Python for Finance

Portfolio theory, Efficient frontier & PCA analysis

Andras Niedermayer

DAUPHINE UNIVERSITÉ PARIS

Outline

1. Portfolio optimization and the efficient frontier
2. Principal Component Analysis

Preliminaries

- Load daily closing prices on five stocks: Apple, Microsoft, Yahoo, Deutsche Bank, and Goldman Sachs.

```
data=DataFrame.from_csv('../L7_Data.csv')
```

- Compute the (log) returns from prices.

```
returns=np.log(data/data.shift(1))
```

- Get the number of assets as a variable.

```
no_assets=len(returns.columns.tolist())
```

Monte Carlo portfolios

Goal

Plot various portfolios of the five stocks in the *expected return – volatility* space. Let us simulate $N = 1000$ portfolios.

- Create (empty) vectors for returns and volatilities.

```
MC_returns=[]
MC_vols=[]
N=1000
```

- In a loop, generate portfolio weights and make sure they add up to 1 (one).

```
for p in range(N):
 weights=np.random.rand(no_assets)
 weights/= np.sum(weights)
```

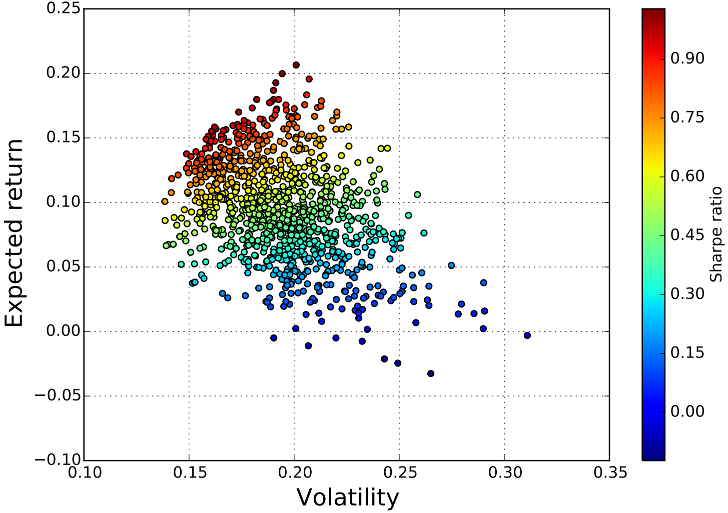
Monte Carlo portfolios (c'td)

Expected returns and volatility of portfolio i

$$\mathbb{E}R_i = w_{i,j}^T \times \mathbb{E}R_j$$
$$\sigma_i^2 = w_{i,j}^T \times \operatorname{cov}(R_j) \times w_{i,j}$$

- For each random weight vector we picked, we compute the expected return and volatility of the corresponding portfolio (annualised).

```
MC_returns.append
 (np.sum(returns.mean()*weights)*252)
MC_vols.append
 (np.sqrt(np.dot(weights.T,
  np.dot(returns.cov()*252, weights))))
```



Expected return
Volatility
Sharpe ratio
0.25
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
0.10
0.15
0.20
0.25
0.30
0.35
0.90
0.75
0.60
0.45
0.30
0.15
0.00

Portfolio summary measures

Objective

Build a function that, for a given vector of portfolio weights w, it returns the portfolio's expected return, volatility, and the Sharpe ratio (as a vector).

```
def portfolio(weights):
  weights=np.array(weights)
  P_ret=np.sum(returns.mean()*weights)*252
  P_vol=np.sqrt(np.dot(weights.T,
    np.dot(returns.cov()*252, weights)))
  return np.array([P_ret,P_vol, P_ret/P_vol])
```

Maximum Sharpe ratio portfolio

- Define a function for the *negative* Sharpe ratio (we want to maximise it).

```
def Sharpe(weights):
        return -portfolio(weights)[2]
```

- Set up the constraint that portfolio weights add up to one.

```
cons=({'type':'eq',
   'fun':lambda x: np.sum(x)-1})
```

- Set up boundaries for the portfolio weights (between 0 and 1).

```
bnds=tuple((0,1) for x in range(no_assets))
```

Maximum Sharpe ratio portfolio (c'td)

- Optimisation function.

```
import scipy.optimize as sco
opt_S=sco.minimize
  (Sharpe, no_assets*[1.0/no_assets],
   method='SLSQP', bounds=bnds,
   constraints=cons)
```

- Print portfolio weights.

```
print opt_S['x'].round(3)
```

- Maximum Sharpe ratio portfolio properties.

```
portfolio(opt_S['x'])
```

Application: Minimum variance portfolio

- Define a function for the portfolio variance.

```
def Variance(weights):
  return portfolio(weights)[1]**2
```

- Set up the constraint that portfolio weights add up to one.

```
cons=({'type':'eq',
    'fun':lambda x: np.sum(x)-1})
```

- Set up boundaries for the portfolio weights (between 0 and 1).

```
bnds=tuple((0,1) for x in range(no_assets))
```

Application: Minimum variance portfolio (c'td)

- Optimisation function.

```
import scipy.optimize as sco
opt_V=sco.minimize
  (Variance, no_assets*[1.0/no_assets],
  method='SLSQP', bounds=bnds,
  constraints=cons)
```

- Print portfolio weights.

```
print opt_V['x'].round(3)
```

- Maximum Sharpe ratio portfolio properties.

```
portfolio(opt_V['x'])
```

Plotting the efficient frontier

Problem

Minimise variance of a portfolio

$$\min_{w_i} {w_i}^T \Sigma w_i \tag{1}$$

subject to a target return:

$${w_i}^T R = R_{\text{target}} \tag{2}$$

- We solve this problem for many levels of the target return.
- Each time we do, we get another point on the efficient frontier.
- We need to specify the constraint in a loop, as the target return is always changing.

Algorithm

- Define a range for target returns.

```
TargetRet=np.linspace(0.0,0.25,50)
```

- Define an (empty) vector for the corresponding minimum volatilities.

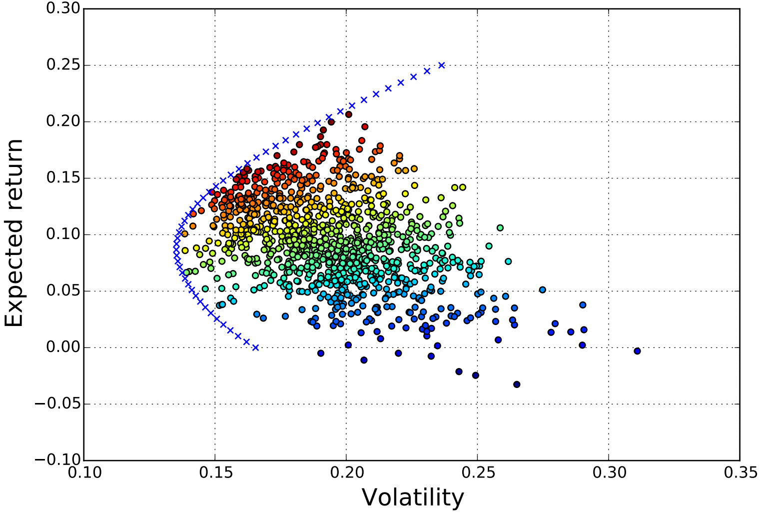
```
MinVols=[]
```

- In a loop of target returns, minimise variance under the constraint that the portfolio return equals the target return.

Algorithm (c'td)

```
for tret in TargetRet:
    cons=({'type':'eq',
    'fun':lambda x: portfolio(x)[0]-tret},
    {'type':'eq', 'fun':lambda x: np.sum(x)-1})
    res=sco.minimize(lambda x: portfolio(x)[1],
    no_assets*[1.0/no_assets], method='SLSQP',
    bounds=bnds, constraints=cons)

MinVols.append(res['fun'])
```



0.30
0.25
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
Expected return
0.10
0.15
0.20
0.25
0.30
0.35
Volatility

Plotting the Capital Market Line

1. Let the risk-free rate, $r_f = 1\%$.
2. The CML is a straight line starting from the risk-free rate and tangent to the upper part of the efficient frontier.
3. We do not have a closed form for the efficient frontier, so we need to use interpolation.

Efficient frontier (non-dominated)

The trick is to remember that the return vector is sorted in ascending order.
Portfolios with returns above the minimum variance portfolio are non-dominated!

```
# minimum variance portolio index
ind=np.argmin(MinVols)
evols=MinVols[ind:]
erets=TargetRet[ind:]
```

Main idea

1. Let $t(x)$ be the capital market line (linear).
2. Let $f(x)$ be the efficient frontier function.
3. Let x^* be the tangency portfolio.
4. At the tangency portfolio, the CML and efficient frontier have equal values and first derivatives.
5. This gives a three-equation system with three unknowns.

$$\begin{aligned} t(x^*) &= a + b \times x^* \text{ (Linearity of CML)} \\ t(x^*) &= f(x^*) \\ t'(x^*) &= f'(x^*) \end{aligned}$$

Numerical approximation of Efficient Frontier

We create numerical approximation via interpolation of

1. the efficient frontier
2. its first derivative

```
import scipy.interpolate as sci
tck=sci.splrep(evols,erets)
def f(x):
  return sci.splev(x,tck,der=0)
def df(x):
  return sci.splev(x,tck,der=1)
```

System of equations

1. We define a function to solve the system of equations.
2. The input is a vector containing
   - the risk-free rate (CML intercept)
   - the maximum Sharpe ratio (CML slope)
   - the tangency portfolio variance.
3. The output is the value of the three equations in the system. For the tangency portfolio, they should all be zero!

```
def equations(p, rf=0.01):
        eq1=rf-p[0]
        eq2=rf+p[1]*p[2]-f(p[2])
        eq3=p[1]-df(p[2])
        return eq1, eq2, eq3
```

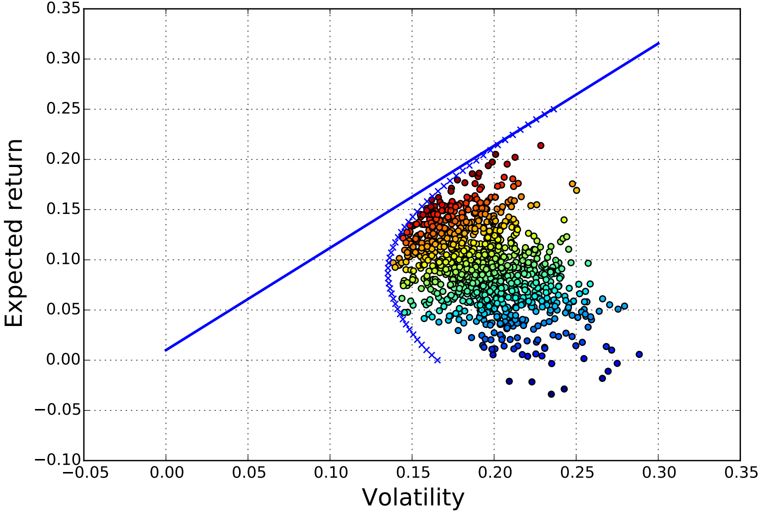
Solve for the CML

1. Find some plausible starting values for the solver (eye-balling, intuition...)
2. Use the `fsolve` function to find the CML.

```
opt=sco.fsolve(equations, [0.01, 0.5, 0.15])
```

We now have the linear CML function:

1. `opt[0]` is the risk-free rate.
2. `opt[1]` is the maximum Sharpe ratio.


Expected return
Volatility
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35

Outline

1. Portfolio optimization and the efficient frontier
2. Principal Component Analysis

Principal Components

1. Assume you have N (potentially) correlated time series, e.g., stock prices.
2. You want to identify K common factors that drive most of the common variation in the stock prices.
3. PCA allows you to find portfolios (linear combinations of stocks) that are orthogonal to each other (zero correlation).
4. The first principal component is the portfolio that accounts for most commonality.
5. The fraction of the variance a PC accounts for is proportional to its *eigenvalue* (λ).

Application

1. Let us load data on all CAC 40 stocks, including the index itself.

```
DataCAC=DataFrame.from_csv('/L7_DataCAC.csv')
```

2. We separate the index into a different DataFrame, and leave all stocks in DataCAC.

```
cac40=DataCAC["^FCHI"]
DataCAC.pop("^FCHI")
```

3. PCA is usually performed on normalised data. Let us define a normalisation function:

```
normalize = lambda x: (x-x.mean())/x.std()
```

Computing PC relative shares

1. The PCA algorithm is available in:

```
from sklearn.decomposition import KernelPCA
```

2. We fit principal components to normalised stock data:

```
pca=KernelPCA().fit(DataCAC.apply(normalize))
```

3. How many principal components do we find?

```
len(pca.lambdas_)
```

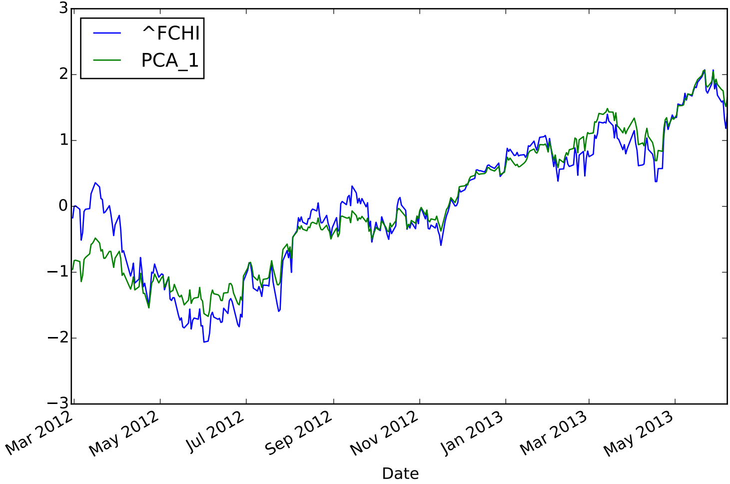
4. Let us compute the relative importance (in %) of the 10-largest eigenvalues.
   We need first to normalise the eigenvalues to add up to one.

```
fractions=lambda x: x/x.sum()
var_ratio=fractions(pca.lambdas_)[:10]
```

Constructing a PCA index

1. We try to replicate the movement in the index using principal components.
2. To do this, we need to transform the data into a single series, using the PCA weights.
3. First, let us just look at the first principal component.

```
pca=KernelPCA(n_components=1)
  .fit(DataCAC.apply(normalize))
cac40['PCA_1']=pca.transform(-DataCAC)
```



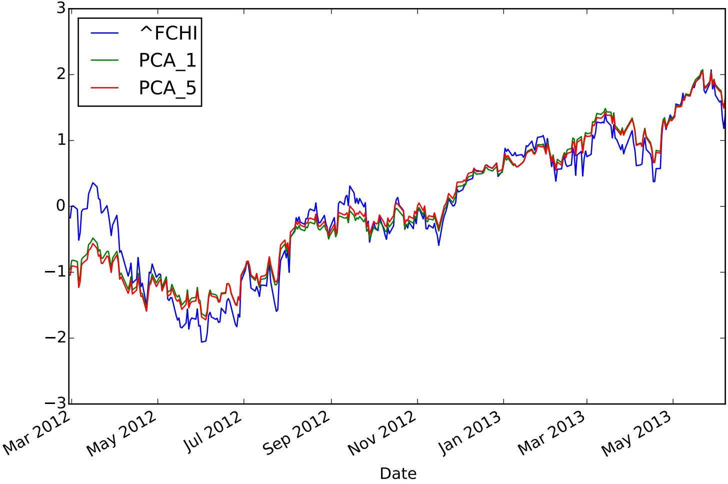
^FCHI
PCA_1
3
2
1
0
−1
−2
−3
Mar 2012
May 2012
Jul 2012
Sep 2012
Nov 2012
Jan 2013
Mar 2013
May 2013
Date

Constructing a PCA index: more than one PC

Application:

1. If we want to use more than one PC to replicate the index, we need to weight the principal components.
2. A natural weight is of course, the proportion of the variance they explain.

```
pca=KernelPCA(n_components=5)
 .fit(DataCAC.apply(normalize))
pca_components=pca.transform(-DataCAC)
weights=fractions(pca.lambdas_)
cac40['PCA_5']=np.dot(pca_components,weights)
```



3
2
1
0
−1
−2
−3
^FCHI
PCA_1
PCA_5
Mar 2012
May 2012
Jul 2012
Sep 2012
Nov 2012
Jan 2013
Mar 2013
May 2013
Date