

# Machine Learning

## Lecture 2

We can now begin to set up training, validation and test data and train a decision tree on our data.

We will first set up training data consisting of a number of years of data, using the relative return as a way to make labels. We then try out the trained model on the validation set.

We aim to maximize the performance on the validation set.

When we are satisfied with the validation performance we run the model on the test set

## Loading the Data Set (you need to put in the file where you have stored the data)

```
1 raw_data = pd.read_pickle(r'C:\Users\niels\data\dataset.pkl')
```

## Restricting to Companies with Market Cap > 1 Billion

```
1 data = raw_data[raw_data['market_cap'] > 1000.0]
```

## The Total Number of Companies w/ Market Cap > 1 Billion that appear during our time horizon

```
1 len(data.index.get_level_values(1).unique())
```

[4]: 4076

## Filling in Missing Values

```
1 data.fillna(0.0,inplace=True)
```

We fill all missing values with 0.0

```
▶ data.fillna(0.0,inplace=True)
```

Next we want to define labels using the `pred_rel_return`. We assign label 1 if the `pred_rel_return` > 1%, -1 if it is < -1% and 0 if it is between -1% and 1%

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

```
▶ data['rel_performance'] = data['pred_rel_return'].apply(f)
```

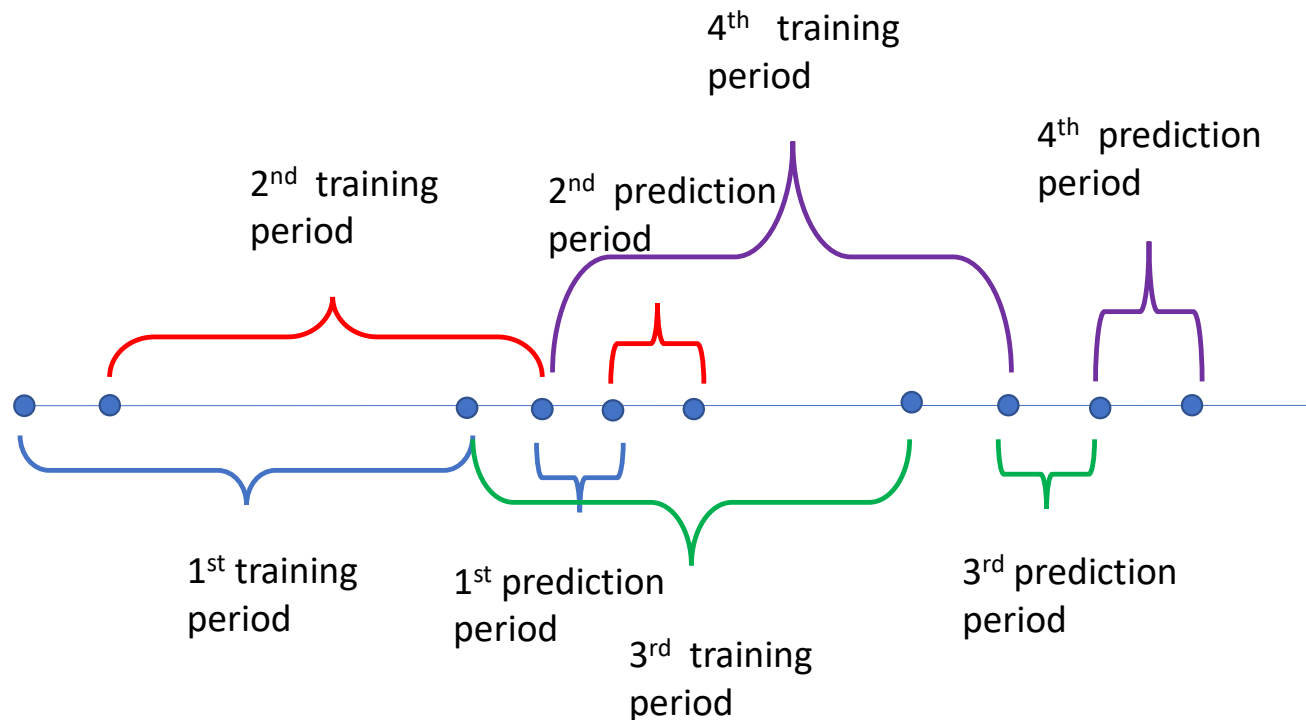
```
data['rel_performance']
```

date	ticker	
2000-02-09	CSCO	-1
	ROP	1
2000-02-10	CMOS	1
2000-02-11	DELL	1
2000-02-15	VAL	1
2000-02-16	AMAT	1
	ANF	-1
	DE	1
	EV	1
	NAV	-1

The strategy is going to work as follows:

