Machine Learning

Lecture 7a

Let’s apply the Adaboost algorithm to our data set

```
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, AdaBoostClassifier
pd.set_option('use_inf_as_na', True)
from collections import Counter
```

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

```
C:\Users\niels\Anaconda3\lib\site-packages\pandas\core\frame.py:3790: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy
  downcast=downcast, **kwargs)
```

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

        return 0
```

class `sklearn.ensemble.` **`AdaBoostClassifier`** (*base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *algorithm='SAMME.R'*, *random_state=None*) [source]

An AdaBoost classifier.

An AdaBoost [1] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

This class implements the algorithm known as AdaBoost-SAMME [2].

As a base estimator we choose a DecisionTreeClassifier with max_depth = 4

```
ada_clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=4),n_estimators=25)
```

It takes a bit of time to train the classifier on the full data set

```
ada_clf.fit(train_1,y_1)
```

```
AdaBoostClassifier(algorithm='SAMME.R',
          base_estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=4,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          learning_rate=1.0, n_estimators=25, random_state=None)
```

```
%timeit ada_clf.fit(train_1,y_1)
```

```
30.8 s ± 1.96 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Feature Importances

```python
def adaboost_feat_importances(m, df):

    return pd.DataFrame({'cols':df.columns, 'feat_imp': m.feature_importances_}
                       ).sort_values('feat_imp', ascending=False)

def plot_fi(fi): return fi.plot('cols', 'feat_imp', 'barh', figsize=(12,7), legend=False)
```

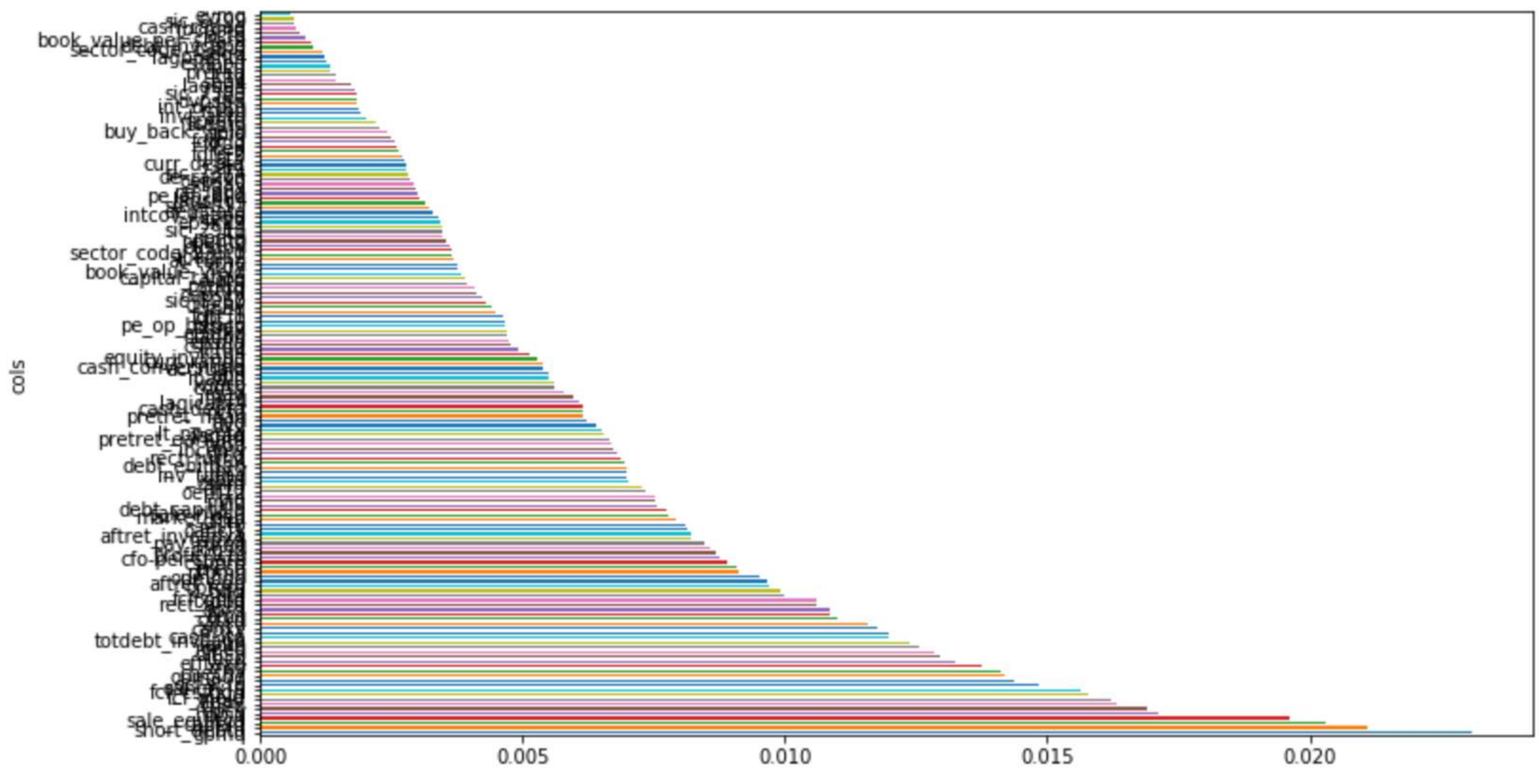
```python
fi = adaboost_feat_importances(ada_clf,train_1)
```

```python
features = fi[(fi['feat_imp'] > 0.0)]
features.shape
```

```
(122, 2)
```

```python
plot_fi(features);
```

```
1 plot_fi(features);
```



```
features
```

	cols	feat_imp
112	gpmq	0.024416
148	short_debtq	0.021091
99	capeiq	0.020853
174	sale_equityq	0.019603
36	nopiq	0.017122
71	dpcy	0.016910
4	cheq	0.016338
155	fcf_csfhdq	0.015632
156	fcf_yield	0.015610
102	oancfy_q	0.014830
152	ocf_lctq	0.014370
111	opmadq	0.014169
66	chechy	0.014123
82	xidoy	0.014021
49	rectq	0.014010
75	miiy	0.013920
120	offtaxq	0.013235
138	debt_invcapq	0.000983
90	book_value_per_share	0.000875
0	actq	0.000767
27	ibcomq	0.000682
165	cash_ratioq	0.000650
540	sic_6799	0.000633
95	evmq	0.000572

153 rows × 2 columns

We cut down to the features that actually have an influence on the classifier

```
train_1 = train_1[features['cols'].values]
valid = valid[features['cols'].values]
```

```
%timeit ada_clf.fit(train_1,y_1)
ada_clf.score(train_1,y_1)
```

```
24 s ± 714 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
0.5565183580514359
```

```
pred_valid = ada_clf.predict(valid)
ada_clf.score(valid,y_valid)
```

```
0.44457142857142856
```

```
(pred_valid * valid_stock_returns).sum()
```

```
27.26727199999997
```

```
Counter(pred_valid)
```

```
Counter({1: 1110, 0: 56, -1: 584})
```

Profit Importance

```python
def profit_importance(m,df,rets):
#     np.random.seed(123)
    profit = []
    for col in df.columns:
        prof = []
        for _ in range(10):
            X = df.copy()
            X[col] = np.random.permutation(X[col].values)
            prediction = m.predict(X)
            prof.append((prediction * rets).sum())
        profit.append(np.mean(prof))
    return profit
```

```python
def adaboost_profit_importance(m, df,rets):
    return pd.DataFrame({'cols':df.columns, 'pi_imp':profit_importance(m,df,rets)}
                        ).sort_values('pi_imp', ascending=True)
```

```python
pi = adaboost_profit_importance(ada_clf,valid,valid_stock_returns)
pi
```

	cols	pi_imp
50	piq	17.245805
6	cheq	17.895002
80	equity_invcapq	22.199241
74	xoprq	22.533351
96	atq	23.067195

Finding the features that optimize the strategy profit on the validation set

```
profits = []
feat=[]

train = train_1.copy()
validation = valid.copy()

while len(train.columns)>1:

    pred_valid = ada_clf.predict(validation)

    print((pred_valid * df_valid['next_period_return']).sum())
    profits.append((pred_valid * df_valid['next_period_return']).sum())
    feat.append(train.columns)

    col_to_drop = pi.iloc[-1]['cols']
    train.drop(col_to_drop,axis=1,inplace=True)
    validation.drop(col_to_drop,axis=1,inplace=True)

    ada_clf.fit(train,y_1)
    pi = adaboost_profit_importance(ada_clf,validation,df_valid['next_period_return'])
```

```
27.267271999999997
```

One of the interesting things about AdaBoosting is that we do not have to use just trees as base estimators. Here is an example boosting a RandomForest

```
rf_clf = RandomForestClassifier(n_estimators=15,min_samples_leaf=2400)
```

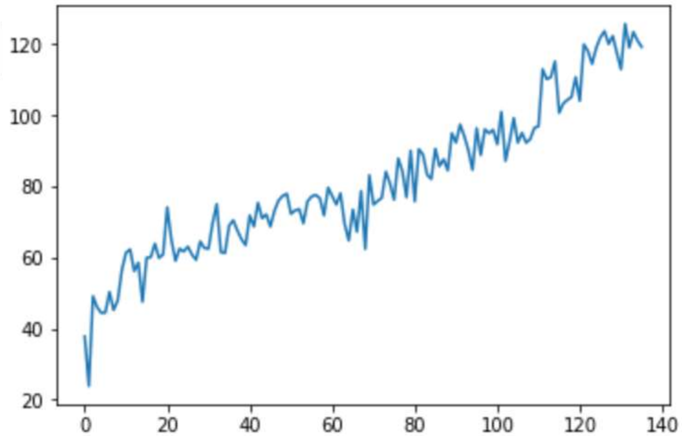
```
ada_clf = AdaBoostClassifier(rf_clf,n_estimators=25)
```

```
%timeit ada_clf.fit(train_1,y_1)
```

```
7.87 s ± 283 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```python
plt.plot(profits);
```

hide




```python
optim_feats = feat[n]
optim_feats
```

```
Index(['cstkq', 'ibadj12', 'fcf_ocfq', 'cshfdq', 'cash_conversionq', 'cshopq'], dtype='object')
```

```python
train_1_optim = train_1[optim_feats]
valid_optim = valid[optim_feats]


ada_clf.fit(train_1_optim,y_1)
print(ada_clf.score(train_1_optim,y_1))
pred_valid_tree = ada_clf.predict(valid_optim)
print(ada_clf.score(valid_optim,y_valid))
(pred_valid_tree * valid_stock_returns).sum()
```

```
0.5075820696550575
0.532
```

]: 119.12096800000003

```python
train_2_tree = train_2[optim_feats]
test_tree = test[optim_feats]
ada_clf.fit(train_2_tree,y_2)
pred_test_tree = ada_clf.predict(test_tree)
(pred_test_tree * test_stock_returns).sum()
```

]: 61.964654999999965

```python
train_1_optim = train_1[optim_feats]
valid_optim = valid[optim_feats]


ada_clf.fit(train_1_optim,y_1)
print(ada_clf.score(train_1_optim,y_1))
pred_valid_tree = ada_clf.predict(valid_optim)
print(ada_clf.score(valid_optim,y_valid))
(pred_valid_tree * valid_stock_returns).sum()
```

```
0.5075820696550575
0.532
```

]: 119.12096800000003

```python
train_2_tree = train_2[optim_feats]
test_tree = test[optim_feats]
ada_clf.fit(train_2_tree,y_2)
pred_test_tree = ada_clf.predict(test_tree)
(pred_test_tree * test_stock_returns).sum()
```

]: 61.964654999999965

```python
train_1_optim = train_1[optim_feats]
valid_optim = valid[optim_feats]


ada_clf.fit(train_1_optim,y_1)
print(ada_clf.score(train_1_optim,y_1))
pred_valid_tree = ada_clf.predict(valid_optim)
print(ada_clf.score(valid_optim,y_valid))
(pred_valid_tree * valid_stock_returns).sum()
```

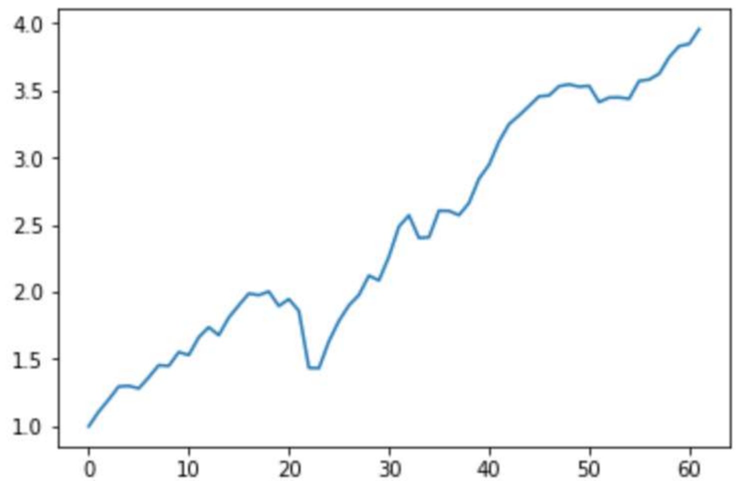
```
0.5075820696550575
0.532
```

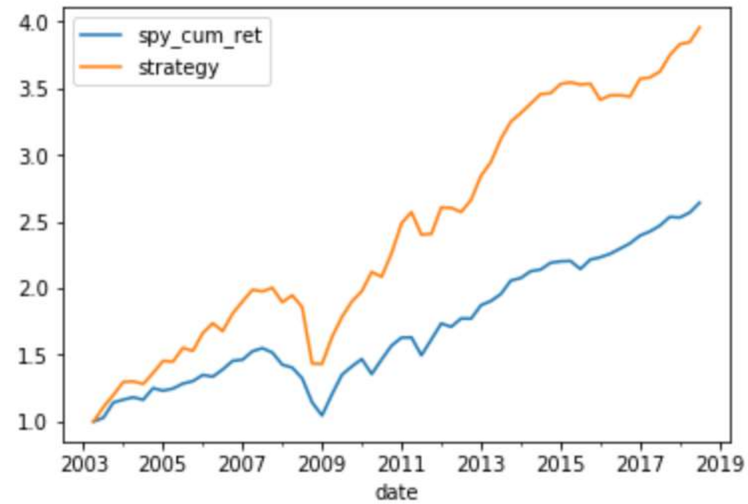
]: 119.12096800000003

```python
train_2_tree = train_2[optim_feats]
test_tree = test[optim_feats]
ada_clf.fit(train_2_tree,y_2)
pred_test_tree = ada_clf.predict(test_tree)
(pred_test_tree * test_stock_returns).sum()
```

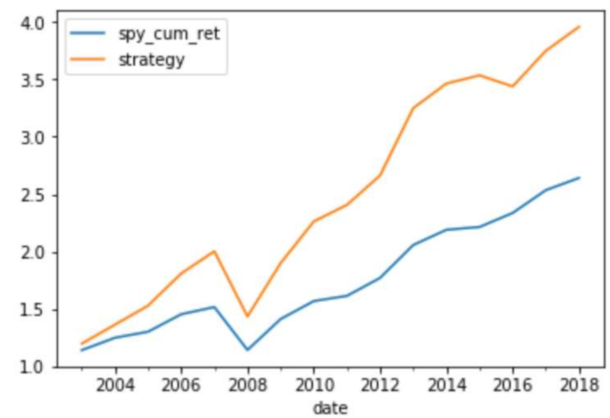
]: 61.964654999999965

```
plt.plot(x);
```

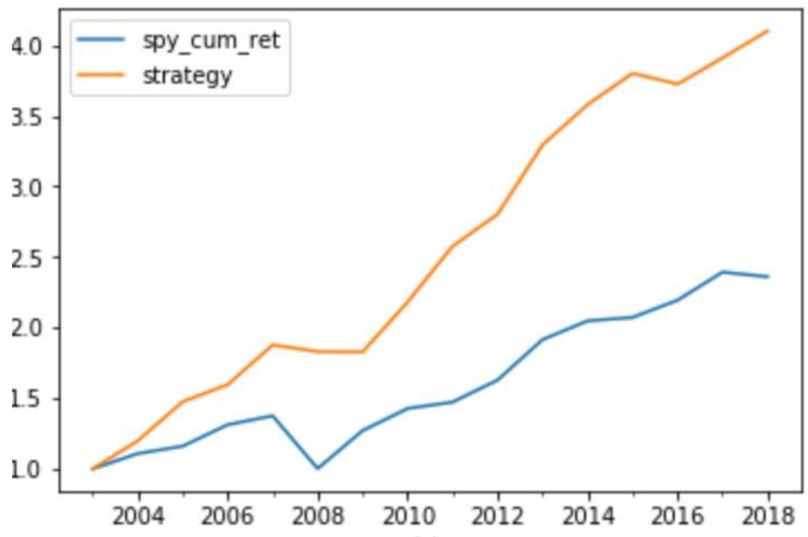





```
SPY = SPY.resample('Y').ffill()
SPY.plot();
```



Variable optimal features




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

```
Strategy Sharpe Ratio:  1.3047801841589615
```

```
strategy_ret = (SPY['strategy'] - 1).diff().values[1:]
spy_ret = (SPY['spy_cum_ret'] - 1).diff().values[1:]

beta = (np.cov(spy_ret,strategy_ret)/np.var(spy_ret))[1,0]
beta
```

```
0.403773768479095
```

```
residual_ret = strategy_ret - beta * spy_ret
IR = np.mean(residual_ret)/np.std(residual_ret)
IR
```

```
1.200388421384077
```

```
alpha = np.mean(residual_ret)
alpha
```

```
0.16120032434776224
```