OldModernPortfolioTheory

May 22, 2019

1 Old Modern portfolio theory

1.1 Efficient frontier, VaR, Expected Shortfall, Bootstrap, Monte-Carlo

In this tutorial, we're going to calculate the efficient frontier based on historical and forecasted data, and then generate some forward-looking returns.

As a starting point we'll use returns of 12 asset classes, namely developed markets bonds(FI.DEV), developed markets equities(EQ.DEV), emerging market bonds (FI.EM), corporate bonds(FI.CORP), emerging market equities(EQ.EM), high yield bonds(FI.HY), inflation-linked bonds(FI.IL), hedge funds(HF), real estate securities(RE.SEC), commodities(COMMOD), private equity(PRIV.EQ), bills(CASH).

[50]:
```
#loadind required libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import norm

import cvxopt as opt
from cvxopt import blas, solvers
```

[51]:
```
#some formatting
pd.options.display.float_format = '{:.3f}%'.format #this is to format pandas␣
↪dataframes nicely
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all" #this is just to show all␣
↪output for any cell, not the last operator output
solvers.options['show_progress'] = False # Turn off progress printing
```

[52]:
```
myPath = r'D:\Serega\Education\!Interviews\Portfolio\SAA_portfolio\Data_Source.
↪xlsx'
```

[53]:
```
returns = pd.read_excel(myPath, index_col=0)
```

[54]:
```
returns.head(2)
print('...')
returns.tail(2)
```

[54]:
```
            FI.DEV  EQ.DEV   FI.EM  FI.CORP  EQ.EM  FI.HY  FI.IL      HF  \
1998-02-28  0.007%  0.068% -0.001%   0.004% 0.104% 0.005% 0.013% -0.032%
```

```
1998-03-31 -0.009%  0.043%  0.009%  -0.005% 0.042% 0.010% 0.023% -0.029%

               RE.SEC  COMMOD  Private EQ   CASH
1998-02-28 -0.055% -0.053%      0.080% 0.004%
1998-03-31 -0.008%  0.010%      0.028% 0.005%


...
```

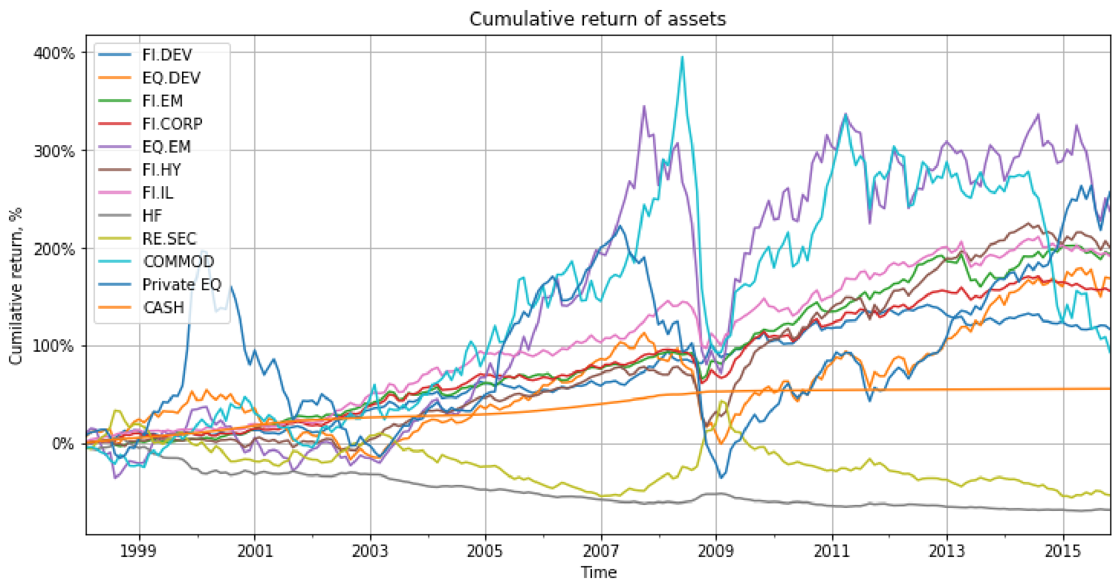
[54]:
```
               FI.DEV  EQ.DEV   FI.EM  FI.CORP   EQ.EM   FI.HY   FI.IL      HF  \
2015-10-31 -0.002%  0.080%  0.027%   0.006%  0.071%  0.030%  0.007% -0.017%
2015-11-30 -0.017% -0.004% -0.008%  -0.011% -0.039% -0.020% -0.012% -0.004%

               RE.SEC  COMMOD  Private EQ    CASH
2015-10-31 -0.062%  0.011%      0.071% -0.000%
2015-11-30 -0.020% -0.075%      0.048% -0.000%
```

As per the output above, in the input file we have monthly returns for a number of assets from February 1998 to November 2015. It is a good data range because it includes the dotcom crysis, the mortgage buble and consequent recoveries. You can choose your own time horizon. If you want do download other data from the internet there is a number of packages to do that. Just don't forget to convert price data to returns. Let's plot this returns to see relative performance of assets.

[55]:
```
cumulative_returns = returns + 1
for i in range(1,returns.shape[0]):
    cumulative_returns.iloc[i,:] = cumulative_returns.iloc[i,:
↪]*cumulative_returns.iloc[i-1,:]
cumulative_returns -= 1
```

[56]:
```
plt.figure();
cumulative_returns.plot(figsize=(12, 6));
plt.title('Cumulative return of assets');
plt.legend(loc='upper left');
plt.xlabel('Time');
plt.ylabel('Cumilative return, %');
plt.gca().set_yticklabels(['{:.0f}%'.format(x*100) for x in plt.gca().
↪get_yticks()]);
plt.grid(True);
```

```
<Figure size 432x288 with 0 Axes>
```



The worst performing classes are hedge funds and real estate securities. Maybe the indices chosen are not representative. However, since it is only an exercise, we'll leave averything as it is. Let's calculate parameters of these returns.

[57]:
```
#function for historical VaR and CVaR calculation
def __return_sorted_columns(df):
    sorted_df = pd.DataFrame(columns=df.columns)
    for col in df:
        sorted_df[col] = sorted(df[col])
    return sorted_df

def var_historical(rtns, confidence=.95):
    sorted_rtns = __return_sorted_columns(rtns)
    ind = int(np.floor(len(rtns)*(1-confidence))) #better to take lower value␣
 ↪to overestimate the risk than to underestimate it
    return sorted_rtns.iloc[ind-1]

def cvar_historical(rtns, confidence=.95):
    sorted_rtns = __return_sorted_columns(rtns)
    ind = int(np.floor(len(rtns)*(1-confidence))) #better to take lower value␣
 ↪to overestimate the risk than to underestimate it
    return np.mean(sorted_rtns[0:ind])

def var_analytical(rtns, confidence=.95):
    mu = rtns.mean() # in some cases mean return may assumed to be zero
    std = rtns.std()
    return mu - std*norm.ppf(confidence)

def cvar_analytical(rtns, confidence=.95):
```

```
    mu = rtns.mean() # in some cases mean return may assumed to be zero
    std = rtns.std()
    return mu - std*norm.pdf(norm.ppf(confidence))/(1-confidence)


def calculateparameters(rtns, confidence=.95):
    """This function returns Mean return, Standard deviation, Historical VaR,␣
 ↪Historical CVaR, Analytical VaR, Analytical CVaR
    Parameters
    ----------
    rtns (pandas dataframe): asset returns
    """
    mean_asset_rtn = rtns.mean()
    std_asset_rtn = rtns.std()
    VaR_hist = var_historical(rtns, confidence)
    CVaR_hist = cvar_historical(rtns, confidence)
    VaR_covar = var_analytical(rtns, confidence)
    CVaR_covar = cvar_analytical(rtns, confidence)
    params = pd.concat([mean_asset_rtn, std_asset_rtn,VaR_hist, CVaR_hist,␣
 ↪VaR_covar, CVaR_covar], axis=1)
    params = params.transpose()
    params.index = ['Mean return','Standard deviation', 'Historical VaR',␣
 ↪'Historical CVaR',
                    'Analytical VaR', 'Analytical CVaR']
    return params
```

[58]:
```
calculateparameters(returns)
```

[58]:

	FI.DEV	EQ.DEV	FI.EM	FI.CORP	EQ.EM	FI.HY	FI.IL
Mean return	0.004%	0.006%	0.005%	0.005%	0.008%	0.006%	0.005%
Standard deviation	0.018%	0.046%	0.019%	0.018%	0.070%	0.029%	0.022%
Historical VaR	-0.030%	-0.085%	-0.028%	-0.025%	-0.105%	-0.040%	-0.034%
Historical CVaR	-0.036%	-0.109%	-0.047%	-0.040%	-0.163%	-0.072%	-0.050%
Analytical VaR	-0.026%	-0.070%	-0.026%	-0.025%	-0.107%	-0.042%	-0.030%
Analytical CVaR	-0.034%	-0.089%	-0.034%	-0.033%	-0.136%	-0.054%	-0.039%

	HF	RE.SEC	COMMOD	Private EQ	CASH
Mean return	-0.005%	-0.002%	0.005%	0.009%	0.002%
Standard deviation	0.020%	0.051%	0.067%	0.074%	0.002%
Historical VaR	-0.035%	-0.070%	-0.112%	-0.113%	0.000%
Historical CVaR	-0.045%	-0.090%	-0.145%	-0.168%	-0.000%
Analytical VaR	-0.038%	-0.087%	-0.105%	-0.113%	-0.001%
Analytical CVaR	-0.047%	-0.108%	-0.133%	-0.144%	-0.002%

As we can see, historical VaR slightly overestimates the risk. It happens because we round the index of the historical return correspondent to the chosen confidence level.

We can generate expected returns using bootstrap or covariance based Monte-Carlo.

[85]:

```python
def montecarlo(rtns, num_simulations = 10000, seed=1):
    '''Covariance based Monte-Carlo, returns are assumed to be normally␣
 ↪distributed
    '''
    n_assets = rtns.shape[1]
    mean_asset_rtn = rtns.mean()
    std_asset_rtn = rtns.std()
    cormat = rtns.corr()
    np.random.seed(seed)
    rand_rtns = (np.random.normal(size=num_simulations*n_assets)).
 ↪reshape(num_simulations,n_assets)
    cholesky_decomposition = (np.linalg.cholesky(cormat)).transpose()
    zscore = np.dot(rand_rtns, cholesky_decomposition)

    rtns_simulations = pd.DataFrame(columns=rtns.columns)
    #haven't found an elegant way to do this. Ended up with a loop. There␣
 ↪should be some convenient function in numpy or pandas...
    for i in range(zscore.shape[0]):
        rtns_simulations.loc[i] = mean_asset_rtn + np.multiply(zscore[i,:
 ↪],std_asset_rtn)
    return rtns_simulations

def bootstrap(rtns, num_simulations = 10000, chunksize = 3, seed=1):
    '''Takes historical data to generate returns
    '''
    n_returns = rtns.shape[0]
    if (chunksize<1):
        chunksize = 1
        print('Chunksize cannot be negative. chunksize is assumed to be 1')

    returns_local = rtns.append(rtns.iloc[0:(chunksize-1),:]) #this is to be␣
 ↪able to take pieces from the end of the series
    chunks = num_simulations//chunksize
    rtns_simulations = pd.DataFrame(columns=rtns.columns)
    np.random.seed(seed)
    for idx in np.random.choice(n_returns, size=chunks, replace=True):
        rtns_simulations = rtns_simulations.append(returns_local.iloc[idx:
 ↪(idx+chunksize),:])

    #adding variables which are lower than
    fraction_period = num_simulations%chunksize
    if fraction_period:
        idx = np.random.randint(n_returns)
        rtns_simulations = rtns_simulations.append(returns_local.iloc[idx:
 ↪(idx+fraction_period),:])
```

```
    return rtns_simulations
```

[88]:
```
bootstrap_returns = bootstrap(returns)
montecarlo_returns = montecarlo(returns)
```

Parameters of returns generated with bootstrap:

[89]:
```
calculateparameters(bootstrap_returns)
```

[89]:

	FI.DEV	EQ.DEV	FI.EM	FI.CORP	EQ.EM	FI.HY	FI.IL	\
Mean return	0.004%	0.006%	0.005%	0.005%	0.008%	0.005%	0.005%	
Standard deviation	0.018%	0.046%	0.019%	0.018%	0.069%	0.029%	0.022%	
Historical VaR	-0.028%	-0.084%	-0.028%	-0.025%	-0.100%	-0.039%	-0.030%	
Historical CVaR	-0.035%	-0.107%	-0.045%	-0.039%	-0.155%	-0.071%	-0.049%	
Analytical VaR	-0.026%	-0.069%	-0.026%	-0.025%	-0.105%	-0.043%	-0.030%	
Analytical CVaR	-0.034%	-0.088%	-0.034%	-0.033%	-0.134%	-0.055%	-0.039%	

	HF	RE.SEC	COMMOD	Private EQ	CASH
Mean return	-0.005%	-0.003%	0.005%	0.008%	0.002%
Standard deviation	0.020%	0.051%	0.067%	0.074%	0.002%
Historical VaR	-0.035%	-0.068%	-0.107%	-0.111%	0.000%
Historical CVaR	-0.045%	-0.086%	-0.144%	-0.168%	-0.000%
Analytical VaR	-0.038%	-0.086%	-0.105%	-0.113%	-0.001%
Analytical CVaR	-0.047%	-0.108%	-0.133%	-0.144%	-0.002%

Parameters of returns generated with Monte-Carlo:

[90]:
```
calculateparameters(montecarlo_returns)
```

[90]:

	FI.DEV	EQ.DEV	FI.EM	FI.CORP	EQ.EM	FI.HY	FI.IL	\
Mean return	0.004%	0.005%	0.005%	0.005%	0.008%	0.006%	0.005%	
Standard deviation	0.018%	0.046%	0.019%	0.018%	0.069%	0.029%	0.022%	
Historical VaR	-0.026%	-0.069%	-0.026%	-0.025%	-0.107%	-0.042%	-0.030%	
Historical CVaR	-0.034%	-0.088%	-0.034%	-0.032%	-0.136%	-0.054%	-0.039%	
Analytical VaR	-0.026%	-0.070%	-0.026%	-0.025%	-0.106%	-0.042%	-0.030%	
Analytical CVaR	-0.034%	-0.089%	-0.034%	-0.033%	-0.135%	-0.054%	-0.039%	

	HF	RE.SEC	COMMOD	Private EQ	CASH
Mean return	-0.005%	-0.002%	0.006%	0.008%	0.002%
Standard deviation	0.020%	0.051%	0.067%	0.074%	0.002%
Historical VaR	-0.038%	-0.086%	-0.105%	-0.113%	-0.001%
Historical CVaR	-0.046%	-0.107%	-0.131%	-0.144%	-0.002%
Analytical VaR	-0.038%	-0.086%	-0.104%	-0.114%	-0.001%
Analytical CVaR	-0.047%	-0.108%	-0.132%	-0.145%	-0.002%

As we can see, generated returns have almost the same parameters as our initial sample, which confirms that the generation functions work correctly.

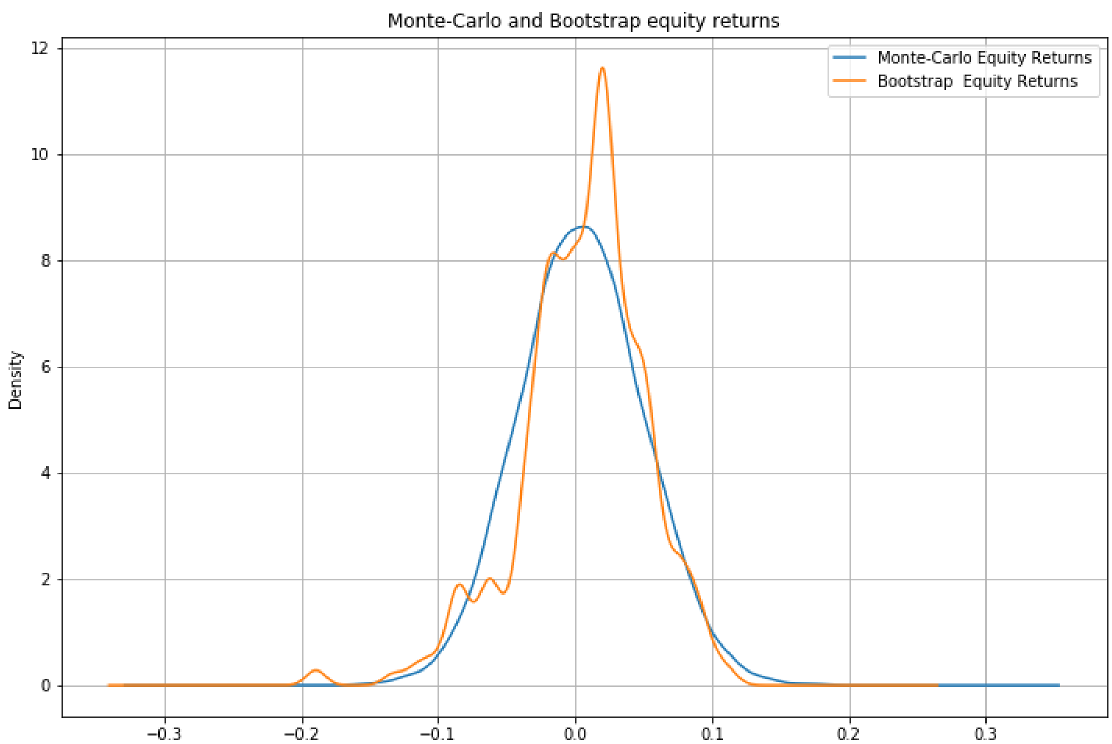
Let's visualize the result, people love it. We can make a density plot for returns of equities.

[108]:
```
equity_returns_for_plotting = pd.concat([montecarlo_returns['EQ.DEV'],
 ↪bootstrap_returns['EQ.DEV']],axis=1)
```

```
equity_returns_for_plotting.columns=['Monte-Carlo Equity Returns','Bootstrap ␣
 ↪Equity Returns']
fig, ax = plt.subplots(figsize=(12, 8))
equity_returns_for_plotting.plot.kde(ax=ax, legend=True, title='Monte-Carlo and␣
 ↪Bootstrap equity returns')
#equity_returns_for_plotting.plot.hist(density=True, ax=ax)
ax.grid(axis='x')
ax.grid(axis='y')
#ax.set_facecolor('#d8dcd6')
```

[108]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1e4d1d0dd68>
```



The return generated by Monte-Carlo is more smooth. Seems like bootstrap returns have fatter tails.

Let's annualize our monthly returns. We'll proceed with bootstrap returns.

[109]:

```
def returns_period_upscale(rtns, periodicity = 12, annualize_last = True):
    new_returns = pd.DataFrame(columns = rtns.columns)
    rtns += 1
    n_steps = rtns.shape[0]//periodicity

    for i in range(n_steps):
        new_returns.loc[i] = np.prod(rtns.iloc[(i*periodicity):
 ↪((i+1)*periodicity)],axis=0)
```

```
    fraction_period = rtns.shape[0]%periodicity
    if fraction_period:
        new_returns.loc[n_steps] = np.prod(rtns.iloc[(n_steps*periodicity):
↪],axis=0)
        if annualize_last: new_returns.loc[n_steps] = np.power(new_returns.
↪loc[n_steps],periodicity/fraction_period)

    rtns -= 1 #python passes this dataframe by reference, and we don't want the␣
↪internal function to make changes. I should've copied this Dataframe at the␣
↪beginning of the function, and work with the copy. But I don't want to)

    return new_returns-1
```

[111]:
```
annual_returns = returns_period_upscale(bootstrap_returns)
```

[112]:
```
covmat, corrmat = [returns.cov(), returns.corr()]
corrmat.style.background_gradient(cmap='coolwarm').set_precision(2)
```

[112]: <pandas.io.formats.style.Styler at 0x1e4d2080978>

image.png

[ ]:
```

```

Below you can see an average return(arithmetic), standard deviation by asset class and correlation and covariation matrix. Geometric returns can be used instead, but the difference is small anyway.

[ ]:
```
mean_asset_rtn, std_asset_rtn = [annual_returns.mean(), annual_returns.std()]
#printing parameters
params = pd.DataFrame(columns=mean_asset_rtn.index, index =␣
↪['Mean_return','Standard_deviation'])
for key, rtn, stdev in zip(mean_asset_rtn.index, mean_asset_rtn, std_asset_rtn):
    params[key] = [f'{rtn*100:.02f}%', f'{stdev*100:.02f}%']
params
```

It seems like equities are doing better than bonds, however equities are more volitile. Makes sense. Let's take a look at correlation matrix

[ ]:
```
covmat, corrmat = [annual_returns.cov(), annual_returns.corr()]
corrmat.style.background_gradient(cmap='coolwarm').set_precision(2)
```

Intuitively, high correlation between assets - a bad thing, low correlation - a good thing.

[116]:
```
def optimal_portfolio(rtns):
    returns = rtns.transpose()
    n = len(returns)
    returns = np.asmatrix(returns.transpose())

    N = 100
    mus = [10**(5.0 * t/N - 1.0) for t in range(N)]
```

```
    # Convert to cvxopt matrices
    S = opt.matrix(np.cov(returns))
    pbar = opt.matrix(np.mean(returns, axis=1))

    # Create constraint matrices
    G = -opt.matrix(np.eye(n))   # negative n x n identity matrix
    h = opt.matrix(0.0, (n ,1))
    A = opt.matrix(1.0, (1, n))
    b = opt.matrix(1.0)

    # Calculate efficient frontier weights using quadratic programming
    portfolios = [solvers.qp(mu*S, -pbar, G, h, A, b)['x']
                  for mu in mus]
    ## CALCULATE RISKS AND RETURNS FOR FRONTIER
    returns = [blas.dot(pbar, x) for x in portfolios]
    risks = [np.sqrt(blas.dot(x, S*x)) for x in portfolios]
    ## CALCULATE THE 2ND DEGREE POLYNOMIAL OF THE FRONTIER CURVE
    m1 = np.polyfit(returns, risks, 2)
    x1 = np.sqrt(m1[2] / m1[0])
    # CALCULATE THE OPTIMAL PORTFOLIO
    wt = solvers.qp(opt.matrix(x1 * S), -pbar, G, h, A, b)['x']
    return np.asarray(wt), returns, risks
```

[117]:
```
optimal_portfolio(annual_returns)
```

```
      ␣
↪---------------------------------------------------------------------------

      TypeError                                 Traceback (most recent call␣
↪last)

      <ipython-input-117-69aeff2bab9e> in <module>
  ----> 1 optimal_portfolio(annual_returns)


      <ipython-input-116-d9676b7184bc> in optimal_portfolio(rtns)
       19     # Calculate efficient frontier weights using quadratic␣
↪programming
       20     portfolios = [solvers.qp(mu*S, -pbar, G, h, A, b)['x']
  ---> 21                   for mu in mus]
       22     ## CALCULATE RISKS AND RETURNS FOR FRONTIER
       23     returns = [blas.dot(pbar, x) for x in portfolios]


      <ipython-input-116-d9676b7184bc> in <listcomp>(.0)
```

```
        19     # Calculate efficient frontier weights using quadratic␣
↪programming
        20     portfolios = [solvers.qp(mu*S, -pbar, G, h, A, b)['x']
   ---> 21                   for mu in mus]
        22     ## CALCULATE RISKS AND RETURNS FOR FRONTIER
        23     returns = [blas.dot(pbar, x) for x in portfolios]


      C:\ProgramData\Anaconda3\lib\site-packages\cvxopt\coneprog.py in qp(P,␣
↪q, G, h, A, b, solver, kktsolver, initvals, **kwargs)
      4485                 'residual as dual infeasibility certificate': dinfres}
      4486
   -> 4487     return coneqp(P, q, G, h, None, A,  b, initvals, kktsolver =␣
↪kktsolver, options = options)


      C:\ProgramData\Anaconda3\lib\site-packages\cvxopt\coneprog.py in␣
↪coneqp(P, q, G, h, dims, A, b, initvals, kktsolver, xnewcopy, xdot, xaxpy,␣
↪xscal, ynewcopy, ydot, yaxpy, yscal, **kwargs)
      1893         if G.typecode != 'd' or G.size != (cdim, q.size[0]):
      1894             raise TypeError("'G' must be a 'd' matrix of size (%d,␣
↪%d)"\
   -> 1895                 %(cdim, q.size[0]))
      1896         def fG(x, y, trans = 'N', alpha = 1.0, beta = 0.0):
      1897             misc.sgemv(G, x, y, dims, trans = trans, alpha = alpha,


      TypeError: 'G' must be a 'd' matrix of size (12, 834)
```

[ ]: