In [56]:
```
import datetime
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="darkgrid")
from datetime import datetime
import cvxpy as cp
import scipy.optimize as sco
import numpy as np
from arch import arch_model
import empyrical as ep
from sklearn.covariance import LedoitWolf, MinCovDet
%matplotlib inline
```

# Data Preparation

In [57]:
```
# set display options
pd.options.display.max_rows, pd.options.display.max_columns = 10, 25
data = pd.read_csv('midterm_daily_adjusted_price.csv')
data['Date'] = pd.to_datetime(data.Date)
data.dropna(inplace = True)
data = data.set_index('Date')
data
```

Out[57]:

| Date | AEZS | A | APD | AA | CF |
|---|---|---|---|---|---|
| 2012-12-31 | 238.0 | 27.141947 | 63.885410 | 19.910763 | 32.126221 |
| 2013-01-02 | 253.0 | 27.765131 | 65.284454 | 20.621868 | 32.771385 |
| 2013-01-03 | 254.0 | 27.864580 | 65.056343 | 20.805374 | 32.616428 |
| 2013-01-04 | 257.0 | 28.414846 | 65.930756 | 21.241209 | 33.347000 |
| 2013-01-07 | 259.0 | 28.209322 | 65.869919 | 20.874187 | 33.222076 |
| ... | ... | ... | ... | ... | ... |
| 2016-12-13 | 3.8 | 44.674984 | 134.638290 | 30.240000 | 24.779669 |
| 2016-12-14 | 3.8 | 44.434235 | 133.789566 | 30.450001 | 24.518003 |
| 2016-12-15 | 3.8 | 44.838707 | 134.611191 | 29.790001 | 24.988997 |
| 2016-12-16 | 3.7 | 44.212730 | 133.112396 | 29.400000 | 25.712940 |
| 2016-12-19 | 3.8 | 43.750469 | 133.374252 | 29.290001 | 24.674997 |

1001 rows × 5 columns

In [58]:
```
data_return_log = np.log(data).diff()
data_return_log.dropna(inplace=True)
data_return_log
```

Out[58]:

| Date | AEZS | A | APD | AA | CF |
|---|---|---|---|---|---|
| **2013-01-02** | 0.061119 | 0.022701 | 0.021663 | 0.035092 | 0.019883 |
| **2013-01-03** | 0.003945 | 0.003575 | -0.003500 | 0.008859 | -0.004740 |
| **2013-01-04** | 0.011742 | 0.019555 | 0.013351 | 0.020732 | 0.022152 |
| **2013-01-07** | 0.007752 | -0.007259 | -0.000923 | -0.017430 | -0.003753 |
| **2013-01-08** | -0.015565 | -0.008023 | 0.001845 | 0.000000 | -0.014769 |
| **...** | ... | ... | ... | ... | ... |
| **2016-12-13** | -0.013072 | 0.005403 | -0.003481 | -0.032534 | -0.020554 |
| **2016-12-14** | 0.000000 | -0.005403 | -0.006324 | 0.006920 | -0.010616 |
| **2016-12-15** | 0.000000 | 0.009062 | 0.006122 | -0.021913 | 0.019028 |
| **2016-12-16** | -0.026668 | -0.014059 | -0.011197 | -0.013178 | 0.028559 |
| **2016-12-19** | 0.026668 | -0.010510 | 0.001965 | -0.003748 | -0.041204 |

1000 rows × 5 columns

In [59]:

```
data_return_lin = data.diff()/data.shift(periods=1)
data_return_lin.dropna(inplace=True)
data_return_lin
```

Out[59]:

| Date | AEZS | A | APD | AA | CF |
|---|---|---|---|---|---|
| **2013-01-02** | 0.063025 | 0.022960 | 0.021899 | 0.035715 | 0.020082 |
| **2013-01-03** | 0.003953 | 0.003582 | -0.003494 | 0.008899 | -0.004728 |
| **2013-01-04** | 0.011811 | 0.019748 | 0.013441 | 0.020948 | 0.022399 |
| **2013-01-07** | 0.007782 | -0.007233 | -0.000923 | -0.017279 | -0.003746 |
| **2013-01-08** | -0.015444 | -0.007991 | 0.001847 | 0.000000 | -0.014660 |
| **...** | ... | ... | ... | ... | ... |
| **2016-12-13** | -0.012987 | 0.005418 | -0.003475 | -0.032010 | -0.020345 |
| **2016-12-14** | 0.000000 | -0.005389 | -0.006304 | 0.006944 | -0.010560 |
| **2016-12-15** | 0.000000 | 0.009103 | 0.006141 | -0.021675 | 0.019210 |
| **2016-12-16** | -0.026316 | -0.013961 | -0.011134 | -0.013092 | 0.028970 |
| **2016-12-19** | 0.027027 | -0.010455 | 0.001967 | -0.003741 | -0.040367 |

1000 rows × 5 columns

In [60]:

```
plt.figure(figsize=(20,10))
plt.plot((data/data.loc["2012-12-31"])['AEZS'], label='AEZS')
```

```
plt.plot((data/data.loc["2012-12-31"])['A'], label='A')
plt.plot((data/data.loc["2012-12-31"])['APD'], label='APD')
plt.plot((data/data.loc["2012-12-31"])['AA'], label='AA')
plt.plot((data/data.loc["2012-12-31"])['CF'], label='CF')
plt.title('Normalized Stock Prices')
plt.legend(loc='upper left', fontsize=12)
plt.xlabel('Date')
plt.ylabel('Normalized Price')
```

Out[60]: Text(0, 0.5, 'Normalized Price')


# Data modelling function

In [61]:
```
def mu(data):
    return data.mean().values
```

In [62]:
```
def Sigma(data):
    return data.cov().values
```

In [63]:
```
def Shrinkage_Sigma(data):
    shrinkage_sigma = LedoitWolf().fit(data)
    return shrinkage_sigma.covariance_
```

In [64]:
```
def Robust_Sigma(data):
    robust_sigma = MinCovDet().fit(data)
    return robust_sigma.covariance_
```

In [65]:
```
def get_mu_and_Sigma_log2lin(mu, Sigma):
    N = np.shape(Sigma)[0]
    mu_ = np.exp(mu + 0.5*np.diag(Sigma)) - 1
    Sigma_ = np.empty((N, N))
    for ii in range(N):
        for jj in range(N):
```

```
                Sigma_[ii, jj] = np.exp(mu[ii] + mu[jj] + 0.5*(Sigma[ii, j

    return mu, Sigma_
```

# Define function for risk-based portfolio

In [66]:
```
# create function for EWP
def EWP(Sigma):
    N = Sigma.shape[0]
    w_EWP = np.ones(N) / N
    return w_EWP
```

In [67]:
```
# create function for GMVP
def GMVP(Sigma):
    w = cp.Variable(len(Sigma))
    variance = cp.quad_form(w, Sigma)
    objective = cp.Minimize(variance)
    constraint_1 = w >= 0
    constraint_2 = cp.sum(w) == 1
    problem = cp.Problem(objective, constraints = [constraint_1, const

    problem.solve()
    return w.value
```

In [68]:
```
from scipy.optimize import minimize, LinearConstraint

def MDP(Sigma):
    N = Sigma.shape[0]
    Diag_sigma = np.sqrt(np.diag(Sigma))
    # define the nonconvex objective function
    def fn_DR(w, Diag_sigma, Sigma):
        x = w.reshape((-1, 1))
        return ((w @ Diag_sigma)/np.sqrt(x.T @ Sigma @ x))[0]

    # initial point
    w0 = np.ones(N)/N

    # define lower bounds for inequality constraints
    ub = np.inf * np.ones(N)
    lb = np.zeros(N)
    A = np.eye(N)
    ineq_lc = LinearConstraint(A=A, lb=lb, ub=ub)

    # define equality constraints
    eq_lc = LinearConstraint(A=np.ones(N), lb=1, ub=1)

    # call solver
    res = minimize(lambda x, *args: -fn_DR(x, *args), w0, args=(Diag_s
    w_MDP = res.x
    return w_MDP
```

In [69]:
```
def MDCP(Sigma):
    C = np.diag(1/np.sqrt(np.diag(Sigma))) @ Sigma @ np.diag(1/np.sqrt
```

```
    return GMVP(Sigma=C)
```

In [70]:

```
# create function for IVP
def IVP(Sigma):
    sigma2 = np.diag(Sigma)
    w_IVP = 1/np.sqrt(sigma2)
    w_IVP = w_IVP/np.sum(w_IVP)
    return w_IVP
```

In [71]:

```
# create function for RPP
def RPP(Sigma,b):
    N = Sigma.shape[0]
    x0 = np.ones(N)/N

    # function definition
    def fn_convex(x, Sigma, b):
        return(0.5 * x.T @ Sigma @ x - np.sum(b * np.log(x)))

    # optimize with general-purpose solver
    result = sco.minimize(fn_convex, x0=x0, args=(Sigma, b), method='B
    x_convex = result.x
    # normalize each column
    w_RPP_convex = x_convex/np.sum(x_convex)
    return w_RPP_convex
```

In [206…

```
slices = np.arange(0,1001,100)

w_matrix_EWP = np.zeros(shape=(10,5))
w_matrix_GMVP = np.zeros(shape=(10,5))
w_matrix_MDP = np.zeros(shape=(10,5))
w_matrix_MDCP = np.zeros(shape=(10,5))
w_matrix_IVP = np.zeros(shape=(10,5))
w_matrix_RPP = np.zeros(shape=(10,5))

Sigma_lin_matrix = np.zeros(shape=(10,5,5))

# 1st date_slice
w_matrix_EWP[:] = np.ones(5)/5
w_matrix_GMVP[0] = np.ones(5)/5
w_matrix_MDP[0] = np.ones(5)/5
w_matrix_MDCP[0] = np.ones(5)/5
w_matrix_IVP[0] = np.ones(5)/5
w_matrix_RPP[0] = np.ones(5)/5


# 2nd - 10st date_slices
for i in range(1,10):


    mu_log = mu(data_return_log[slices[i-1]:slices[i]])
    Sigma_log = Sigma(data_return_log[slices[i-1]:slices[i]])

    mu_lin, Sigma_lin = get_mu_and_Sigma_log2lin(mu_log, Sigma_log)

    # this function can now be used as
```

```
    w_EWP = EWP(Sigma_lin)
    w_matrix_EWP[i] = w_EWP
    w_matrix_EWP

    w_GMVP = GMVP(Sigma_lin)
    w_matrix_GMVP[i] = w_GMVP
    w_matrix_GMVP

    w_MDP = MDP(Sigma_lin)
    w_matrix_MDP[i] = w_MDP
    w_matrix_MDP

    w_MDCP = MDCP(Sigma_lin)
    w_matrix_MDCP[i] = w_MDCP
    w_matrix_MDCP

    w_IVP = IVP(Sigma_lin)
    w_matrix_IVP[i] = w_IVP
    w_matrix_IVP

    w_RPP = RPP(Sigma_lin, [1/5 for i in range(5)])
    w_matrix_RPP[i] = w_RPP
    w_matrix_RPP


for i in range(0,10):
    mu_log = mu(data_return_log[slices[i]:slices[i+1]])
    Sigma_log = Sigma(data_return_log[slices[i]:slices[i+1]])

    mu_lin, Sigma_lin = get_mu_and_Sigma_log2lin(mu_log, Sigma_log)
    Sigma_lin_matrix[i] = Sigma_lin
```

