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
```

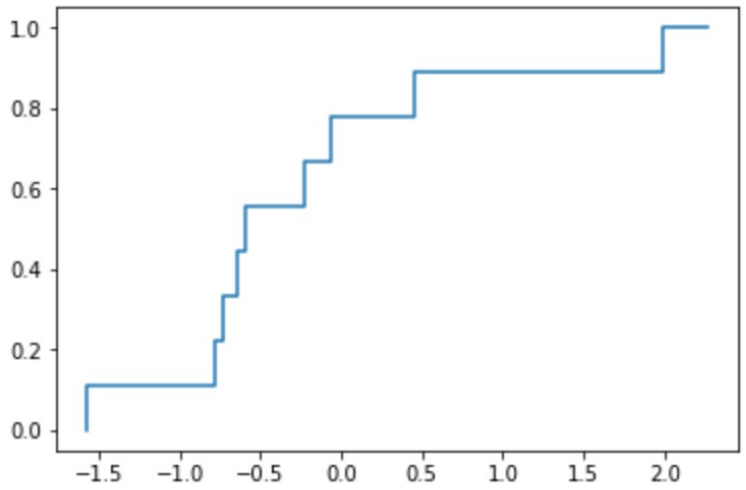
```
1  X = sorted(X)
```

```
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)
```

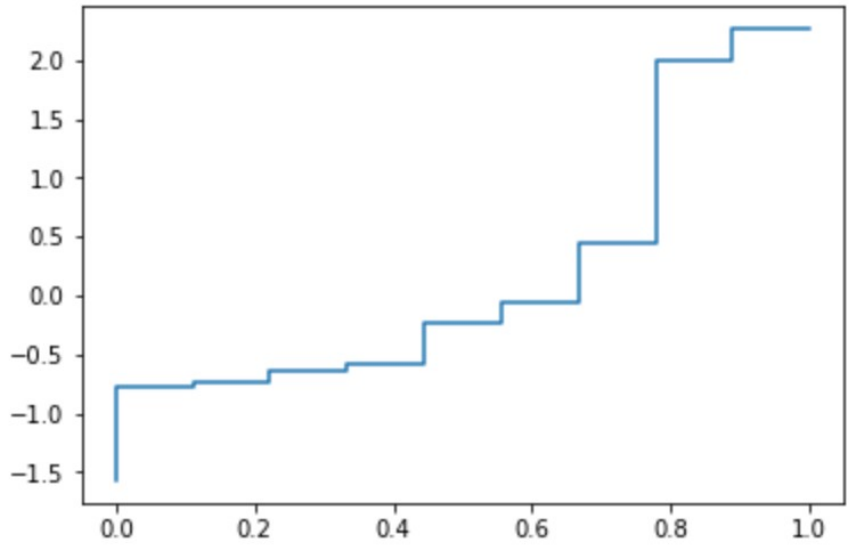
```
1  plt.step(X,y);
```



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.

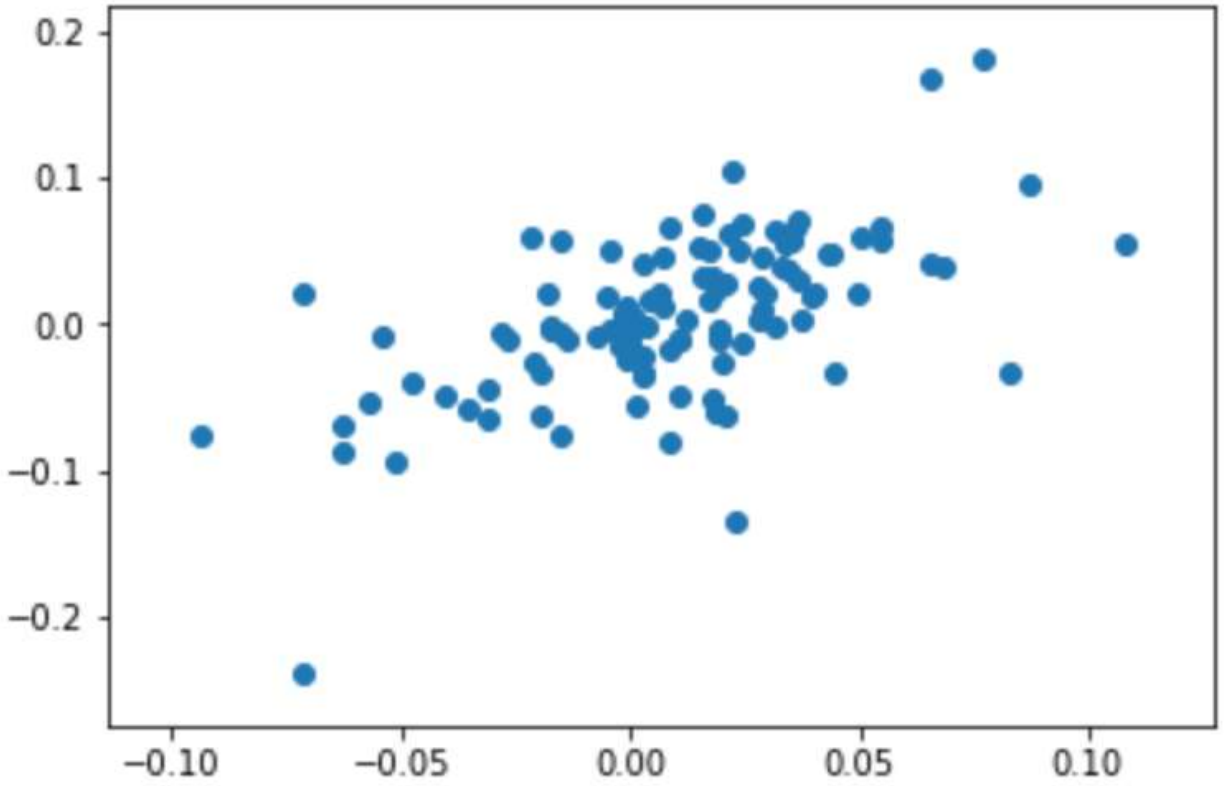
We then replace the returns with the excess returns

```
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)
```

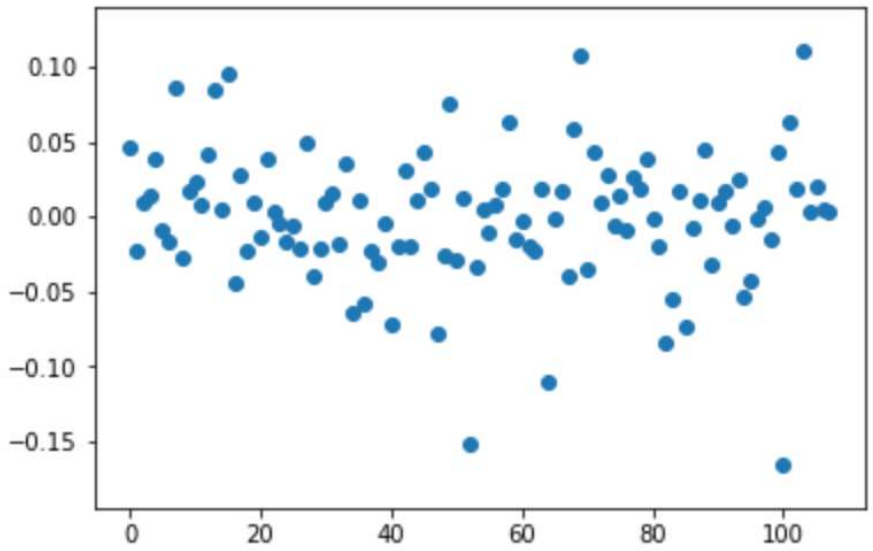
```
0.9655071347625888 -0.004005175608166194
```

Next we compute the residuals

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

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

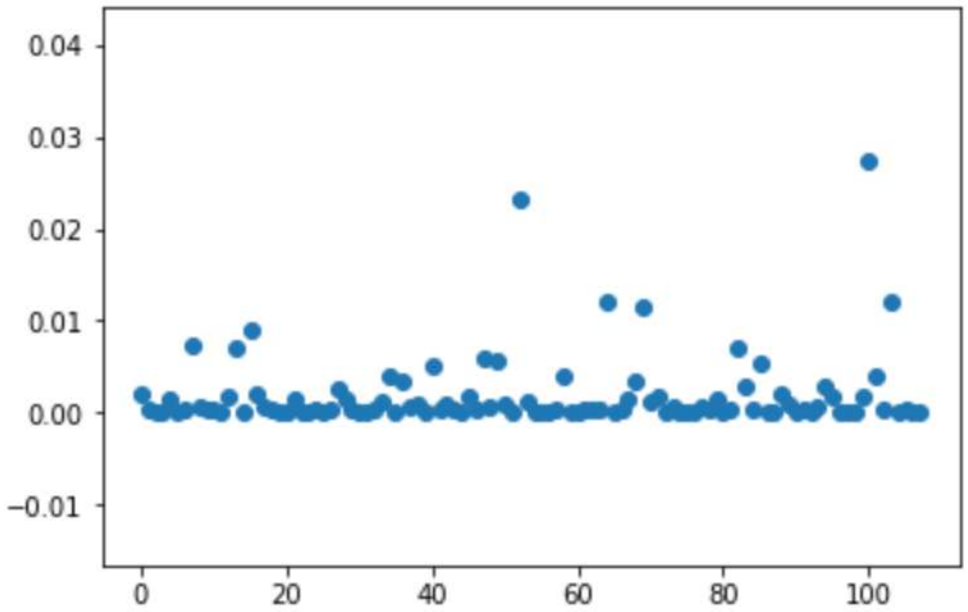
```
<matplotlib.collections.PathCollection at 0x1c922e8c9b0>
```



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.

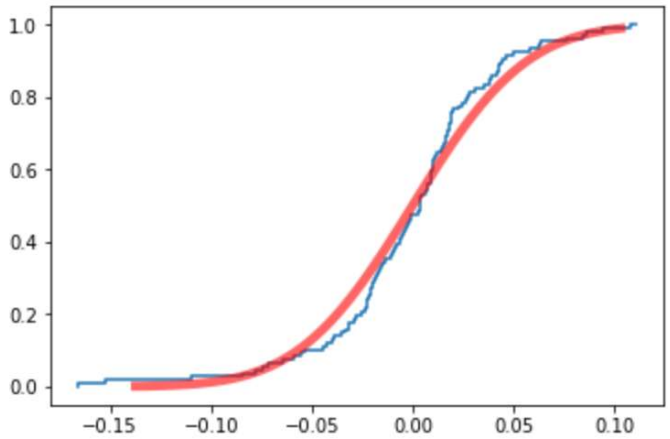
We can now plot the empirical cdf of the residuals and overlay it with the cdf of the normal distribution

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

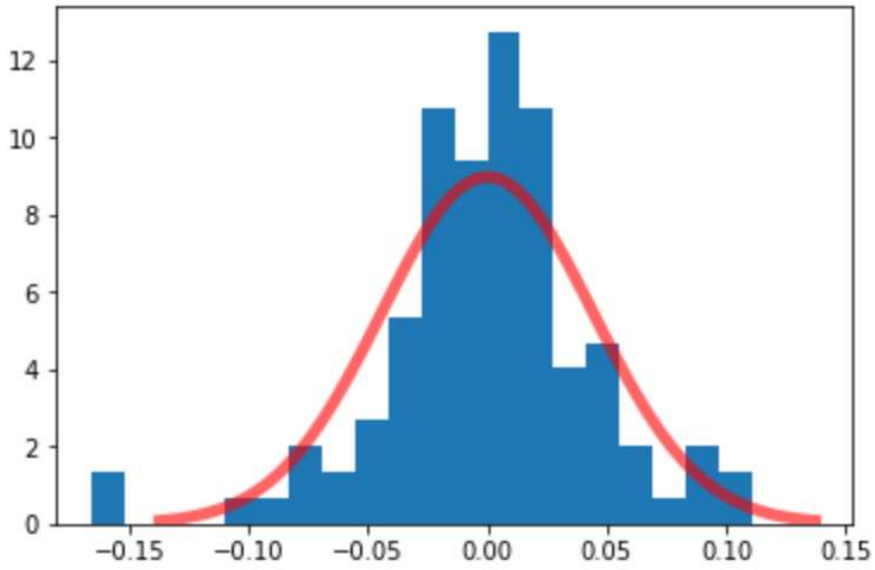
```
p = np.linspace(0,1,len(df))
```

```
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

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

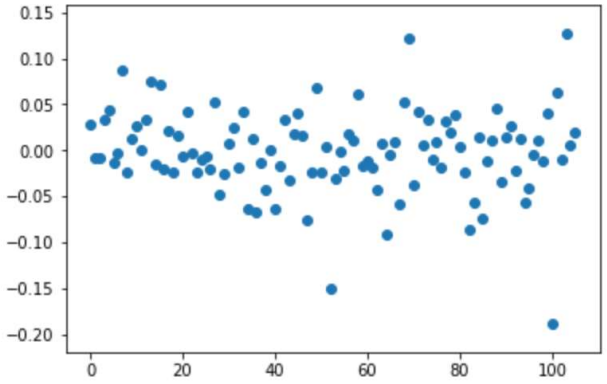
Under the normality assumption the F-statistic follows an F(1,m-2) distribution, which we can use to test the hypothesis

$$slope = b$$

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)

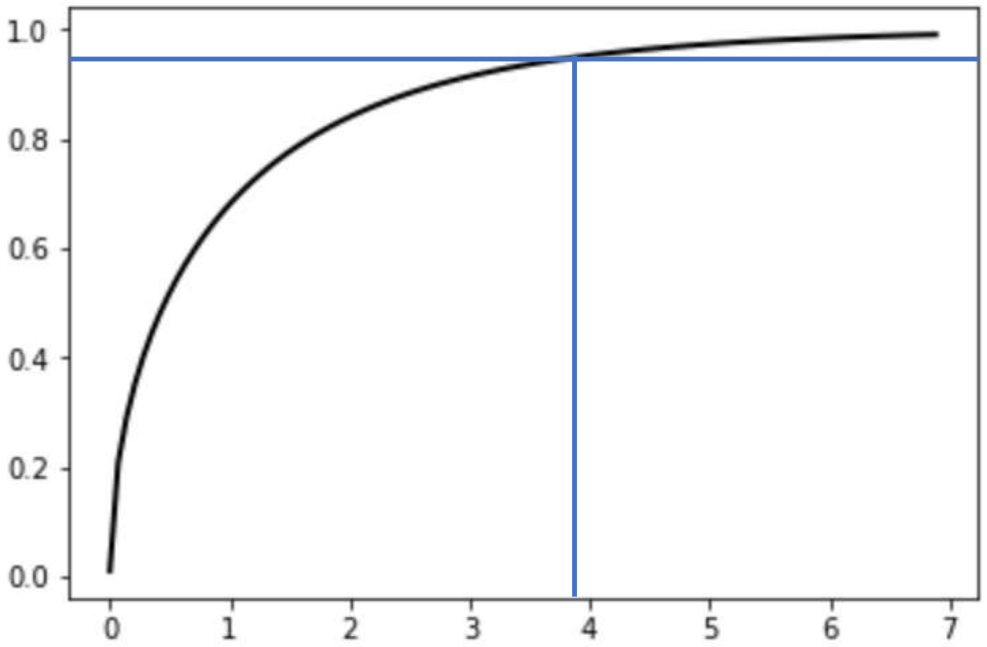
```
1  from scipy.stats import f
```

```
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.

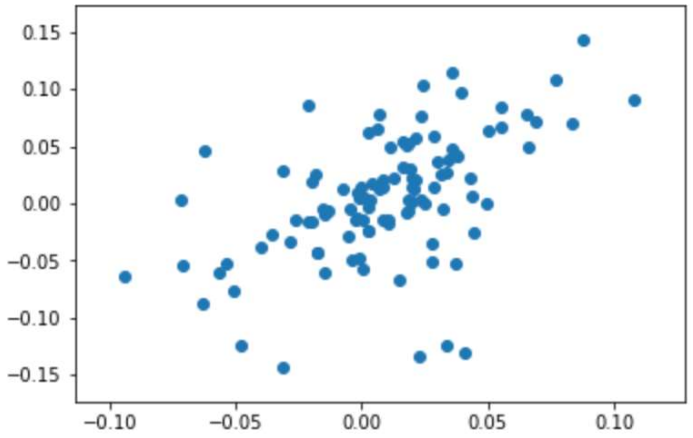
The bootstrapping works by resampling the residuals and compute a new set of values for the dependent variable

```
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)
```

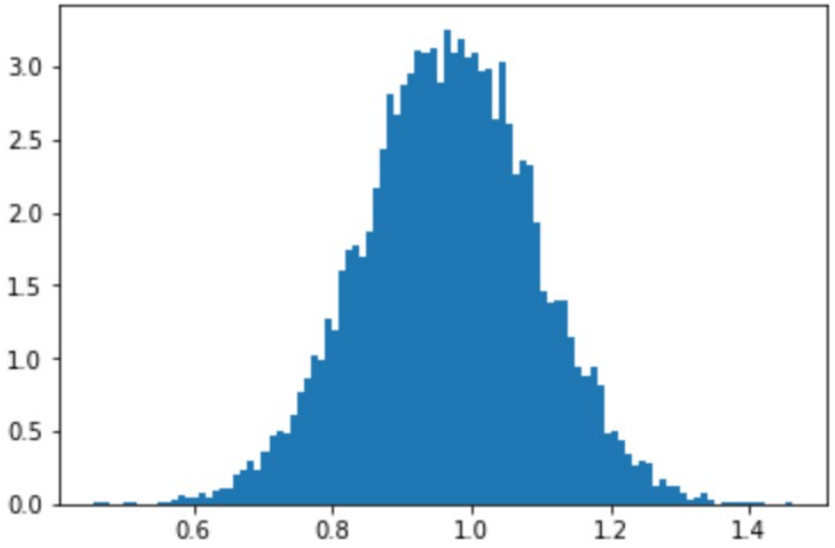
```
<matplotlib.collections.PathCollection at 0x1c9231fc240>
```



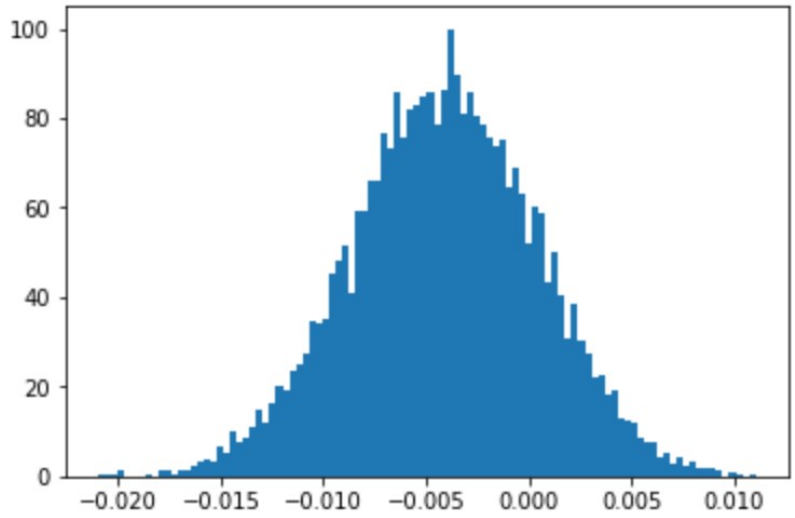
We can then run linear regression to get bootstrapped values for the slope and the intercept. Doing this many times give us empirical distributions for the coefficients in the regression

```
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);
```

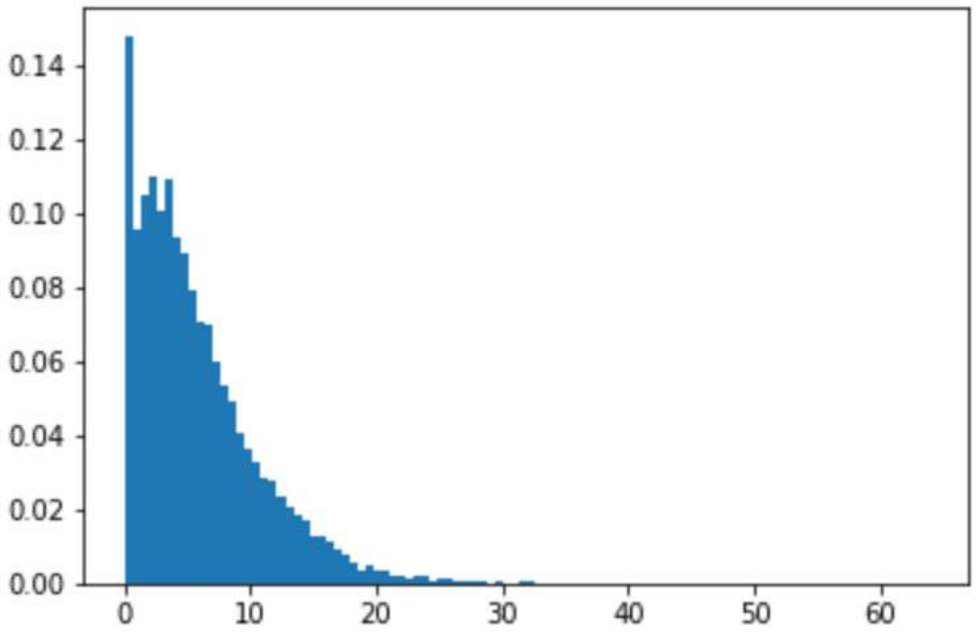


Bootstrapped values of the distribution of the F-statistic under the hypothesis slope = 0.7

```
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);
```



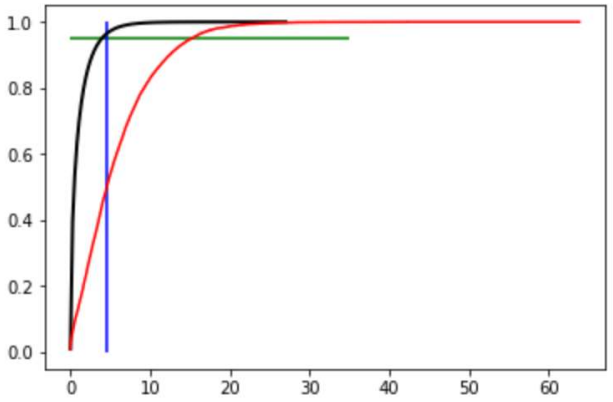
CDFs of F-distribution and bootstrapped distribution

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

```
F_stats.sort()
```

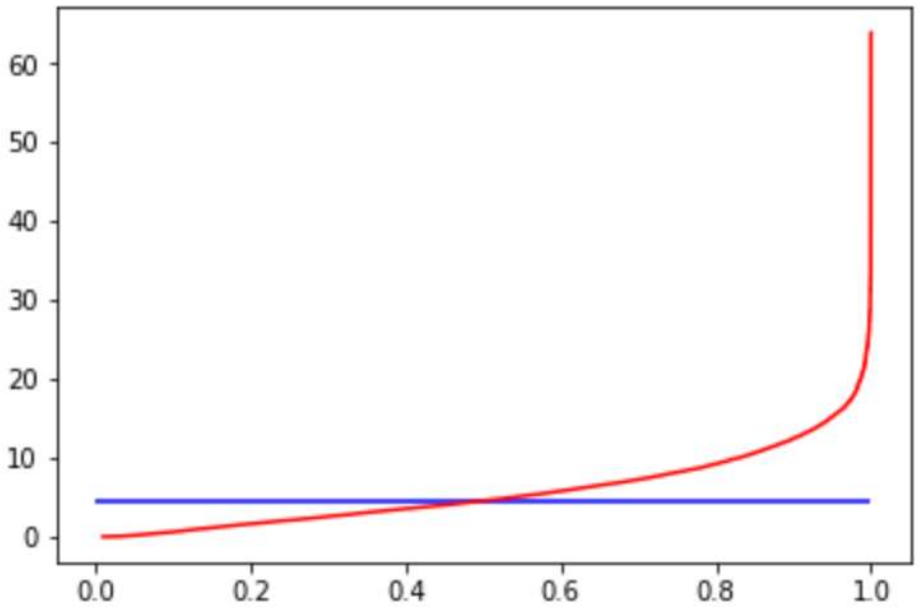
```

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:

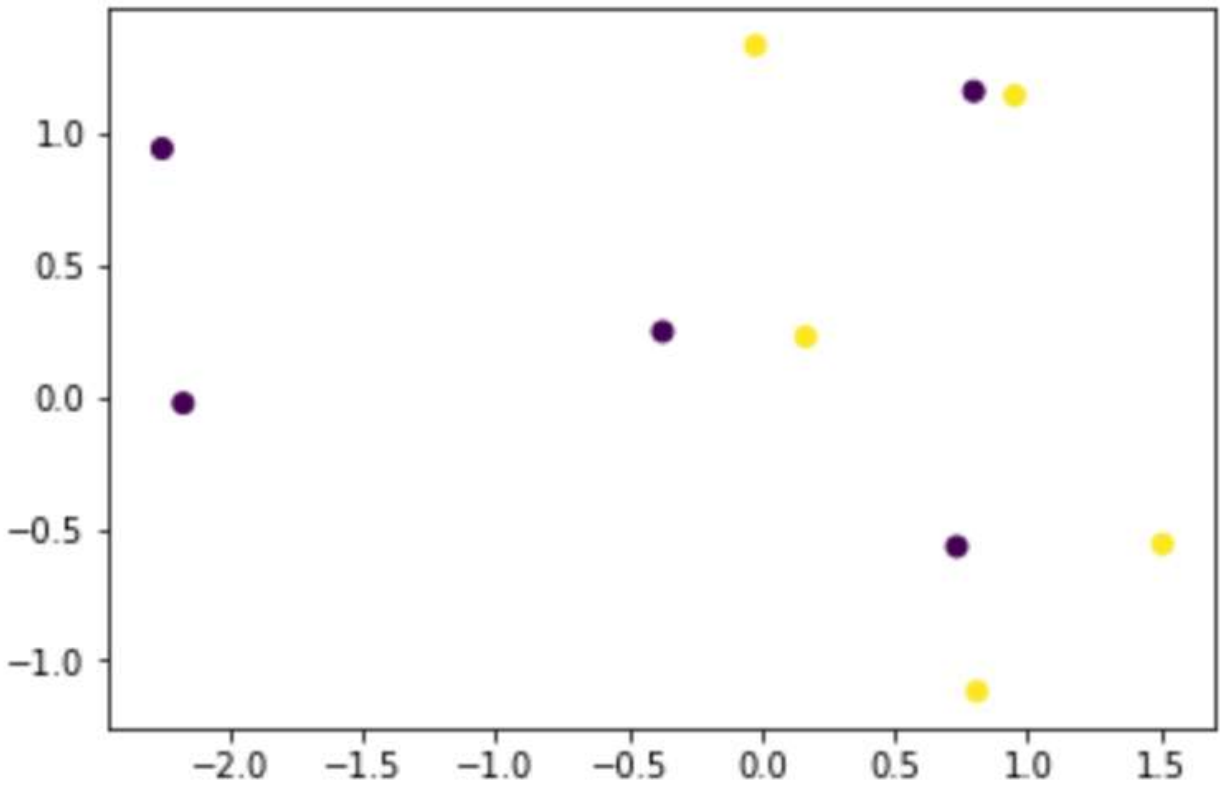
We first make a small data set of points

```
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
```

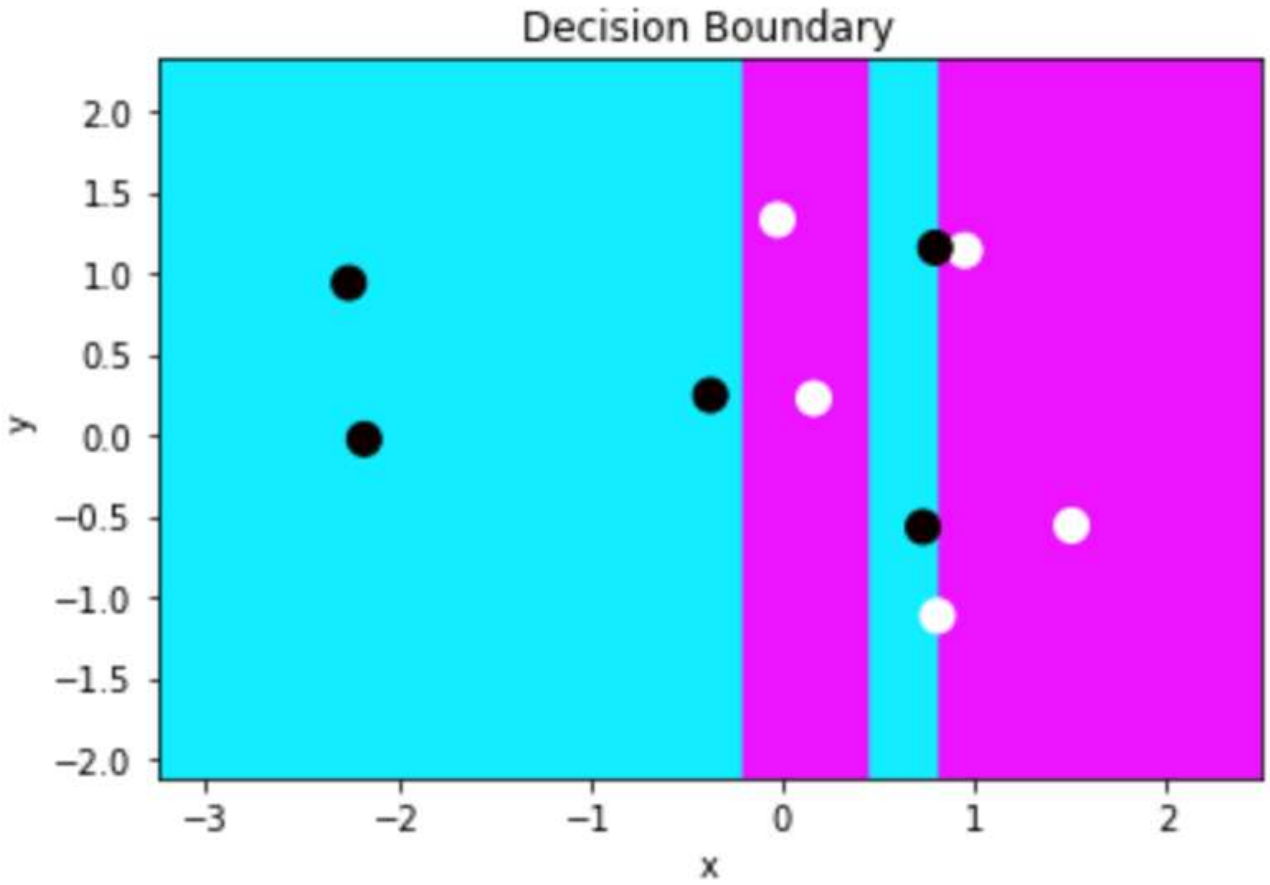
```
t_0 = DecisionTreeClassifier()
t_0.fit(X,y)
```

```
]: 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

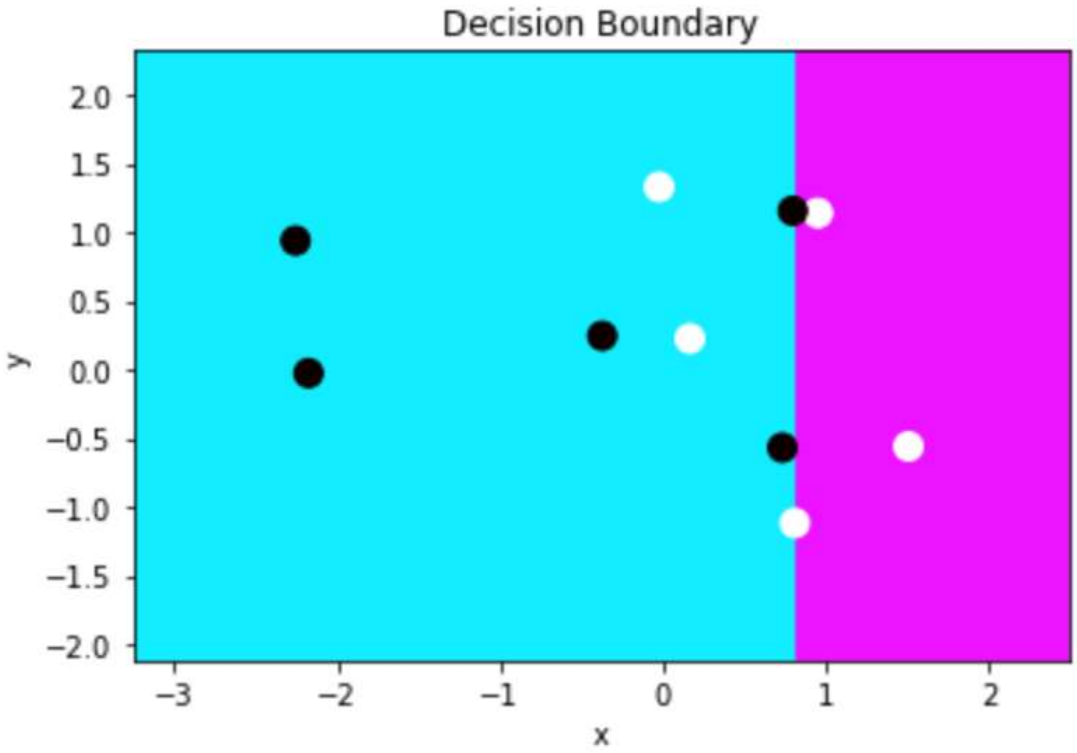
We can then choose resampled data

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

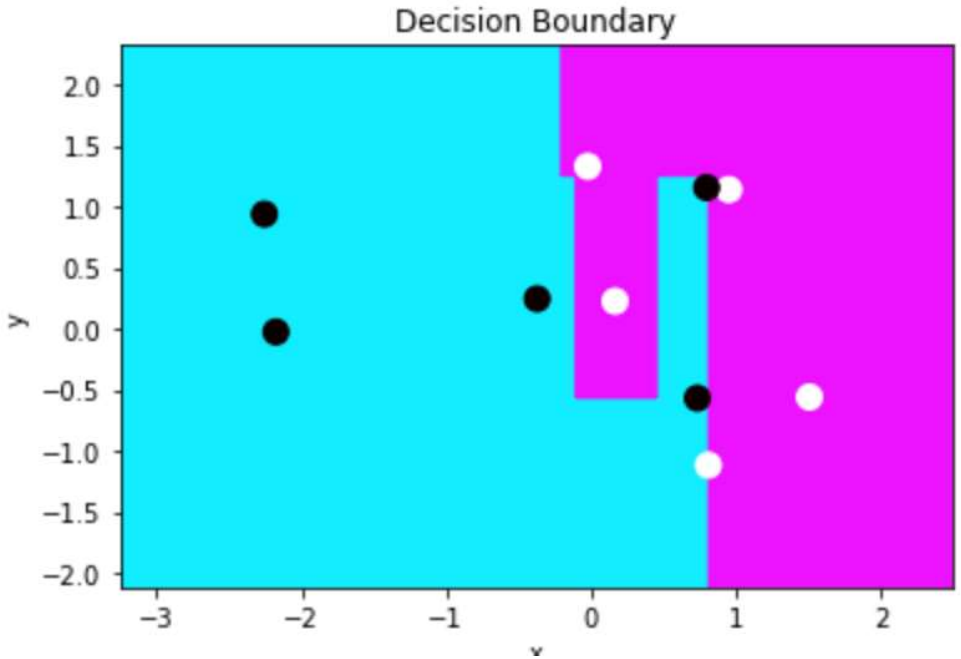
```
class bt_clf():

    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)
```

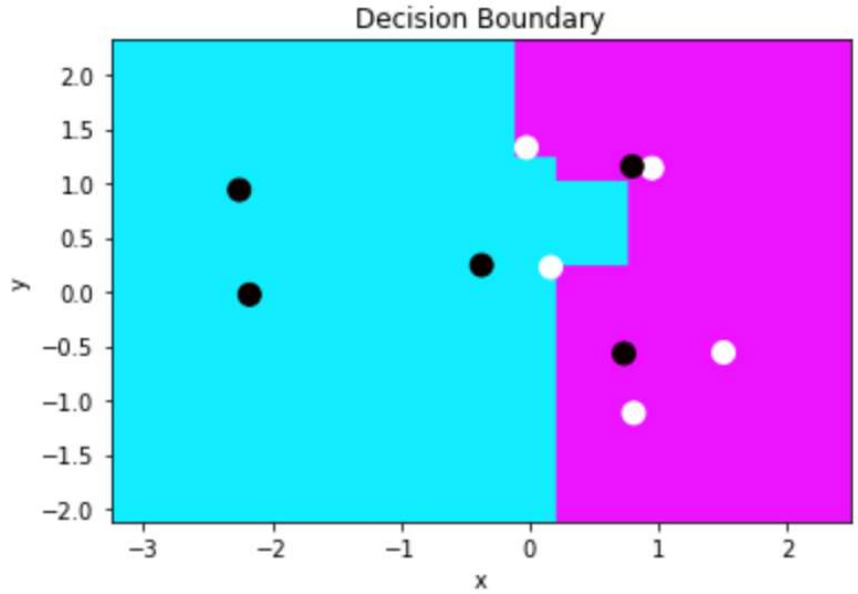


If we run it again we get a different result because the bootstrapped samples will likely be different

```
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

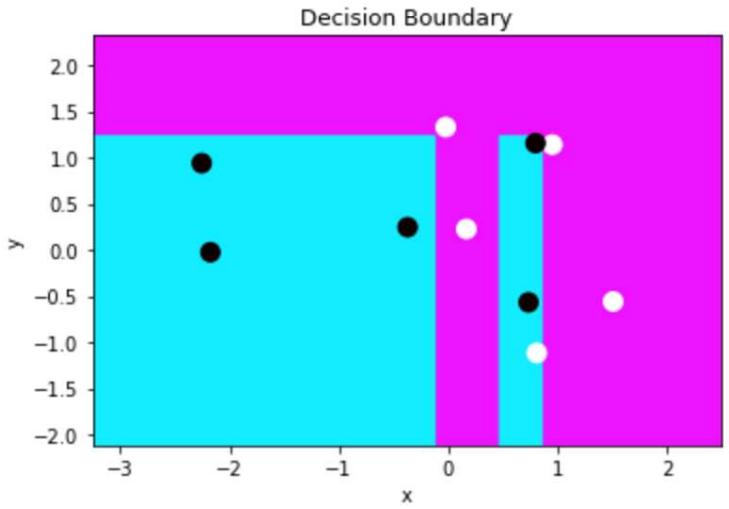
```
from sklearn.ensemble import BaggingClassifier
```

```
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:

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

We download the data sets we constructed in last lecture

```
train_1 = pd.read_pickle(r'C:\Users\niels\data\train_1.pkl')
train_2 = pd.read_pickle(r'C:\Users\niels\data\train_2.pkl')
valid = pd.read_pickle(r'C:\Users\niels\data\valid.pkl')
test = pd.read_pickle(r'C:\Users\niels\data\test.pkl')
train_1_stock_returns = pd.read_pickle(r'C:\Users\niels\data\train_1_stock_returns.pkl')
train_2_stock_returns = pd.read_pickle(r'C:\Users\niels\data\train_2_stock_returns.pkl')
y_1 = pd.read_pickle(r'C:\Users\niels\data\y_1.pkl')
y_2 = pd.read_pickle(r'C:\Users\niels\data\y_2.pkl')
y_valid = pd.read_pickle(r'C:\Users\niels\data\y_valid.pkl')
y_test = pd.read_pickle(r'C:\Users\niels\data\y_test.pkl')
valid_stock_returns = pd.read_pickle(r'C:\Users\niels\data\valid_stock_returns.pkl')
test_stock_returns = pd.read_pickle(r'C:\Users\niels\data\test_stock_returns.pkl')
data = pd.read_pickle(r'C:\Users\niels\data\data.pkl')
```

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)

```
t_clf = DecisionTreeClassifier(min_samples_leaf=1200,max_depth=8)
```

```
bg_clf = BaggingClassifier(t_clf,n_estimators=60,oob_score=True,random_state=123,n_jobs=-1)
```

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

```
BaggingClassifier(base_estimator=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=1200, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
         bootstrap=True, bootstrap_features=False, max_features=1.0,
         max_samples=1.0, n_estimators=60, n_jobs=-1, oob_score=True,
         random_state=123, verbose=0, warm_start=False)
```

```
bg_clf.score(train_1,y_1)
```

```
0.5193882216034247
```

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

```
0.47058823529411764
```

If we use the default base classifier it will fit it perfectly to the training set and we get very bad overfitting

```
t_clf = DecisionTreeClassifier(min_samples_leaf=1200,max_depth=8)
```

```
bg_clf = BaggingClassifier(n_estimators=60,oob_score=True,random_state=123,n_jobs=-1)
```

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

```
]: BaggingClassifier(base_estimator=None, bootstrap=True,
                    bootstrap_features=False, max_features=1.0, max_samples=1.0,
                    n_estimators=60, n_jobs=-1, oob_score=True, random_state=123,
                    verbose=0, warm_start=False)
```

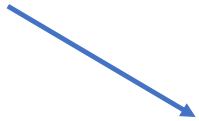
```
bg_clf.score(train_1,y_1)
```

```
]: 1.0
```

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

```
]: 0.4715370018975332
```

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



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

```
Counter(pred_valid)
```

```
]: Counter({1: 1461, -1: 647})
```

The oob_score (=out-of-bag score) is computed as follows: for each point x in the training set we look at the trees which do not contain x in their resampled data set and compute the average prediction accuracy over those trees. We then average these numbers over the whole data set

```
bg_clf.oob_score_
```

```
]: 0.5018079049083579
```

To illustrate the oob_score we return to our simple example

Data point 0: does not occur in t0,t3,t4
Data point 1: does not occur in t1,t2,t3
Data point 2: occurs in all
Data point 3: does not occur in t4
Data point 4: does not occur in t1,t2,t3,t4
Data point 5: does not occur in t0,t2,t3,t4
Data point 6: occurs in all
Data point 7: does not occur in t4
Data point 8: does not occur in t0
Data point 9: does not occur in t3

```
def make_boot_strapped_tree_with_resampled():
    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, sample
```

```
 S = np.array([make_boot_strapped_tree_with_resampled() for _ in range(5)])
```

```
S[:,1]
```

```
array([array([6, 9, 7, 1, 1, 7, 2, 1, 3, 4]),
       array([2, 7, 6, 5, 6, 0, 5, 8, 3, 9]),
       array([0, 9, 7, 2, 8, 3, 7, 6, 0, 0]),
       array([3, 7, 3, 3, 8, 8, 2, 3, 7, 6]),
       array([8, 1, 6, 6, 2, 6, 9, 1, 9, 2])], dtype=object)
```

```
predictions = []
for i in range(10):
    for j in range(5):
        if i not in S[:,1][j]:
            predictions.append([i,j,S[:,0][j].predict([X[i]]).item(),y[i]])
```

```
predictions
```

```
3]: [[0, 0, 0, 0],
     [0, 3, 0, 0],
     [0, 4, 1, 0],
     [1, 1, 1, 1],
     [1, 2, 1, 1],
     [1, 3, 1, 1],
     [3, 4, 1, 0],
     [4, 1, 1, 0],
     [4, 2, 1, 0],
     [4, 3, 1, 0],
     [4, 4, 1, 0],
     [5, 0, 0, 1],
     [5, 2, 0, 1],
     [5, 3, 1, 1],
     [5, 4, 0, 1],
     [7, 4, 1, 1],
     [8, 0, 1, 1],
     [9, 3, 1, 0]]
```

Point no. prediction

Does no occur in tree no.

label

oob_score = 2/3 + 1 + 0 + 0 + 0+ 1/4 + 0 + 1 + 1 = 3.915/10 =0.3915

oob_score does not indicate severe overfitting

```
bg_clf.oob_score_
```

```
0.5018079049083579
```

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 baggingtree_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)
    print(feature_importances.shape)

    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

```
fi = baggingtree_feat_importance(bg_clf,train_1)
```

```
(60, 725)
```

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

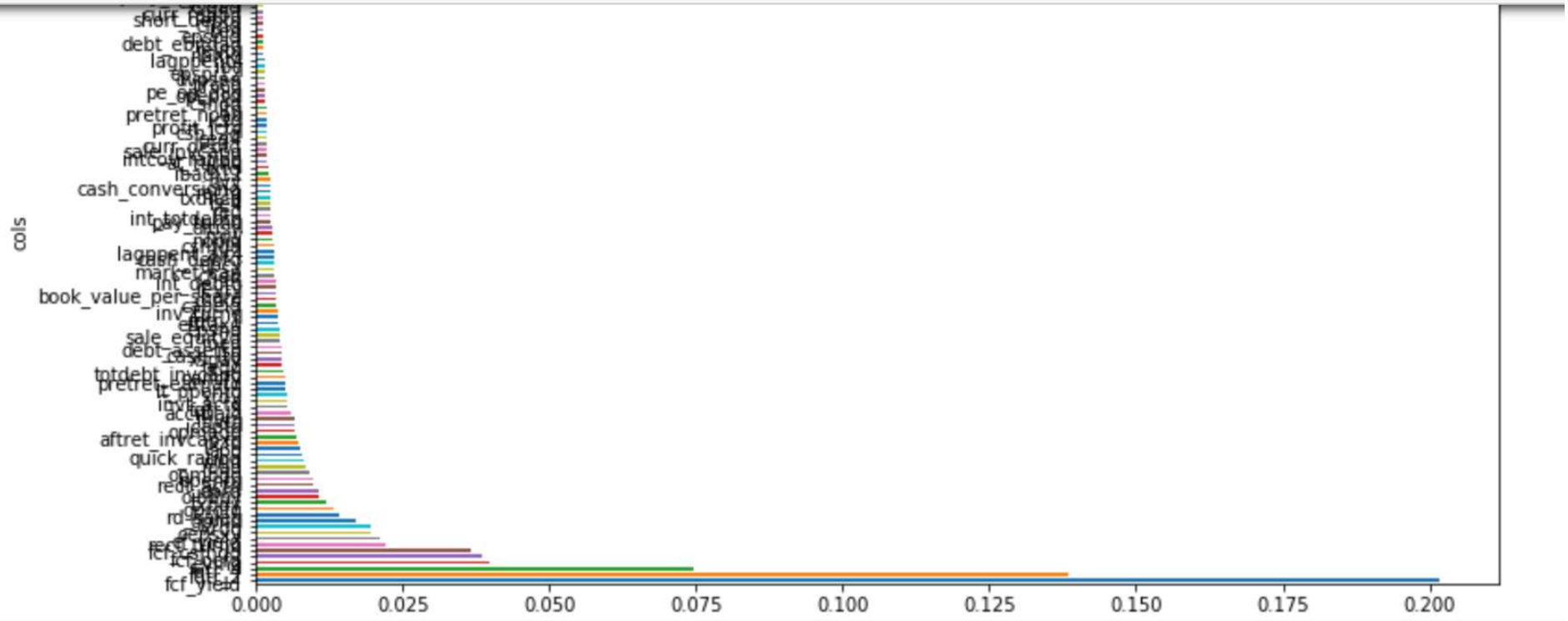
]:

	cols	feat_imp
156	fcf_yield	0.201645
180	fqtr_2	0.138606
182	fqtr_4	0.074448
95	evmq	0.039772
154	fcf_ocfq	0.038652
155	fcf_csfhdq	0.036534
171	rect_turnq	0.022033
58	txpq	0.000811
163	intcovq	0.000804
40	oiadpq	0.000763
14	dlttq	0.000754
66	chechy	0.000748
94	yearly_sales	0.000674
102	oancfy_q	0.000663
113	ptpmq	0.000630

118 rows × 2 columns

We are left with 118 features

```
plot_fi(features);
```



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

```
]: 118
```

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

```
bg_clf.fit(train_1,y_1)
bg_clf.score(train_1,y_1)
```

```
]: 0.5193882216034247
```

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

```
]: 0.47058823529411764
```

```
bg_clf.oob_score_
```

```
]: 0.5017663438759818
```

We are mostly interested in the P/L from the strategy

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

```
]: 48.185897999999895
```

We compute the profit_importance as before by taking each column, scrambling it and use the trained model to predict on the scrambled data and compute the P/L from this prediction

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

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

The pi of a feature indicates what the P/L would be if the feature is eliminated. Hence a low score means that the P/L would decrease a lot by eliminating the feature and so a lower pi score means higher importance

```
bg_clf.fit(train_1,y_1)
pi = baggingtree_profit_importance(bg_clf,valid,valid_stock_returns)
pi
```

]:

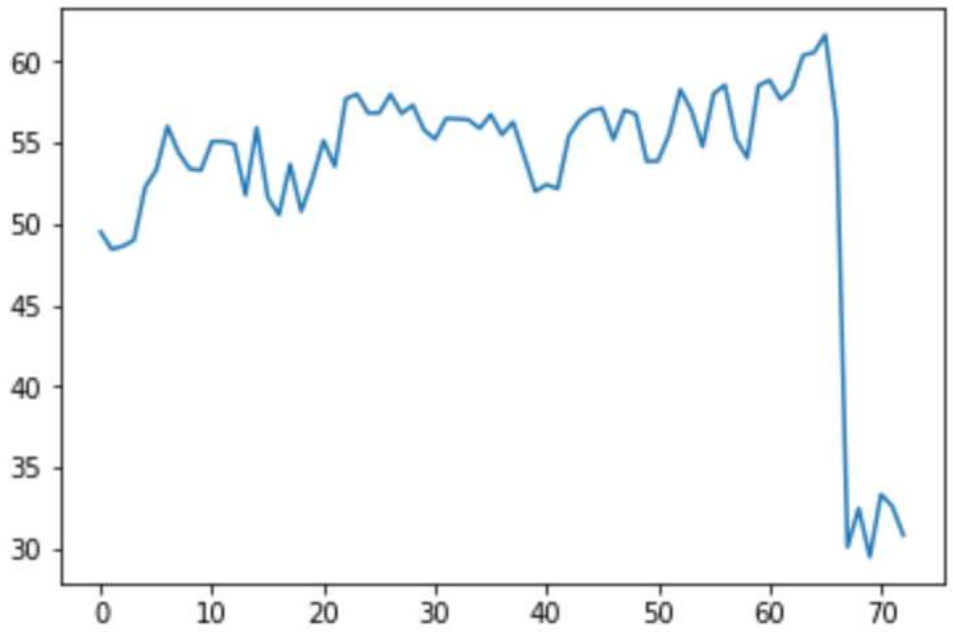
	cols	pi_imp
0	fcf_yield	33.780170
4	fcf_ocfq	41.115940
11	rd_saleq	42.486102
9	xrdq	43.202310
41	sale_equityq	44.319570
3	evmq	44.686826
10	gpmq	44.966260
1	fqtr_2	45.162428
47	cstkq	45.897548
30	invt_actq	45.959832
17	ppentq	46.190702
87	lagppent4	46.491808
105	at4	46.638764
39	debt_assetsq	46.685646
19	roaq	46.823166
74	ceq4	46.889956
96	xsgaq	46.911422
107	roceq	46.931298
33	pretret_earnatq	46.952168
111	intcovq	47.173192

Next we run the same loop as last time. Each time through the loop we drop the feature with the highest pi score (=least important). Retrain the model and drop all features with fi = 0.0 and compute the P/L. We store the profit and the feature set

```
profits = []
feat=[]
train = train_1.copy()
validation = valid.copy()
while len(train.columns)>2:
    bg_clf.fit(train,y_1)
    pi = baggingtree_profit_importance(bg_clf,validation,valid_stock_returns)

    col_to_drop = pi[pi['pi_imp'] == pi['pi_imp'].max()]['cols']
    train.drop(col_to_drop,axis=1,inplace=True)
    validation.drop(col_to_drop,axis=1,inplace=True)
    bg_clf.fit(train,y_1)
    fi = baggingtree_feat_importance(bg_clf,train)
    features = fi[(fi['feat_imp'] > 0.00)]
    train = train[features['cols'].values]
    validation = validation[features['cols'].values]
    bg_clf.fit(train,y_1)
    pred_valid = bg_clf.predict(validation)
    profits.append((pred_valid * valid_stock_returns).sum())
    feat.append(features['cols'].values)
```

After we have run through all the features we graph the profits for the various features

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



We can then find the maximal profit and the feature set that generates the maximal profit

```
n = np.argmax(profits)
max_profits = profits[n]
optim_feats = feat[n]
```

```
profits
```

```
]: [49.49441799999991,
    48.42615199999989,
    48.6093999999999,
    48.98855599999989,
    52.2383159999999,
    60.34153999999999,
    60.53448399999991,
    61.61372199999991,
    56.29200799999991,
    30.102918000000003,
    32.484728,
    29.502100000000006,
    33.34514800000001,
    32.62800999999999,
    30.841660000000054]
```

We print out the optimal feature set and retrain the model on these features

```
print(max_profits)
print(optim_feats)
```

```
61.61372199999991
['fcf_yield' 'fqtr_2' 'evmq' 'oibdpy' 'fcf_ocfq' 'oepsxy' 'gprofq' 'cfmq'
 'ppentq']
```

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


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

```
0.5143593366859233
0.4796015180265655
```

```
61.23308399999988
```

Next we train and evaluate the profit on the test set

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

```
]: -5.141207000000049
```

This does not look too promising. But let's run a back test from 2003 - 2018

This code is exactly the same as last week's

```
start_dates = [pd.to_datetime('2000-01-01') + pd.DateOffset(months = 3 * i) for i in range(63)]
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]
test_frames = [data.loc[d + pd.DateOffset(months=3):d+pd.DateOffset(months = 6)] 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]
```

```
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]
```

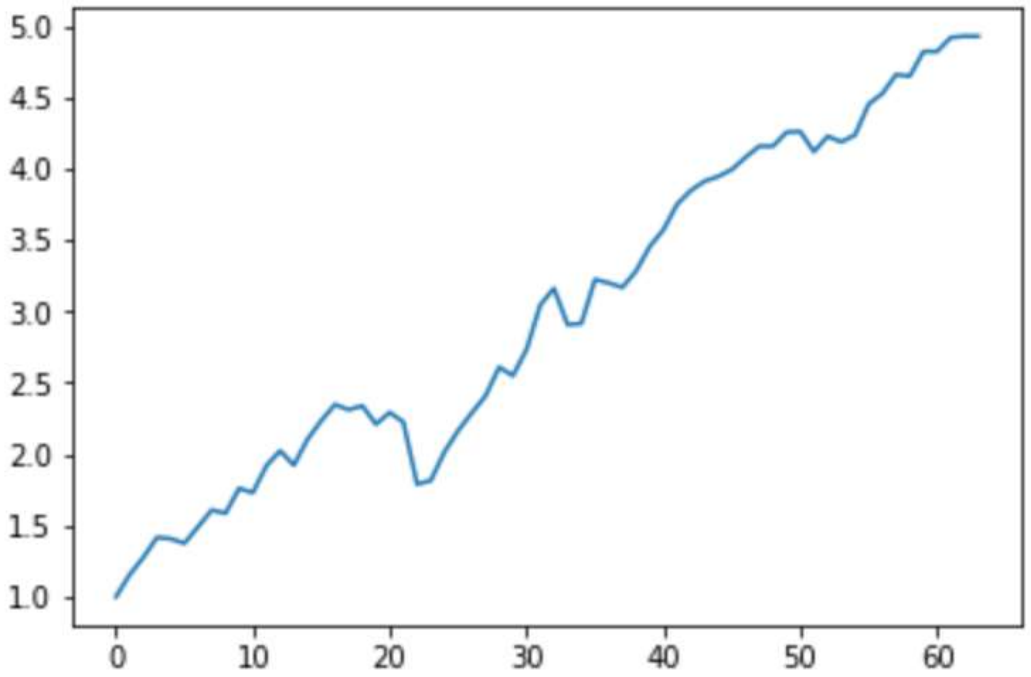
```
training_labels = [d['rel_performance'].values for d in training_frames]
```

We then cut down the feature set to the optimal features and compute the cumulative quarterly profits for following the strategy

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

for i in range(len(training_labels)):
        bg_clf.fit(opt_training_data[i],training_labels[i])
        pred_i = bg_clf.predict(opt_test_data[i])
        profit_i = (pred_i * test_frames[i]['next_period_return']).sum()
        P_L.append(profit_i)
        num_positions = len(pred_i)
        ret.append((1.0/num_positions) * profit_i)
        x.append(x[i] + (x[i]/num_positions) * profit_i)
```

The graph of the cumulative returns from 2003 - 2018

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



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

```
SPY = pd.read_pickle(r'C:\Users\niels\data\SPY_cum_ret.pkl')
SPY = SPY.loc['2003-01-01':]
SPY = SPY.resample('Q').ffill()
```

```
SPY
```

]:

date	spy_cum_ret
2003-03-31	-0.414239
2003-06-30	-0.265502
2003-09-30	-0.235779
2003-12-31	-0.122240
2004-03-31	-0.100766
2004-06-30	-0.082935
2004-09-30	-0.101756
2004-12-31	-0.014375

Starting at 2003-03-31 we get the cumulative returns

```
SPY['spy_cum_ret'] = (SPY['spy_cum_ret'] - SPY['spy_cum_ret'][0] + 1)
```

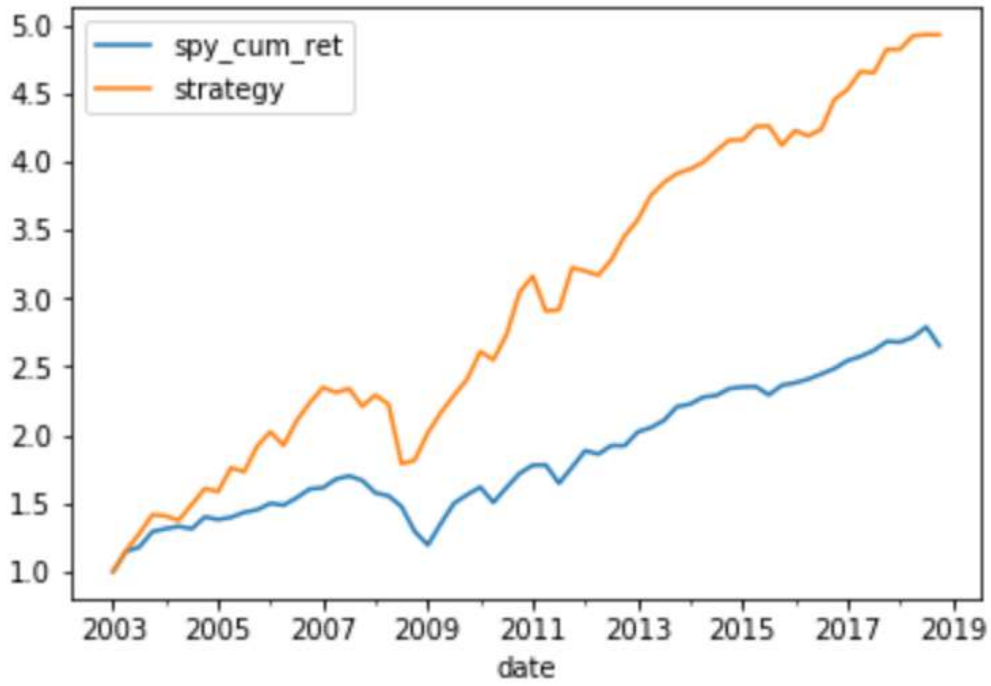
```
SPY['spy_cum_ret']
```

```
]: date
2003-03-31    1.000000
2003-06-30    1.148737
2003-09-30    1.178460
2003-12-31    1.291999
2004-03-31    1.313473
2004-06-30    1.331304
2004-09-30    1.312483
2004-12-31    1.399864
2005-03-31    1.380715
2005-06-30    1.396615
2005-09-30    1.433715
2005-12-31    1.452315
2006-03-31    1.499256
2006-06-30    1.485589
2006-09-30    1.539767
2006-12-31    1.604438
2016-12-31    2.485679
2017-03-31    2.543806
2017-06-30    2.574735
2017-09-30    2.618566
2017-12-31    2.684401
2018-03-31    2.679207
2018-06-30    2.716031
2018-09-30    2.790417
2018-12-31    2.652200
Freq: Q-DEC, Name: spy_cum_ret, Length: 64, dtype: float64
```

We add the cumulative profits from the strategy as a column in the SPY data frame and plot

```
SPY['strategy'] = x
```

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



The strategy outperforms the index by almost a factor of 2 (again this is without taking transaction costs into account).

```
x[-1]
```

```
4.928878587587322
```

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

```
2.652199999999999
```

Let's also compute the Sharpe Ratio = Average Return/STD(Returns)

```
strategy_mean_ret = (SPY['strategy'] - 1).diff().mean()
```

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

```
strategy_sr = strategy_mean_ret/strategy_std
strategy_sr
```

```
0.5088835757103606
```

We also compute the Sharpe Ratio of the index

```
(SPY['spy_cum_ret'] - 1).diff().mean()
```

```
0.026225396825396805
```

```
(SPY['spy_cum_ret'] - 1).diff().std()
```

```
0.06961855074029269
```

```
(SPY['spy_cum_ret'] - 1).diff().mean()/(SPY['spy_cum_ret'] - 1).diff().std()
```

```
0.37670127496949596
```