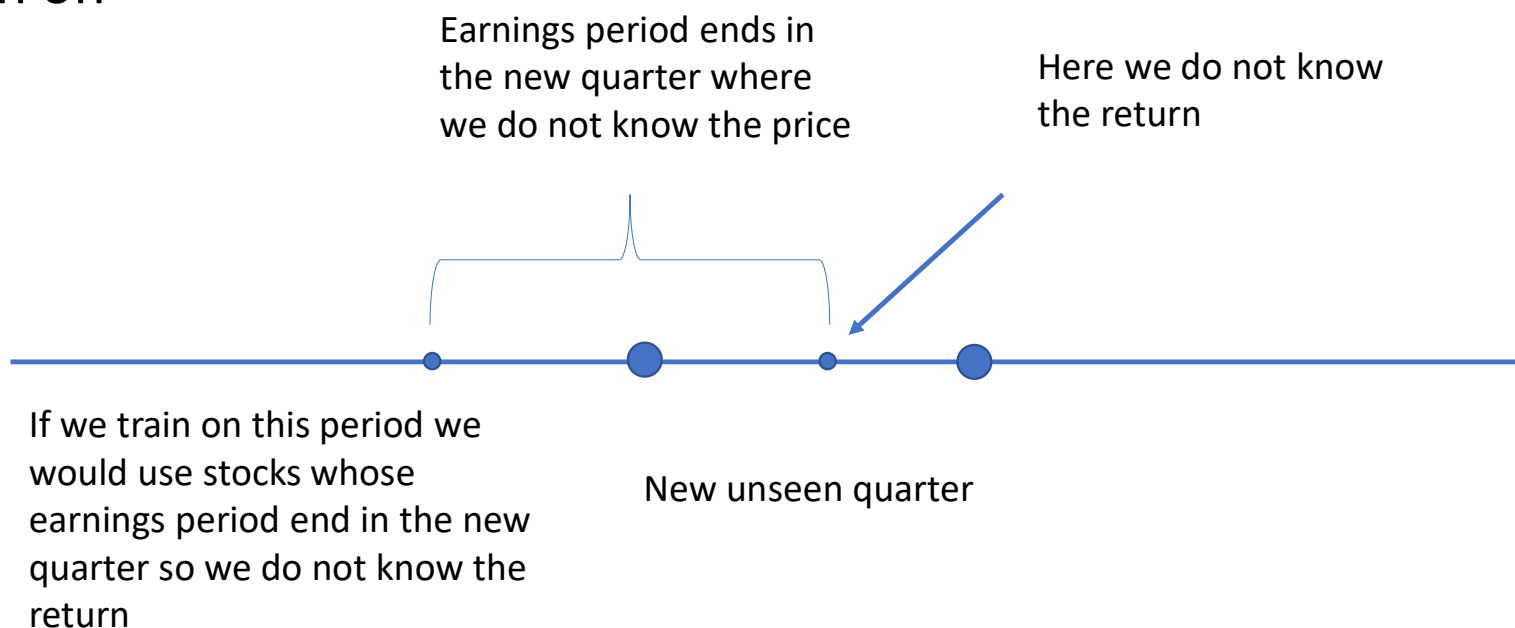
We train on a period of 3 years and then predict on the quarter starting 3 months after the end date of the training period

Then we move the 3 year period forward 1 quarter and train again and predict on the following quarter etc.



## Why are we skipping a quarter?

This is because when we train we use the returns for a period that ends in the quarter immediately following the training period and so when we test on an unseen quarter this information would not be available to train on



During each prediction period we use the model that has been trained on the data in the preceding training period. During the prediction period, when a company releases quarterly earnings we use the trained model to tell us whether to buy, sell or not do anything.

We are first going to try it out over 2 periods and find the best parameters for our model

```
▶ 1 df_1 = data.loc['2008-01-01':'2011-01-01']  
  2 df_2 = data.loc['2015-04-01':'2018-04-01']
```

Each training period is 3 years



Next we have the two prediction periods

```
▶ 1 df_valid = data.loc['2012-04-01':'2012-07-01']  
  2 df_test = data.loc['2018-07-01':'2018-10-01']
```

and the labels for the prediction periods

```
▶ y_1 = df_1['rel_performance'].values  
  y_2 = df_2['rel_performance'].values
```

```
▶ Counter(y_1)
```

```
5]: Counter({-1: 15542, 1: 15974, 0: 2888})
```

The data we train the model on, should not contain any features that use information not revealed before the end of the training period. This includes next\_period\_return, spy\_next\_period\_return, rel\_return, cum\_return, spy\_cum\_return. We also leave out the ticker symbol and the date.

```
train_1 = df_1.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)  
train_2 = df_2.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)  
  
valid = df_valid.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)  
test = df_test.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)
```

We want to save the actual returns from both the training and prediction periods as well as the labels for the validation and test periods

```
▶ train_1_stock_returns = df_1['next_period_return']  
  valid_stock_returns = df_valid['next_period_return']  
  train_2_stock_returns = df_2['next_period_return']  
  test_stock_returns = df_test['next_period_return']
```

---

```
▶ y_valid = df_valid['rel_performance'].values  
  y_test = df_test['rel_performance'].values
```

---

We are now ready to start training our first model.

```
► from sklearn.tree import DecisionTreeClassifier  
from sklearn.metrics import accuracy_score, confusion_matrix
```

We instantiate a tree classifier and train it on train\_1,

```
► t_clf = DecisionTreeClassifier(min_samples_leaf=600, max_depth=8, random_state=123)
```

```
► t_clf.fit(train_1, y_1)
```

```
.]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=8,  
    max_features=None, max_leaf_nodes=None,  
    min_impurity_decrease=0.0, min_impurity_split=None,  
    min_samples_leaf=600, min_samples_split=2,  
    min_weight_fraction_leaf=0.0, presort=False, random_state=123,  
    splitter='best')
```

the number of features is

```
► len(train_1.columns)
```

```
|: 725
```

The large number of features is due to the one-hot-encoding of the categorical variables which associate to each category a new column consisting of only 0s and 1s.

We can estimate how well the trained model fits the data by making the model predict the rel\_performance on the data and compare that with the actual, which is the array y\_1

```
1 t_clf.score(train_1,y_1)
```

```
21]: 0.5537500796533487
```

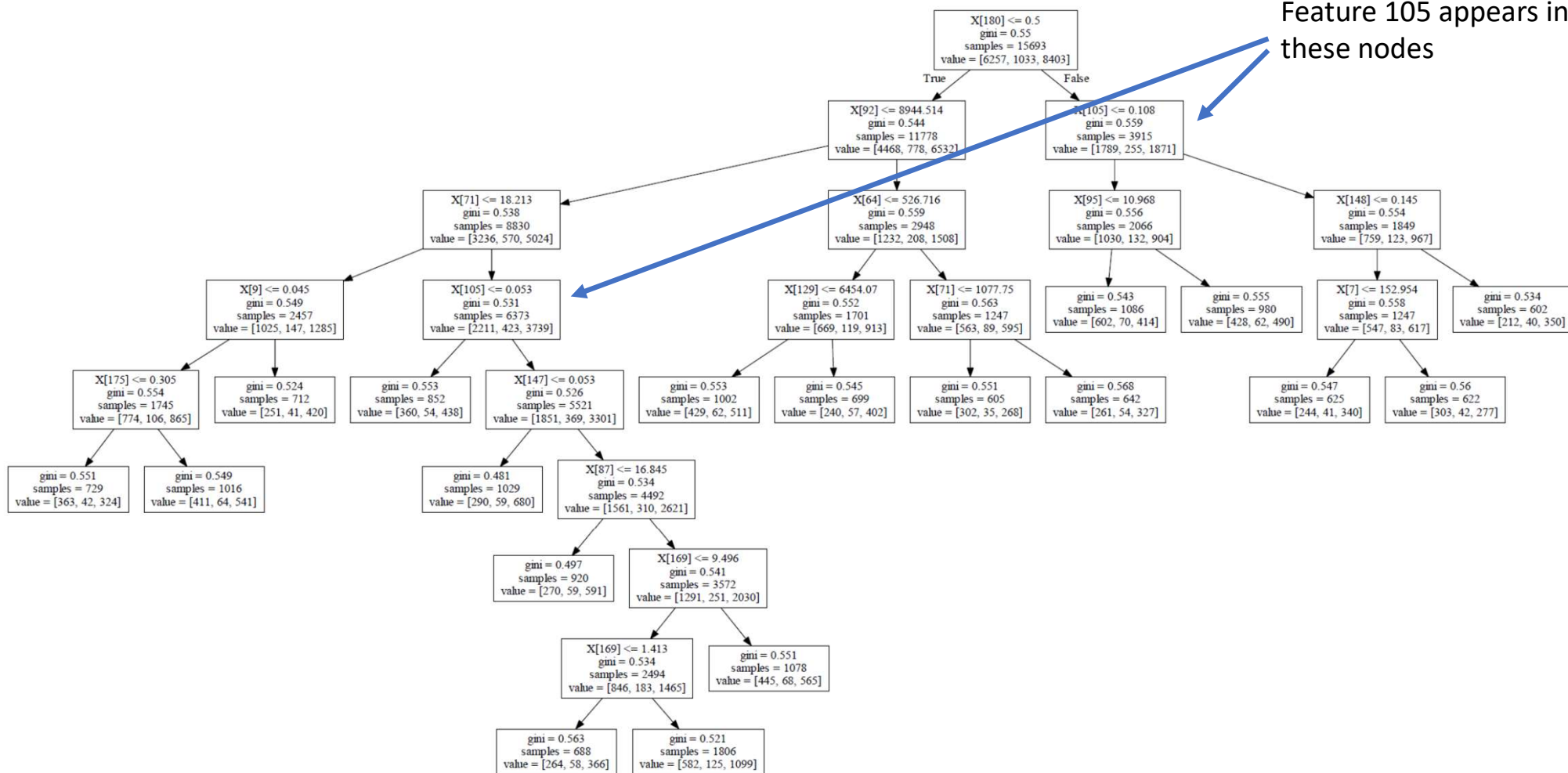
**and predicts the validation data (not very well)**

Fraction of correct  
predictions

```
1 t_clf.score(valid,y_valid)
```

```
22]: 0.4275947793660659
```

Feature 105 appears in these nodes



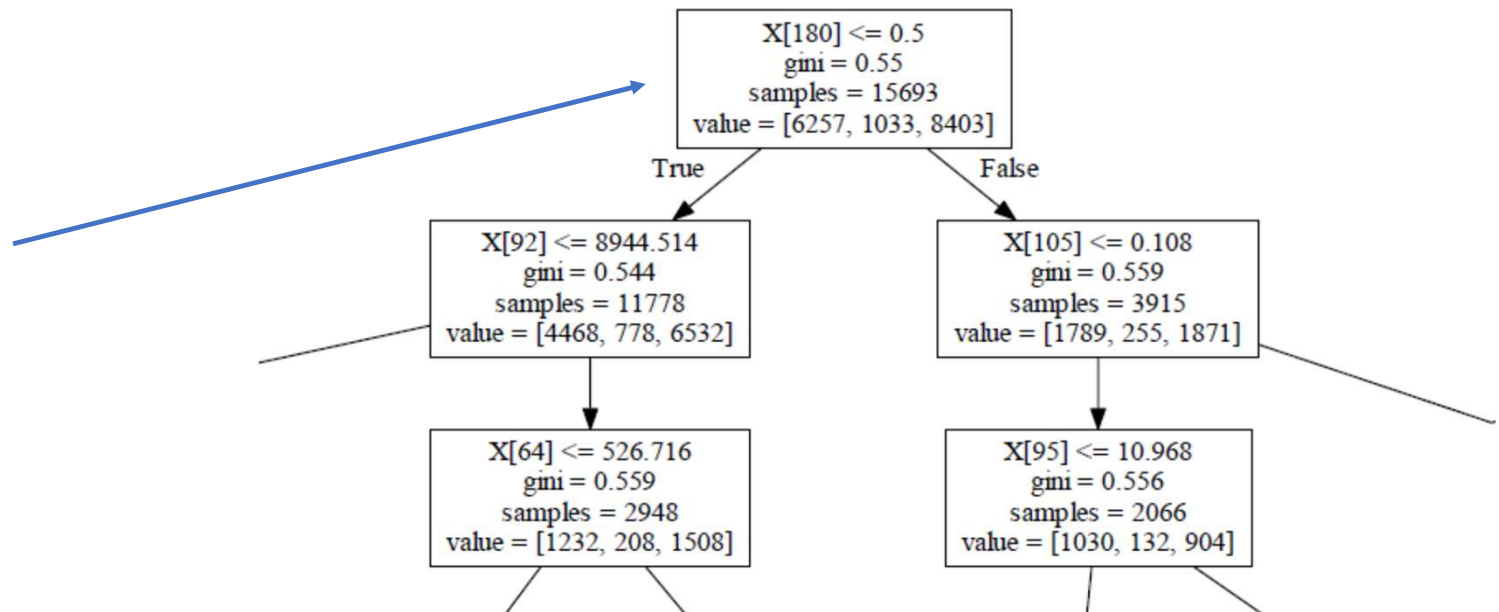
First we are going to reduce the number of features.

We can compute how important a given feature is in the tree by looking at the nodes in the tree where the split is on the feature and look at how much the split reduces the Gini index. The sum of these is the feature importance.

0.0012757  
0.0012757

## Small segment of the tree

Node splits on feature  
180 =fqtr\_2



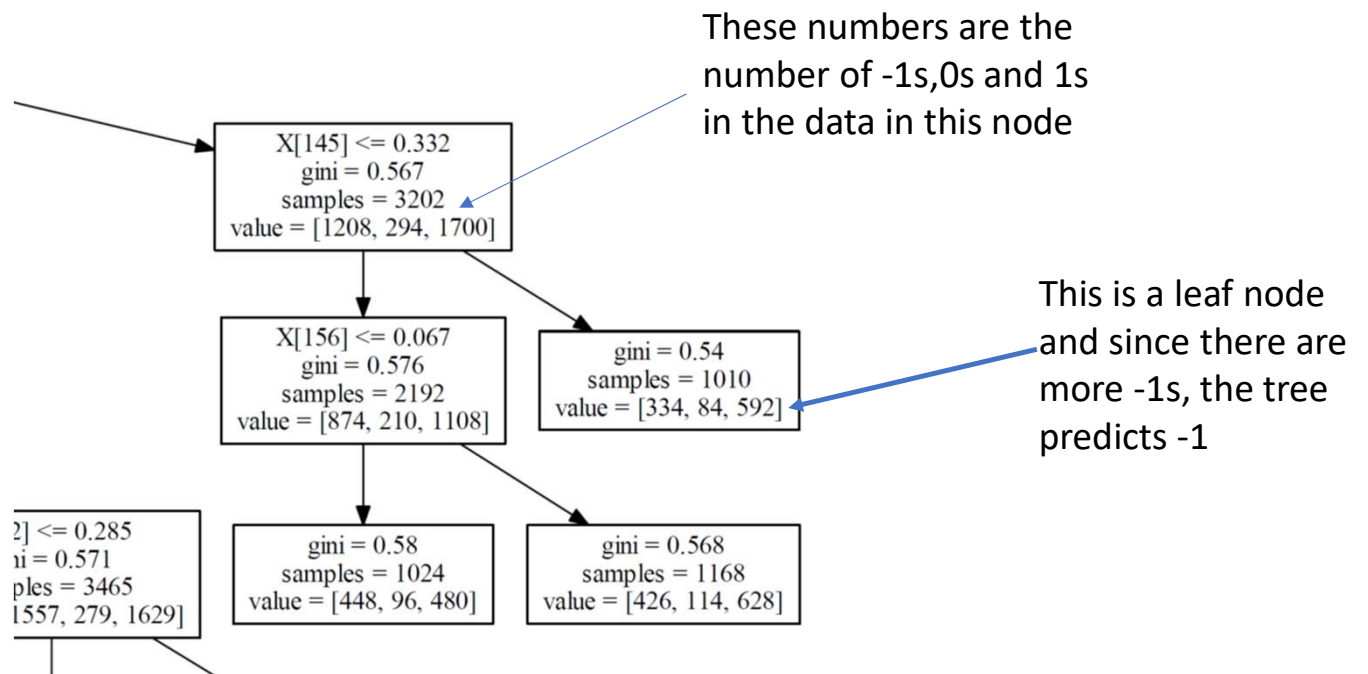
Gini index of the split=  
 $(11778/15693)*0.544 +$   
 $(3915/15693)*0.559 =$   
 $0.4083 + 0.1395 =$   
 $0.5478$

Reduction in Gini index =  
 $0.55 - 0.5478 = 0.001226$



By adding up all the Gini index reductions over all the nodes (remark that a feature can occur in several nodes) we get the feature importance.

```
► def tree_feat_importance(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)
```



1	fi
---	----

]:

	cols	feat_imp
180	fqtr_2	0.200806
105	cf_yield	0.146429
71	dpcy	0.116419
95	evmq	0.082621
92	market_cap	0.075873
169	inv_turnq	0.069376
147	debt_ebitdaq	0.050145
9	cshopq	0.049831
87	prccq	0.042593
175	sale_nwcq	0.040217

248	sic_2590	0.000000
257	sic_2731	0.000000
249	sic_2611	0.000000
250	sic_2621	0.000000
251	sic_2631	0.000000
252	sic_2650	0.000000
253	sic_2670	0.000000
254	sic_2673	0.000000
255	sic_2711	0.000000
256	sic_2721	0.000000
724	sector_code_850.0	0.000000

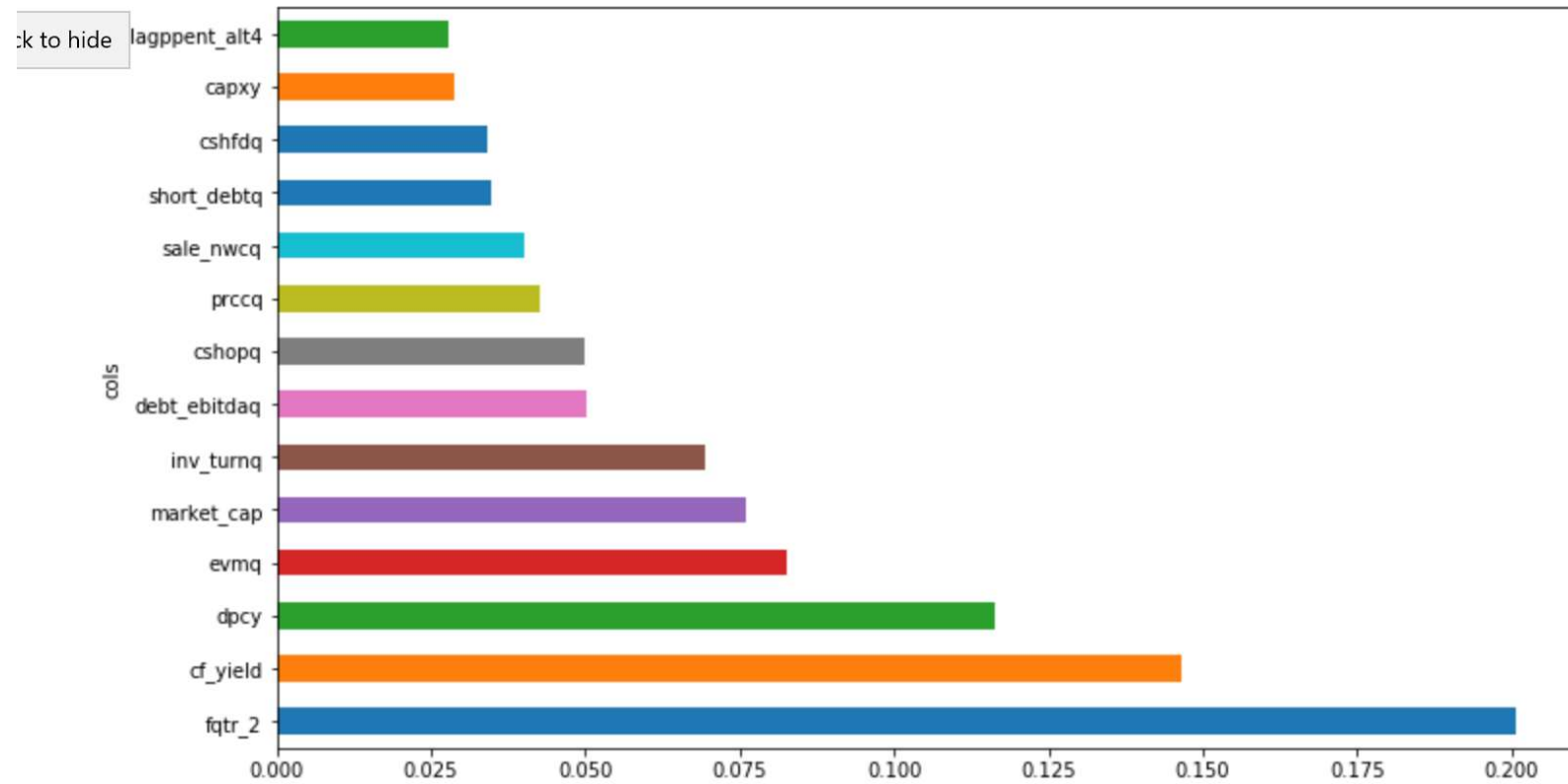
725 rows × 2 columns

Features with importance 0.0 do not appear at all in the tree so can safely be deleted

```
► features = fi[(fi['feat_imp'] > 0.00)]
```

## Feature importance spectrum

1 `plot_fi(features);`



There are 14 features actually occurring in the tree

```
1 len(features['cols'].values)
```

```
] 14
```

**We cut down the training and validation data sets to only include the relevant features and retrain the tree on the reduced data set**

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

```
1 t_clf.fit(train_1,y_1)
```

```
] DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=8,
                        max_features=None, max_leaf_nodes=None,
                        min_impurity_decrease=0.0, min_impurity_split=None,
                        min_samples_leaf=600, min_samples_split=2,
                        min_weight_fraction_leaf=0.0, presort=False, random_state=123,
                        splitter='best')
```

```
1 t_clf.score(train_1,y_1)
```

```
] 0.5537500796533487
```

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

```
1 t_clf.score(valid,y_valid)
```

```
] 0.4275947793660659
```

The accuracy on the training data is identical to the accuracy with all the features

What we are really interested in is of course how well our strategy performs in terms of P/L

When the model predicts +1 we buy \$1 of the stock and when it predicts -1 we sell (short) \$1 of the stock. If the prediction is 0 we do not invest in the stock.

To compute the performance of the strategy we can simply compute

**We are really only interested in how much return the strategy generates over the validation period**

```
1 (pred_valid * df_valid['next_period_return']).sum()
```

```
0]: -19.835171000000038
```

Can we improve on this score?

To try to improve the profit of the strategy we define a new importance measure of each feature: `profit_importance`.

The idea is that we remove a feature and see if that improves the profit.

The amount by which it improves the profit is the `profit_importance`.  
Remark that here we want to remove features with high `profit_importance`



The way we compute the pi is to take the given feature and then permute the values in the column so that the feature no longer has an effect on the prediction



```
1 def profit_importance(t,df,rets):
2     # np.random.seed(123)
3     profit = []
4     for col in df.columns:
5         X = df.copy()
6         X[col] = np.random.permutation(df[col].values)
7         prediction = t.predict(X)
8         profit.append((prediction * rets).sum())
9     return profit
```



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

For each column we randomly permute the values, use the trained tree to predict on the scrambled data set and use this prediction to compute the profit.

This method avoid having to retrain the tree for every feature

```
1 pi = tree_profit_importance(t_clf,valid,df_valid['next_period_return'])
2 pi
```

	cols	pi_imp
3	evmq	-22.807007
11	csbfdq	-20.433543
1	cf_yield	-20.284441
5	inv_turnq	-19.835171
6	debt_ebitdaq	-19.835171
8	prccq	-19.835171
13	lagppent_alt4	-19.835171
2	dpcy	-19.419867
0	fqtr_2	-19.195297
10	short_debtq	-19.024541
12	capxy	-18.710075
7	cshopq	-17.345881
4	market_cap	-17.071709
9	sale_nwcq	-11.928787

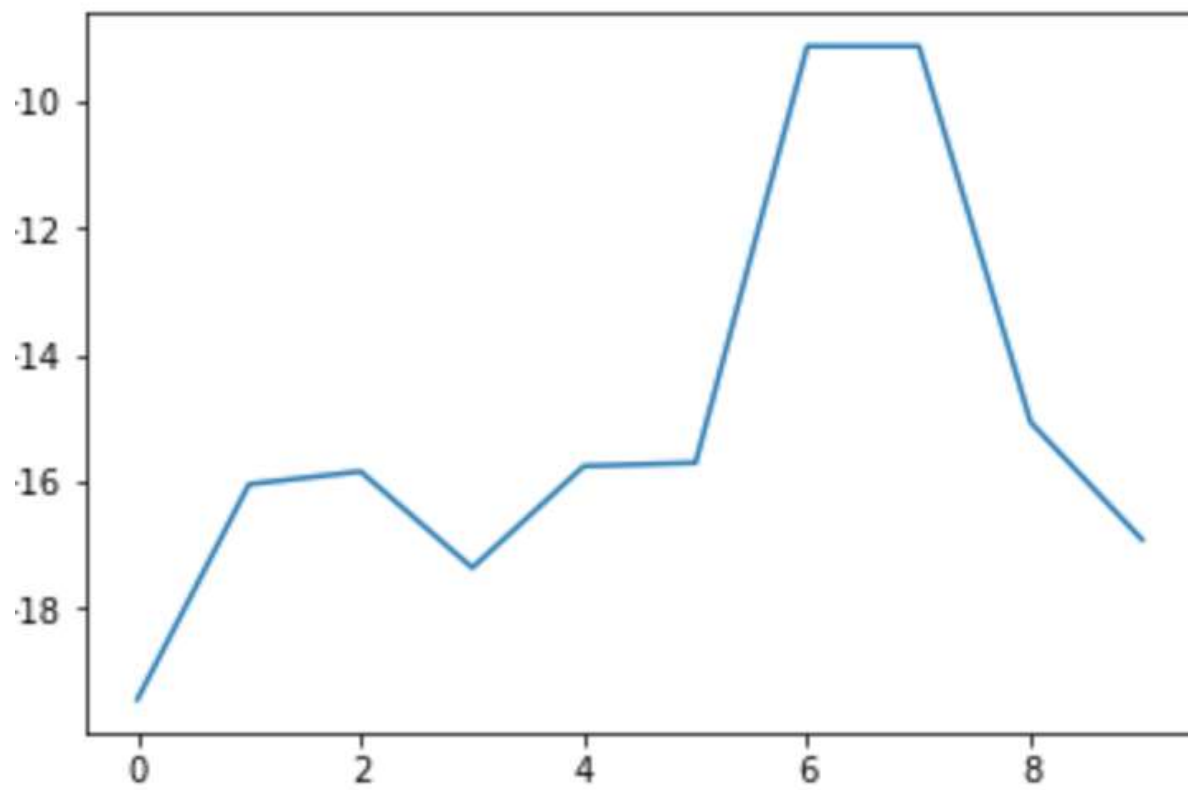
Removing evmq would reduce the profit to -\$22.807 so we do not want to remove this feature, while removing sale\_nwcq would improve the profit so we would want to remove this feature

The features strongly interact with each other so removing a feature affects the importance and the profit\_importance of all the other features so it is difficult to determine the optimal set of features.

We have a small algorithm that sequentially removes the feature with the highest pi score and then retrains the tree on the remaining features, removes those with feature importance = 0.0 and predicts the profits. We loop through all the features and we can then choose the feature set which yields the highest profit.

This gives an indication of what the optimal features might be but to find the actual set is intractable

```
1
2 profits = []
3 feat=[]
4 train = train_1.copy()
5 validation = valid.copy()
6 while len(train.columns)>1:
7     t_clf.fit(train,y_1)
8     pi = tree_profit_importance(t_clf,validation,valid_stock_returns)
9     col_to_drop = pi.iloc[-1]['cols']
10    train.drop(col_to_drop,axis=1,inplace=True)
11    validation.drop(col_to_drop,axis=1,inplace=True)
12    t_clf.fit(train,y_1)
13    pred_valid = t_clf.predict(validation)
14    profits.append((pred_valid * valid_stock_returns).sum())
15    feat.append(validation.columns)
16
17
```



```
▶ 1 n = np.argmax(profits)
   2 max_profits = profits[n]
   3 optim_feats = feat[n]
```

```
▶ 1 print(max_profits)
   2 print(optim_feats)
```

-9.123655000000026

['cf\_yield' 'dpcy' 'prccq' 'evmq' 'cshfdq']

---

The features that give us the highest profit on the validation set are:

cf\_yield = Cash Flow Yield

evmq = Enterprise Value Multiple

dpcy = Depreciation and Amortization

prccq = Price Close Quarter

cshfdq = Common Shares for Diluted EPS

The real test of the feasibility of the strategy is to see how the strategy with these features performs on the test set

```
► 1 train_1_optim = train_1[optim_feats]
   2 valid_optim = valid[optim_feats]
   3
   4 t_clf.fit(train_1_optim,y_1)
   5 pred_valid_tree = t_clf.predict(valid_optim)
   6 (pred_valid_tree * df_valid['next_period_return']).sum()

): -9.123655000000026
```

**How well does it do on the test set?**

```
► 1 train_2_tree = train_2[optim_feats]
   2 test_tree = test[optim_feats]
   3 t_clf.fit(train_2_tree,y_2)
   4 pred_test_tree = t_clf.predict(test_tree)
   5 (pred_test_tree * df_test['next_period_return']).sum()

): 1.8368159999999243
```

If we were 100% correct the profit would be

```
1 (y_valid * df_valid['pred_rel_return']).sum()  
: 167.41809499999999
```

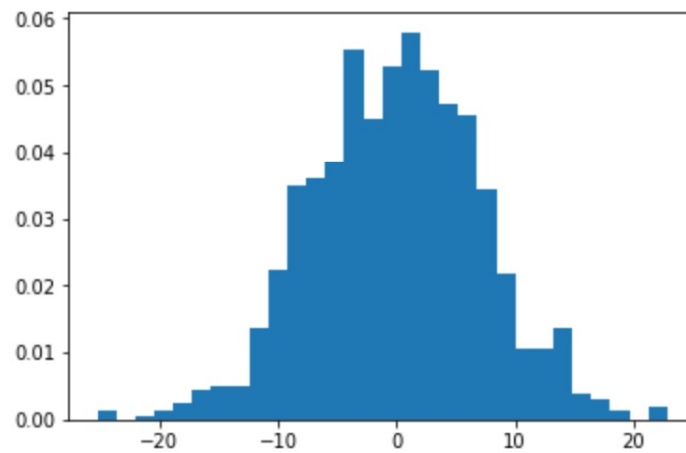
and if we made a random choice of which stock to buy or sell we would in the mean make

```
1 np.mean(random_predictions)  
: 0.03039984200000059
```



```
1 random_predictions = []
2 for _ in range(1000):
3     pred_random = np.random.choice([-1,0,1],m)
4     random_predictions.append((pred_random * df_test['next_period_return']).sum())
5
```

```
1 plt.hist(random_predictions,bins=30,density=True);
```



```
1 np.mean(random_predictions)
```

```
]: 0.03039984200000059
```

As a rule of thumb: a small set of optimal features will generalize better

We will back test our strategy over the range 2000-01-01 to 2018-10-01


First we need the starting dates and the end dates for the training periods, each training period is 36 months and each training period is shifted one quarter from the previous

```
1 start_dates = [pd.to_datetime('2000-01-01') + pd.DateOffset(months = 3 * i) for i in range(62)]
2 end_dates = [d + pd.DateOffset(months = 36) for d in start_dates]
```

```
1 training_frames = [data.loc[d:d+pd.DateOffset(months = 36)] for d in start_dates]
2 test_frames = [data.loc[d + pd.DateOffset(months=3):d+pd.DateOffset(months = 6)] for d in end_dates]
```

Then we can select the data frames for each training and validation period

```
▶ training_frames = [data.loc[d:d+pd.DateOffset(months = 36)] for d in start_dates]  
validation_frames = [data.loc[d + pd.DateOffset(months=3):d+pd.DateOffset(months = 6)] for d in end_dates]
```



The validation period corresponding to a training period starts 3 months after the end date of the training date and runs for 3 months

We drop the features from each of the training\_frames and the validation\_frames

```
▶ 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]
```

```
▶ validation_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 validation_frames]
```

and we get the labels for each training set

---

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

Now we can run through all the dates, train on each training\_data and use the trained tree to predict the profit on each validation\_data.

We also see how starting with \$1 will perform over the 17 year period.

If we have \$x we invest (long or short)  $x/\text{\#securities}$  and so the total profit is  $(x/\text{\#securities}) * \text{profit}$ .

First we restrict to the optimal features we found

```
▶ opt_training_data = [t[optim_feats] for t in training_data]
  opt_validation_data = [v[optim_feats] for v in validation_data]
```

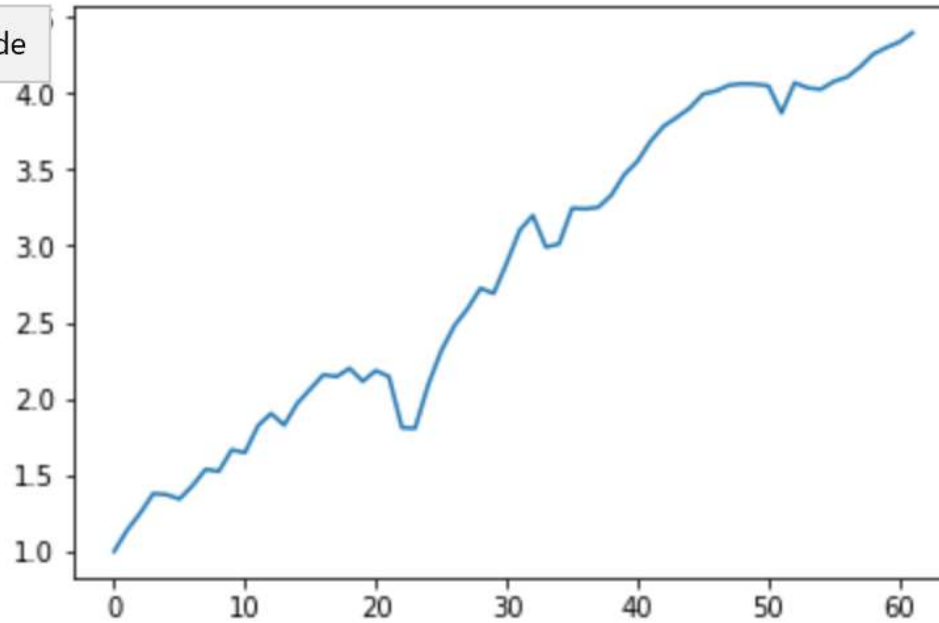
```
▶ 1 P_L = []
   2 x = [1]
   3 ret = []
   4
   5 for i in range(len(training_labels)-1):
   6     t_clf.fit(opt_training_data[i], training_labels[i])
   7     pred_i = t_clf.predict(opt_test_data[i])
   8     profit_i = (pred_i * test_frames[i]['next_period_return']).sum()
   9     P_L.append(profit_i)
  10     num_positions = len(pred_i)
  11     ret.append((1.0/num_positions) * profit_i)
  12     x.append(x[i] + (x[i]/num_positions) * profit_i)
```

\$1 investment in the strategy becomes ~ \$4 over 17 years



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

to hide





We will compare this to a \$1 investment in SPY

We have the cumulative returns on SPY in the raw\_data data frame and we only want one return for each date (for each date there will be as many copies of the cumulative SPY returns as there are companies reporting on that date)

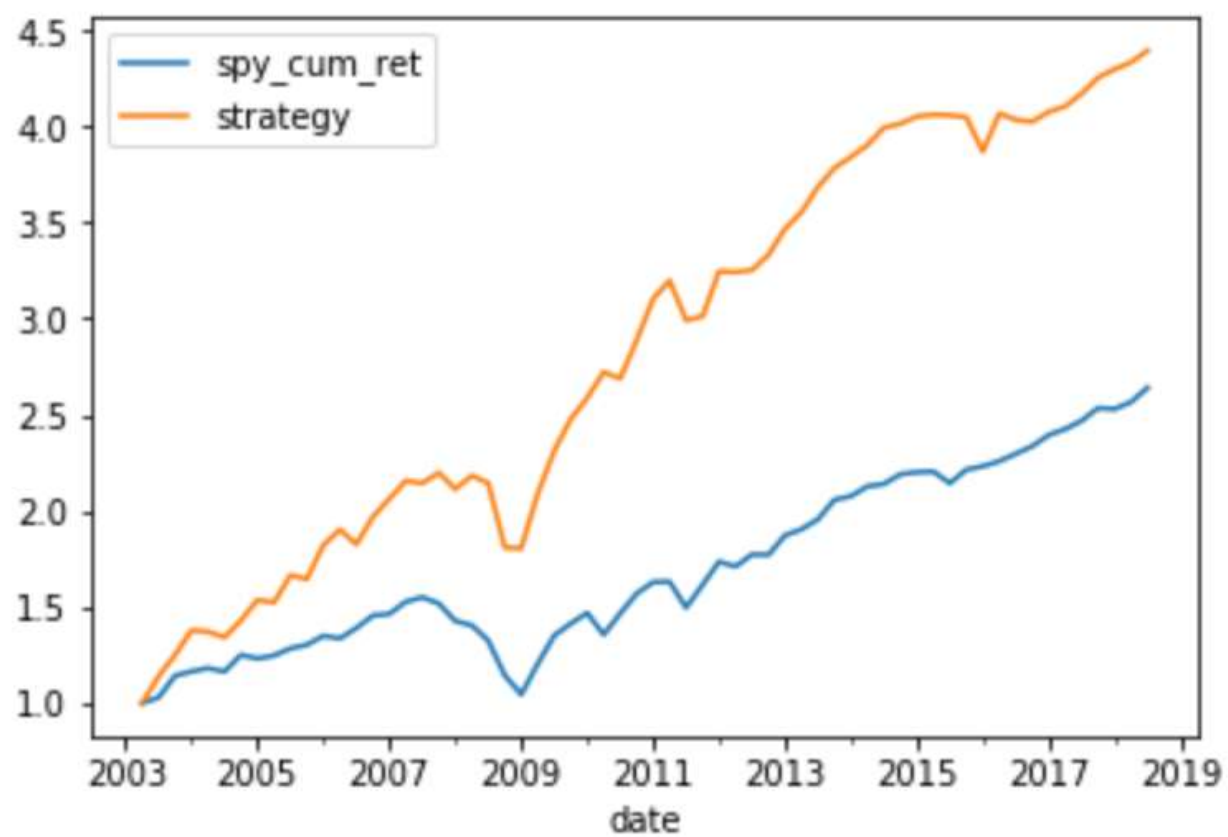
```
1 SPY = pd.read_pickle(r'C:\Users\niels\data\SPY_cum_ret.pkl')
2 SPY = SPY.loc['2003-04-01':'2018-09-30']
3 SPY = SPY.resample('Q').ffill()
4 SPY['spy_cum_ret'] = (SPY['spy_cum_ret'] - SPY['spy_cum_ret'][0] + 1)
5 SPY['strategy'] = x
```

We want quarterly returns so we resample to quarters

```
spy = spy.resample('Q').ffill()  
spy
```

date	
2003-06-30	1.133396
2003-09-30	1.163119
2003-12-31	1.276658
2004-03-31	1.298132
2004-06-30	1.315963
2004-09-30	1.297142
2004-12-31	1.384523
2005-03-31	1.365374
2005-06-30	1.381274
2005-09-30	1.418374
2005-12-31	1.436974
2006-03-31	1.483915
2006-06-30	1.470248
2006-09-30	1.524426
2006-12-31	1.589097
2007-03-31	1.507750

```
1 SPY.plot();
```



The strategy handily beats the SPY but remark that we have not included transaction costs which might change the result significantly.

The Sharpe Ratios:

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

Strategy Sharpe Ratio: 0.5296623613542152

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

SPY Sharpe Ratio: 0.4100834330712901

## Total Returns

1 `x[-1]`

`57]: 4.395936951063365`

1 `SPY['spy_cum_ret'][-1]`

`58]: 2.64168`

## Computing the (quarterly) $\alpha$ of the strategy

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

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

0]: 0.9936537611546371

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

1]: 0.3502843897055849

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

2]: 0.028929105646738038

There is, however a serious problem with our reasoning.

Can you spot it ?