Udacity P3 Smart Beta

def generate\_dollar\_volume\_weights(close, volume):

"""

Generate dollar volume weights.

Parameters

----------

close : DataFrame

Close price for each ticker and date

volume : str

Volume for each ticker and date

Returns

-------

dollar\_volume\_weights : DataFrame

The dollar volume weights for each ticker and date

"""

assert close.index.equals(volume.index)

assert close.columns.equals(volume.columns)

#TODO: Implement function

print("close:", close.shape, close)#(4,3)

print("volume:", volume.shape, volume)#(4,3)

dollar\_volume = close \* volume

print("dollar\_volume:",dollar\_volume.shape, dollar\_volume)#(4,3)

total\_dollar\_volume = dollar\_volume.sum( axis=1 )

dollar\_volume\_weights = dollar\_volume.div( total\_dollar\_volume, axis = 0 )

return dollar\_volume\_weights

project\_tests.test\_generate\_dollar\_volume\_weights(generate\_dollar\_volume\_weights)

—————------------------------------------------------------------------------------------

def calculate\_dividend\_weights(dividends):

"""

Calculate dividend weights.

Parameters

----------

dividends : DataFrame

Dividend for each stock and date

Returns

-------

dividend\_weights : DataFrame

Weights for each stock and date

"""

#TODO: Implement function

print("Dividiens:",dividends.shape,dividends)

cumulate\_div = np.cumsum(dividends, axis = 0)#by col 3 col 3 value

print("cumulate\_div:", cumulate\_div.shape, cumulate\_div)

total\_div = cumulate\_div.sum(axis = 1)#by row one value

print("total\_div:", total\_div.shape, total\_div)

dividend\_weights = cumulate\_div.div( total\_div, axis = 0)

print("dividend\_weights", dividend\_weights.shape, dividend\_weights)

return dividend\_weights

project\_tests.test\_calculate\_dividend\_weights(calculate\_dividend\_weights)

————————————————————————————————————————

# Since we're not dealing with volatility, we don't have to use log returns.

def generate\_returns(prices):

"""

Generate returns for ticker and date.

Parameters

----------

prices : DataFrame

Price for each ticker and date

Returns

-------

returns : Dataframe

The returns for each ticker and date

"""

#TODO: Implement function

print("prices:", prices.shape, prices)

lastprice = prices.shift(1)

print("prices shift one day",lastprice.shape, lastprice)

returns = (prices - lastprice)/lastprice

print("return:",returns.shape, returns)

return returns

project\_tests.test\_generate\_returns(generate\_returns)

————————————————————————————————————————————————————

def generate\_weighted\_returns(returns, weights):

"""

Generate weighted returns.

Parameters

----------

returns : DataFrame

Returns for each ticker and date

weights : DataFrame

Weights for each ticker and date

Returns

-------

weighted\_returns : DataFrame

Weighted returns for each ticker and date

"""

assert returns.index.equals(weights.index)

assert returns.columns.equals(weights.columns)

#TODO: Implement function

print("returns:", returns.shape, returns)

print("weights:", weights.shape, weights)

return returns\*weights

project\_tests.test\_generate\_weighted\_returns(generate\_weighted\_returns)

————————————————————————————————————————————def get\_covariance\_returns(returns):

"""

Calculate covariance matrices.

Parameters

----------

returns : DataFrame

Returns for each ticker and date

Returns

-------

returns\_covariance : 2 dimensional Ndarray

The covariance of the returns

"""

#TODO: Implement function

# return np.cov(returns) #AssertionError: Wrong shape for output returns\_covariance. Got (4, 4), expected (3, 3)

# returns\_covariance = np.cov(returns.fillna(0)) #AssertionError: Wrong shape for output returns\_covariance. Got (4, 4), expected (3, 3)

# # print(returns\_covariance.shape) #(4, 4)

# print(returns\_covariance)

# [[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]

# [0.00000000e+00 1.64036278e-03 1.95188774e-04 1.51535604e-02]

# [0.00000000e+00 1.95188774e-04 2.75514630e-05 1.08867748e-03]

# [0.00000000e+00 1.51535604e-02 1.08867748e-03 2.57993031e-01]]

returns\_covariance = np.cov(returns.fillna(0), rowvar = False)

# print(returns\_covariance.shape)#(3, 3)

# print(returns\_covariance)#(3, 3)

# [[0.89856076 0.7205586 0.8458721 ]

# [0.7205586 0.78707297 0.76450378]

# [0.8458721 0.76450378 0.83182775]]

return returns\_covariance

project\_tests.test\_get\_covariance\_returns(get\_covariance\_returns)

——————————

import cvxpy as cvx

def get\_optimal\_weights(covariance\_returns, index\_weights, scale=2.0):

"""

Find the optimal weights.

Parameters

----------

covariance\_returns : 2 dimensional Ndarray

The covariance of the returns

index\_weights : Pandas Series

Index weights for all tickers at a period in time

scale : int

The penalty factor for weights the deviate from the index

Returns

-------

x : 1 dimensional Ndarray

The solution for x

"""

assert len(covariance\_returns.shape) == 2

assert len(index\_weights.shape) == 1

assert covariance\_returns.shape[0] == covariance\_returns.shape[1] == index\_weights.shape[0]

#TODO: Implement function

x = cvx.Variable(len(index\_weights))

print("x:",x)

objective = cvx.Minimize(cvx.quad\_form(x,covariance\_returns) + scale\*cvx.norm(x-index\_weights, p=2, axis=None))

print("objective:",objective)

# objective: minimize QuadForm

# (var0, [[0.143123 0.0216755 0.014273 ]

# [0.0216755 0.0401826 0.00663152]

# [0.014273 0.00663152 0.044963 ]])

# + 2.0 \* Pnorm(var0 + -[0.23623892 0.0125628 0.7511982 ], 2)

constrains = [x >= 0, sum(x) == 1]

print("constrains:",constrains)

# constrains: [NonPos(Expression(AFFINE, UNKNOWN, (3,))), Zero(Expression(AFFINE, UNKNOWN, ()))]

#(..) Combine objective & constrains

problem = cvx.Problem(objective, constrains)

print("problem:",problem)

# problem:

# minimize QuadForm(var0, [[0.143123 0.0216755 0.014273 ]

# [0.0216755 0.0401826 0.00663152]

# [0.014273 0.00663152 0.044963 ]])

# + 2.0 \* Pnorm(var0 + -[0.23623892 0.0125628 0.7511982 ], 2)

#(..) returns the minimum of the solution

result = problem.solve()

print("result:",result)

# result: 0.03868611091516786

print("x.value",x.value)

# x.value [0.23623897 0.01256285 0.75119817]

return x.value

project\_tests.test\_get\_optimal\_weights(get\_optimal\_weights)

——————————————

def rebalance\_portfolio(returns, index\_weights, shift\_size, chunk\_size):

"""

Get weights for each rebalancing of the portfolio.

Parameters

----------

returns : DataFrame

Returns for each ticker and date

index\_weights : DataFrame

Index weight for each ticker and date

shift\_size : int

The number of days between each rebalance

chunk\_size : int

The number of days to look in the past for rebalancing

Returns

-------

all\_rebalance\_weights : list of Ndarrays

The ETF weights for each point they are rebalanced

"""

assert returns.index.equals(index\_weights.index)

assert returns.columns.equals(index\_weights.columns)

assert shift\_size > 0

assert chunk\_size >= 0

#TODO: Implement function

print("returns:",returns)

returns = returns.fillna(0)

print("returns fillna:",returns)

all\_rebalance\_weights = []

for i in range(len(returns)):

if i >= shift\_size and i % shift\_size == 0 and i >= chunk\_size:

covariance\_returns = get\_covariance\_returns(returns.iloc[i-chunk\_size:i])

rebalance\_weights = get\_optimal\_weights(covariance\_returns, index\_weights.iloc[i-1], scale=2.0)

all\_rebalance\_weights.append(rebalance\_weights)

print("len(all\_rebalance\_weights):", len(all\_rebalance\_weights) )

# len(all\_rebalance\_weights): 4

print("all\_rebalance\_weights:",all\_rebalance\_weights)

# all\_rebalance\_weights: [array([0.29341237, 0.41378419, 0.29280344]), array([0.29654088, 0.40731481, 0.29614432]), array([0.29868214, 0.40308791, 0.29822995]), array([0.30100044, 0.39839644, 0.30060312])]

return all\_rebalance\_weights

project\_tests.test\_rebalance\_portfolio(rebalance\_portfolio)

—————————————————

def get\_portfolio\_turnover(all\_rebalance\_weights, shift\_size, rebalance\_count, n\_trading\_days\_in\_year=252):

"""

Calculage portfolio turnover.

Parameters

----------

all\_rebalance\_weights : list of Ndarrays

The ETF weights for each point they are rebalanced

shift\_size : int

The number of days between each rebalance

rebalance\_count : int

Number of times the portfolio was rebalanced

n\_trading\_days\_in\_year: int

Number of trading days in a year

Returns

-------

portfolio\_turnover : float

The portfolio turnover

"""

assert shift\_size > 0

assert rebalance\_count > 0

#TODO: Implement function

# n\_trading\_days\_in\_year

all\_rebalance\_weights = np.array(all\_rebalance\_weights)

print("all\_rebalance\_weights:", all\_rebalance\_weights )

# [[1.22050335e-04 3.01991574e-04 9.99575958e-01]

# [1.30570982e-05 8.11299880e-06 9.99978830e-01]

# [3.91748175e-01 5.60768785e-01 4.74830401e-02]]

sum\_total\_turnover = np.sum( abs(np.diff(all\_rebalance\_weights, axis=0)) )

print("sum\_total\_turnover:", sum\_total\_turnover )

#sum\_total\_turnover: 1.9057973231726242

number\_of\_rebalance\_events\_per\_year = n\_trading\_days\_in\_year / shift\_size

print("number\_of\_rebalance\_events\_per\_year:",number\_of\_rebalance\_events\_per\_year)

#number\_of\_rebalance\_events\_per\_year: 84.0

portfolio\_turnover = (sum\_total\_turnover / rebalance\_count) \* number\_of\_rebalance\_events\_per\_year

print("portfolio\_turnover:",portfolio\_turnover)

#portfolio\_turnover: 80.04348757325022

return portfolio\_turnover

project\_tests.test\_get\_portfolio\_turnover(get\_portfolio\_turnover)