_cum_ret'] - 1).diff().mean()
spy_std = (SPY['spy_cum_ret'] - 1).diff().std()
print('SPY Sharpe Ratio                              : ',spy_mean_ret/spy_std)

strategy_mean_ret = (SPY['random forest strategy'] - 1).diff().mean()
strategy_std = (SPY['random forest strategy'] - 1).diff().std()
strategy_sr = strategy_mean_ret/strategy_std
print('Random Forest Strategy Sharpe Ratio    : ',strategy_sr)

strategy_mean_ret = (SPY['gradient boosting strategy'] - 1).diff().mean()
strategy_std = (SPY['gradient boosting strategy'] - 1).diff().std()
strategy_sr = strategy_mean_ret/strategy_std
print('Gradient Boosting Strategy Sharpe Ratio: ',strategy_sr)
```

```
SPY Sharpe Ratio                        :  0.6324646748042164
Random Forest Strategy Sharpe Ratio     :  0.8151625107247469
Gradient Boosting Strategy Sharpe Ratio:  1.1367787320017024
```

So we see that the final returns of our both our strategies are better than that of the SPY, with both above 4.5 at the end of our testing period.

Further, the Sharpe Ratios of both our strategies our better, but it is evident that the Gradient

Boosting classifier performed the best with a Sharpe Ratio of approximately 1.14.

2 Problem 2

2.1 Combining strategies by choosing weights

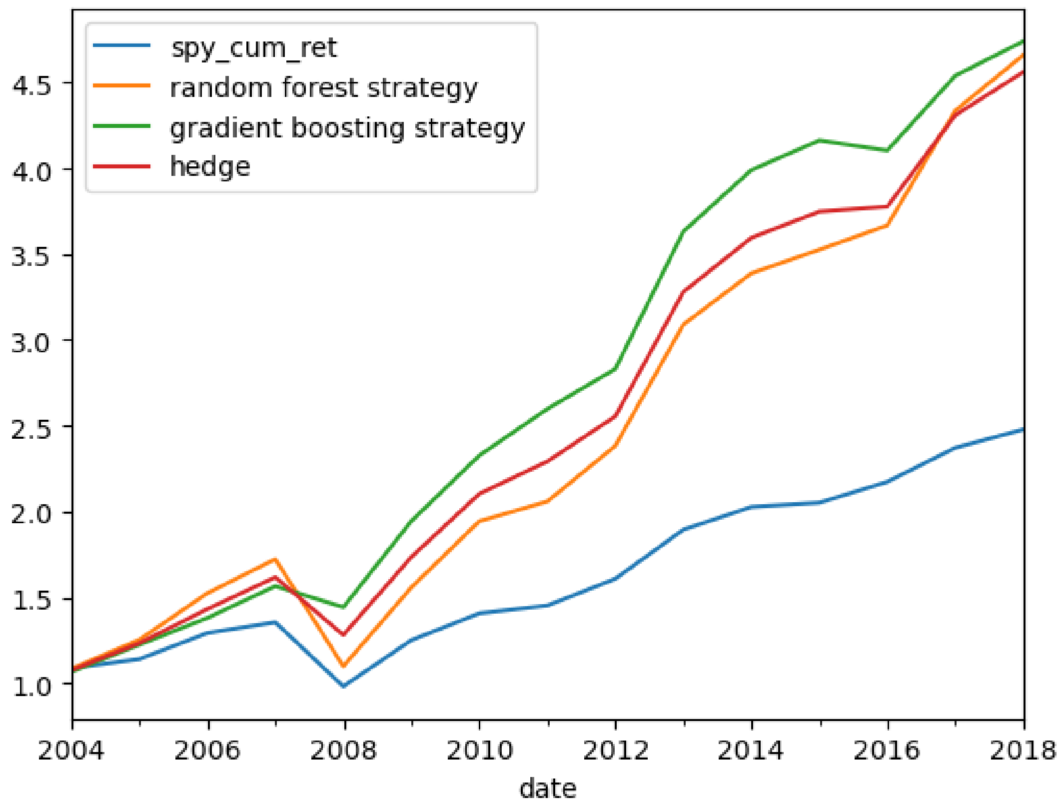
2.1.1 Next we will try to combine our existing strategies to find one with the best Sharpe Ratio.

For example if we naively choose weights: - 0.5 for gradient boosting - 0.5 for random forest strategy - -0.1 for SPY (indicating that we are shorting it)

We obtain the following:

[44]:
```
SPY['hedge'] = 0.5*SPY['gradient boosting strategy'] + 0.5*SPY['random forest␣
 ↪strategy'] - 0.1 * (SPY['spy_cum_ret'] - SPY['spy_cum_ret'].iloc[0])
SPY.plot()

strategy_mean_ret = (SPY['hedge'] - 1).diff().mean()
strategy_std = (SPY['hedge'] - 1).diff().std()
strategy_sr = strategy_mean_ret/strategy_std
print('Hedge Strategy Sharpe Ratio: ',strategy_sr)
```

```
Hedge Strategy Sharpe Ratio:  1.0124814959394992
```



2.1.2 However, we can try to find the weights that give us the best Sharpe ratio by using the optimize function in Scipy:

[45]:
```
def weighted_hedge(weights):
    w1 = weights[0]
    w2 = weights[1]
    # w1 corresponds to the weight for the gradient boosted strategy
    # 1-w1 corresponds to the weight for the random forest strategy
    # w2 corresponds to the negative weight (i.e. shorting) for the SPY

    # Combine the strategies according to the weights
    SPY['hedge'] = w1 * SPY['gradient boosting strategy'] + (1-w1) *␣
 ↪SPY['random forest strategy'] + w2 * (SPY['spy_cum_ret'] -␣
 ↪SPY['spy_cum_ret'].iloc[0])

    # Compute the negative Sharpe ratio (As we need to minimize this)
    strategy_mean_ret = (SPY['hedge'] - 1).diff().mean()
    strategy_std = (SPY['hedge'] - 1).diff().std()
    strategy_sr = strategy_mean_ret/strategy_std
    return -strategy_sr
```

[46]:
```
initial_guess = [0.5, -0.1]
result = optimize.minimize(weighted_hedge, initial_guess)
result
```

[46]:
```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: -1.1921177528413918
        x: [ 7.184e-01 -8.103e-01]
      nit: 10
      jac: [ 1.490e-07  8.643e-07]
 hess_inv: [[ 2.103e+00  2.398e+00]
            [ 2.398e+00  3.800e+00]]
     nfev: 36
     njev: 12
```

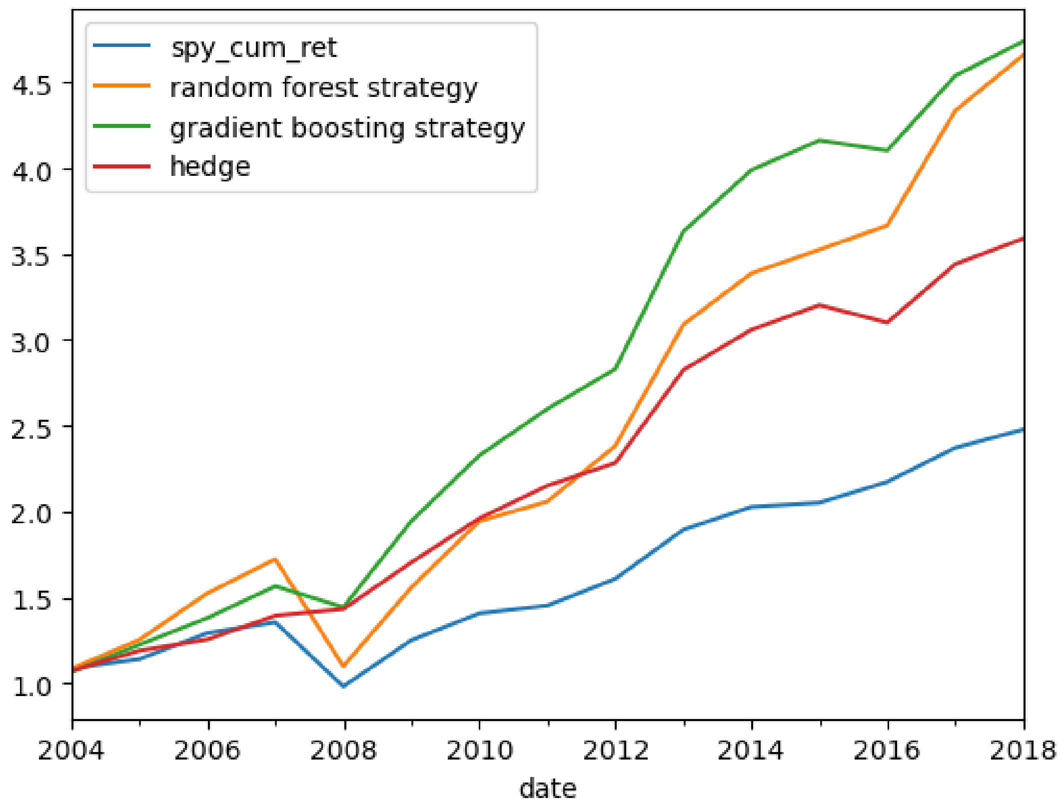
[47]:
```
w1 = result.x[0]
w2 = result.x[1]
print(f" Weight for gradient boosting strategy: {w1}")
print(f" Weight for random forest strategy    : {1-w1}")
print(f" Weight for shorting SPY              : {w2}")
```

```
 Weight for gradient boosting strategy: 0.7184243263033968
 Weight for random forest strategy    : 0.2815756736966032
 Weight for shorting SPY              : -0.8103018464035477
```

2.1.3 Using the optimal weights:

[48]:
```
SPY['hedge'] = w1*SPY['gradient boosting strategy'] + (1-w1)*SPY['random forest␣
↪strategy'] + w2 * (SPY['spy_cum_ret'] - SPY['spy_cum_ret'].iloc[0])
SPY.plot()

strategy_mean_ret = (SPY['hedge'] - 1).diff().mean()
strategy_std = (SPY['hedge'] - 1).diff().std()
strategy_sr = strategy_mean_ret/strategy_std
print('Hedge Strategy Sharpe Ratio: ',strategy_sr)
```

```
Hedge Strategy Sharpe Ratio:  1.1921177528413918
```



2.1.4 Conclusion

So by combining all three of the strategies, we have found a strategy that has a higher Sharpe Ratio (approximately 1.2) than each of them individually.

Although the combined strategy has a lower return at the end of the period compared to our two classifier strategies, it still outperforms the SPY.
In addition, having a higher Sharpe ratio is more desirable for investors who may be more risk averse.

If we wanted to explore further, we could look at other metrics such as such as the alpha and the information ratio. In addition, we could try using a neural network model in order to capture intricacies in the data that were not accounted for and evaluate the performance against the existing models.