In [207…

```
EWP_wide = pd.DataFrame(w_matrix_EWP)
GMVP_wide = pd.DataFrame(w_matrix_GMVP)
MDP_wide = pd.DataFrame(w_matrix_MDP)
MDCP_wide = pd.DataFrame(w_matrix_MDCP)
IVP_wide = pd.DataFrame(w_matrix_IVP)
RPP_wide = pd.DataFrame(w_matrix_RPP)

EWP_wide.columns =['AEZS', 'A', 'APD', 'AA', 'CF']
GMVP_wide.columns =['AEZS', 'A', 'APD', 'AA', 'CF']
MDP_wide.columns =['AEZS', 'A', 'APD', 'AA', 'CF']
MDCP_wide.columns =['AEZS', 'A', 'APD', 'AA', 'CF']
IVP_wide.columns =['AEZS', 'A', 'APD', 'AA', 'CF']
RPP_wide.columns =['AEZS', 'A', 'APD', 'AA', 'CF']

EWP_wide.index = np.arange(1,11)
GMVP_wide.index = np.arange(1,11)
MDP_wide.index = np.arange(1,11)
MDCP_wide.index = np.arange(1,11)
IVP_wide.index = np.arange(1,11)
RPP_wide.index = np.arange(1,11)

EWP_long = EWP_wide.unstack().reset_index()
GMVP_long = GMVP_wide.unstack().reset_index()
MDP_long = MDP_wide.unstack().reset_index()
MDCP_long = MDCP_wide.unstack().reset_index()
```

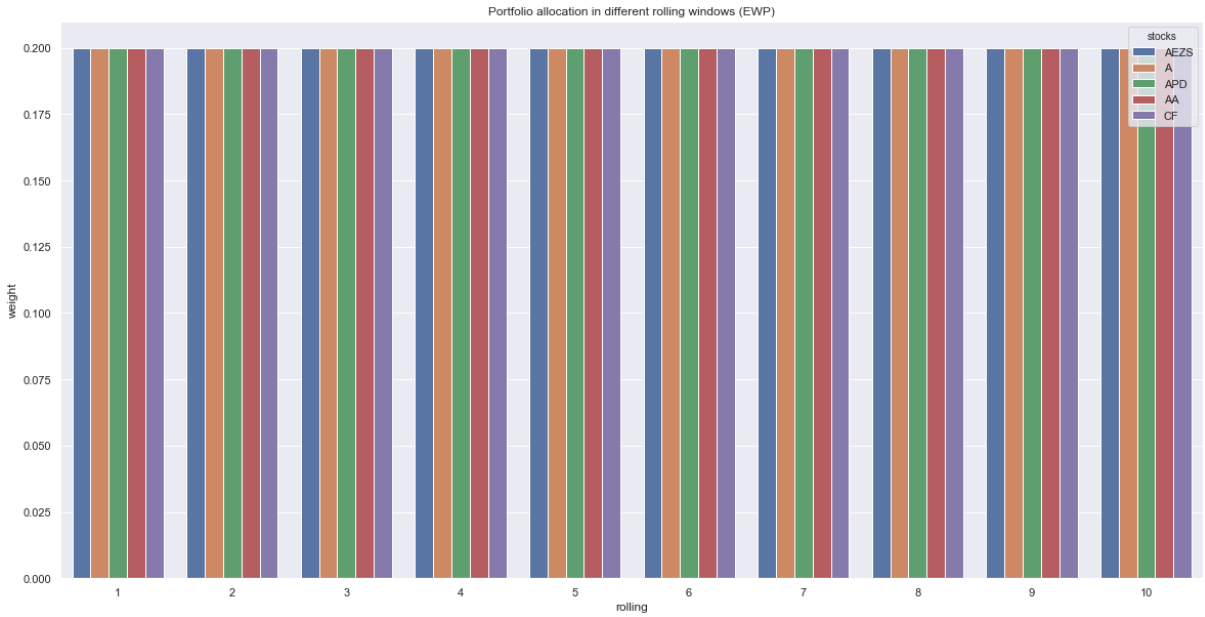
```
IVP_long = IVP_wide.unstack().reset_index()
RPP_long = RPP_wide.unstack().reset_index()

EWP_long.columns = ['stocks','slices','weights']
GMVP_long.columns = ['stocks','slices','weights']
MDP_long.columns = ['stocks','slices','weights']
MDCP_long.columns = ['stocks','slices','weights']
IVP_long.columns = ['stocks','slices','weights']
RPP_long.columns = ['stocks','slices','weights']
```

# Create a grouped bar chart of the portfolio matrix like the following figure

In [208…

```
# plot to compare the allocations
sns.set(rc={'figure.figsize':(20,10)})

ax = sns.barplot(data=EWP_long, x='slices', y='weights', hue='stocks')
ax.set_title("Portfolio allocation in different rolling windows (EWP)")
ax.set(xlabel='rolling', ylabel='weight')
```

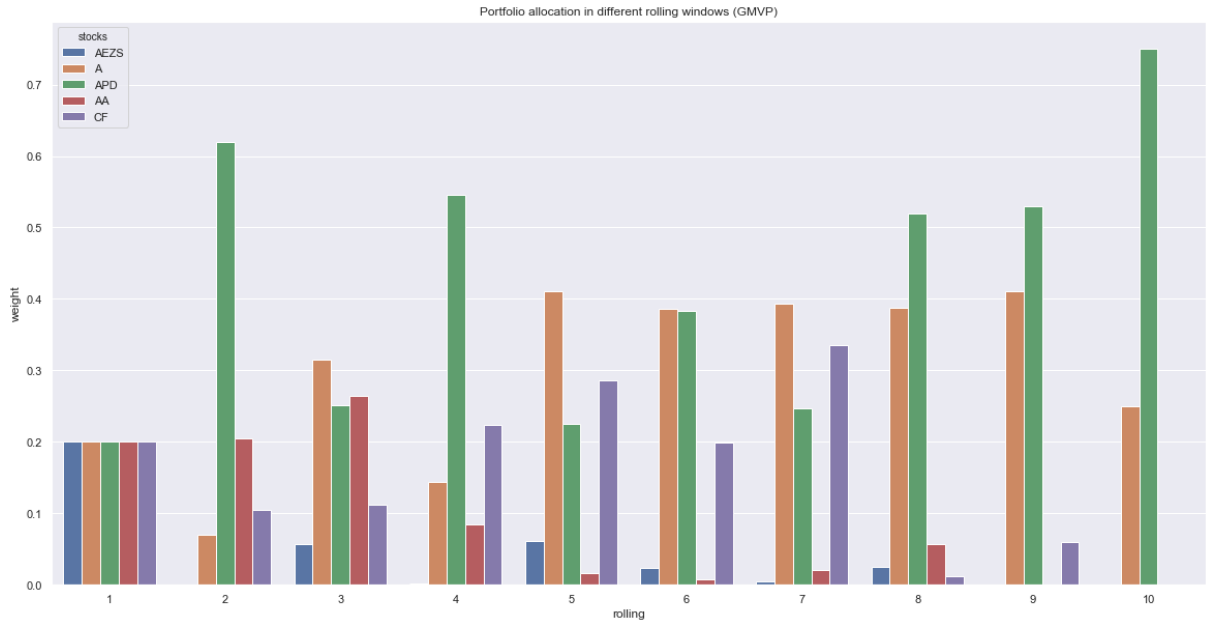
Out[208… [Text(0, 0.5, 'weight'), Text(0.5, 0, 'rolling')]




In [209…

```
# plot to compare the allocations
sns.set(rc={'figure.figsize':(20,10)})

ax = sns.barplot(data=GMVP_long, x='slices', y='weights', hue='stocks')
ax.set_title("Portfolio allocation in different rolling windows (GMVP)")
ax.set(xlabel='rolling', ylabel='weight')
```

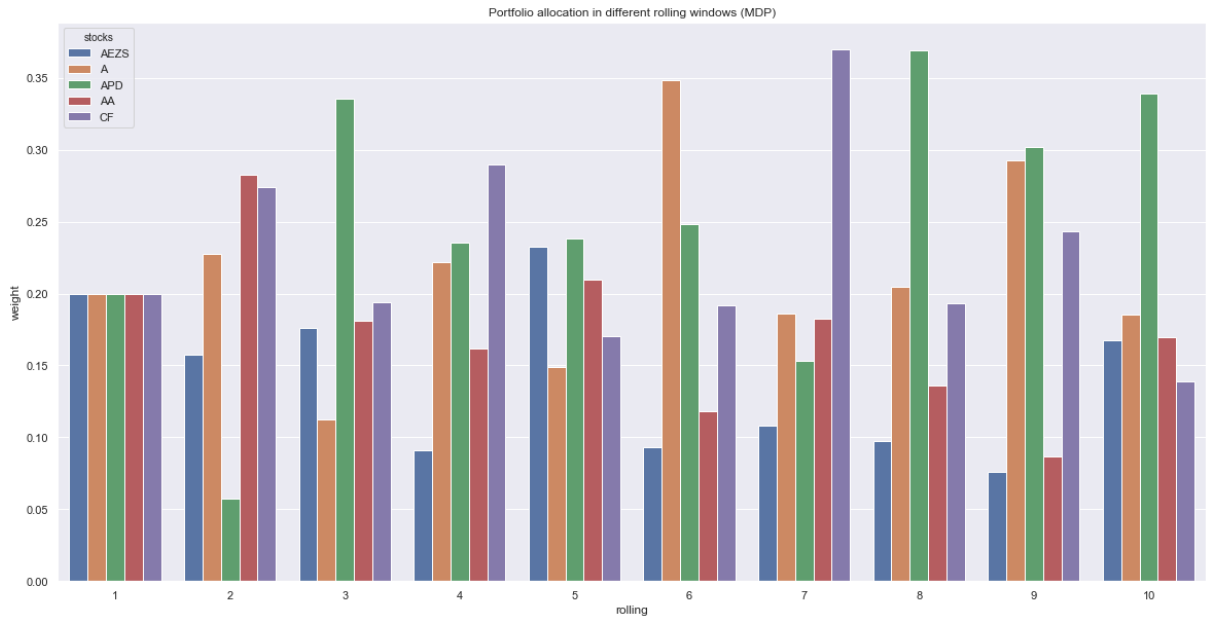
Out[209… [Text(0, 0.5, 'weight'), Text(0.5, 0, 'rolling')]




In [210…

```
# plot to compare the allocations
sns.set(rc={'figure.figsize':(20,10)})

ax = sns.barplot(data=MDP_long, x='slices', y='weights', hue='stocks')
ax.set_title("Portfolio allocation in different rolling windows (MDP)"
ax.set(xlabel='rolling', ylabel='weight')
```

