Machine Learning

Lecture 3

Bootstrapping

Bootstrapping is a technique to randomize a finite dataset to generate new samples. It is best illustrated with an example:

Consider a dataset with 10 samples. We assume they are drawn from some unknown distribution we would like to determine.

$$\{-0.2322, -0.7313, -0.0603, 1.9935, -0.5873, 0.4529, -1.5701, -0.7856, 2.2676, -0.6444\}$$

We can draw the empirical distribution function by first ordering the data set

$$\{-1.5701, -0.7856, -0.7313, -0.6444, -0.5873, -0.2322, -0.0603, 0.4529, 1.9935, 2.2676\}$$

We order the data

$$\{-1.5701, -0.7856, -0.7313, -0.6444, -0.5873, -0.2322, -0.0603, 0.4529, 1.9935, 2.2676\}$$

and draw the empirical distribution

```
1  X = np.array([-0.2322,-0.7313,-0.0603,1.9935,-0.5873,0.4529,-1.5
```

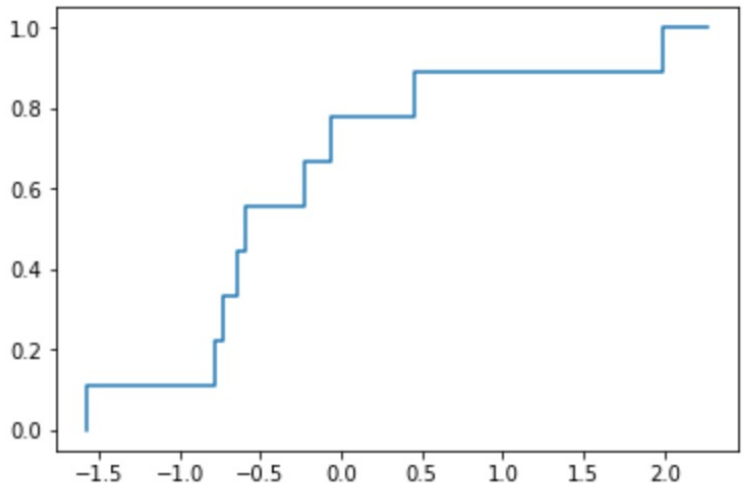
```
1  X = sorted(X)
```

```
1  X
```

```
[-1.5701,
 -0.7856,
 -0.7313,
 -0.6444,
 -0.5873,
 -0.2322,
 -0.0603,
 0.4529,
 1.9935,
 2.2616]
```

```
1  y = np.linspace(0,1,10)
```

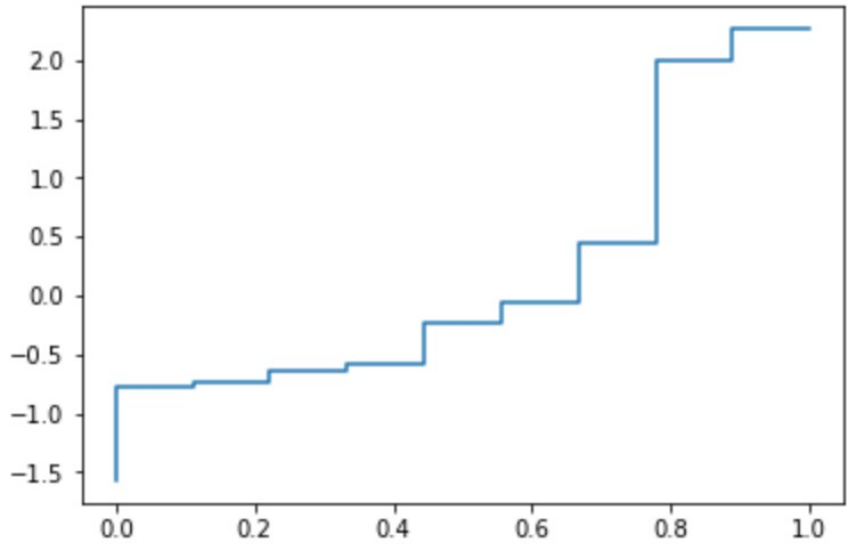
```
1  plt.step(X,y);
```



If we let Ψ denote this step function, we let $\Psi^{-1} : (0,1) \to \mathbb{R}$ be the function defined by

$$\Psi^{-1}(x) = X[i], \text{ if } x \in [i/10, i + 1/10)$$

```
1  plt.step(y,X);
```



The key observation is that if we generate 10 samples from a uniform distribution $\{x_0, x_1, \ldots, x_9\}$ then $\{\Psi^{-1}(x_0), \Psi^{-1}(x_1), \ldots, \Psi^{-1}(x_9)\}$ are samples from the distribution Ψ .

This way we can generate as many datasets, sampled from Ψ, as we want.

```
1  X_1 = np.random.uniform(0,1,10)
```

```
1  X_1
```

```
array([0.02951883, 0.14922467, 0.99969761, 0.09383227, 0.08465161,
       0.90996896, 0.64453885, 0.58084102, 0.27975876, 0.87437016])
```

```
1  np.array([psi_inv(x) for x in X_1])
```

```
array([-1.5701, -0.7856,  2.2616, -1.5701, -1.5701,  2.2616, -0.0603,
       -0.2322, -0.7313,  1.9935])
```

Let's try another, more elaborate example.

Recall the CAPM model

$$r_{es} = \alpha + \beta_{sm} r_{em}$$

Here r_{es}, r_{em} are the excess returns on the stock, *s* and the market *m* i.e. the returns above the risk-free return at the time.

α, β_{sm} can be estimated by running a linear regression.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
df = pd.read_csv(r'C:\Users\niels\data\IBM.csv',usecols=['Date','Adj Close'])

df.set_index(['Date'],inplace=True)

df['IBM'] = df['Adj Close']
df.drop(['Adj Close'],axis=1,inplace=True)

df['SP500'] = pd.read_csv(r'C:\Users\niels\data\^GSPC.csv',usecols=['Adj Close']).values

df['Risk Free Rate'] = pd.read_csv(r'C:\Users\niels\data\^IRX.csv',usecols=['Adj Close']).values

df['IBM'] = df['IBM'].pct_change()
df['SP500'] = df['SP500'].pct_change()
df['Risk Free Rate'] = df['Risk Free Rate'] *0.01/12

df = df.iloc[2:]

df.head()
```

Date	IBM	SP500	Risk Free Rate
2010-06-01	-0.009158	-0.053882	0.000142
2010-07-01	0.039844	0.068778	0.000117
2010-08-01	-0.041043	-0.047449	0.000113
2010-09-01	0.094816	0.087551	0.000129
2010-10-01	0.070524	0.036856	0.000092

We use monthly returns for IBM and use the SP500 index as a proxy for the market portfolio. For the risk free rate we use the annualized rate/12 of 3 month Treasury Bills.

We compute monthly returns by using the .pct_change method.

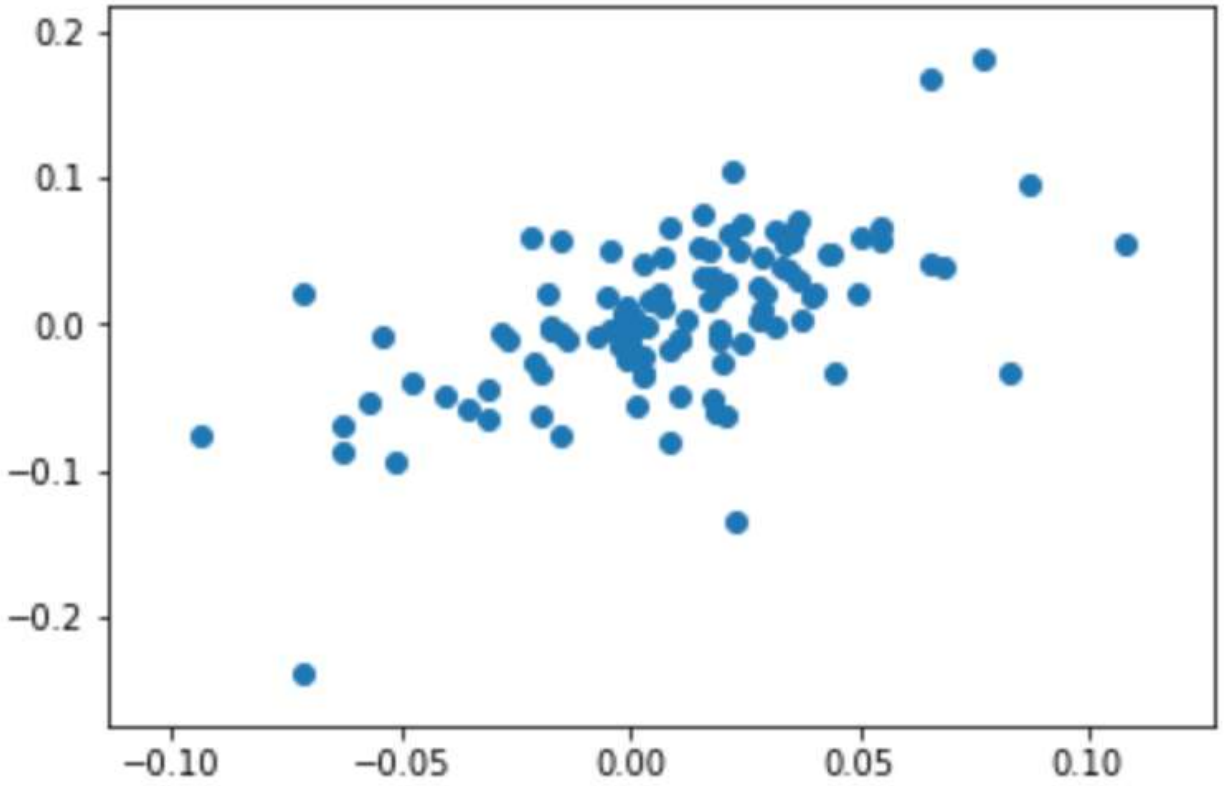
We then replace the returns with the excess returns

```
df['IBM'] = df['IBM'] - df['Risk Free Rate']
df['SP500'] = df['SP500'] - df['Risk Free Rate']
```

```
df.head()
```

Date	IBM	SP500	Risk Free Rate
2010-06-01	-0.009299	-0.054024	0.000142
2010-07-01	0.039727	0.068661	0.000117
2010-08-01	-0.041156	-0.047562	0.000113
2010-09-01	0.094687	0.087422	0.000129
2010-10-01	0.070432	0.036764	0.000092

```
1 plt.scatter(df['SP500'],df['IBM']);
```



We can then run the linear regression

```
1 from scipy.stats import linregress
```

```
1 slope, intercept, r_value, p_value, std_err = linregress(df[['SP500','IBM']])
```

```
1 print(slope,intercept)
```

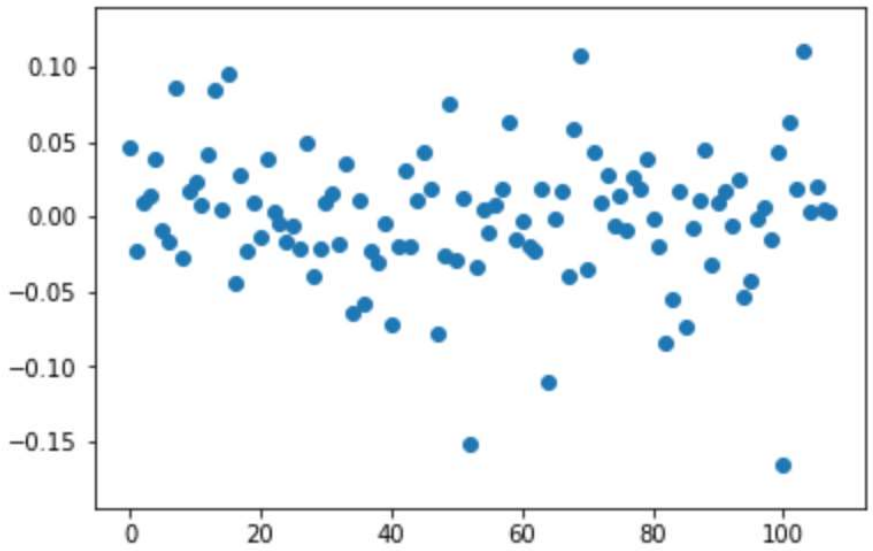
```
0.9655071347625888 -0.004005175608166194
```

Next we compute the residuals

```
1 residuals = df['IBM'] - (intercept + slope * df['SP500'])
```

```
1 plt.scatter(range(len(df)),residuals)
```

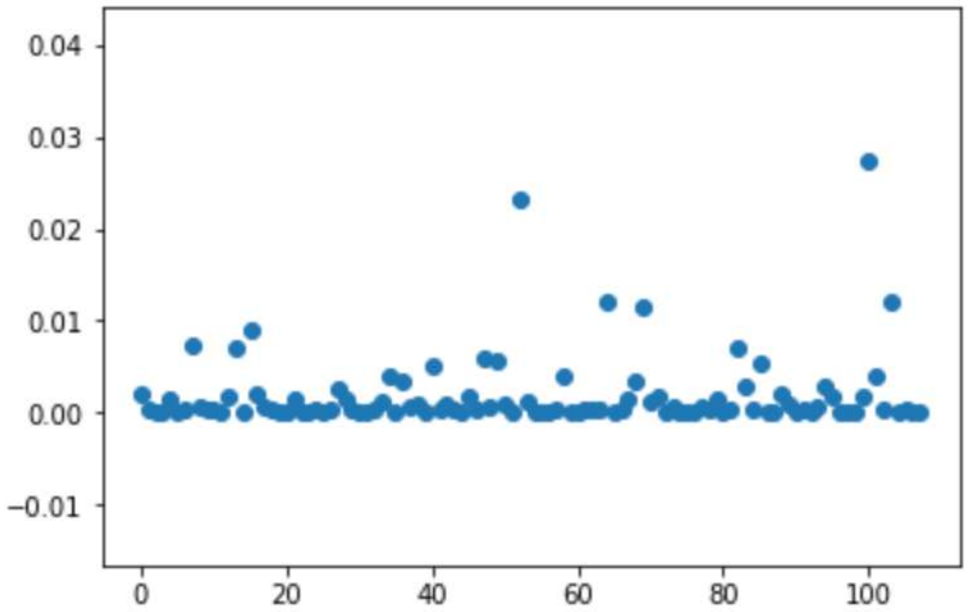
```
<matplotlib.collections.PathCollection at 0x1c922e8c9b0>
```



They look reasonably random and the plot of the squares (=variance) also looks reasonably constant.

```
1  plt.scatter(range(len(df)),residuals**2)
```

```
]: <matplotlib.collections.PathCollection at 0x1c922ef2d30>
```



According to the OLS models the residuals should be normally distributed with mean 0 and variance

$$\sigma^2 = \frac{1}{n-1} \sum \varepsilon_i^2$$

where the sum is over the residuals.

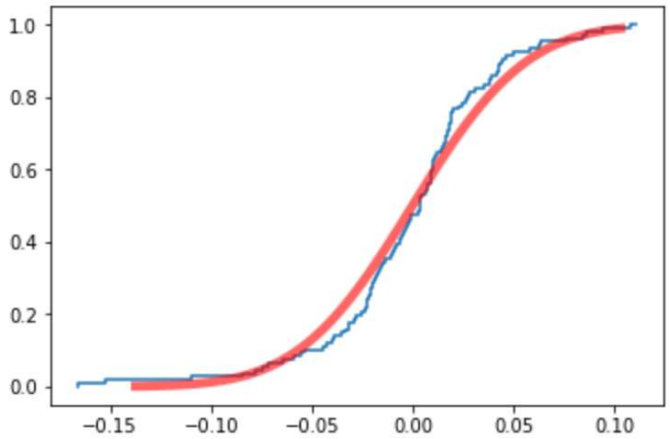
We can now plot the empirical cdf of the residuals and overlay it with the cdf of the normal distribution

```
residuals = residuals.values
residuals.sort()
```

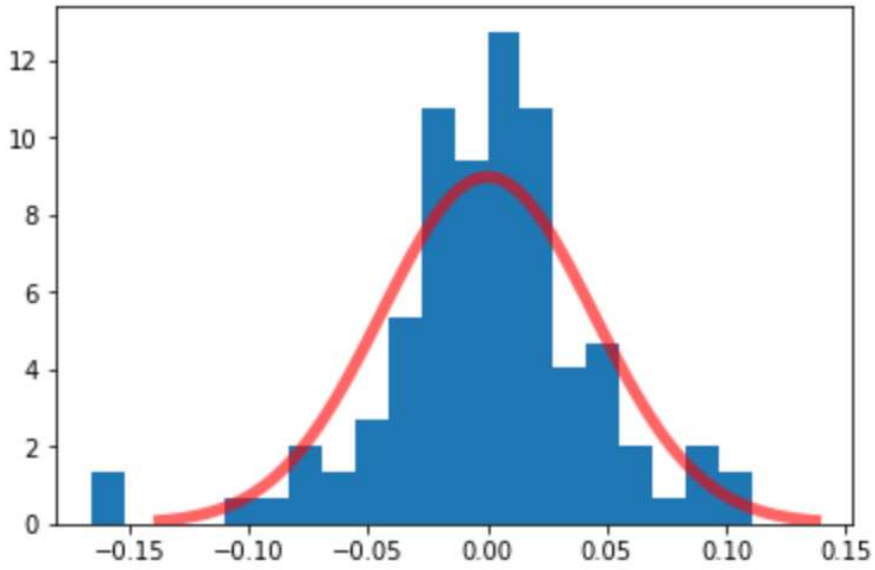
```
p = np.linspace(0,1,len(df))
```

```
mu = residuals.mean()
sigma = residuals.std()
```

```
from scipy.stats import norm
fig, ax = plt.subplots()
plt.step(residuals,p);
x = np.linspace(sigma*norm.ppf(0.001),
                sigma * norm.ppf(0.99), 100)
ax.plot(x, norm.cdf(x/sigma),
      'r-', lw=5, alpha=0.6, label='norm pdf');
```



```
fig, ax = plt.subplots()
plt.hist(residuals,bins=20,density=True);

x = np.linspace(sigma*norm.ppf(0.001),
                sigma * norm.ppf(0.999), 100)
ax.plot(x, norm.pdf(x/sigma)/sigma,
      'r-', lw=5, alpha=0.6, label='norm pdf');

```



We can also run a Kolmogorov-Smirnov test for normality

```
1  from scipy.stats import kstest
2
3  kstest(residuals,'norm')
```

```
KstestResult(statistic=0.45600637892237195, pvalue=0.0)
```

which strongly rejects normality.

Recall that the p-value is p(statistic|null-hypothesis), so in this case, under the hypothesis of normality the probability of observing the test statistic 0.456 is 0.

Suppose we wanted to test a hypothesis that β has some specific value *b*. Assuming homoscedasticity and normality we could use an F-test.

We compute the F-statistic

$$F = (RSSR - USSR)/(USSR/m - 2)$$

Here $RSSR$ is the *Restricted Sum of Squared Residuals* which is computed by estimating the restricted model

$$\alpha = r_{se} - b \cdot r_{me}$$

i.e. the slope is fixed to be *b.* The estimator $\widehat{\alpha}$ is just the mean of the right hand side

The restricted residuals are then

$$residuals_{res} = \widehat{\alpha} - (r_{se} - b \cdot r_{me})$$

where $\widehat{\alpha}$ is the mean of $(r_{se} - b \cdot r_{me})$

The unrestricted residuals are the residuals from the unrestricted model

$$residuals = r_{se} - (intercept + slope \cdot r_{me})$$

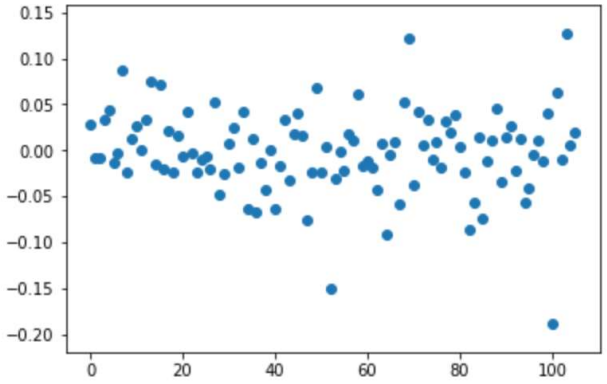
Under the normality assumption the F-statistic follows an F(1,m-2) distribution, which we can use to test the hypothesis

$$slope = b$$

Suppose we want to test the hypothesis $\beta = 0.7$

```
[ ]: 1 z = (df['IBM'] - 0.7 * df['SP500']).values
```

```
[ ]: 1 plt.scatter(range(len(z)),z);
```




```
[ ]: 1 residuals_res = z.mean() - z
```

```
[ ]: 1 RSSR = (residuals_res**2).sum()
     2 RSSR
```

```
: 0.22244998441787733
```

```
[ ]: 1 F = (RSSR - USSR)/(USSR/(len(z)-2))
```

```
[ ]: 1 F
```

```
: 4.42534076575516
```

The 90% confidence interval for F(1,105) (remark that the F-test is a one-sided test)

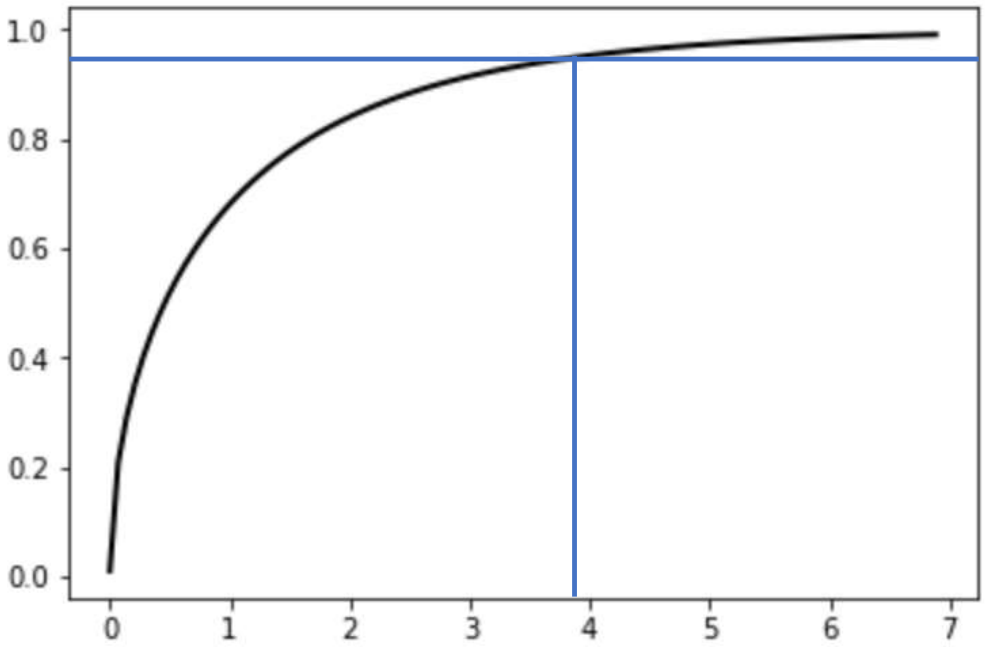
```
1  from scipy.stats import f
```

```
1  f.interval(0.90, 1, 105, loc=0, scale=1)
```

```
(0.00395098311823501, 3.9315564099949247)
```

Thus the null-hypothesis is rejected.

```
x = np.linspace(f.ppf(0.01, 1, 105),
                f.ppf(0.99, 1, 105), 100)

fig, ax = plt.subplots(1,1)
rv = f(1, 105)
ax.plot(x, rv.cdf(x), 'k-', lw=2, label='frozen pdf');
```



Using bootstrapping we can find the empirical distribution of the F-statistic and use that to perform a test.

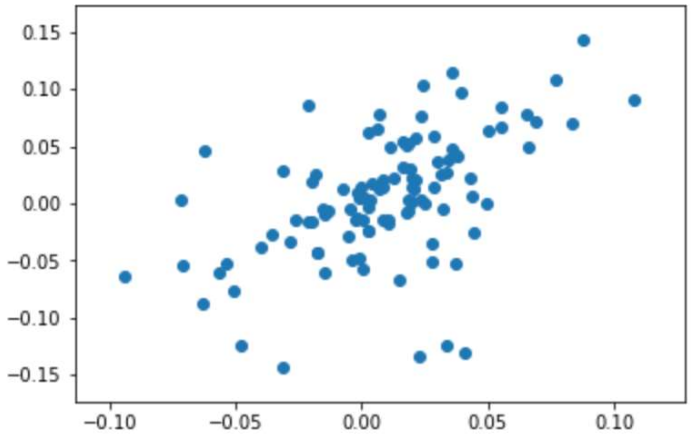
The bootstrapping works by resampling the residuals and compute a new set of values for the dependent variable

```
1 from sklearn.utils import resample
```

```
1 u = resample(residuals)
2 y_resampled = intercept + slope * df['SP500'].values + u
```

```
1 plt.scatter(df['SP500'],y_resampled)
```

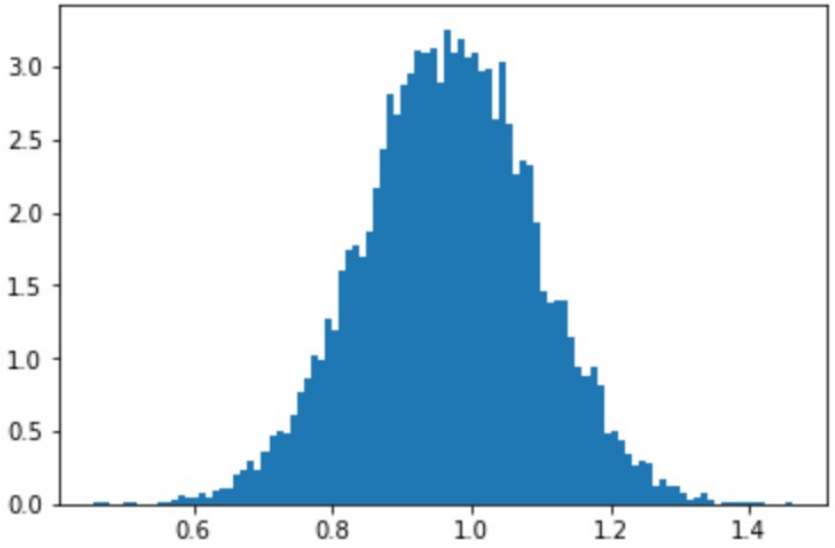
```
<matplotlib.collections.PathCollection at 0x1c9231fc240>
```



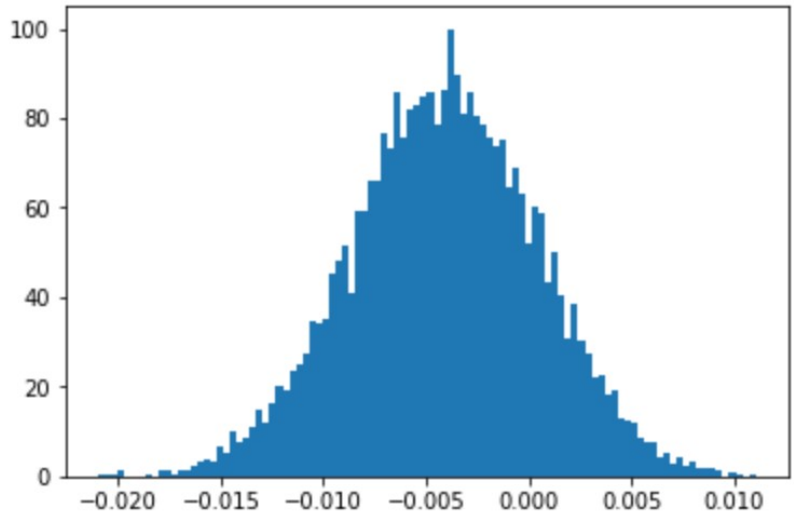
We can then run linear regression to get bootstrapped values for the slope and the intercept. Doing this many times give us empirical distributions for the coefficients in the regression

```
slopes = []
intercepts = []

for _ in range(10000):
    u = resample(residuals)
    y_ = (intercept + slope * df['SP500']).values + u
    slope_resamp, intercept_resamp,_,_,_ = linregress(df['SP500'],y_)
    slopes.append(slope_resamp)
    intercepts.append(intercept_resamp)


```

```
1 plt.hist(slopes,bins=100,density=True);
```




```
1 plt.hist(intercepts,bins=100,density=True);
```

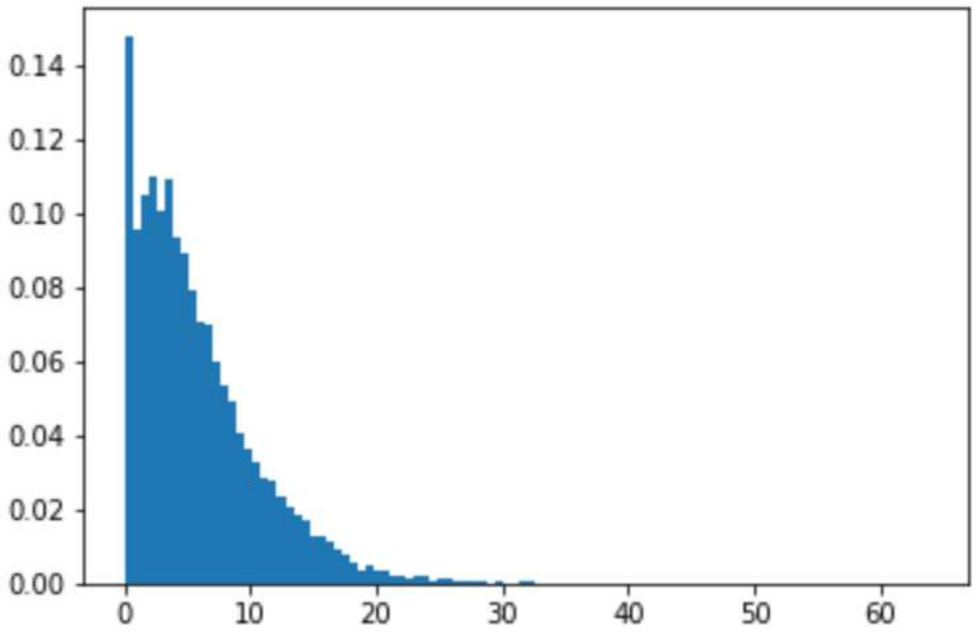


Bootstrapped values of the distribution of the F-statistic under the hypothesis slope = 0.7

```
gamma = 0.7
F_stats = []

for _ in range(10000):
    u = resample(residuals)
    y_ = (intercept + slope * df['SP500']).values + u
    slope_resamp, intercept_resamp,_,_,_ = linregress(df['SP500'],y_)
    res_resamp = y_ - (intercept_resamp + slope_resamp * df['SP500'].val
    z = y_ - gamma * df['SP500'].values
    resid_rest_resampl = z.mean() - z
    USSR = (res_resamp**2).sum()
    RSSR = (resid_rest_resampl**2).sum()

    F = (RSSR - USSR)/(USSR/(len(z)-2))


    F_stats.append(F)
```

Histogram of the bootstrapped F-statistics

```
1  plt.hist(F_stats,bins=100,density=True);
```



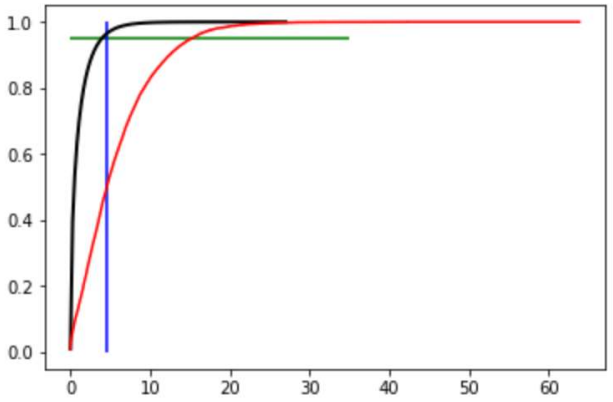
CDFs of F-distribution and bootstrapped distribution

```
t = np.linspace(0.01,1,10000)
```

```
F_stats.sort()
```

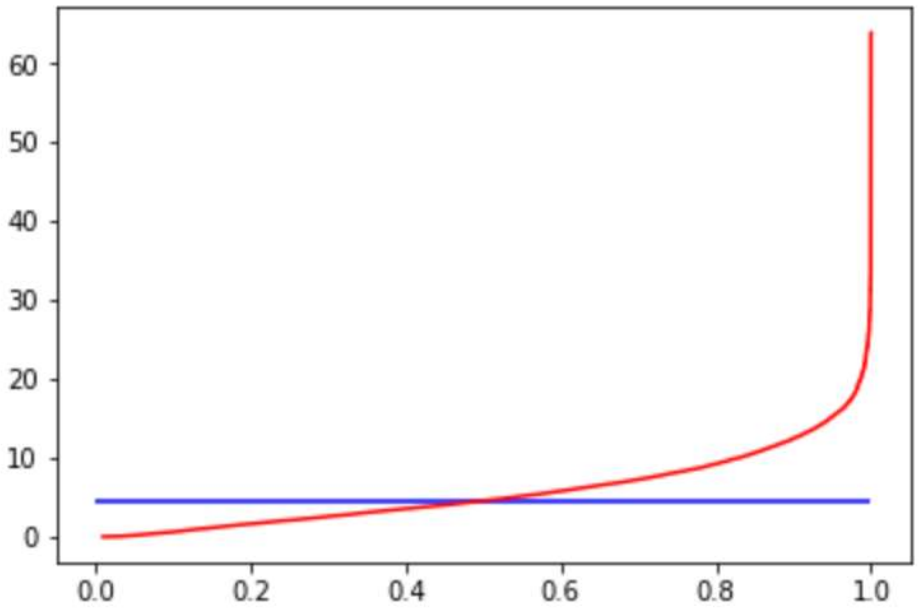
```

x = np.linspace(f.ppf(0.01, 1, 105),
                f.ppf(0.999999, 1, 105), 100)

fig, ax = plt.subplots(1,1)
rv = f(1, 105)
ax.plot(x, rv.cdf(x), 'k-', lw=2, label='frozen pdf');
plt.step(F_stats[:10000],t,c='r');
plt.hlines(0.95,0,35,colors='g')
plt.vlines(4.478786,0,1,colors='b')
```

```
<matplotlib.collections.LineCollection at 0x1cc88bc6b00>
```



Inverse cdf of bootstrapped distribution

```
1  plt.step(t,F_stats[:10000],c='r');
2  plt.hlines(4.478786,0,1,colors='b');
```



So how can we use bootstrapping in our tree model

To improve the performance of the model, we want to train several trees and then average over the predictions (actually voting in this case). This is called an ensemble model.

The idea behind ensemble models is that each estimator (tree in this case) is not in itself a very good predictor and may have a high error. The error is a random variable with mean 0. If our estimators are independent and produce independent samples of the error then the average will approach the mean =0.

So we want to create estimators that are as independent as possible

One of the simplest ways to do this is to use a BaggingClassifier.

We randomly select, with replacement, data from the training set, so each random sample is the same size as the original data set but with repetitions i.e. we create bootstrap samples

For each bootstrap sample we train a tree and take the most common prediction among the ensemble.

We try a simple example:

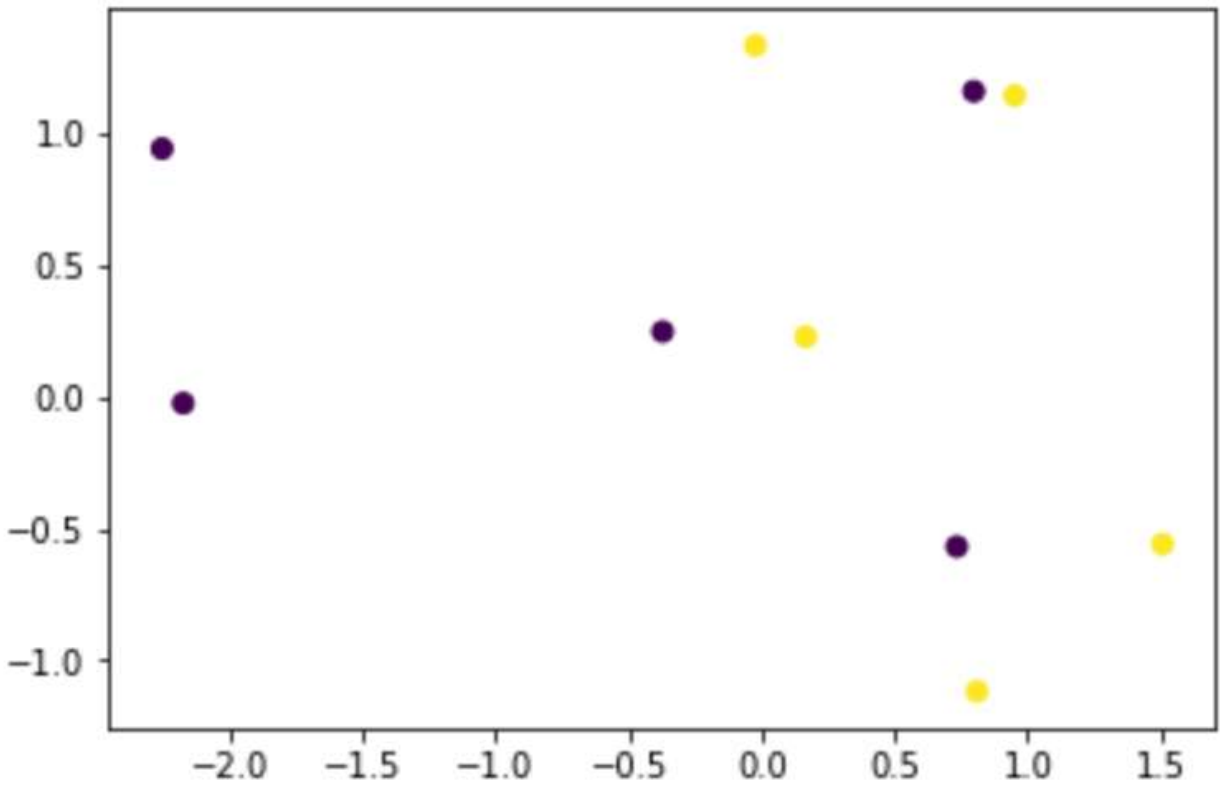
We first make a small data set of points

```
from sklearn.datasets import make_classification,make_moons
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
```

```
N = 10
```

```
X,y = make_moons(n_samples=N,noise=1.0)
```

```
plt.scatter(X[:,0],X[:,1],c=y);
```



We fit a decision tree to the data

```
from sklearn.tree import DecisionTreeClassifier
```

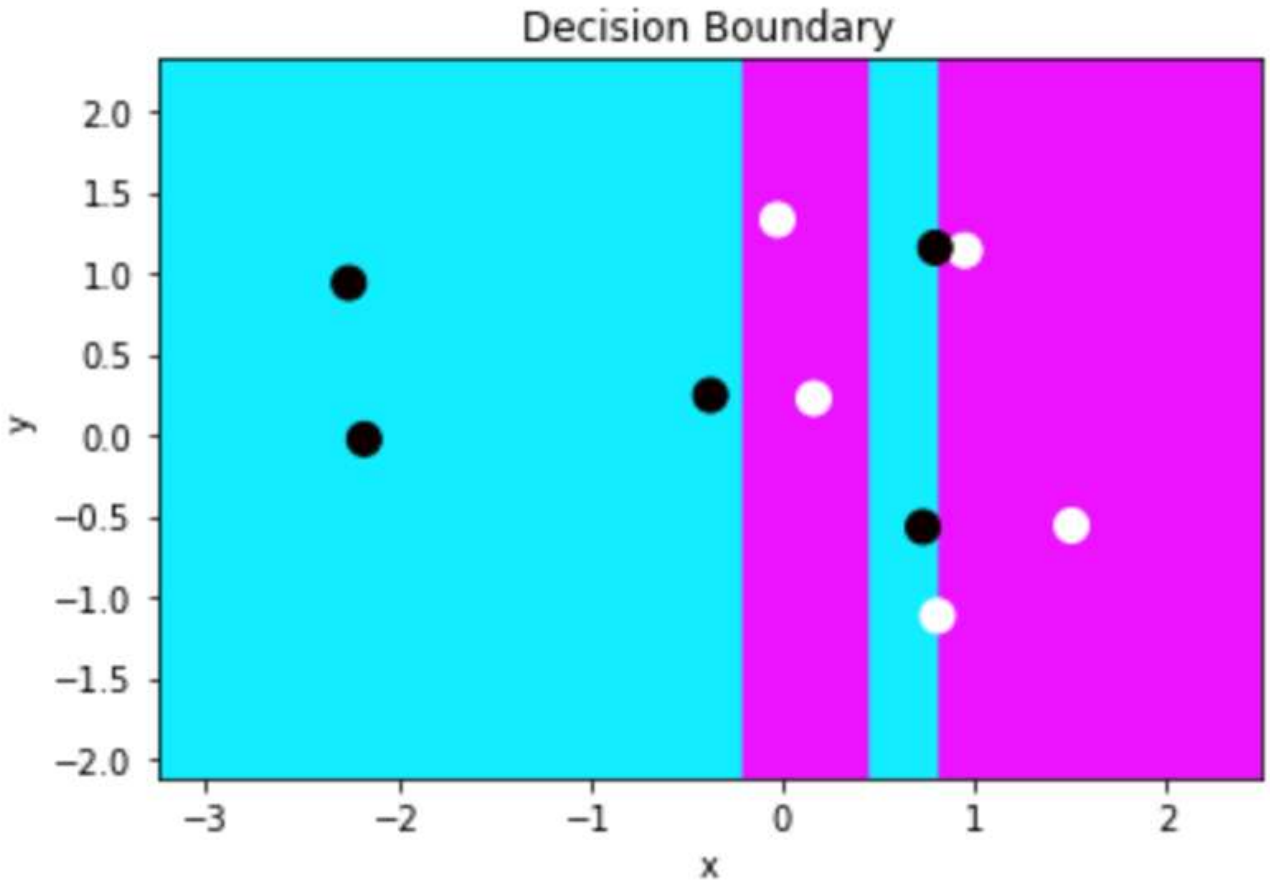
```
t_0 = DecisionTreeClassifier()
t_0.fit(X,y)
```

```
]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            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')
```

We write a small function to display the decision boundary which can be used in many other examples so it may be a good idea to save it separately

```
def plot_decision_boundary(data,labels,clf):
    plot_step = 0.02
    x_min, x_max = data[:,0].min() -1, data[:,0].max() + 1
    y_min,y_max = data[:,1].min() -1 , data[:,1].max() + 1

    xx,yy = np.meshgrid(np.arange(x_min,x_max,plot_step),
                        np.arange(y_min,y_max,plot_step))
    Z = clf.predict(np.c_[xx.ravel(),yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx,yy,Z,cmap='cool')

    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Decision Boundary')

    plt.scatter(data[:,0],data[:,1],c=labels,cmap='hot',s=100)
    plt.show()
```

```
plot_decision_boundary(X,y,t_0)
```



We write a function to generate bootstrapped trees:

This generates a sample, with replacement, of numbers 0 to N-1

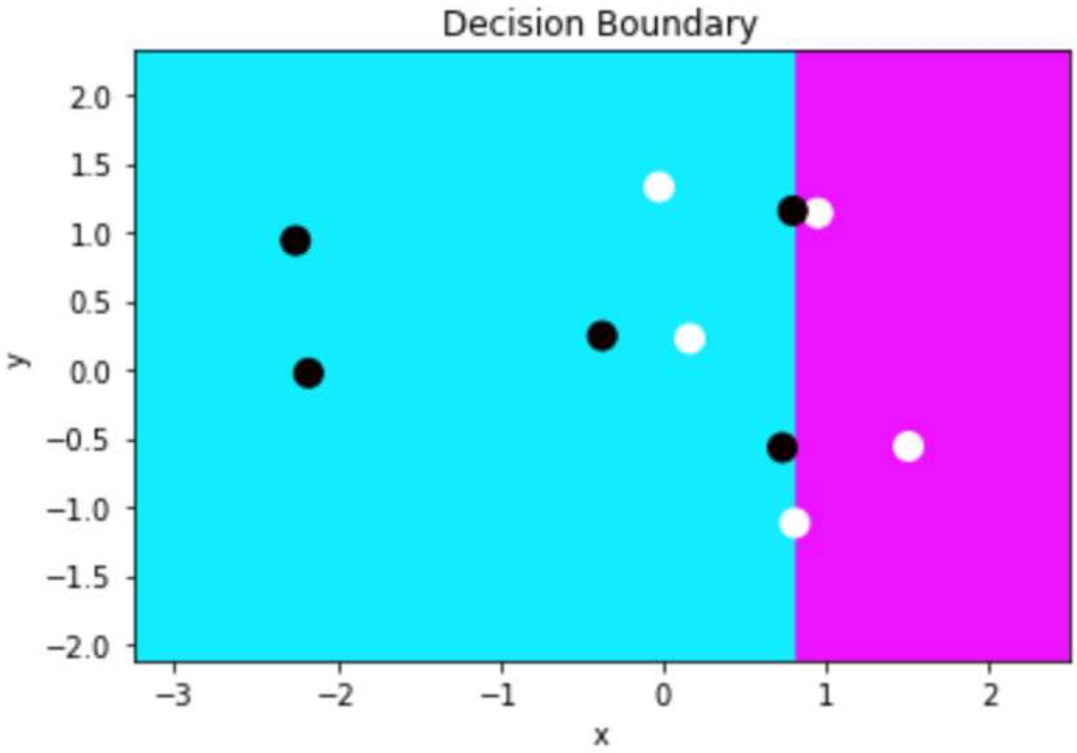
We can then choose resampled data

Fit a tree to the resampled data and return the tree

```
def make_boot_strapped_tree():
    sample = np.random.choice(range(N),N)
    X_resampled = X[sample]
    y_resampled = y[sample]
    t = DecisionTreeClassifier()
    t.fit(X_resampled,y_resampled)
    return t
```

```
t = make_boot_strapped_tree()
```

```
plot_decision_boundary(X,y,t)
```



Remark that the bootstrapped tree no longer classifies all the points correctly

Next we write a *bagged tree classifier* class

The class takes as parameter an array of bootstrapped trees

The *predict* method takes an array of points and for each point, we take the predictions of the bootstrapped trees and then take the most commonly occurring prediction (0 or 1) among the trees. This is called *voting*.
We return the array of predictions

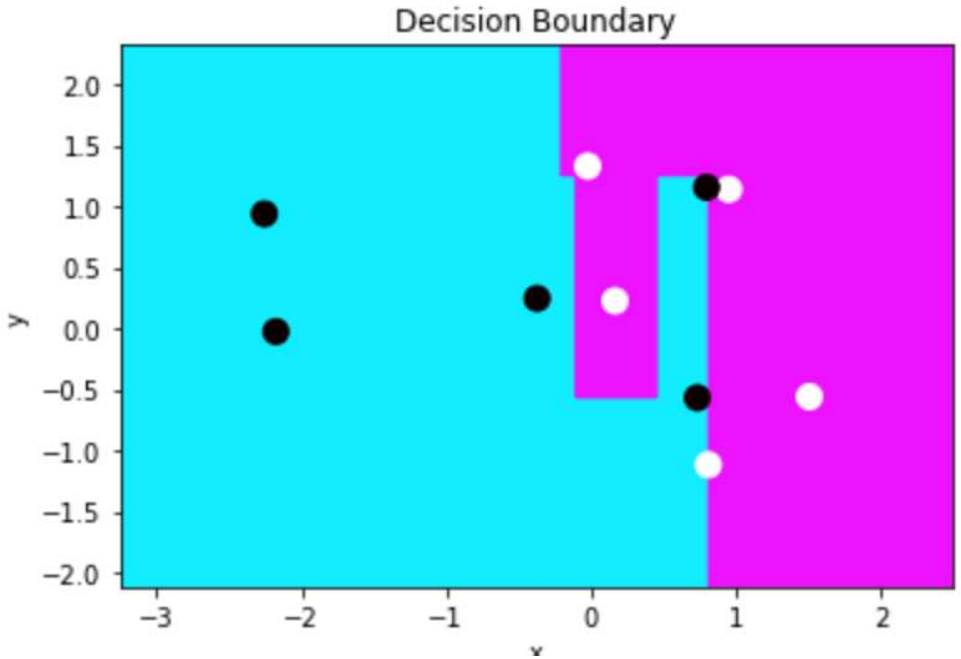
```
class bt_clf():

    def __init__(self,T):
        self.T = T

    def predict(self,X):
        p = np.array([t.predict(X) for t in self.T])
        return np.array([Counter(p_).most_common(1)[0][0] for p_ in p.T])
```

Here is what the decision boundary for a bagged tree classifier looks like

```
T = [make_boot_strapped_tree() for _ in range(5)]
```

```
b = bt_clf(T)
```

```
plot_decision_boundary(X,y,b)
```

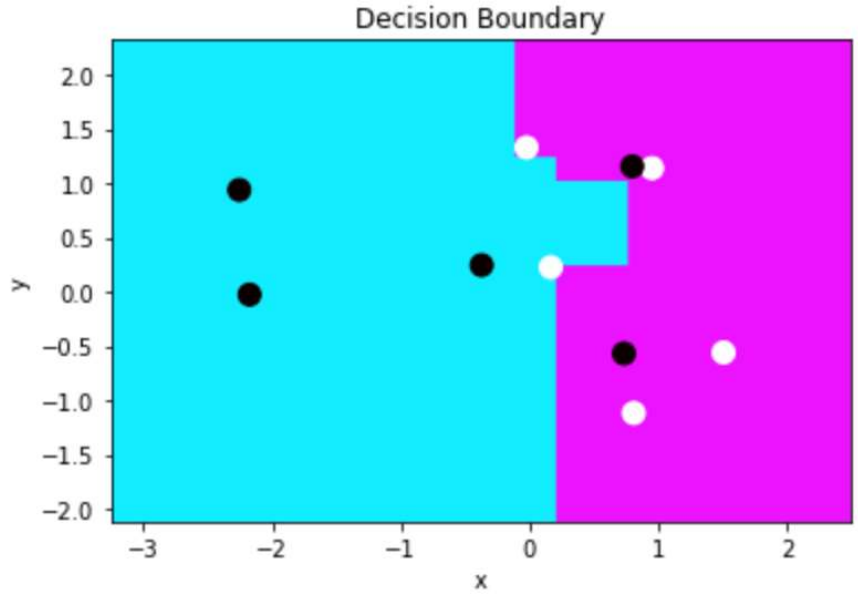


If we run it again we get a different result because the bootstrapped samples will likely be different

```
T = [make_boot_strapped_tree() for _ in range(5)]
```

```
b = bt_clf(T)
```

```
plot_decision_boundary(X,y,b)
```



sklearn.ensemble already contains a BaggingClassifier class so we do not need to write our own

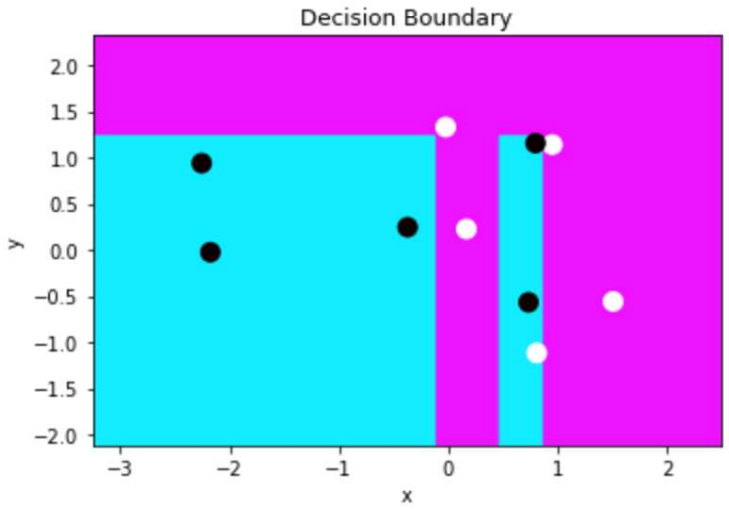
```
from sklearn.ensemble import BaggingClassifier
```

```
bt = BaggingClassifier(n_estimators=5)
```

```
bt.fit(X,y)
```

```
BaggingClassifier(base_estimator=None, bootstrap=True,
         bootstrap_features=False, max_features=1.0, max_samples=1.0,
         n_estimators=5, n_jobs=None, oob_score=False, random_state=None,
         verbose=0, warm_start=False)
```

```
plot_decision_boundary(X,y,bt)
```



We can apply this technique to our dataset of stock data:

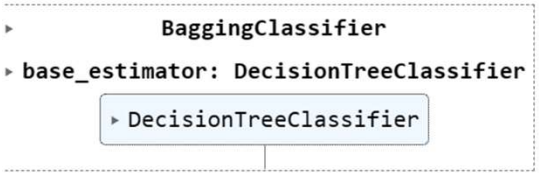
```
data.reset_index(inplace=True,)
data.set_index('date',inplace=True)

data.sort_index()
data.head()

df_train = data.loc['2001-01-01':'2004-01-01']

df_valid = data.loc['2004-04-01':'2004-07-01']
df_test = data.loc['2004-07-01':'2004-10-01']

train = df_train.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)

train_stock_returns = df_train['next_period_return']
valid_stock_returns = df_valid['next_period_return']
test_stock_returns = df_test['next_period_return']

y_train = df_train['rel_performance']
y_valid = df_valid['rel_performance']
y_test = df_test['rel_performance']

y_train = y_train.values
y_valid = y_valid.values
y_test = y_test.values
```

The BaggingClassifier needs a base classifier (in none is specified it defaults to a DecisionTree and it will fit this tree to maximal precision on the training set but this will result in massive overfitting on the validation set. This is the reason we specify a tree with a lower precision)

```
1 t_clf = DecisionTreeClassifier(max_depth=6,min_samples_leaf=200)
```

```
1 bg_clf = BaggingClassifier(t_clf,n_estimators=40,random_state=123,n_jobs=-1)
```

```
1 bg_clf.fit(train,y_train)
```



We use the trained classifier to get predictions on the validation set

```
1 bg_clf.score(train,y_train)
```

```
0.5914504662542701
```

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

```
0.5619047619047619
```

```
1 pred_valid = bg_clf.predict(valid)
```

```
1 Counter(pred_valid)
```

```
Counter({1: 1127, -1: 28})
```

```
1 Counter(y_valid)
```

```
Counter({1: 628, -1: 402, 0: 125})
```

```
1 profit = (pred_valid*valid_stock_returns).sum()
2 profit
```

```
-12.83161799999998
```

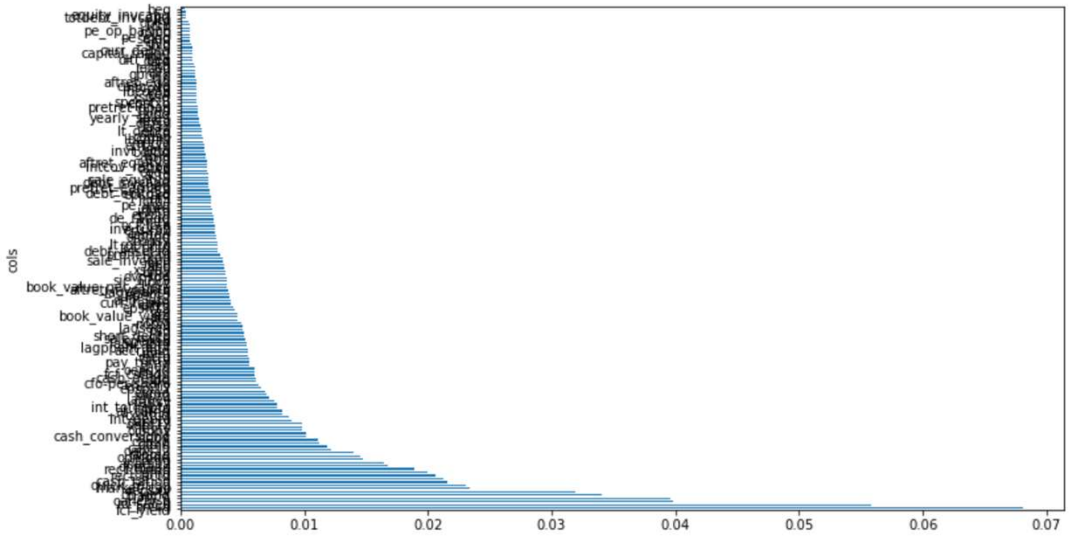
Next we look at feature importances. The BaggingClassifier does not have a feature_importances_ attribute. Instead we get the feature_importances_ for each of the trees in the ensemble and average

```
def bagging_feat_importance(m, df):
    feature_importances = []
    for est in m.estimators_:
        fi = est.feature_importances_
        feature_importances.append(fi)
    feature_importances = np.array(feature_importances)

    return pd.DataFrame({'cols':df.columns, 'feat_imp':np.mean(feature_importances,axis=0)}
                        ).sort_values('feat_imp', ascending=False)


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

We eliminate those features that do not occur in any of the trees, i.e. those with feature importance = 0.0

```
1 fi = bagging_feat_importance(bg_clf,train)
```

```
1 features = fi[(fi['feat_imp'] > 0.0)]
2 plot_fi(features);
```




```
1 len(features)
```

```
155
```

We are left with 155 features

Next we run an Optuna study to find optimal hyperparameters

```
def objective(trial:Trial,train=None,labels=None,val=None,val_labels=None,val_rets=None):

    t_min_samples_leaf = trial.suggest_int('min_samples_leaf',100,1200,step=100)
    t_max_depth = trial.suggest_int('max_depth',5,25,step=5)
    t_n_estimators = trial.suggest_int('n_estimators',5,50,step=5)


    t_clf = DecisionTreeClassifier(min_samples_leaf = t_min_samples_leaf,max_depth=t_max_depth,random_state=123)
    bg_clf = BaggingClassifier(t_clf,n_estimators=t_n_estimators,random_state=123,n_jobs=1)
    bg_clf.fit(train,labels)

    preds = bg_clf.predict(val)
    profit = (preds * val_rets).sum()

#     score = bg_clf.score(val,val_labels)

    return profit
```

We need hyperparameters for the base tree and for the ensemble (which is just the number of trees in the ensemble)

```
study.best_params
```

```
{'min_samples_leaf': 500, 'max_depth': 20, 'n_estimators': 5}
```

```
t_cfl = DecisionTreeClassifier(**{'min_samples_leaf': 500, 'max_depth': 5},random_state=123)
bg_clf = BaggingClassifier(t_clf,n_estimators=5,random_state=123)
```

```
bg_clf.fit(train,y_train)
```

BaggingClassifier

base_estimator: DecisionTreeClassifier

DecisionTreeClassifier

```
pred_valid = bg_clf.predict(valid)
```

```
bg_clf.score(train,y_train)
```

```
0.5923737420367464
```

```
bg_clf.score(valid,y_valid)
```

```
0.5645021645021645
```

```
confusion_matrix(pred_valid,y_valid)
```

```
array([[ 33,   2,   9],
       [  0,   0,   0],
       [369, 123, 619]], dtype=int64)
```

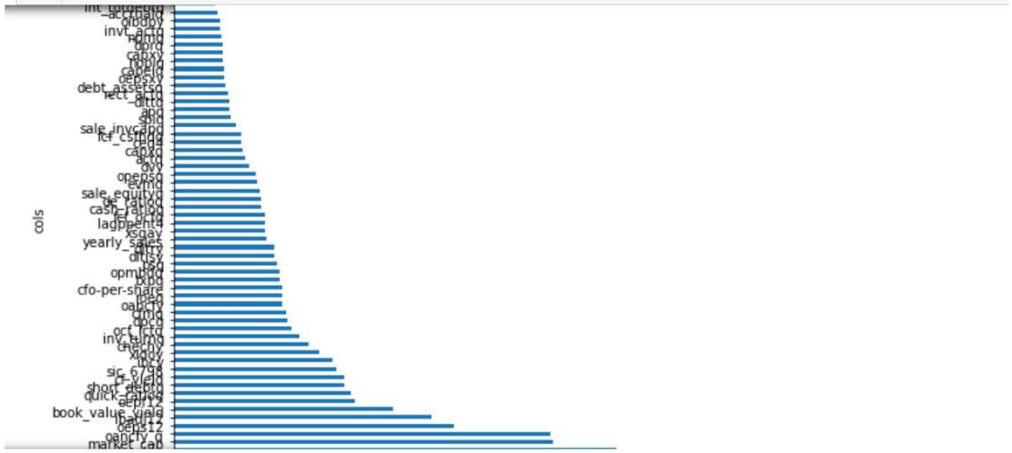
```
(pred_valid * df_valid['next_period_return'].values).sum()
```

```
-10.95675999999997
```

Using the optimal hyperparameters, we lower the number of features appearing in any of the trees in the ensemble to 62

```
1 fi = bagging_feat_importance(bg_clf,train)
```

```
1 features = fi[(fi['feat_imp'] > 0.0)]
2 plot_fi(features);
```




```
1 features
```

	cols	feat_imp
156	fcf_yield	0.103983
40	oiadpq	0.072066
176	rd_saleq	0.054744
92	market_cap	0.046889
102	oancfy_q	0.046559
...	...	...
142	int_totdebtq	0.005082
59	txtq	0.004987
112	gpmq	0.004585
132	aftret_invcapxq	0.003935
15	dpq	0.003563

62 rows × 2 columns

Let’s compute the Shapley values

```
1 import shap
```

```
1 def model(features):
2     tree_features = features[features.columns[:-1].values]
3
4     pred = bg_clf.predict(tree_features)
5
6     ret = pred * features[features.columns[-1]]
7
8     return ret
```

```
1 explainer = shap.explainers.Permutation(model,valid_1_norm,)
```

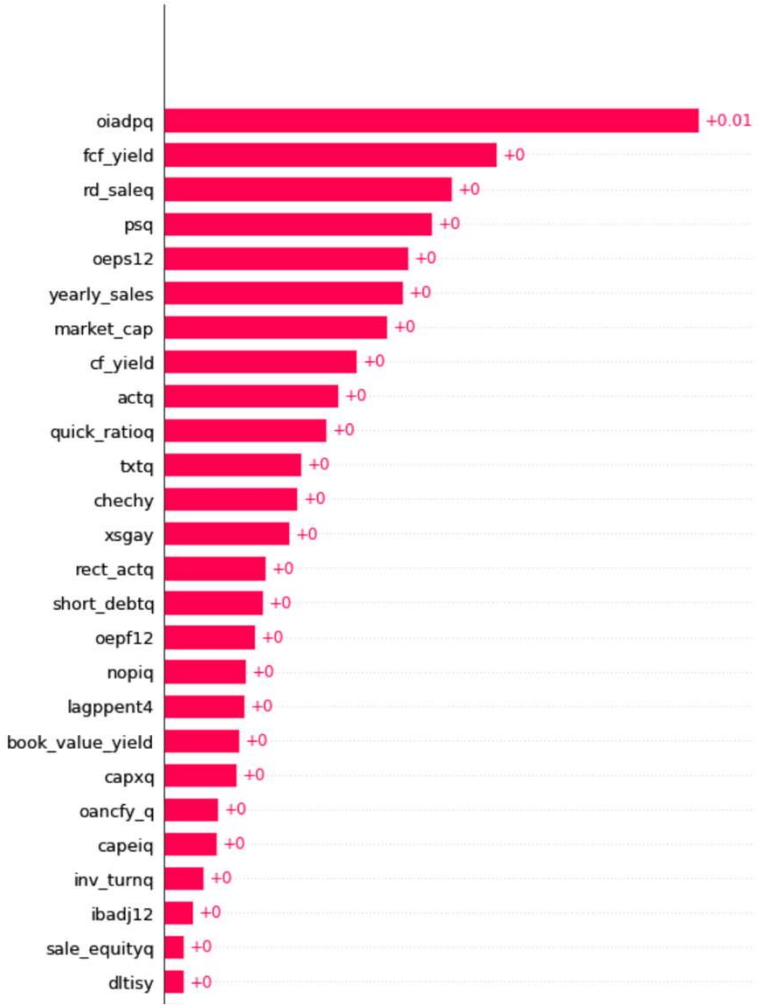
```
1 shap_values = explainer(valid_1_norm,max_evals=2000,)
```

```
Permutation explainer:  50%|████████████████████▌                  | 572/1155 [1:06:16<1:02:39,  6.45s/it]
```

```
1 shap_values.values.shape
```

```
(1155, 63)
```

```
1 shap.plots.bar(shap_values[:,:-1],max_display=60,)
```



We actually get a lot of features with non-zero Shapley values

```
1 shap_cols = cols[np.abs(shap_values[:,:-1].values).mean(axis=0)>0.000]
```

```
1 shap_cols
```

```
array(['fcf_yield', 'oiadpq', 'rd_saleq', 'market_cap', 'oancfy_q',
       'oeps12', 'ibadj12', 'book_value_yield', 'oepf12', 'quick_ratioq',
       'short_debtq', 'cf_yield', 'sic_6798', 'ibcy', 'xidoy', 'chechy',
       'inv_turnq', 'ocf_lctq', 'dpcq', 'cfmq', 'oancfy', 'roeq',
       'cfo-per-share', 'txpq', 'opmbdq', 'psq', 'dltisy', 'dltry',
       'yearly_sales', 'xsgay', 'lagppent4', 'fcf_ocfq', 'cash_ratioq',
       'de_ratioq', 'sale_equityq', 'evmq', 'opepsq', 'dvy', 'actq',
       'capxq', 'ceq4', 'fcf_csfhdq', 'sale_invcapq', 'apq', 'dlttq',
       'rect_actq', 'capeiq', 'nopiq', 'capxy', 'npmq', 'invt_actq',
       'oibdpy', 'accrualq', 'int_totdebtq', 'txtq', 'gpmq',
       'aftret_invcapxq', 'dpq'], dtype=object)
```

```
1 len(shap_cols)
```

```
58
```

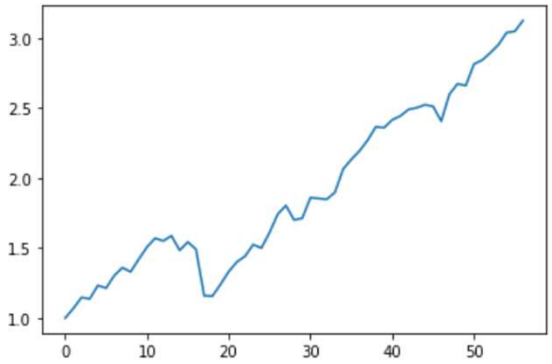
Now we can backtest our model exactly like last lecture

```
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]

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

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]
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]
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]

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

```
scalers = [StandardScaler() for _ in range(len(training_data))]

opt_training_data = [pd.DataFrame(scalers[i].fit_transform(training_frames[i][shap_cols].values),columns=shap_cols) for
opt_valid_data = [pd.DataFrame(scalers[i].transform(valid_frames[i][shap_cols].values),columns=shap_cols) for i in range
opt_test_data = [pd.DataFrame(scalers[i].transform(test_frames[i][shap_cols].values),columns=shap_cols) for i in range(l
```

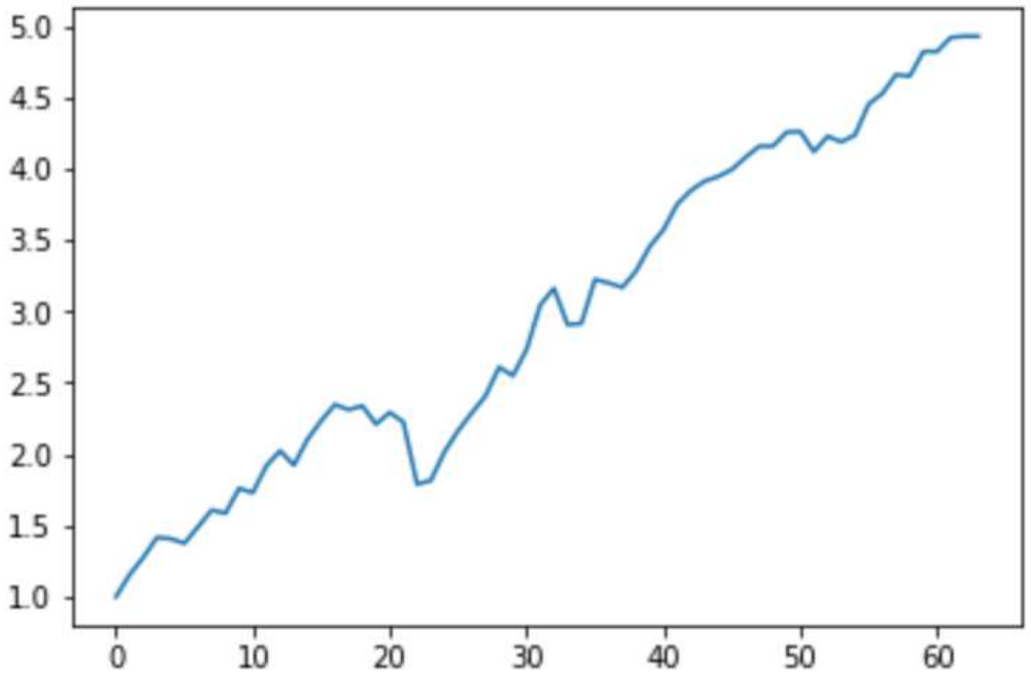
```
x = [1]
ret = []

for i in range(len(start_dates)-1):
        t_clf.fit(opt_training_data[i],training_labels[i])

        preds = t_clf.predict(opt_test_data[i])
        profit_i = (preds*test_frames[i]['next_period_return']).sum()
        ret.append(profit_i)
        num_names = len(opt_test_data[i])
        x.append(x[i] + (x[i]/num_names)*profit_i)


```

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



The graph of the cumulative returns from 2003 - 2018

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

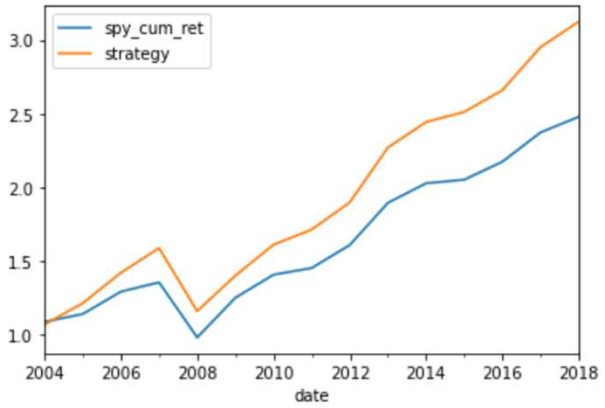


We compare this to the cumulative returns on the index (SPY) and annualize it

```
SPY = pd.read_pickle(r'C:\Users\niels\OneDrive\Machine Learning 2022\Lecture 2\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['strategy'] = x
```

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

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



Computing annualized Sharpe Ratio

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

```
Strategy Sharpe Ratio:  0.8072158791958496
```

```
1 strategy_std = (SPY['strategy'] - 1).diff().std()
2 strategy_std
```

```
0.1817757218883073
```

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

```
SP Sharpe Ratio:  0.6324646748042164
```

```
1 x[-1]
```

```
3.123310489952453
```

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

```
2.4779340000000003
```