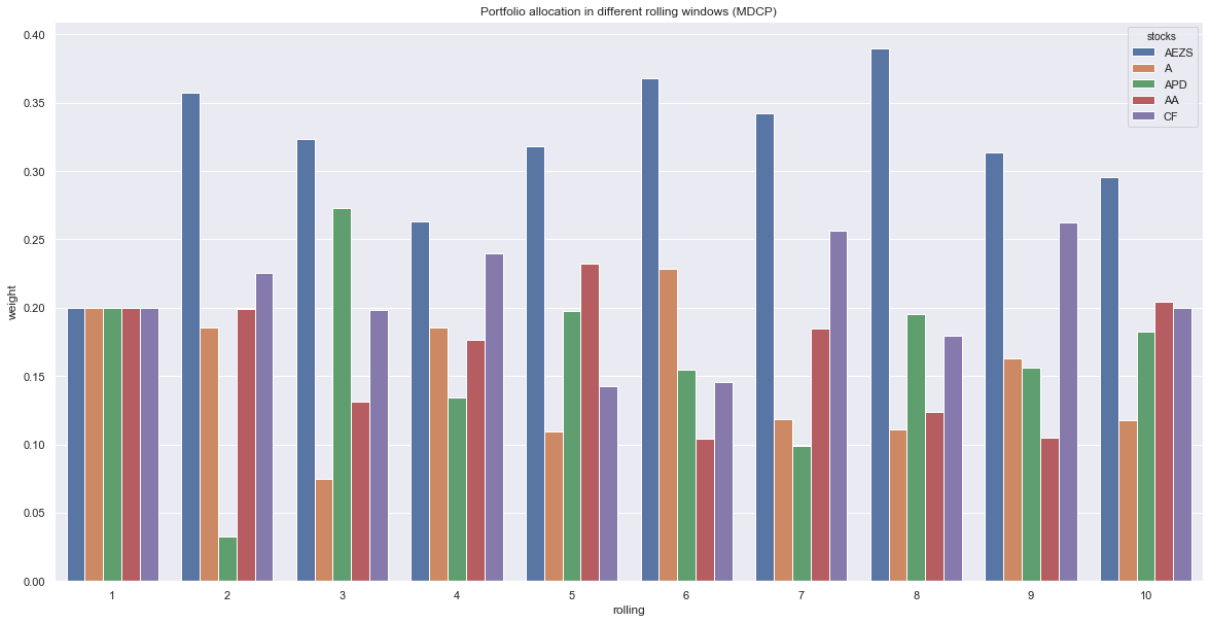
Out[210… [Text(0, 0.5, 'weight'), Text(0.5, 0, 'rolling')]




In [211…

```
# plot to compare the allocations
sns.set(rc={'figure.figsize':(20,10)})

ax = sns.barplot(data=MDCP_long, x='slices', y='weights', hue='stocks'
ax.set_title("Portfolio allocation in different rolling windows (MDCP)
ax.set(xlabel='rolling', ylabel='weight')
```

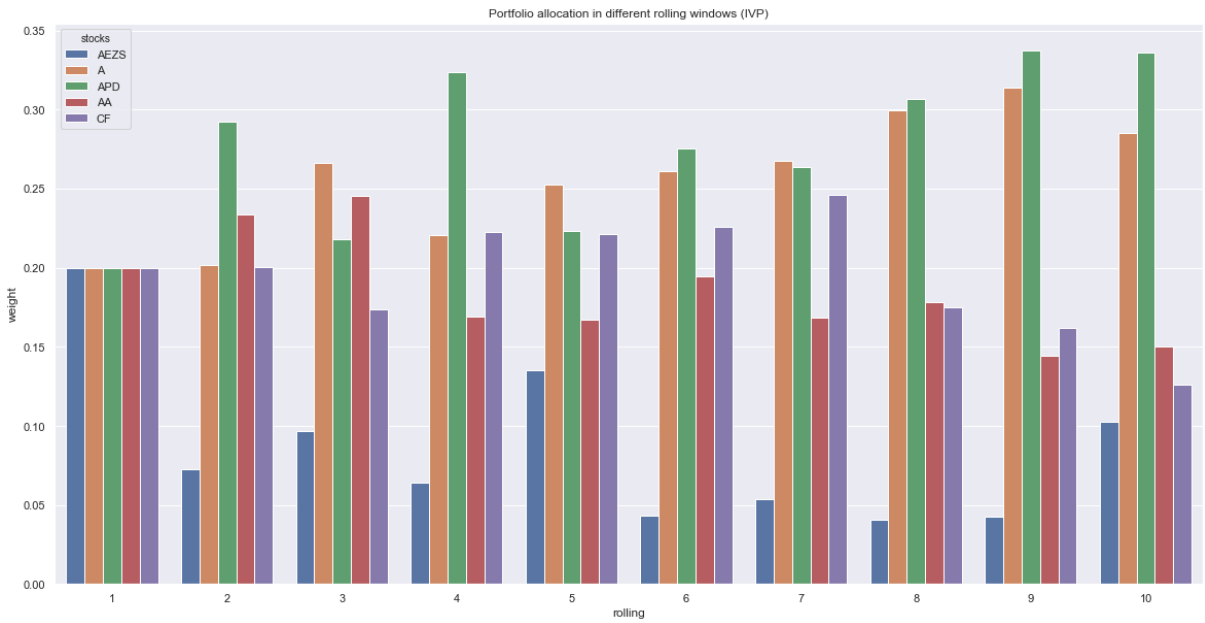
Out[211… [Text(0, 0.5, 'weight'), Text(0.5, 0, 'rolling')]




In [212…

```
# plot to compare the allocations
sns.set(rc={'figure.figsize':(20,10)})

ax = sns.barplot(data=IVP_long, x='slices', y='weights', hue='stocks')
ax.set_title("Portfolio allocation in different rolling windows (IVP)"
ax.set(xlabel='rolling', ylabel='weight')
```

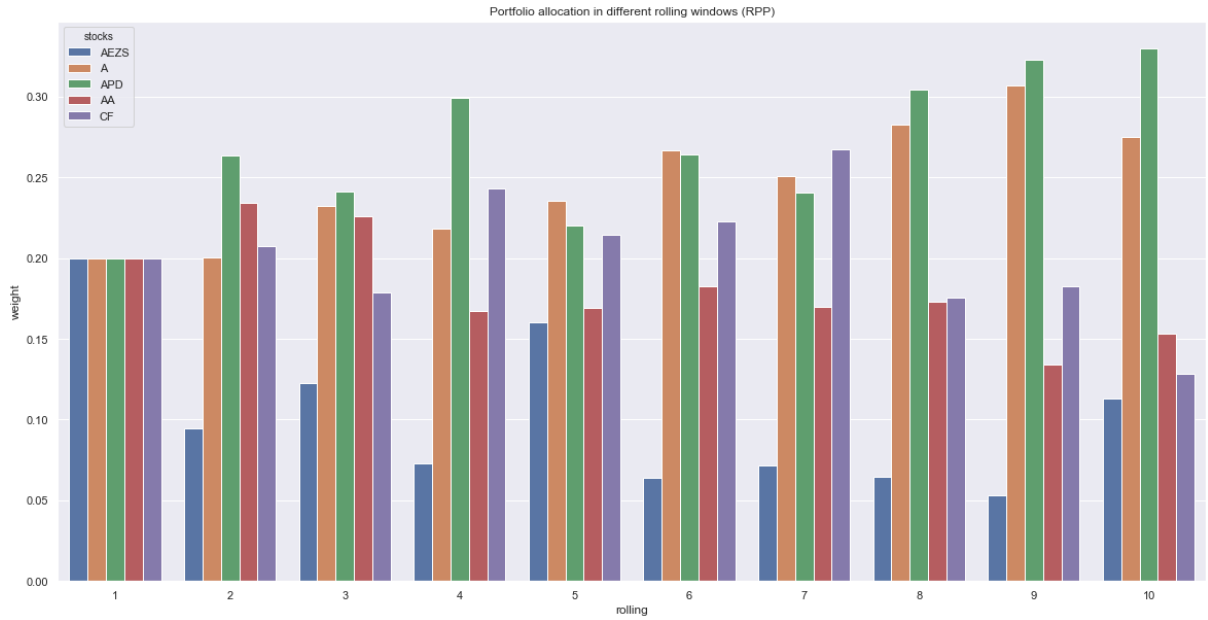
Out[212… [Text(0, 0.5, 'weight'), Text(0.5, 0, 'rolling')]




In [213…

```
# plot to compare the allocations
sns.set(rc={'figure.figsize':(20,10)})

ax = sns.barplot(data=RPP_long, x='slices', y='weights', hue='stocks')
ax.set_title("Portfolio allocation in different rolling windows (RPP)"
ax.set(xlabel='rolling', ylabel='weight')
```

Out[213… [Text(0, 0.5, 'weight'), Text(0.5, 0, 'rolling')]




In [ ]:

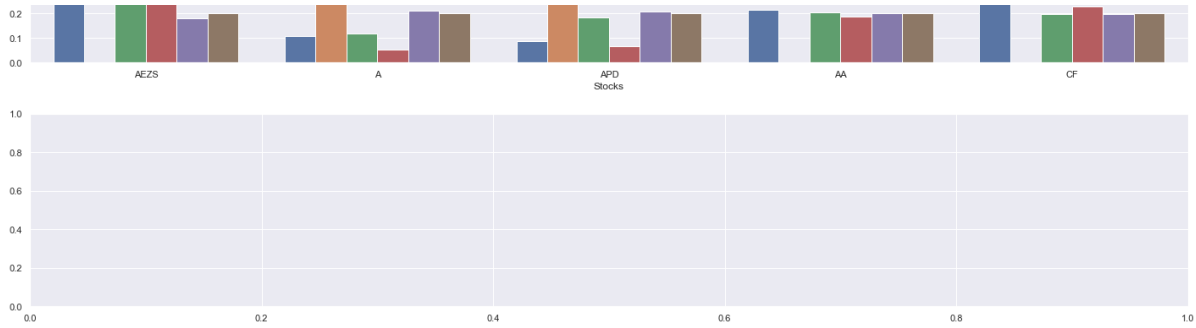
# Relative risk contributions

## In-Sample

In [214…

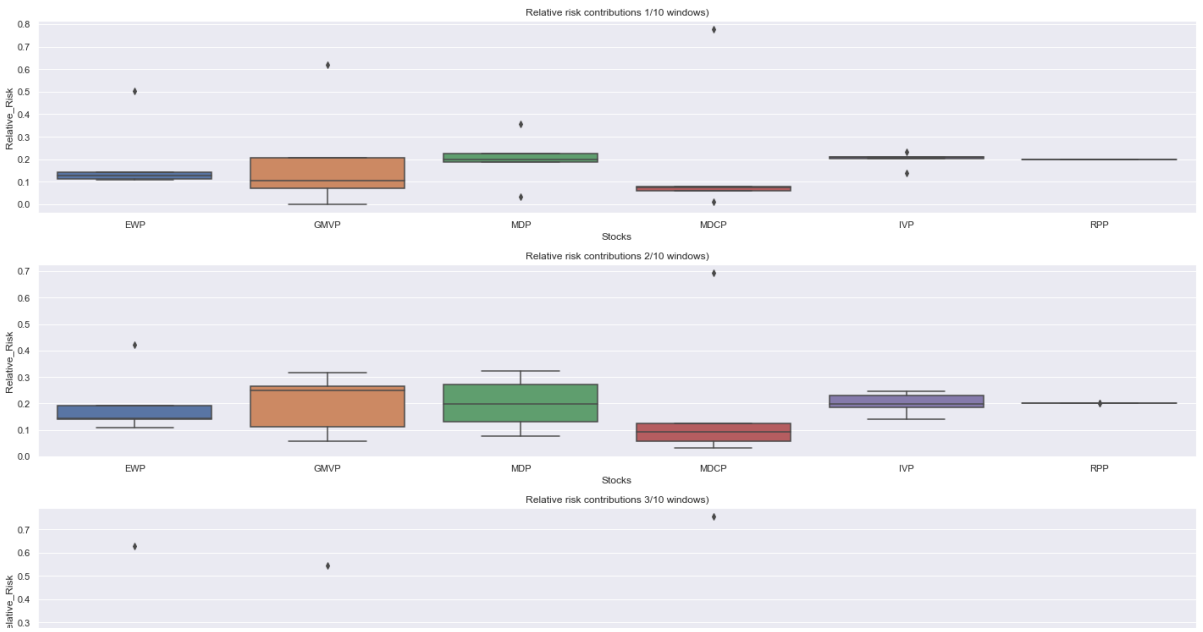
```
# compute risk contributions
sns.set(rc={'figure.figsize':(20,40)})
fig, axes = plt.subplots(10, 1)


for i in range(0,9):
    Total_Risk = pd.DataFrame(np.asarray([w_matrix_EWP[i+1] * (Sigma_l
                                          w_matrix_GMVP[i+1] * (Sigma_li
                                          w_matrix_MDP[i+1] * (Sigma_lir
                                          w_matrix_MDCP[i+1] * (Sigma_li
                                          w_matrix_IVP[i+1] * (Sigma_lir
                                          w_matrix_RPP[i+1] * (Sigma_lir
                              columns=['EWP', 'GMVP', 'MDP', 'MDCP', 'IV
                              index=data.columns)

    Relative_risk = Total_Risk / Total_Risk.sum()
    # plt.bar(Relative_risk, rot=0, title="Relative risk contributions
    # Relative_risk.plot.bar(rot=0, title="Relative risk contributions

    Relative_risk_long = Relative_risk.unstack().reset_index()
    Relative_risk_long.columns = ['Portfolio','Stocks','Relative_Risk'

    # plot to compare the allocations

    sns.barplot(ax=axes[i], data=Relative_risk_long, x='Stocks', y='Re
    axes[i].set_title('Relative risk contributions ' + str(i+1) + '/10
    axes[i].set(xlabel='Stocks', ylabel='Relative_Risk')

fig.tight_layout()
plt.show()
```

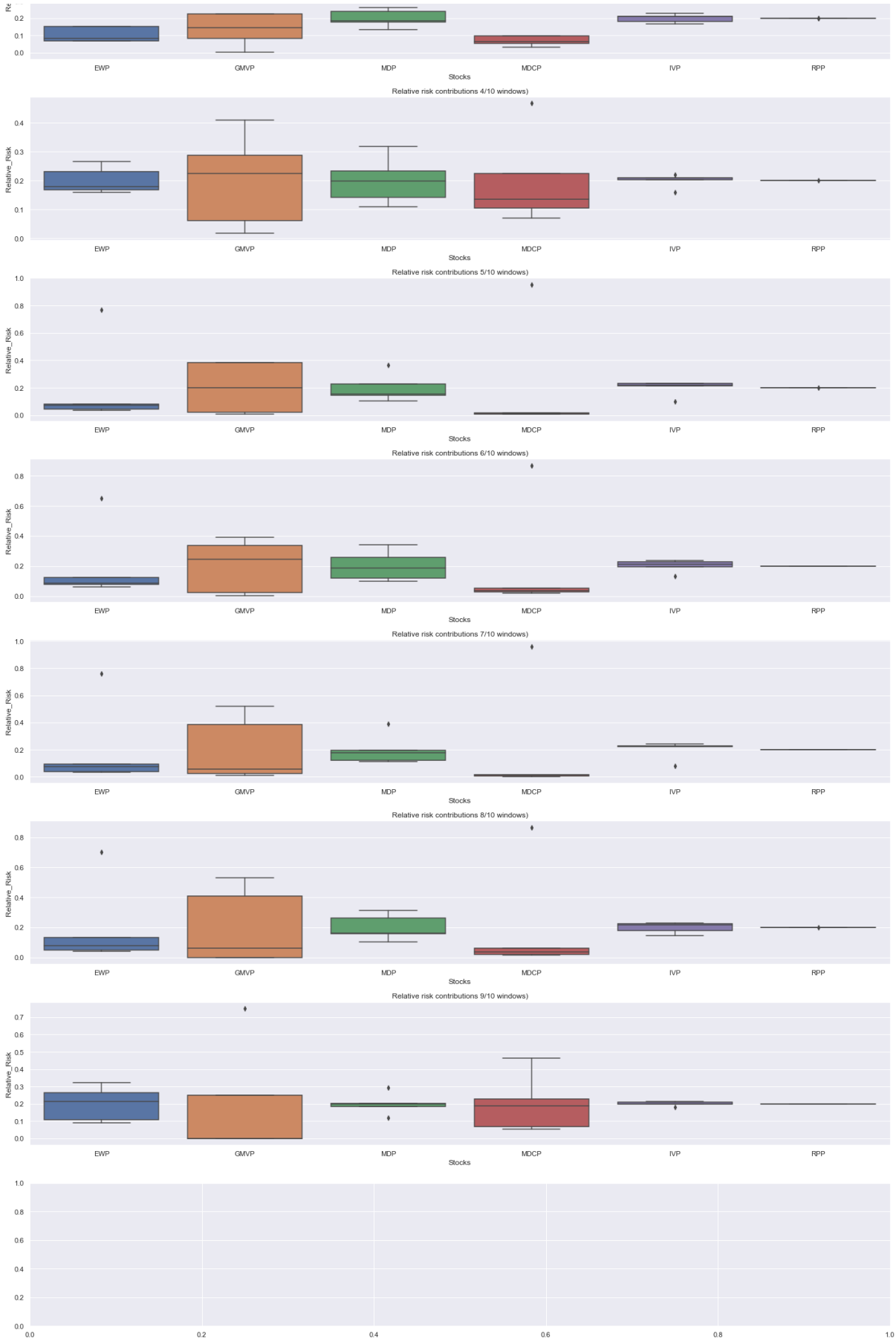

Relative risk contributions 2/10 windows)
Relative risk contributions 3/10 windows)
Relative risk contributions 4/10 windows)
Relative risk contributions 5/10 windows)
Relative risk contributions 6/10 windows)
Relative risk contributions 7/10 windows)
Relative risk contributions 8/10 windows)
Relative risk contributions 9/10 windows)
Relative_Risk
Stocks
AEZS
A
APD
AA
CF
Portfolio
EWP
GMVP
MDP
MDCP
IVP
RPP




In [215…

```
# compute risk contributions
sns.set(rc={'figure.figsize':(20,40)})
fig, axes = plt.subplots(10, 1)


for i in range(0,9):
    Total_Risk = pd.DataFrame(np.asarray([w_matrix_EWP[i+1] * (Sigma_
                                          w_matrix_GMVP[i+1] * (Sigma_
                                          w_matrix_MDP[i+1] * (Sigma_
                                          w_matrix_MDCP[i+1] * (Sigma_
                                          w_matrix_IVP[i+1] * (Sigma_
                                          w_matrix_RPP[i+1] * (Sigma_
                               columns=['EWP', 'GMVP', 'MDP', 'MDCP', 'I
                               index=data.columns)

    Relative_risk = Total_Risk / Total_Risk.sum()
    # plt.bar(Relative_risk, rot=0, title="Relative risk contributions
    # Relative_risk.plot.bar(rot=0, title="Relative risk contributions

    Relative_risk_long = Relative_risk.unstack().reset_index()
    Relative_risk_long.columns = ['Portfolio','Stocks','Relative_Risk

    # plot to compare the allocations

    sns.boxplot(ax = axes[i], data=Relative_risk)
    axes[i].set_title('Relative risk contributions ' + str(i+1) + '/10
    axes[i].set(xlabel='Stocks', ylabel='Relative_Risk')

fig.tight_layout()
plt.show()
```




Relative risk contributions 4/10 windows)
Relative risk contributions 5/10 windows)
Relative risk contributions 6/10 windows)
Relative risk contributions 7/10 windows)
Relative risk contributions 8/10 windows)
Relative risk contributions 9/10 windows)
Relative_Risk
Stocks
EWP
GMVP
MDP
MDCP
IVP
RPP

In [ ]:

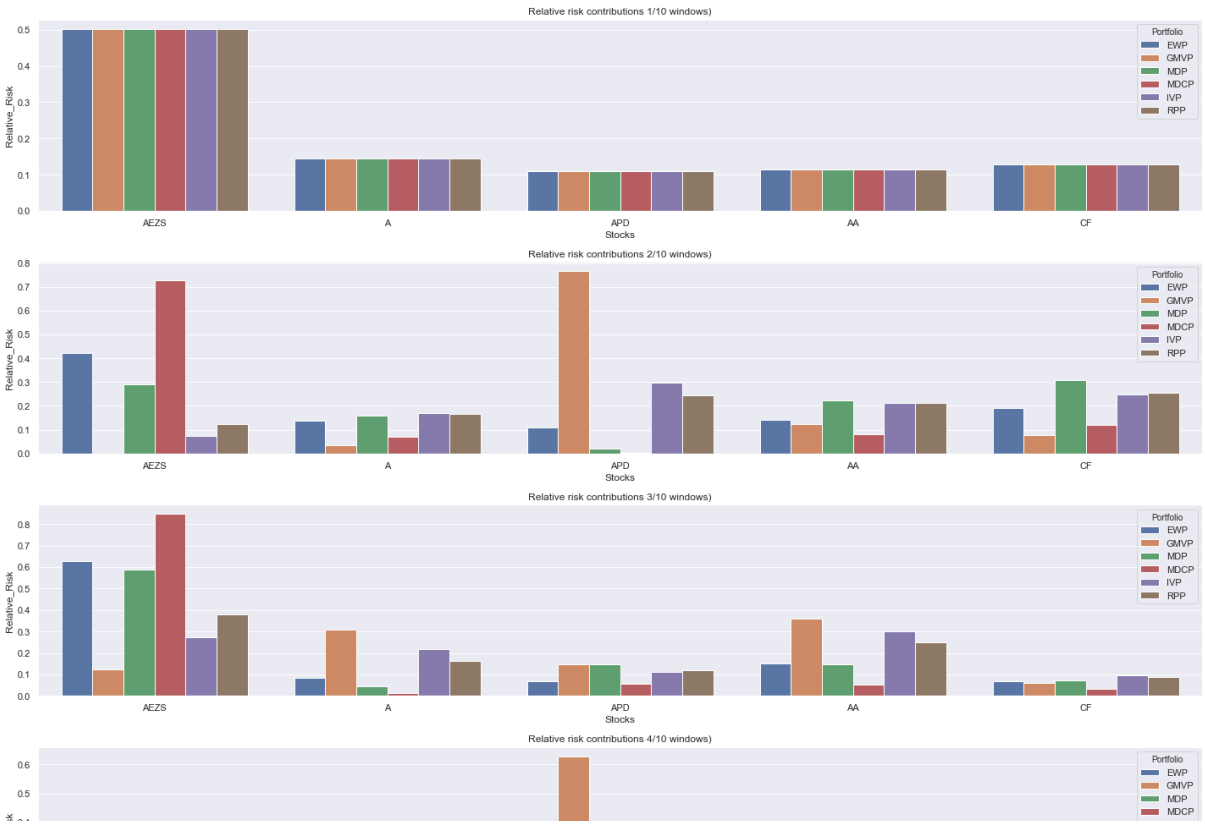
In [ ]:

In [ ]:

## Out-Sample

In [216…

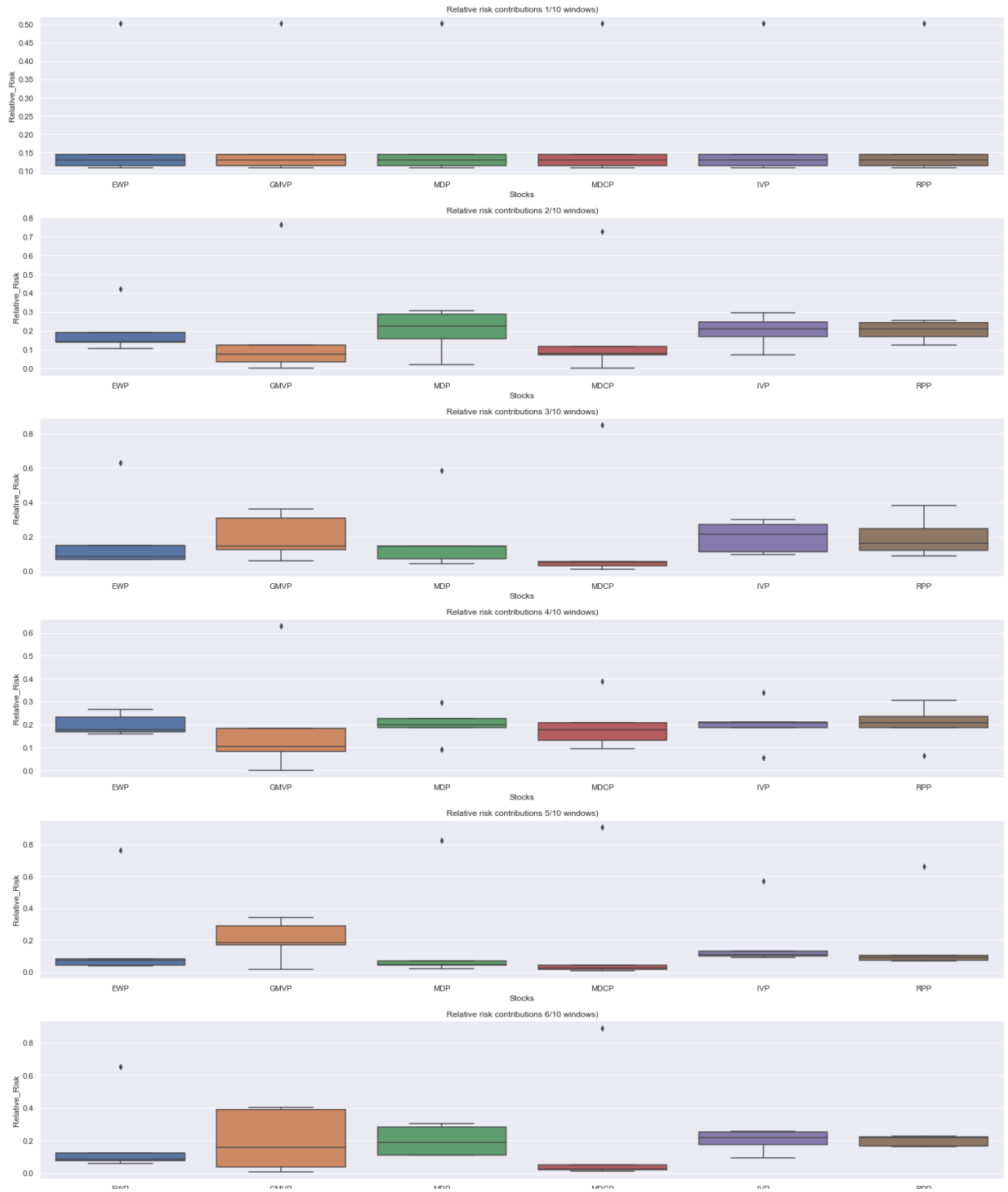
```
# compute risk contributions
sns.set(rc={'figure.figsize':(20,40)})
fig, axes = plt.subplots(10, 1)


for i in range(10):
    Total_Risk = pd.DataFrame(np.asarray([w_matrix_EWP[i] * (Sigma_lir
                                          w_matrix_GMVP[i] * (Sigma_lin_
                                          w_matrix_MDP[i] * (Sigma_lin_r
                                          w_matrix_MDCP[i] * (Sigma_lin_
                                          w_matrix_IVP[i] * (Sigma_lin_r
                                          w_matrix_RPP[i] * (Sigma_lin_r
                              columns=['EWP', 'GMVP', 'MDP', 'MDCP', 'IV
                              index=data.columns)

    Relative_risk = Total_Risk / Total_Risk.sum()
    # plt.bar(Relative_risk, rot=0, title="Relative risk contributions
    # Relative_risk.plot.bar(rot=0, title="Relative risk contributions

    Relative_risk_long = Relative_risk.unstack().reset_index()
    Relative_risk_long.columns = ['Portfolio','Stocks','Relative_Risk'

    # plot to compare the allocations

    sns.barplot(ax=axes[i], data=Relative_risk_long, x='Stocks', y='Re
    axes[i].set_title('Relative risk contributions ' + str(i+1) + '/10
    axes[i].set(xlabel='Stocks', ylabel='Relative_Risk')

fig.tight_layout()
plt.show()
```

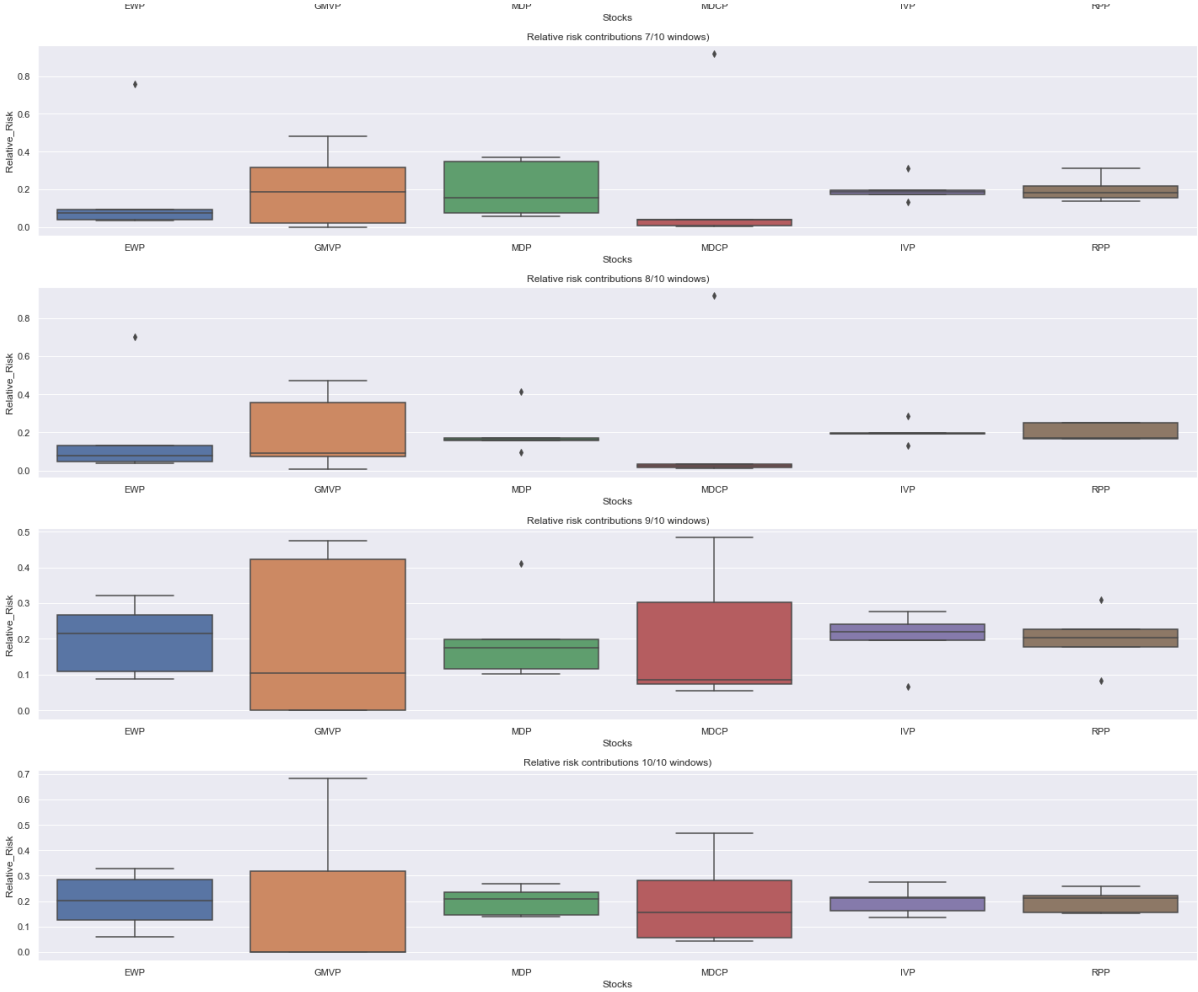



In [217…

```
# compute risk contributions
sns.set(rc={'figure.figsize':(20,40)})
fig, axes = plt.subplots(10, 1)


for i in range(10):
    Total_Risk = pd.DataFrame(np.asarray([w_matrix_EWP[i] * (Sigma_lir
                                          w_matrix_GMVP[i] * (Sigma_lin_
                                          w_matrix_MDP[i] * (Sigma_lin_r
                                          w_matrix_MDCP[i] * (Sigma_lin_
                                          w_matrix_IVP[i] * (Sigma_lin_r
                                          w_matrix_RPP[i] * (Sigma_lin_r
```

```
                                columns=['EWP', 'GMVP', 'MDP', 'MDCP', 'IV
                                index=data.columns)

    Relative_risk = Total_Risk / Total_Risk.sum()
    # plt.bar(Relative_risk, rot=0, title="Relative risk contributions
    # Relative_risk.plot.bar(rot=0, title="Relative risk contributions

    Relative_risk_long = Relative_risk.unstack().reset_index()
    Relative_risk_long.columns = ['Portfolio','Stocks','Relative_Risk'

    # plot to compare the allocations

    sns.boxplot(ax = axes[i], data=Relative_risk)
    axes[i].set_title('Relative risk contributions ' + str(i+1) + '/10
    axes[i].set(xlabel='Stocks', ylabel='Relative_Risk')

fig.tight_layout()
plt.show()
```






In [ ]:

## (c) Compute the simple return (not compounded) of GMVP in this backtest. Plot the return path compared with the EWP.

In [218…

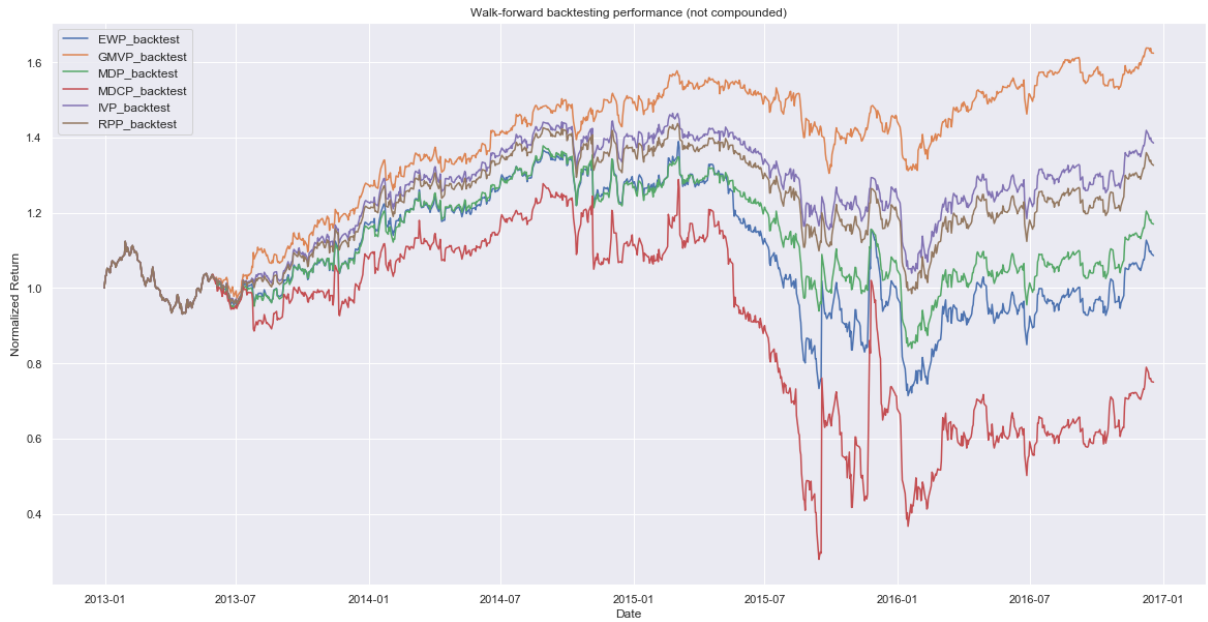
```
EWP_backtest = np.array([1])
GMVP_backtest = np.array([1])
MDP_backtest = np.array([1])
MDCP_backtest = np.array([1])
IVP_backtest = np.array([1])
RPP_backtest = np.array([1])

for i in range(1000):
    EWP_backtest = np.append(EWP_backtest, (EWP_backtest[-1] -1) + np.
    GMVP_backtest = np.append(GMVP_backtest, (GMVP_backtest[-1] -1) +
    MDP_backtest = np.append(MDP_backtest, (MDP_backtest[-1] -1) + np.
    MDCP_backtest = np.append(MDCP_backtest, (MDCP_backtest[-1] -1) +
    IVP_backtest = np.append(IVP_backtest, (IVP_backtest[-1] -1) + np.
    RPP_backtest = np.append(RPP_backtest, (RPP_backtest[-1] -1) + np.

# EWP_backtest, GMVP_backtest, IVP_backtest, RPP_backtest
```

In [219…

```
plt.figure(figsize=(20,10))
```

```
plt.plot(data.index, EWP_backtest, label='EWP_backtest')
plt.plot(data.index, GMVP_backtest, label='GMVP_backtest')
plt.plot(data.index, MDP_backtest, label='MDP_backtest')
plt.plot(data.index, MDCP_backtest, label='MDCP_backtest')
plt.plot(data.index, IVP_backtest, label='IVP_backtest')
plt.plot(data.index, RPP_backtest, label='RPP_backtest')

plt.legend(loc='upper left', fontsize=12)
plt.title('Walk-forward backtesting performance (not compounded)')
plt.xlabel('Date')
plt.ylabel('Normalized Return')
```

Out[219… Text(0, 0.5, 'Normalized Return')




In [220…

```
EWP_return_log = np.diff(np.log(EWP_backtest))
GMVP_return_log = np.diff(np.log(GMVP_backtest))
MDP_return_log = np.diff(np.log(MDP_backtest))
MDCP_return_log = np.diff(np.log(MDCP_backtest))
IVP_return_log = np.diff(np.log(IVP_backtest))
RPP_return_log = np.diff(np.log(RPP_backtest))
```

In [221…

```
print('The mean return of EWP_backtest: ', np.mean(EWP_return_log)*252
print('The mean return of GMVP_backtest: ', np.mean(GMVP_return_log)*2
print('The mean return of MDP_backtest: ', np.mean(MDP_return_log)*252
print('The mean return of MDCP_backtest: ', np.mean(MDCP_return_log)*2
print('The mean return of IVP_backtest: ', np.mean(IVP_return_log)*252
print('The mean return of RPP_backtest: ', np.mean(RPP_return_log)*252
```

```
The mean return of EWP_backtest:  0.020791955225549226
The mean return of GMVP_backtest:  0.1221291328613562
The mean return of MDP_backtest:  0.03950362846822808
The mean return of MDCP_backtest:  -0.07253026266485857
The mean return of IVP_backtest:  0.08208096603003597
The mean return of RPP_backtest:  0.07105684707770556
```

In [222…

```
print('The volatility of EWP_backtest: ', np.std(EWP_return_log)*np.sc
print('The volatility of GMVP_backtest: ', np.std(GMVP_return_log)*np.
print('The volatility of MDP_backtest: ', np.std(MDP_return_log)*np.sc
```

```
print('The volatility of MDCP_backtest: ', np.std(MDCP_return_log)*np.
print('The volatility of IVP_backtest: ', np.std(IVP_return_log)*np.sc
print('The volatility of RPP_backtest: ', np.std(RPP_return_log)*np.sc
```

```
The volatility of EWP_backtest:  0.32403826809119696
The volatility of GMVP_backtest:  0.144775231186643
The volatility of MDP_backtest:  0.22818689806949163
The volatility of MDCP_backtest:  0.7802350433166365
The volatility of IVP_backtest:  0.17171836874843596
The volatility of RPP_backtest:  0.18487740007530737
```

In [223…

```
print('The Sharpe ratio of EWP_backtest: ', np.mean(EWP_return_log)*np
print('The Sharpe ratio of GMVP_backtest: ', np.mean(GMVP_return_log)*
print('The Sharpe ratio of MDP_backtest: ', np.mean(MDP_return_log)*np
print('The Sharpe ratio of MDCP_backtest: ', np.mean(MDCP_return_log)*
print('The Sharpe ratio of IVP_backtest: ', np.mean(IVP_return_log)*np
print('The Sharpe ratio of RPP_backtest: ', np.mean(RPP_return_log)*np
```

```
The Sharpe ratio of EWP_backtest:  0.06416512268142836
The Sharpe ratio of GMVP_backtest:  0.8435775364358311
The Sharpe ratio of MDP_backtest:  0.17311961730685219
The Sharpe ratio of MDCP_backtest:  -0.09295950404450651
The Sharpe ratio of IVP_backtest:  0.477997587726232
The Sharpe ratio of RPP_backtest:  0.3843457721103904
```

In [224…

```
import empyrical as ep

print('The max_drawdown of EWP_log_return: ', ep.max_drawdown(EWP_retu
print('The max_drawdown of GMVP_log_return: ', ep.max_drawdown(GMVP_re
print('The max_drawdown of MDP_log_return: ', ep.max_drawdown(MDP_retu
print('The max_drawdown of MDCP_log_return: ', ep.max_drawdown(MDCP_re
print('The max_drawdown of IVP_log_return: ', ep.max_drawdown(IVP_retu
print('The max_drawdown of RPP_log_return: ', ep.max_drawdown(RPP_retu
```

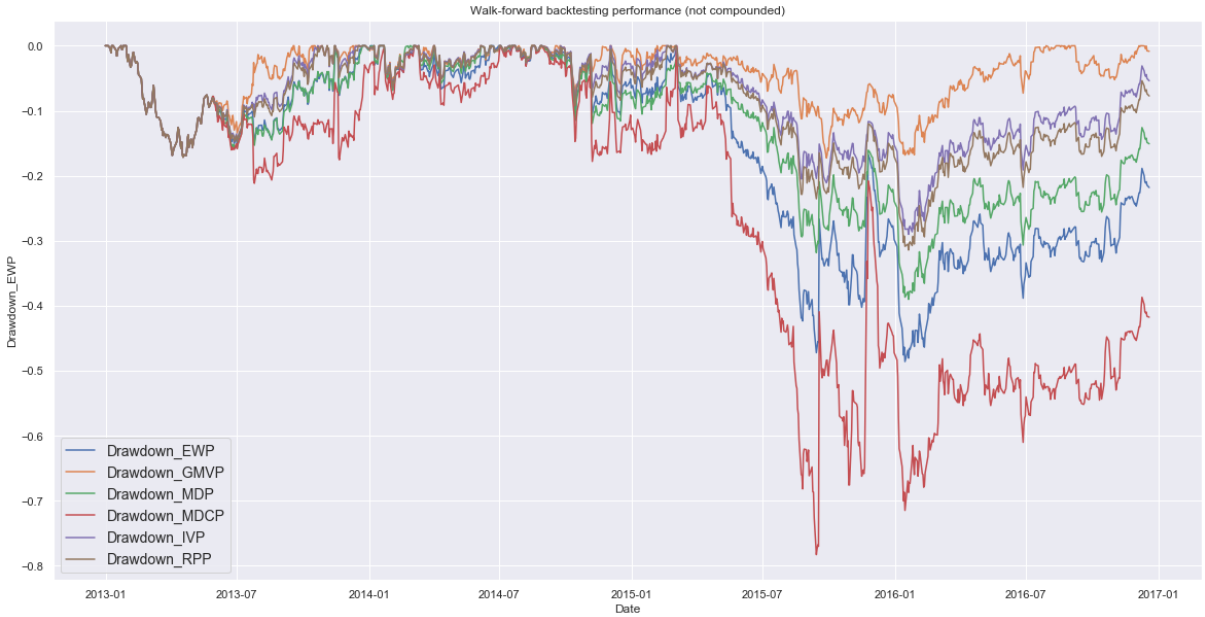
```
The max_drawdown of EWP_log_return:  -0.5431247638652615
The max_drawdown of GMVP_log_return:  -0.17783189716375583
The max_drawdown of MDP_log_return:  -0.4220098884053489
The max_drawdown of MDCP_log_return:  -0.8710096580991978
The max_drawdown of IVP_log_return:  -0.30187805912705423
The max_drawdown of RPP_log_return:  -0.3292613978781877
```

In [225…

```
df_EWP = pd.DataFrame(EWP_backtest)
df_GMVP = pd.DataFrame(GMVP_backtest)
df_MDP = pd.DataFrame(MDP_backtest)
df_MDCP = pd.DataFrame(MDCP_backtest)
df_IVP = pd.DataFrame(IVP_backtest)
df_RPP = pd.DataFrame(RPP_backtest)

Roll_Max_EWP = df_EWP.cummax()
Roll_Max_GMVP = df_GMVP.cummax()
Roll_Max_MDP = df_MDP.cummax()
Roll_Max_MDCP = df_MDCP.cummax()
Roll_Max_IVP = df_IVP.cummax()
Roll_Max_RPP = df_RPP.cummax()

Daily_Drawdown_EWP = df_EWP/Roll_Max_EWP - 1.0
Daily_Drawdown_GMVP = df_GMVP/Roll_Max_GMVP - 1.0
Daily_Drawdown_MDP = df_MDP/Roll_Max_MDP - 1.0
Daily_Drawdown_MDCP = df_MDCP/Roll_Max_MDCP - 1.0
```

```
Daily_Drawdown_IVP = df_IVP/Roll_Max_IVP - 1.0
Daily_Drawdown_RPP = df_RPP/Roll_Max_RPP - 1.0

Max_Daily_Drawdown_EWP = Daily_Drawdown_EWP.cummin()
Max_Daily_Drawdown_GMVP = Daily_Drawdown_GMVP.cummin()
Max_Daily_Drawdown_MDP = Daily_Drawdown_MDP.cummin()
Max_Daily_Drawdown_MDCP = Daily_Drawdown_MDCP.cummin()
Max_Daily_Drawdown_IVP = Daily_Drawdown_IVP.cummin()
Max_Daily_Drawdown_RPP = Daily_Drawdown_RPP.cummin()


plt.figure(figsize=(20,10))
plt.plot(data.index, Daily_Drawdown_EWP, label='Drawdown_EWP')
plt.plot(data.index, Daily_Drawdown_GMVP, label='Drawdown_GMVP')
plt.plot(data.index, Daily_Drawdown_MDP, label='Drawdown_MDP')
plt.plot(data.index, Daily_Drawdown_MDCP, label='Drawdown_MDCP')
plt.plot(data.index, Daily_Drawdown_IVP, label='Drawdown_IVP')
plt.plot(data.index, Daily_Drawdown_RPP, label='Drawdown_RPP')

plt.legend(loc='lower left', fontsize=14)
plt.title('Walk-forward backtesting performance (not compounded)')
plt.xlabel('Date')
plt.ylabel('Drawdown_EWP')
```

Out[225… `Text(0, 0.5, 'Drawdown_EWP')`

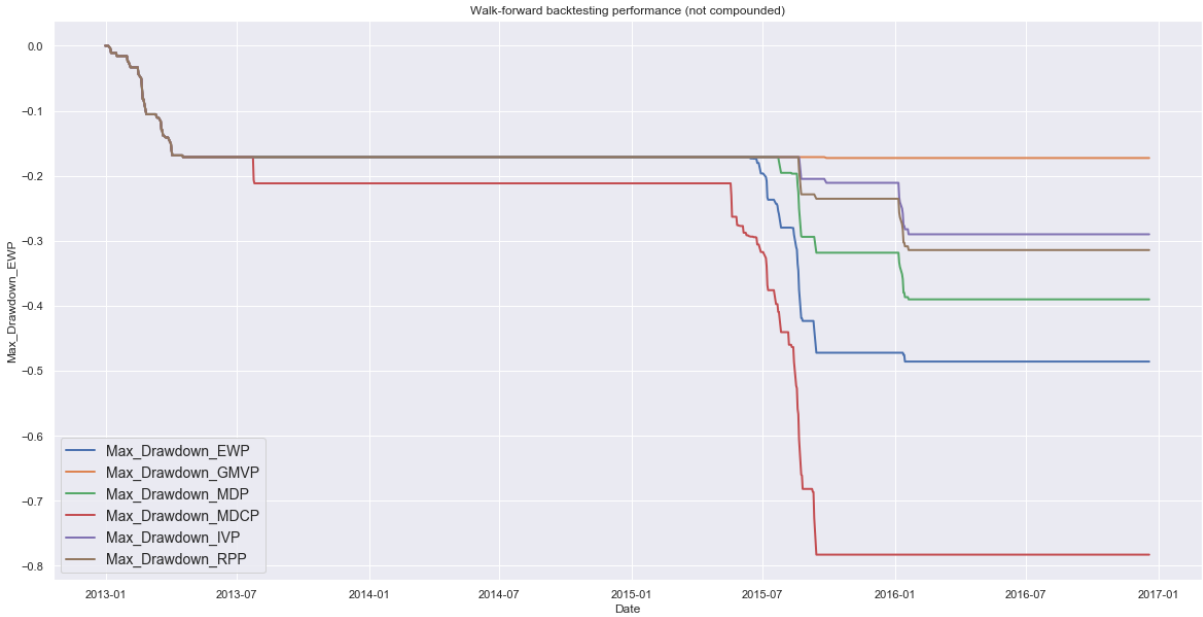



In [226…

```
plt.figure(figsize=(20,10))
plt.plot(data.index, Max_Daily_Drawdown_EWP, lw=2, label='Max_Drawdown
plt.plot(data.index, Max_Daily_Drawdown_GMVP, lw=2, label='Max_Drawdow
plt.plot(data.index, Max_Daily_Drawdown_MDP, lw=2, label='Max_Drawdown
plt.plot(data.index, Max_Daily_Drawdown_MDCP, lw=2, label='Max_Drawdow
plt.plot(data.index, Max_Daily_Drawdown_IVP, lw=2, label='Max_Drawdown
plt.plot(data.index, Max_Daily_Drawdown_RPP, lw=2, label='Max_Drawdown

plt.legend(loc='lower left', fontsize=14)
plt.title('Walk-forward backtesting performance (not compounded)')
plt.xlabel('Date')
plt.ylabel('Max_Drawdown_EWP')
```

`Text(0, 0.5, 'Max_Drawdown_EWP')`

Out[226…




# (d) Implement a GARCH(1,1) model on the log-returns of the APD stock. In class, we have shown the usage of relevant R packages. In the problem, you have to find a Python package online, install it, and use it correctly by yourself. Plot the log-returns and the estimated volatility in one subplot:

In [227…

```
from arch import arch_model
```

In [228…

```
data_return_log
```

Out[228…

| Date | AEZS | A | APD | AA | CF |
|---|---|---|---|---|---|
| **2013-01-02** | 0.061119 | 0.022701 | 0.021663 | 0.035092 | 0.019883 |
| **2013-01-03** | 0.003945 | 0.003575 | -0.003500 | 0.008859 | -0.004740 |
| **2013-01-04** | 0.011742 | 0.019555 | 0.013351 | 0.020732 | 0.022152 |
| **2013-01-07** | 0.007752 | -0.007259 | -0.000923 | -0.017430 | -0.003753 |
| **2013-01-08** | -0.015565 | -0.008023 | 0.001845 | 0.000000 | -0.014769 |
| **…** | … | … | … | … | … |
| **2016-12-13** | -0.013072 | 0.005403 | -0.003481 | -0.032534 | -0.020554 |
| **2016-12-14** | 0.000000 | -0.005403 | -0.006324 | 0.006920 | -0.010616 |
| **2016-12-15** | 0.000000 | 0.009062 | 0.006122 | -0.021913 | 0.019028 |
| **2016-12-16** | -0.026668 | -0.014059 | -0.011197 | -0.013178 | 0.028559 |
| **2016-12-19** | 0.026668 | -0.010510 | 0.001965 | -0.003748 | -0.041204 |

1000 rows × 5 columns

In [229…
```
# define model
model = arch_model(data_return_log['APD'], vol='GARCH', p=1, q=1)
```

In [230…
```
# fit model
model_fit = model.fit()

print(model_fit.summary())
```

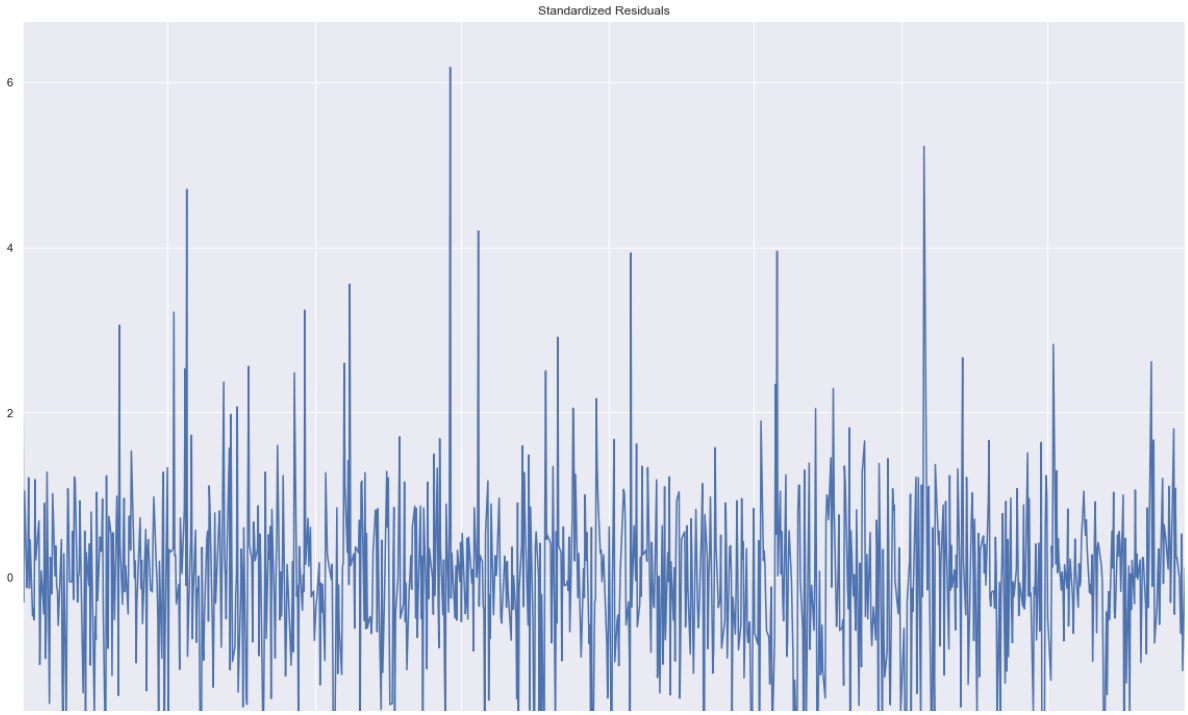
```
Iteration:      1,   Func. Count:      6,   Neg. LLF: 369191348.434713
3
Iteration:      2,   Func. Count:     18,   Neg. LLF: 142010.351114428
51
Iteration:      3,   Func. Count:     30,   Neg. LLF: 2784.55487106210
8
Iteration:      4,   Func. Count:     40,   Neg. LLF: 3015291.47930144
Iteration:      5,   Func. Count:     52,   Neg. LLF: 4314.48182658647
8
Iteration:      6,   Func. Count:     62,   Neg. LLF: 3369665.77157655
1
Iteration:      7,   Func. Count:     73,   Neg. LLF: 162190.541360264
7
Iteration:      8,   Func. Count:     84,   Neg. LLF: 85190350373.9645
5
Iteration:      9,   Func. Count:     96,   Neg. LLF: 4263210065.36615
37
Optimization terminated successfully    (Exit mode 0)
            Current function value: -2966.0366464936933
            Iterations: 10
            Function evaluations: 105
            Gradient evaluations: 9
                     Constant Mean - GARCH Model Results
==============================================================================
========
Dep. Variable:                     APD   R-squared:
0.000
Mean Model:             Constant Mean   Adj. R-squared:
0.000
Vol Model:                      GARCH   Log-Likelihood:
2966.04
Distribution:                  Normal   AIC:
-5924.07
Method:            Maximum Likelihood   BIC:
-5904.44
                                        No. Observations:
1000
Date:                Tue, May 04 2021   Df Residuals:
999
Time:                        12:57:50   Df Model:
1
                                  Mean Model
==============================================================================
=======
                 coef    std err          t      P>|t|       95.0% Con
f. Int.
------------------------------------------------------------------------------
-------
mu          5.8898e-04  3.662e-04      1.609      0.108 [-1.287e-04,1.3
```

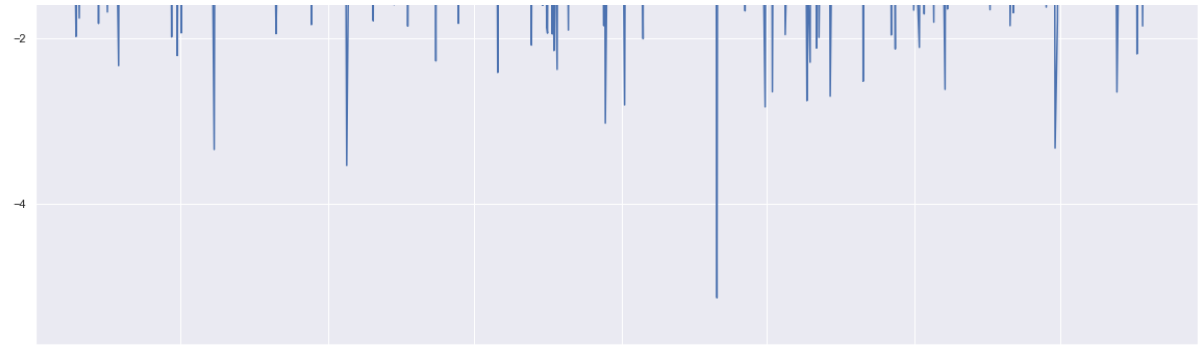
```
07e-03]
                              Volatility Model
==============================================================================
======
                 coef    std err          t      P>|t|      95.0% Con
f. Int.
------------------------------------------------------------------------------
------
omega      4.8390e-05  5.252e-06      9.214  3.156e-20 [3.810e-05,5.86
8e-05]
alpha[1]       0.2000  6.545e-02      3.056  2.245e-03   [7.172e-02,
0.328]
beta[1]        0.5000  6.272e-02      7.973  1.554e-15     [  0.377,
0.623]
==============================================================================
======

Covariance estimator: robust
/Users/kenneth/anaconda3/lib/python3.7/site-packages/arch/univariate/b
ase.py:317: DataScaleWarning: y is poorly scaled, which may affect con
vergence of the optimizer when
estimating the model parameters. The scale of y is 0.0001611. Paramete
r
estimation work better when this value is between 1 and 1000. The reco
mmended
rescaling is 100 * y.

This warning can be disabled by either rescaling y before initializing
the
model or by setting rescale=False.

  data_scale_warning.format(orig_scale, rescale), DataScaleWarning
/Users/kenneth/anaconda3/lib/python3.7/site-packages/scipy/optimize/op
timize.py:283: RuntimeWarning: Values in x were outside bounds during
a minimize step, clipping to bounds
  "minimize step, clipping to bounds", RuntimeWarning)
```

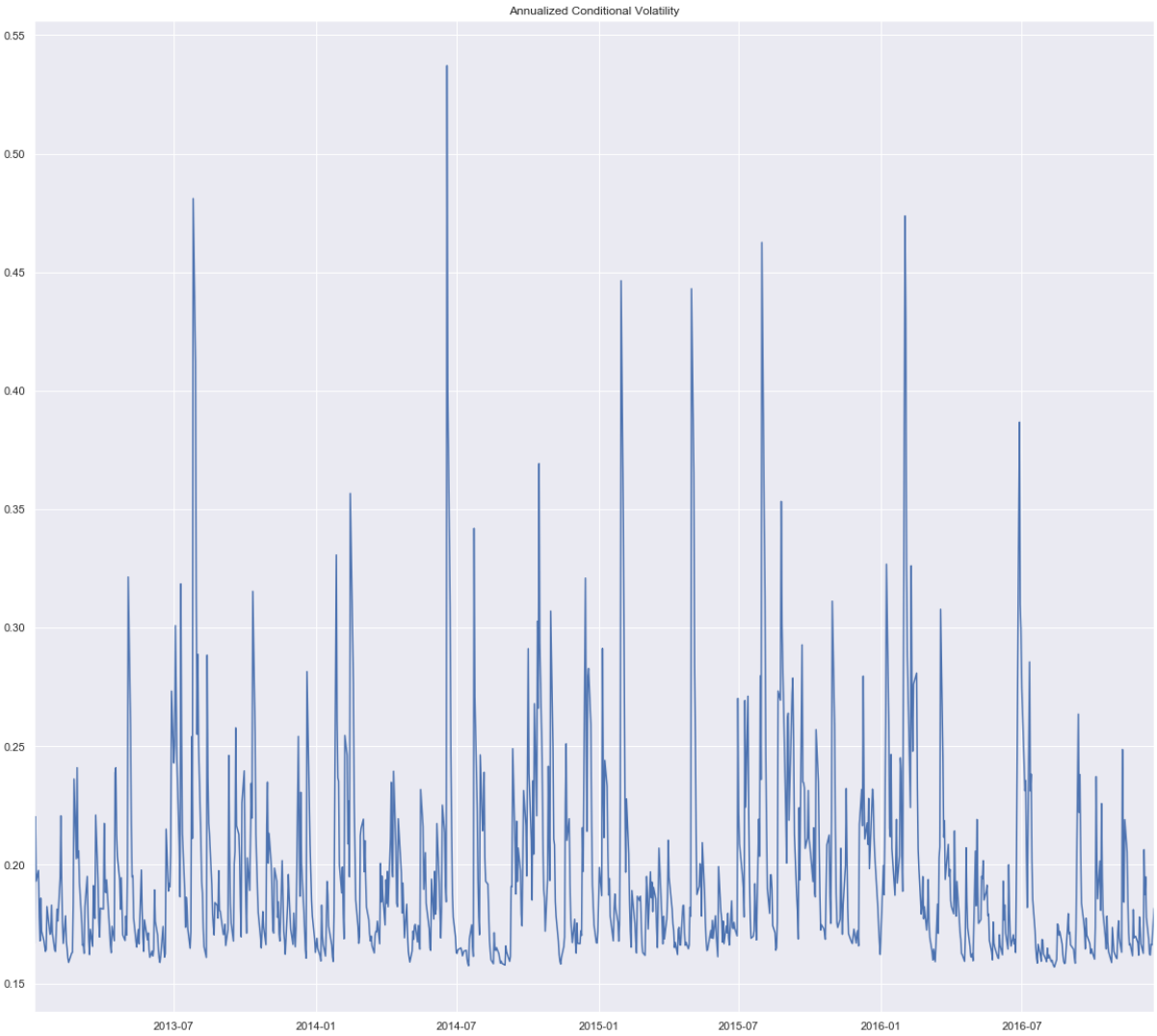
In [231…

```
plt.figure(figsize=(20,10))
fig = model_fit.plot(annualize="D")
```

```
<Figure size 1440x720 with 0 Axes>
```



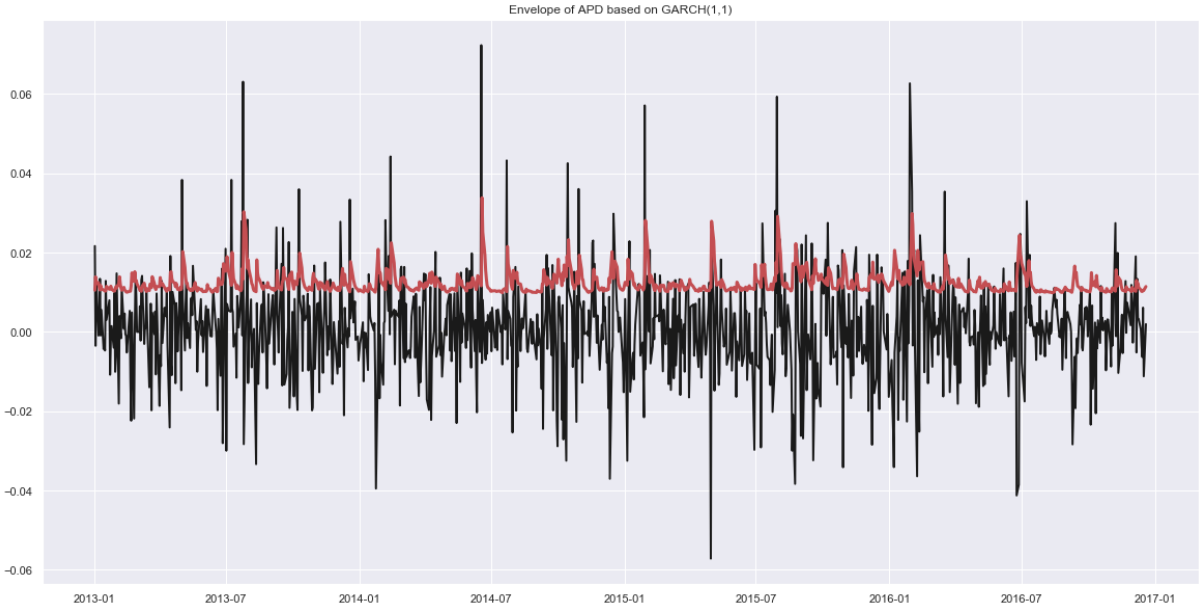



In [232…

```
plt.figure(figsize=(20,10))
plt.plot(data_return_log['APD'], color='k', linewidth=2)
plt.plot(model_fit.conditional_volatility, color='r', linewidth=3)
plt.title('Envelope of APD based on GARCH(1,1)')
```

Out[232… Text(0.5, 1.0, 'Envelope of APD based on GARCH(1,1)')




In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: