Allopy Documentation

Release 0.4.7+1.g620c5ad.dirty

Daniel Bok

Jun 26, 2020

CONTENTS:

The Allopy toolbox contains methods for financial risk modelling and portfolio optimization.

Primarily, it has routines for

- Non-Linear and Linear Optimization
- Basic Financial Metrics (such as expected returns or CVaR calculations)

CHAPTER

ONE

GETTING STARTED

1.1 Python Support

Only Python 3.6 and above are supported. We recommend using the the Anaconda distribution as it bundles all the required software nicely for you.

Otherwise, you'll have to manage your environment setup (i.e. C-compiler setup and others) yourself if you choose to use pip.

You can download the Anaconda or the Miniconda distribution to get started. Miniconda is more bare-bones (smaller) and is thus faster to download and setup.

1.2 Installing the Packages

You can install the packages via `conda` or `pypi`. If installing via `conda`, make sure you have added the **conda-forge** channel. The details to do so are listed in the *Configuring Conda* section.

```
# conda
conda install -c danielbok allopy

# pip
pip install allopy
```

1.3 Configuring Conda

We require packages from both the `conda-forge` and `danielbok` channels. Before anything, open your command prompt and type this command in:

```
conda config --prepend channels conda-forge --append channels danielbok
```

This command places the **conda-forge** channel to the top of list while the **danielbok** channel will be placed at the bottom. It means that whenever you install packages, `conda` will first look for the package from **conda-forge**. If it can't find the package, it will move down the list to find the package in the other channels. Once it finds the package, it will install it. Otherwise, it will throw an error.

You may get an error message that reads

```
'conda' is not recognized as an internal or external command, operable program or␣
↪batch file.
```

In this case, it means that you have not added conda to your path. What you need to do is find the folder you installed the Miniconda or Anaconda package and add them to path.

Assuming you're using a Windows machine and have installed Miniconda to the folder `C:\\Miniconda3\\`, there are 2 ways to add `conda` to your path.

1.3.1 Method 1

1. In the **Start Menu**, search for **edit environment variables for your account**
2. In the top prompt titled **User variables for <NTID>**, search for **PATH**
3. Double click **PATH**
4. In the **Variable Value** section, go to the end and add the following line `;C:\Miniconda3\;C:\Miniconda3\condabin`.
5. Ensure that you have **added** and **not replaced**!!
6. Click **Okay** to everything

1.3.2 Method 2

The second way is to run the following line in your command prompt. However this is not recommended.

```
setx PATH "C:\Miniconda3\;C:\Miniconda3\condabin;%PATH%"
```

1.4 Using Environments

A tutorial on how to manage your `conda` environment can be found here.

It is best practice to start your project in a new environment.

CHAPTER

TWO

EXAMPLES

A listing of the various ways we can use `Allopy`. You should able to get the gist of what's going on here. For more in depth details, check out the API documentation.

2.1 Adding Uncertainty Penalty

This tutorial will guide you on how to add the uncertainty penalty to the `PortfolioOptimizer` classes. As of writing the two optimizer class that supports penalty functions are `PortfolioOptimizer` and `ActivePortfolioOptimizer`.

To start off, let's load in all the required packages.

[1]:
```
from muarch.funcs import get_annualized_sd

from allopy import OptData, PortfolioOptimizer
from allopy.datasets import load_monte_carlo
from allopy.penalty import UncertaintyPenalty

import numpy as np


np.set_printoptions(linewidth=200)
```

Let's load in a sample dataset.

[2]:
```
data = OptData(load_monte_carlo(), 'q')
data.shape
```

[2]:
```
(80, 10000, 9)
```

We'll only use the first 7 asset classes. For them we will also set the lower and upper bounds respectively.

[3]:
```
opt = PortfolioOptimizer(data.take_assets(7))
opt.set_bounds(
    0,  # lower bounds all set  to 0
    [0.4, 0.3, 0.13, 0.11, 0.25, 0.04, 0.05]  # custom upper bounds
)
```

[3]:
```
<allopy.optimize.portfolio.portfolio.optimizer.PortfolioOptimizer at 0x263fc627208>
```

For simplicity, we will use the current volatility as the uncertainty vector. But remember, you can set a uncertainty matrix for the penalty class.

[4]:
```
vol = get_annualized_sd(data, 'quarter')
vol.round(4)
```

[4]:
```
array([0.1849, 0.2648, 0.2026, 0.0961, 0.078 , 0.1403, 0.0428, 0.0613, 0.185 ])
```

[5]:
```
penalty = UncertaintyPenalty(vol, lambda_=1.0)
print(penalty)
```

```
UncertaintyPenalty(
    lambda=1.0,
    uncertainty=[[0.1849, 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
↪   ],
       [0.    , 0.2648, 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ],
       [0.    , 0.    , 0.2026, 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ],
       [0.    , 0.    , 0.    , 0.0961, 0.    , 0.    , 0.    , 0.    , 0.    ],
       [0.    , 0.    , 0.    , 0.    , 0.078 , 0.    , 0.    , 0.    , 0.    ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.1403, 0.    , 0.    , 0.    ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.0428, 0.    , 0.    ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.0613, 0.    ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.185 ]],
    method=direct
)
```

Note that we have transformed the vector to a diagonal matrix. Now, let's add the penalty to the optimizer.

[6]:
```
try:
    opt.penalty = penalty
except AssertionError as e:
    print("An error has occurred:", '. '.join(e.args))
```

```
An error has occurred: dimension of the penalty does not match the data
```

Oops, why is there an error? It's because the penalty vector has a dimension of 9, which means that there should be 9 asset classes. However, at the top, we have chosen to use only the first 7 asset classes. To fix that, we must initialize the `UncertaintyPenalty` correctly. Let's do it again.

[7]:
```
penalty = UncertaintyPenalty(vol[:7], lambda_=1.0)
print(penalty)
```

```
UncertaintyPenalty(
    lambda=1.0,
    uncertainty=[[0.1849, 0.    , 0.    , 0.    , 0.    , 0.    ],
       [0.    , 0.2648, 0.    , 0.    , 0.    , 0.    ],
       [0.    , 0.    , 0.2026, 0.    , 0.    , 0.    ],
       [0.    , 0.    , 0.    , 0.0961, 0.    , 0.    ],
       [0.    , 0.    , 0.    , 0.    , 0.078 , 0.    ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.1403],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.0428]],
    method=direct
)
```

[8]:
```
opt.penalty = penalty
optimal_weights = opt.maximize_sharpe_ratio()
optimal_weights.round(4)
```

[8]:
```
array([0.2678, 0.1522, 0.13  , 0.11  , 0.25  , 0.04  , 0.05  ])
```

That's it, we're done. To remove any penalty you have set by accident, set it to `None`.

[9]:
```
opt.penalty = None
```

2.2 Basic Introduction to BaseOptimizer

In this tutorial, we show how to use the `BaseOptimizer` to optimize a hypothetical portfolio.

In this portfolio, we have 2 assets with different expected returns and volatility. Our task is to find the optimal weights subject to some risk constraints. Let's assume Asset A has an annual return of 12% with volatility at 4%, Asset B has an historical annual returns of 4% with volatility at 0.14% and both of them has a covariance of 0.2%. We start off by simulating **500** instances of their one-year ahead returns.

[1]:
```
import numpy as np
from scipy.stats import multivariate_normal as mvn

assets_mean = [0.12, 0.04]  # asset mean returns vector
assets_std = [
    [0.04, 0.002],
    [0.002, 0.0014]
]  # asset covariance matrix

# hypothetical returns series
returns = mvn.rvs(mean=assets_mean, cov=assets_std, size=500, random_state=88)
```

Now that we have the returns series, our job is to optimize the portfolio where our objective is to maximize the expected returns subject to certain risk budgets. Let's assume we are only comfortable with taking a volatility of at most 10%.

Our problem is thus given by

Looks complicated but let's simplify it with some vector notations. Allowing r_n to be the returns at trial n after accounting for the weights (w), μ to be the mean return across trials, the problem can be specified as

[2]:
```
from allopy.optimize import BaseOptimizer


def objective(w):
    return (returns @ w).mean()


def constraint(w):
    # we need to convert the constraint to standard form. So c(w) - K <= 0
    return (returns @ w).std() - 0.1


prob = BaseOptimizer(2)  # initialize the optimizer with 2 asset classes

# set the objective function
prob.set_max_objective(objective)

# set the inequality constraint function
```

(continues on next page)

(continued from previous page)

```
prob.add_inequality_constraint(constraint)

# set lower and upper bounds to 0 and 1 for all free variables (weights)
prob.set_bounds(0, 1)

# set equality matrix constraint, Ax = b. Weights sum to 1
prob.add_equality_matrix_constraint([[1, 1]], [1])

sol = prob.optimize()
print('Solution: ', sol)
```

```
Solution:  [0.47209577 0.52790423]
```

Don't be alarmed if you noticed the print outs, `Setting gradient for ...`. By default, you actually have to set the gradient and possibly the hessian for your function. In fact, you could if you wanted to. This will give you more control over the optimization program. However, understanding that it may be tedious, we have opted to set the gradient for you if you didn't do so.

This assumes you're using a gradient based optimizer. In case you did, the default gradient is set using a second-order numerical derivative.

Also notice the solution given above. This means that the optimizer has successfully found the solution. To get even more information, we can use the `.summary()` method as seen below.

[3]:
```
prob.summary()
```

[3]:
```
                                Portfolio Optimizer

================================================================================



Algorithm: Sequential Quadratic Programming (SQP) (local, derivative)

--------------------------------------------------------------------------------

Optimizer Setup                             Options
objective                    maximize     xtol_abs                          1e-06
n_var                               2     xtol_rel                            0.0
n_eq_con                            1     ftol_abs                          1e-06
n_ineq_con                          1     ftol_rel                            0.0
                                          max_eval                         100000
                                          stop_val                            inf

--------------------------------------------------------------------------------

   Lower Bounds   Upper Bounds
       0.000000       1.000000
       0.000000       1.000000



Results

--------------------------------------------------------------------------------

Program found a solution
Solution: [0.4720957710945860.5279042289054587]
```

(continues on next page)

(continued from previous page)

```
The following inequality constraints were tight:
  1: constraint
```

2.3 Simulation and Optimization

We'll go through the entire simulation and optimization process in this tutorial. The general procedure on optimizing the portfolio is as follows:

1. Determine the asset classes (and thus the dataset used)
2. Simulate the returns
3. Run an optimization program on the simulated cube (tensor)

More details on each section will be covered.

2.3.1 Getting Started

To start off, install the required packages by running the following commands in your command prompt.

Feel free to name `my_env` with whatever you like.

```
conda config --prepend channels conda-forge
conda config --append channels bashtage
conda config --append channels danielbok

conda create -y -n my_env python=3.7 muarch copulae allopy pandas

# you may need to init if you're using conda for the first time
conda init --all

conda activate my_env
```

Remember to activate the environment.

2.3.2 Simulation

For the simuluation The simulation follows the following procedure:

1. Load the **returns** dataset of the asset classes you want to simulate
2. For each asset class, specify the AR-GARCH model
3. Fit the AR-GARCH models with the **log-returns** data
4. Fit the standardized residuals of the AR-GARCH models to a Student Copula
5. Overwrite the correlation matrix of the Student Copula with the **log-returns correlation**
6. Simulate future returns using the AR-GARCH models with distributions "inversed" from the copula
7. Truncate then calibrate the first 2 moments (returns and vol) of the cube

Exploring the Data

Let's start by loading and exploring the sample policy index dataset. These are the monthly returns for the 7 assets we will simulate. The data ranges from 01 Jan 1985 to 01 Oct 2017.

[1]:
```
import numpy as np
from allopy.datasets import load_index

returns = load_index()
log_returns = (returns + 1).apply(np.log)
_, num_assets = log_returns.shape

log_returns.head()
```

[1]:
```
                CASH        NB      EILB      DMEQ      EMEQ        PE  \
DATE
1985-01-01  0.004109  0.008796  0.028741  0.058068 -0.014185  0.067508
1985-01-02  0.024052 -0.002144 -0.052592  0.008257 -0.041216  0.007606
1985-01-03 -0.030244  0.006433 -0.000043 -0.013792  0.017271  0.028742
1985-01-04  0.002630  0.005479  0.020288 -0.083561 -0.005833 -0.016684
1985-01-05  0.007554  0.042372  0.046989  0.052969 -0.008937  0.071804

                  RE
DATE
1985-01-01 -0.000949
1985-01-02 -0.018895
1985-01-03 -0.014230
1985-01-04 -0.022006
1985-01-05  0.035114
```

[2]:
```
log_returns.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 394 entries, 1985-01-01 to 2017-01-10
Data columns (total 7 columns):
CASH    394 non-null float64
NB      394 non-null float64
EILB    394 non-null float64
DMEQ    394 non-null float64
EMEQ    394 non-null float64
PE      394 non-null float64
RE      394 non-null float64
dtypes: float64(7)
memory usage: 24.6 KB
```

Forming the Models

By right, we are supposed to form a distinct AR-GARCH model for each asset class. However, we'll take some short cuts by assuming every model is an AR(1)-GARCH(1,1)-Skew_T model with the exception of **CASH**. This exception is meant to serve as an example of how to change each model separately.

[3]:
```
from muarch import MUArch, UArch

# The settings defined will set the default for all models.
arch_models = MUArch(num_assets,
                     mean='ARX',
```

(continues on next page)

(continued from previous page)

```
                    lags=1,
                    vol='GARCH',
                    p=1,
                    q=1,
                    power=2.0,
                    dist='skewt',
                    scale=100)

# You can specify each model separately. We'll use CASH as an example
# CASH is the model in the MUArch object
arch_models[0] = UArch(mean='CONSTANT',
                       lags=0,
                       vol='CONSTANT',
                       dist='skewt',
                       scale=100)
```

Fitting the Model

We can call the fit function and subsequently check the quality of each fit through the `.summary()` method.

[4]:
```
arch_models.fit(log_returns)
arch_models.summary()
```

[4]:
```
<class 'muarch.summary.SummaryList'>
"""
CASH

                     Constant Mean - Constant Variance Model Results
==============================================================================================
Dep. Variable:                                   y   R-squared:                          -0.
↪000
Mean Model:                          Constant Mean   Adj. R-squared:                     -0.
↪000
Vol Model:                       Constant Variance   Log-Likelihood:                  -734.
↪606
Distribution:      Standardized Skew Student's t   AIC:                                 ⌟
↪1477.21
Method:                         Maximum Likelihood   BIC:                                 ⌟
↪1493.12
                                                     No. Observations:                        ⌟
↪394
Date:                             Sun, Jan 12 2020   Df Residuals:                            ⌟ ⌟
↪390
Time:                                    11:54:38   Df Model:                                ⌟
↪  4
                                 Mean Model
============================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
----------------------------------------------------------------------------
mu         -4.6322e-03  8.148e-02 -5.685e-02      0.955 [ -0.164,  0.155]
                           Volatility Model
============================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
----------------------------------------------------------------------------
sigma2         2.6251      0.274      9.575  1.019e-21 [  2.088,  3.162]
                              Distribution
```

(continues on next page)

(continued from previous page)

```
==============================================================================
                  coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------------
nu               5.4964      1.387      3.964  7.368e-05 [  2.779,  8.214]
lambda     -6.7665e-04  6.797e-02 -9.955e-03      0.992 [ -0.134,  0.133]
==============================================================================

Covariance estimator: robust

********************************************************************************************************

NB

                              AR - GARCH Model Results
==========================================================================================
Dep. Variable:                                y   R-squared:                          0.
↪009
Mean Model:                                  AR   Adj. R-squared:                     0.
↪006
Vol Model:                                GARCH   Log-Likelihood:                -651.
↪331
Distribution:      Standardized Skew Student's t   AIC:                                 
↪1316.66
Method:                     Maximum Likelihood   BIC:                                 
↪1344.48
                                                 No. Observations:                    
↪393
Date:                        Sun, Jan 12 2020   Df Residuals:                         
↪386
Time:                                11:54:38   Df Model:                            
↪  7
                               Mean Model
============================================================================
                coef    std err          t      P>|t|     95.0% Conf. Int.
----------------------------------------------------------------------------
Const          0.1950  7.099e-02      2.746  6.029e-03  [5.582e-02,  0.334]
y[1]           0.0896  5.778e-02      1.550      0.121 [-2.368e-02,  0.203]
                           Volatility Model
============================================================================
                coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0415  8.911e-02      0.465      0.642      [ -0.133,  0.216]
alpha[1]       0.0000  4.415e-02      0.000      1.000 [-8.653e-02,8.653e-02]
beta[1]        0.9723      0.100      9.698  3.080e-22      [  0.776,  1.169]
                            Distribution
============================================================================
                coef    std err          t      P>|t|  95.0% Conf. Int.
----------------------------------------------------------------------------
nu             8.2346      3.400      2.422  1.543e-02 [  1.571, 14.898]
lambda        -0.0813      0.105     -0.774      0.439 [ -0.287,  0.125]
==============================================================================

Covariance estimator: robust

********************************************************************************************************

EILB
```

(continues on next page)

(continued from previous page)

```
                              AR - GARCH Model Results
==============================================================================================
Dep. Variable:                                  y   R-squared:                           0.
↪002
Mean Model:                                    AR   Adj. R-squared:                      -0.
↪001
Vol Model:                                 GARCH   Log-Likelihood:                    -
↪1024.23
Distribution:      Standardized Skew Student's t   AIC:                                 ⌊
↪2062.46
Method:                        Maximum Likelihood   BIC:                                 ⌊
↪2090.27
                                                    No. Observations:                      ⌊
↪393
Date:                           Sun, Jan 12 2020   Df Residuals:                          ⌊
↪386
Time:                                  11:54:39   Df Model:                              ⌊
↪  7
                                 Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
Const          0.7238      0.177      4.081  4.477e-05    [  0.376,  1.071]
y[1]           0.0202  5.402e-02      0.375      0.708 [-8.565e-02,  0.126]
                            Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          7.2805      1.601      4.549  5.402e-06    [  4.143, 10.418]
alpha[1]       0.1777      0.109      1.634      0.102 [-3.549e-02,  0.391]
beta[1]        0.1816      0.153      1.187      0.235    [ -0.118,  0.481]
                              Distribution
============================================================================
                 coef    std err          t      P>|t|    95.0% Conf. Int.
----------------------------------------------------------------------------
nu             8.8789      3.717      2.389  1.691e-02   [  1.594, 16.164]
lambda        -0.0437  6.861e-02     -0.636      0.524 [ -0.178,9.080e-02]
============================================================================

Covariance estimator: robust

**********************************************************************************************

DMEQ

                              AR - GARCH Model Results
==============================================================================================
Dep. Variable:                                  y   R-squared:                          -0.
↪004
Mean Model:                                    AR   Adj. R-squared:                      -0.
↪006
Vol Model:                                 GARCH   Log-Likelihood:                    -
↪1142.18
Distribution:      Standardized Skew Student's t   AIC:                                 ⌊
↪2298.36
Method:                        Maximum Likelihood   BIC:                                 ⌊
↪2326.18
```

(continues on next page)

(continued from previous page)

```
                                                  No. Observations:                 ↵
↪393
Date:                          Sun, Jan 12 2020   Df Residuals:                     ↵
↪386
Time:                                 11:54:39   Df Model:                         ↵
↪  7
                              Mean Model
================================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
--------------------------------------------------------------------------------
Const          0.3412      0.219      1.558      0.119 [-8.804e-02,  0.770]
y[1]          -0.0145  5.746e-02     -0.252      0.801  [ -0.127,9.814e-02]
                         Volatility Model
================================================================================
                 coef    std err          t      P>|t|    95.0% Conf. Int.
--------------------------------------------------------------------------------
omega          1.3369      0.568      2.354  1.860e-02   [  0.224,  2.450]
alpha[1]       0.0762  3.183e-02      2.393  1.672e-02 [1.378e-02,  0.139]
beta[1]        0.8594  4.086e-02     21.032  3.314e-98   [  0.779,  0.939]
                           Distribution
================================================================================
                 coef    std err          t      P>|t|     95.0% Conf. Int.
--------------------------------------------------------------------------------
nu             9.6054      4.727      2.032  4.215e-02    [  0.341, 18.870]
lambda        -0.2163  7.351e-02     -2.943  3.255e-03 [ -0.360,-7.224e-02]
================================================================================

Covariance estimator: robust

********************************************************************************************************************

EMEQ

                            AR - GARCH Model Results
================================================================================================
Dep. Variable:                                 y   R-squared:                        0.
↪019
Mean Model:                                   AR   Adj. R-squared:                   0.
↪016
Vol Model:                                 GARCH   Log-Likelihood:                -
↪1309.97
Distribution:      Standardized Skew Student's t   AIC:                            ↵ ↵
↪2633.94
Method:                        Maximum Likelihood   BIC:                            ↵
↪2661.75
                                                  No. Observations:                 ↵
↪393
Date:                          Sun, Jan 12 2020   Df Residuals:                     ↵
↪386
Time:                                 11:54:39   Df Model:                         ↵
↪  7
                              Mean Model
================================================================================
                 coef    std err          t      P>|t|    95.0% Conf. Int.
--------------------------------------------------------------------------------
Const          0.9325      0.357      2.609  9.080e-03   [  0.232,  1.633]
y[1]           0.1289  4.625e-02      2.787  5.317e-03 [3.826e-02,  0.220]
```

(continues on next page)

(continued from previous page)

```
                              Volatility Model
==============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
omega          1.6863      1.348      1.251      0.211      [ -0.956,  4.329]
alpha[1]       0.0322  1.742e-02      1.846  6.490e-02 [-1.985e-03,6.629e-02]
beta[1]        0.9367  3.036e-02     30.856 4.717e-209      [  0.877,  0.996]
                              Distribution
==============================================================================
                 coef    std err          t      P>|t|     95.0% Conf. Int.
------------------------------------------------------------------------------
nu             6.0618      1.870      3.242  1.186e-03    [  2.397,  9.726]
lambda        -0.2297  7.166e-02     -3.205  1.349e-03 [ -0.370,-8.925e-02]
==============================================================================

Covariance estimator: robust

********************************************************************************************************

PE

                               AR - GARCH Model Results
==========================================================================================
Dep. Variable:                                  y   R-squared:                         0.
↪028
Mean Model:                                    AR   Adj. R-squared:                    0.
↪026
Vol Model:                                 GARCH   Log-Likelihood:                -
↪1180.08
Distribution:      Standardized Skew Student's t   AIC:                             
↪2374.16
Method:                      Maximum Likelihood   BIC:                             
↪2401.98
                                                  No. Observations:                  
↪393
Date:                          Sun, Jan 12 2020   Df Residuals:                      
↪386
Time:                                 11:54:39   Df Model:                          
↪  7
                               Mean Model
==============================================================================
                 coef    std err          t      P>|t|     95.0% Conf. Int.
------------------------------------------------------------------------------
Const          0.4321      0.246      1.760  7.845e-02 [-4.916e-02,  0.913]
y[1]           0.1118  5.112e-02      2.187  2.872e-02  [1.162e-02,  0.212]
                              Volatility Model
==============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
omega          0.9024      0.581      1.552      0.121     [ -0.237,  2.042]
alpha[1]       0.0562  1.982e-02      2.834  4.595e-03 [1.733e-02,9.502e-02]
beta[1]        0.9098  2.973e-02     30.603 1.105e-205     [  0.852,  0.968]
                              Distribution
==============================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------------
nu             8.5291      3.344      2.550  1.076e-02 [  1.974, 15.084]
```

(continues on next page)

(continued from previous page)

```
lambda        -0.2765  8.300e-02     -3.332  8.626e-04 [ -0.439, -0.114]
==============================================================================

Covariance estimator: robust

********************************************************************************************************************

RE

                               AR - GARCH Model Results
==============================================================================================
Dep. Variable:                                   y   R-squared:                          0.
↪010
Mean Model:                                     AR   Adj. R-squared:                     0.
↪007
Vol Model:                                  GARCH   Log-Likelihood:                 -891.
↪715
Distribution:      Standardized Skew Student's t   AIC:                                 ⌊ ⌊
↪1797.43
Method:                         Maximum Likelihood   BIC:                                 ⌊
↪1825.25
                                                     No. Observations:                      ⌊
↪393
Date:                            Sun, Jan 12 2020   Df Residuals:                          ⌊ ⌊
↪386
Time:                                    11:54:39   Df Model:                              ⌊
↪  7
                                 Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
Const          0.1584      0.127      1.251      0.211 [-8.968e-02,  0.407]
y[1]           0.0813  5.794e-02      1.404      0.160 [-3.221e-02,  0.195]
                            Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          1.0374      0.783      1.325      0.185    [ -0.497,  2.572]
alpha[1]       0.0937  4.863e-02      1.926  5.409e-02 [-1.645e-03,  0.189]
beta[1]        0.7221      0.164      4.391  1.129e-05    [  0.400,  1.044]
                              Distribution
============================================================================
                  coef    std err          t      P>|t|  95.0% Conf. Int.
----------------------------------------------------------------------------
nu             15.0314      8.833      1.702  8.880e-02 [ -2.281, 32.344]
lambda     -6.9447e-03  8.766e-02 -7.922e-02      0.937 [ -0.179,  0.165]
============================================================================

Covariance estimator: robust
"""
```

Fitting the Copula

The next thing we need to do is to fit the copula with the standardized residuals of the arch models' fits. We can also check the fit of the copula with the `.summary()` method.

[5]:
```
from copulae import TCopula

residuals = arch_models.residuals()  # defaults to standardized residuals
cop = TCopula(num_assets)
cop.fit(residuals)
cop.summary()
```

[5]:
```
Student Copula Summary
================================================================================
Student Copula with 7 dimensions


Parameters
--------------------------------------------------------------------------------
 Degree of Freedom  : 23.07912545920342

 Correlation Matrix
 1.000000  0.132323 -0.383796  0.088988  0.095665  0.069652  0.233252
 0.132323  1.000000  0.399767 -0.019967 -0.072441 -0.048850  0.178103
-0.383796  0.399767  1.000000  0.014854  0.038000  0.070871  0.015642
 0.088988 -0.019967  0.014854  1.000000  0.506254  0.699188  0.467597
 0.095665 -0.072441  0.038000  0.506254  1.000000  0.579042  0.317568
 0.069652 -0.048850  0.070871  0.699188  0.579042  1.000000  0.442605
 0.233252  0.178103  0.015642  0.467597  0.317568  0.442605  1.000000



                                  Fit Summary
================================================================================
Log. Likelihood      : -388.78612323077846
Variance Estimate    : Not Implemented Yet
Method               : Maximum pseudo-likelihood
Data Points          : 393

Optimization Setup
--------------------------------------------------------------------------------
    bounds           : [(0.0, inf), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0),␣
↪(-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0,
↪ 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0),
↪ (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0)]
    options          : {'maxiter': 20000, 'ftol': 1e-06, 'iprint': 1, 'disp': False,
↪'eps': 1.5e-08}
    method           : SLSQP

Results
--------------------------------------------------------------------------------
    x                : [ 2.30791255e+01  1.32323371e-01 -3.83796066e-01  8.89879525e-02
  9.56646977e-02  6.96515171e-02  2.33251735e-01  3.99767177e-01
 -1.99674055e-02 -7.24414193e-02 -4.88504786e-02  1.78102666e-01
  1.48544670e-02  3.79998706e-02  7.08714854e-02  1.56419634e-02
  5.06253610e-01  6.99188239e-01  4.67596725e-01  5.79042392e-01
  3.17568429e-01  4.42605381e-01]
    fun              : -388.78612323077846
```

(continues on next page)

(continued from previous page)

```
    jac             : [-0.00053054 -0.03081292  0.02364686  0.04192392 -0.02807307 -0.
↪03360583
  0.00718501 -0.03806235 -0.05757859  0.0176783   0.03008912 -0.00658247
  0.07304    -0.01542351 -0.05668426  0.00757912 -0.00902673  0.01783368
  0.01313841 -0.02499595 -0.02411677  0.02380602]
    nit             : 35
    nfev            : 907
    njev            : 35
    status          : 0
    message         : Optimization terminated successfully.
    success         : True
```

All elliptical copula comes with a correlation matrix. In this case, we would like to overwrite the correlation matrix of the `TCopula` with the correlation matrix of the log returns.

[6]:
```
np.set_printoptions(linewidth=160)
cop.sigma
```

[6]:
```
array([[ 1.        ,  0.13232337, -0.38379607,  0.08898795,  0.0956647 ,  0.06965152,␣
↪ 0.23325173],
       [ 0.13232337,  1.        ,  0.39976718, -0.01996741, -0.07244142, -0.04885048,␣
↪ 0.17810267],
       [-0.38379607,  0.39976718,  1.        ,  0.01485447,  0.03799987,  0.07087149,␣
↪ 0.01564196],
       [ 0.08898795, -0.01996741,  0.01485447,  1.        ,  0.50625361,  0.69918824,␣
↪ 0.46759673],
       [ 0.0956647 , -0.07244142,  0.03799987,  0.50625361,  1.        ,  0.57904239,␣
↪ 0.31756843],
       [ 0.06965152, -0.04885048,  0.07087149,  0.69918824,  0.57904239,  1.        ,␣
↪ 0.44260538],
       [ 0.23325173,  0.17810267,  0.01564196,  0.46759673,  0.31756843,  0.44260538,␣
↪ 1.        ]])
```

[7]:
```
# this is how you overwrite the correlation matrix

cop[:] = log_returns.corr()
cop.sigma
```

[7]:
```
array([[ 1.        ,  0.13569432, -0.35933974,  0.10954112,  0.1281724 ,  0.09676939,␣
↪ 0.24841052],
       [ 0.13569432,  1.        ,  0.34423657, -0.01144135, -0.11482461, -0.06096223,␣
↪ 0.17947296],
       [-0.35933974,  0.34423657,  1.        ,  0.0869657 ,  0.05494202,  0.11039716,␣
↪ 0.03943829],
       [ 0.10954112, -0.01144135,  0.0869657 ,  1.        ,  0.59790511,  0.76230971,␣
↪ 0.49290886],
       [ 0.1281724 , -0.11482461,  0.05494202,  0.59790511,  1.        ,  0.67644088,␣
↪ 0.3457239 ],
       [ 0.09676939, -0.06096223,  0.11039716,  0.76230971,  0.67644088,  1.        ,␣
↪ 0.43281785],
       [ 0.24841052,  0.17947296,  0.03943829,  0.49290886,  0.3457239 ,  0.43281785,␣
↪ 1.        ]])
```

Simulating Future returns

Now that we have the copula to generate a dependency structure for each of the univariate GARCH models, we can start simulating future returns. The simulated returns data will be a 3-dimensional cube with axis time, trials and asset class respectively.

For 10000 trials and 120 periods (10 years), this will usually take some time, so give it a while. For demonstration purposes, we'll be simulating a (120, 1000, 7) cube.

Note that the order is very important in the subsequent optimization procedure, so don't reshape it!

[8]:
```
simulated_log_returns = arch_models.simulate_mc(120, 1000, custom_dist=cop.random)
cube = np.exp(simulated_log_returns) - 1  # recover as returns, instead of log returns
```

Removing Outliers

For each asset class, we define outliers as a data point that is more than 6 standard deviations away from the asset class's mean. We replace these outliers with the asset class mean.

As an aside, there probably is a better approach for multi-variate outlier detection and replacement

[9]:
```
from muarch.calibrate import truncate_outliers

cube = truncate_outliers(cube, 6)
```

Calibrating the Moments of the Cube

After removing outliers in the cube, the next step is to calibrate the first 2 moments of the cube according to the forward looking assumptions that you have. The listing below shows the function to calculate the annualized mean and volatility for the data cube.

[10]:
```
import pandas as pd


def show_moments(data):
    t, n, a = data.shape

    df = pd.DataFrame({
        'Asset': returns.columns,
        'Mean (%)': ((cube + 1).prod(0) ** 0.1).mean(0) - 1,
        'Vol (%)': ((cube + 1).reshape(t // 12, 12, n, a).prod(1) - 1).std(0).mean(0)
    })

    df.iloc[:, 1:] = df.iloc[:, 1:].apply(lambda x: round(100 * x, 4))
    return df
```

[11]:
```
# before calibration

show_moments(cube)
```

[11]:

	Asset	Mean (%)	Vol (%)
0	CASH	-0.0091	5.0712
1	NB	2.6050	4.2896
2	EILB	9.3393	11.9668
3	DMEQ	4.2744	14.8196

(continues on next page)

(continued from previous page)

```
4  EMEQ   14.3037  30.2308
5    PE    6.5389  19.0356
6    RE    2.1290   8.2400
```

For expedience, the calibration code is shown below.

```
from scipy import optimize as opt

def calibrate_data(data,
                   mean = None,
                   sd = None,
                   time_unit=12):
    data = data.copy()
    y, n, num_assets = data.shape
    y //= time_unit

    def set_target(target_values, typ, default):
        if target_values is None:
            return default, [0 if typ == 'mean' else 1] * num_assets

        best_guess = []
        target_values = np.asarray(target_values)
        assert num_assets == len(target_values) == len(
            default), "vector length must be equal to number of assets in data cube"
        for i, v in enumerate(target_values):
            if v is None:
                target_values[i] = default[i]
                best_guess.append(0 if typ == 'mean' else 1)
            else:
                best_guess.append(target_values[i] - default[i] if typ == 'mean' else
↪default[i] / target_values[i])
        return target_values, best_guess

    d = (data + 1).prod(0)
    default_mean = (np.sign(d) * np.abs(d) ** (1 / y)).mean(0) - 1
    default_vol = ((data + 1).reshape(y, time_unit, n, num_assets).prod(1) - 1).
↪std(1).mean(0)

    target_means, guess_mean = set_target(mean, 'mean', default_mean)
    target_vols, guess_vol = set_target(sd, 'vol', default_vol)

    sol = np.asarray([opt.root(
        fun=_asset_moments,
        x0=[gv, gm],
        args=(data[..., i], tv, tm, time_unit)
    ).x for i, tv, tm, gv, gm in zip(range(num_assets), target_vols, target_means,
↪guess_vol, guess_mean)])

    for i in range(num_assets):
        if sol[i, 0] < 0 or False:
            # negative vol adjustments would alter the correlation between asset
↪classes (flip signs)
            # in such instances, we fall back to using the a simple affine transform
↪where
            # R' = (tv/cv) * r  # adjust vol first
            # R' = (tm - mean(R'))  # adjust mean
```

(continues on next page)

(continued from previous page)

```
            # adjust vol
            tv = default_vol[i]
            cv = sd[i] if sd[i] is not None else tv
            data[..., i] *= tv / cv  # tv / cv

            # adjust mean
            tm, cm = target_means[i], data[..., i].mean()
            data[..., i] += (tm - cm)  # (tm - mean(R'))
        else:
            data[..., i] = data[..., i] * sol[i, 0] + sol[i, 1]
    return data


def _asset_moments(x: np.ndarray, asset: np.ndarray, t_vol: float, t_mean: float,␣
↪time_unit: int):
    t, n = asset.shape  # time step (month), trials
    y = t // time_unit

    calibrated = asset * x[0] + x[1]

    d = (calibrated + 1).prod(0)
    mean = (np.sign(d) * np.abs(d) ** (1 / y)).mean() - t_mean - 1
    vol = ((calibrated.reshape((y, time_unit, n)) + 1).prod(1) - 1).std(1).mean(0) -␣
↪t_vol

    return vol, mean
```

[12]:
```
# calibration

from muarch.calibrate import calibrate_data

target_mean = [0, 0.03, 0.06, 0.08, 0.11, 0.12, 0.05]
target_vol =  [0.055, 0.073, 0.14, 0.09, 0.22, 0.18, 0.1]

cube = calibrate_data(cube, target_mean, target_vol)

show_moments(cube)
```

[12]:

	Asset	Mean (%)	Vol (%)
0	CASH	0.0	5.0818
1	NB	3.0	6.7131
2	EILB	6.0	12.8631
3	DMEQ	8.0	8.1705
4	EMEQ	11.0	20.1085
5	PE	12.0	16.1869
6	RE	5.0	9.1215

2.3.3 Optimization

The data used for the optimization is the truncated and calibrated cube from the simulation step before. We have 2 optimizers, the `BaseOptimizer` and the `PortfolioOptimizer`.

The `PortfolioOptimizer` is built on top of the `BaseOptimizer`. So if you need to do common optimization operations, use the `PortfolioOptimizer` as it probably already has the routines. If you need to work from scratch on something custom, use the `BaseOptimizer`.

We'll explore the `BaseOptimizer` first to understand how to code out a Maximize Expected Returns subject to CVaR and Volatility constraints.

$$\max_{w_{1,\dots,A}} \frac{1}{N} \sum_{i}^{N} \prod_{i}^{T} \sum_{k}^{A} (\mathbf{C}_{i,j,k} w_k + 1)$$

$$s.t.$$

$$\text{Vol}(\mathbf{C}, \mathbf{w}) \leq 0.1$$

$$\text{CVaR}(\mathbf{C}, \mathbf{w}) \geq -0.33$$

$$0 \leq w_k \leq 1 \quad \forall \quad k \in 1, \dots, A$$

An important thing to note is that while we simulated monthly data, for optimization, we use quarterly data. Here's a fast way to convert monthly to quarterly data.

[13]:
```
t, n, _ = cube.shape

q_cube = (cube + 1).reshape(t // 3, 3, n, num_assets).prod(1) - 1
```

Let's start with the `BaseOptimizer`

[14]:
```
from allopy import BaseOptimizer

opt = BaseOptimizer(num_assets)

bounds = np.array([
    (0, 0.05),
    (0, 0.4),
    (0, 0.1),
    (0, 0.35),
    (0, 0.2),
    (0, 0.15),
    (0, 0.15),
])

opt.set_bounds(bounds[:, 0], bounds[:, 1])
max_vol = 0.1
max_cvar = -0.33

def obj_fun(w):
    # Maximize returns assuming there's rebalancing every quarter
    return ((q_cube @ w) + 1).prod(0).mean() - 1

def vol_c_fun(w):
    years = len(q_cube) // 4
    annual_data = (q_cube @ w + 1).reshape(years, 4, n).prod(1) - 1
    annual_vols = q_cube.std(0)
    return annual_vols.mean() - max_vol
```

(continues on next page)

(continued from previous page)

```
def cvar_c_fun(w):
    # cvar usually calculated for 3 years
    returns = ((q_cube[:3 * 12] @ w) + 1).prod(0) - 1
    cutoff = np.percentile(returns, 5)  # get the 5%
    cvar = returns[returns <= cutoff].mean()

    return max_cvar - cvar


opt.add_equality_constraint(lambda w: sum(w) - 1)
opt.set_max_objective(obj_fun)
opt.add_inequality_constraint(vol_c_fun)
opt.add_inequality_constraint(cvar_c_fun)

weights = opt.optimize()

bopt = pd.DataFrame({
    'Asset': returns.columns,
    'Weights': np.round(weights * 100, 2)
})

bopt
```

[14]:

	Asset	Weights
0	CASH	0.0
1	NB	5.0
2	EILB	10.0
3	DMEQ	35.0
4	EMEQ	20.0
5	PE	15.0
6	RE	15.0

Using the `PortfolioOptimizer`.

[15]:
```
from allopy import PortfolioOptimizer, OptData

main_data = OptData(q_cube, time_unit='quarterly')
opt = PortfolioOptimizer(main_data, cvar_data=main_data[:12])
opt.set_bounds(bounds[:, 0], bounds[:, 1])
weights = opt.maximize_returns(max_vol=max_vol, max_cvar=max_cvar)

aopt = pd.DataFrame({
    'Asset': returns.columns,
    'Weights': np.round(weights * 100, 2)
})

aopt
```

[15]:

	Asset	Weights
0	CASH	0.00
1	NB	5.34
2	EILB	10.00
3	DMEQ	35.00
4	EMEQ	20.00
5	PE	15.00
6	RE	14.66

2.4 Regret Optimization

Regret optimization is a technique to optimize a portfolio given that one is uncertain about future prospects. Since one must eventually settle for a single portfolio out of all the possible scenarios, the best portfolio in this case is determine by the one that will have the *least* regret. This will be elaborated further

In general, this is a 2-stage optimization. The algorithm works as such,

1. Generate several possible scenarios in the future
    - For each scenario, give it a discrete probability of occurrence
    - These probabilities must sum to 1 across all scenarios!
2. In the first stage, derive the optimal weights for the portfolio for each scenario
    - If there are 5 scenarios, there will be 5 sets of these weights
    - Thus a model with 5 scenarios and 3 assets will yield a 5 x 3 matrix
3. In the second stage, solve for the minimal regret portfolio, these time using the previous sets optimal weights
    - You will receive the final set of of weights here (a 1 x 3 vector)

2.4.1 Regret

Regret is defined as the cost of choosing one portfolio (which is optimal for a scenario) when another different scenario occurs instead. Functionally, the simplest mathematical formuation is

$$D(R(w_s) - R(w_o))$$

where D is a distance function, R is the profit function, w_s is the optimal weights for scenario s and w_o is the "optimal" weights that was chosen. Thus to solve for the minimal regret function, the exact problem formulation is as listed below

$$\min_{w} \sum_{s}^{S} p_s D(R(w_s) - R(w))$$

where p_s is the probability of scenario s occurring.

Distance function

The distance function could be anything that makes sense. Some common examples include a linear function, absolute function or quadratic function. Different functions will penalise regret differently and lead to different outcomes. For example, if a portfolio that does not have a wide swings in terms of objectives is desired, a quadratic function will work better than a linear function.

$$\text{Assuming} \quad R(w) \geq 1 \quad \forall w \in W \tag{2.1}$$

$$[R(w_s) - R(w_o)]^2 \geq R(w_s) - R(w_o) \tag{2.2}$$

Linear Approximations

Suppose we have convex objective and constraints functions for the first stage, we alter the second stage optimization a little. The outcome will be similar but it will give another nice interpretation of the results which will be explained later. Ideally these functions should be **strictly linear**. However, in practice, the differences are usually negligible.

Suppose we want to maximize the returns of the portfolio subject to some CVaR constraints. For simplicity, the returns function will be R and CVaR constraint functions will be C. Thus our first stage optimization will be

For every single scenario s in S

$$\begin{aligned} &\max_{w_s} R(w_s) \\ &\text{subject to} \\ &C_s(w_s) \geq 0 \\ &\sum_i w_{s,i} = 1 \\ 0 \leq w \leq 1 \quad &\forall w \in w_s \end{aligned}$$

From this we would get

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{s,1} & w_{s,2} & \dots & w_{s,n} \end{bmatrix}$$

where s is the number of scenarios adn n is the number of assets. We would then tweak our second (Regret) optimization to

$$\begin{aligned} &\min_a \sum_s^S p_s D(R(w_s) - R(W \cdot a)) \\ &\text{subject to} \\ &\sum_s^S a_s = 1 \end{aligned}$$

The solution of the problem, a, will represent the proportion of importance that is taken from each scenario. Suppose there are 3 scenarios - **X**, **Y**, **Z** and that the final proportion derived is `[0.2, 0.3, 0.5]`. This means that **20%** of the weights are taken from **X**, **30%** from **Y** and **50%** from **Z**. In essence, it weights the importance of each scenario for the final outcome.

To get the final weights, simply do a dot product of $W \cdot a$.

2.4.2 Example

We will run through the Regret Optimization using both the `RegretOptimizer` and `PortfolioRegretOptimizer` classes. The `PortfolioRegretOptimizer` is a helper class that has several common built-in problems within itself. Underneath the hood, it uses the `RegretOptimizer` for the same operations. The `RegretOptimizer` is the more flexible tool that is useful for modelling more exotic scenarios.

[1]:
```
import numpy as np
from muarch.calibrate import calibrate_data

from allopy import OptData, RegretOptimizer
from allopy.datasets import load_monte_carlo
```

[2]:
```
# Generate different scenarios
num_assets = 7
num_scenarios = 4
scenario_probability = [0.57, 0.1, 0.14, 0.19]

main_adjustments = np.array([
    [ 0.0061,  0.0601,  0.0466,  0.0051, -0.0066, -0.0013, -0.0026],
    [ 0.0642,  0.0537,  0.0818,  0.0713,  0.0177,  0.0099,  0.0116],
    [-0.0219, -0.0381, -0.0059,  0.0242, -0.0153,  0.0164, -0.001 ],
    [-0.0333, -0.0617, -0.0405, -0.0251,  0.0084,  0.0054, -0.0035]
])

cvar_adjustments = np.array([
    [-0.0436,  0.0586, -0.0081,  0.0051, -0.0078,  0.0135, -0.0081],
    [ 0.0662,  0.079 ,  0.0896,  0.0501, -0.0265, -0.0572,  0.0025],
    [-0.1192, -0.1701, -0.1143,  0.0736, -0.0831,  0.0134, -0.0047],
    [-0.1728, -0.257 , -0.1933, -0.1432,  0.0342,  0.0079,  0.003 ]
])


def make_scenarios(adjustments, truncate=False):
    scenarios = []
    for adj in adjustments:
        data = OptData(load_monte_carlo()[..., :num_assets], 'quarterly')

        if truncate:  # cut for CVaR
            data = data.cut_by_horizon(3)

        scenarios.append(data.calibrate_data(adj))

    return scenarios

main_scenarios = make_scenarios(main_adjustments)
cvar_scenarios = make_scenarios(cvar_adjustments, True)
```

[3]:
```
# objective and constraint functions

def make_max_returns_obj_fun(cube: OptData):
    def obj_fun(w):
        return 1e2 * cube.expected_return(w, True)

    return obj_fun


def make_cvar_constraint_fun(cube: OptData, limit: float):
    def cvar_fun(w):
        return 1e3 * (limit - cube.cvar(w, True, 5.0))

    return cvar_fun


# limits and bounds
lb = [0, 0, 0.13, 0.11, 0, 0.05, 0.04]
ub = [1, 0.18, 0.13, 0.11, 1, 0.05, 0.04]

cvar_limit = [-0.34, -0.253, -0.501, -0.562]
```

[4]:
```
# optimization model formulation and execution
opt = RegretOptimizer(num_assets, num_scenarios, scenario_probability, sum_to_1=True)
opt.set_bounds(lb, ub)

obj_funcs = []
constraint_funcs = []
for m, c, limit in zip(main_scenarios, cvar_scenarios, cvar_limit):
    obj_funcs.append(make_max_returns_obj_fun(m))
    constraint_funcs.append(make_cvar_constraint_fun(c, limit))

opt.set_max_objective(obj_funcs)
opt.add_inequality_constraint(constraint_funcs)

final_weights = opt.optimize()
```

You can get the summary of the results. The first table show the optimal weight for each scenario. The second shows the proportion of each scenario and is only available when the approx option is set to `True`. The final table shows the *final* optimal weights.

[5]:
```
opt.summary()
```

[5]:
```
<class 'allopy.optimize.regret.summary.RegretSummary'>
"""
                 Regret Optimizer
=================================================
        Scenario_1 Scenario_2 Scenario_3 Scenario_4
-------------------------------------------------
Asset_1     0.2499     0.3495     0.1003     0.0000
Asset_2     0.1800     0.0946     0.0000     0.0000
Asset_3     0.1300     0.1300     0.1300     0.1300
Asset_4     0.1100     0.1100     0.1100     0.1100
Asset_5     0.2401     0.2259     0.5697     0.6700
Asset_6     0.0500     0.0500     0.0500     0.0500
Asset_7     0.0400     0.0400     0.0400     0.0400
-------------------------------------------------------------------------
                           Scenario                          Proportion (%)
-------------------------------------------------------------------------
0                         Scenario_1                               68.5900
1                         Scenario_2                                0.0000
2                         Scenario_3                                0.0000
3                         Scenario_4                               47.5900
-------------------------------------------------
                                          Weight
-------------------------------------------------
Asset_1                                   0.1714
Asset_2                                   0.1235
Asset_3                                   0.1510
Asset_4                                   0.1278
Asset_5                                   0.4835
Asset_6                                   0.0581
Asset_7                                   0.0465
=================================================
Optimization completed successfully
"""
```

The `PortfolioRegretOptimizer` contains a number of common optimization regimes with respect to regret optimization. We can apply the same optimization we did with the `maximize_returns()` method.

[6]:
```
from allopy import PortfolioRegretOptimizer

opt = PortfolioRegretOptimizer(main_scenarios,
                               cvar_scenarios,
                               scenario_probability,
                               rebalance=True,
                               sum_to_1=True,
                               time_unit="quarterly")

opt.set_bounds(lb, ub)
opt.maximize_returns(max_cvar=cvar_limit)
opt.summary()
```

[6]:
```
<class 'allopy.optimize.regret.summary.RegretSummary'>
"""
                  Regret Optimizer
=================================================
        Scenario_1 Scenario_2 Scenario_3 Scenario_4
-------------------------------------------------
Asset_1     0.2499     0.3494     0.1003     0.0000
Asset_2     0.1800     0.0947     0.0000     0.0000
Asset_3     0.1300     0.1300     0.1300     0.1300
Asset_4     0.1100     0.1100     0.1100     0.1100
Asset_5     0.2401     0.2259     0.5697     0.6700
Asset_6     0.0500     0.0500     0.0500     0.0500
Asset_7     0.0400     0.0400     0.0400     0.0400
-------------------------------------------------------------------------
                           Scenario                          Proportion (%)
-------------------------------------------------------------------------
0                         Scenario_1                               68.5900
1                         Scenario_2                                0.0000
2                         Scenario_3                                0.0000
3                         Scenario_4                               47.5900
-------------------------------------------------
                                          Weight
-------------------------------------------------
Asset_1                                   0.1714
Asset_2                                   0.1235
Asset_3                                   0.1510
Asset_4                                   0.1278
Asset_5                                   0.4835
Asset_6                                   0.0581
Asset_7                                   0.0465
=================================================
Optimization completed successfully
"""
```

CHAPTER

THREE

API

The optimizers are the classes responsible for finding the ideal weights. The underlying optimizer is built from the nlopt package. The optimizers are classified into 2 categories, **Deterministic** and **Discrete State Uncertainty**.

Deterministic optimizers are suitable for instances where the problem scenario is known. For example, if it is expected to only have one market scenario, then it is suitable to use the optimizers under this category.

Discrete State Uncertainty optimizers are suitable for instances where there are multiple problem scenarios and when a discrete probability can be assigned to each scenario. For instance, there could be 3 scenarios that would happen in the future. Baseline at 50%, Upside at 30% and Downside at 20%. In this case, this class of optimizers will be most suitable to model the optimization problem.

Presently, **Continuous State Uncertainty optimizers** are not implemented in the package.

3.1 Base Optimizer

The `BaseOptimizer` is the underlying object that is used to optimize anything. All other optimizers inherits this class. It offers the most flexibility in modelling.

class allopy.optimize.**BaseOptimizer**(*n*, *algorithm=40*, **args*, ***kwargs*)

__init__(*n*, *algorithm=40*, **args*, ***kwargs*)

The BaseOptimizer is the raw optimizer with minimal support. For advanced users, this class will provide the most flexibility. The default algorithm used is Sequential Least Squares Quadratic Programming.

Parameters

- **n** (*int*) – number of assets
- **algorithm** (*int or str*) – the optimization algorithm
- **args** – other arguments to setup the optimizer
- **kwargs** – other keyword arguments

add_equality_constraint(*fn*, *tol=None*)

Adds the equality constraint function in standard form, A = b. If the gradient of the constraint function is not specified and the algorithm used is a gradient-based one, the optimizer will attempt to insert a smart numerical gradient for it.

Parameters

- **fn** (Callable[[ndarray], float]) – Constraint function
- **tol** (*float, optional*) – A tolerance in judging feasibility for the purposes of stopping the optimization

Returns Own instance

Return type *BaseOptimizer*

add_equality_matrix_constraint(*Aeq*, *beq*, *tol=None*)
Sets equality constraints in standard matrix form.

For equality, $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$

Parameters

- **Aeq** – Equality matrix. Must be 2 dimensional
- **beq** – Equality vector or scalar. If scalar, it will be propagated
- **tol** – A tolerance in judging feasibility for the purposes of stopping the optimization

Returns Own instance

Return type *BaseOptimizer*

add_inequality_constraint(*fn*, *tol=None*)
Adds the equality constraint function in standard form, A <= b. If the gradient of the constraint function is not specified and the algorithm used is a gradient-based one, the optimizer will attempt to insert a smart numerical gradient for it.

Parameters

- **fn** (`Callable[[ndarray], float]`) – Constraint function
- **tol** (*float, optional*) – A tolerance in judging feasibility for the purposes of stopping the optimization

Returns Own instance

Return type *BaseOptimizer*

add_inequality_matrix_constraint(*A*, *b*, *tol=None*)
Sets inequality constraints in standard matrix form.

For inequality, $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$

Parameters

- **A** – Inequality matrix. Must be 2 dimensional.
- **b** – Inequality vector or scalar. If scalar, it will be propagated.
- **tol** – A tolerance in judging feasibility for the purposes of stopping the optimization

Returns Own instance

Return type *BaseOptimizer*

property lower_bounds
Lower bound of each variable

property model
The underlying optimizer. Use this if you need to access lower level settings for the optimizer

optimize(*x0=None*, **args*, *initial_solution='random'*, *random_state=None*)
Runs the optimizer and returns the optimal results if any.

Notes

An initial vector must be set and the quality of any solution (especially gradient-based ones) will lie on this initial vector. Alternatively, the optimizer will ATTEMPT to randomly generate a feasible one if the `initial_solution` argument is set to "random". However, there is no guarantee in the feasibility. In general, it is a tough problem to find a feasible solution in high-dimensional spaces, much more an optimal one. Thus use the random initial solution at your own risk.

The following lists the options for finding an initial solution for the optimization problem. It is best if the user supplies an initial value instead of using the heuristics provided if the user already knows the region to search.

random Randomly generates "bound-feasible" starting points for the decision variables. Note that these variables may not fulfil the other constraints. For problems where the bounds have been tightly defined, this often yields a good solution.

min_constraint_norm Solves the optimization problem listed below. The objective is to minimize the L_2 norm of the constraint functions while keeping the decision variables bounded by the original problem's bounds.

$$\begin{aligned} &\min |constraint|^2 \\ &s.t. \\ &LB \leq x \leq UB \end{aligned}$$

Parameters

- **x0** (*iterable float*) – Initial vector. Starting position for free variables. In many cases, especially for derivative-based optimizers, it is important for the initial vector to be already feasible.
- **args** – other arguments to pass into the optimizer
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied. See notes on Initial Solution for more information
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

Returns Values of free variables at optimality

Return type ndarray

remove_all_constraints()
Removes all constraints

set_bounds(*lb*, *ub*)
Sets the lower and upper bound

Parameters

- **lb** (`Union[ndarray, Iterable, int, float, complex]`) – Vector of lower bounds. If array, must be same length as number of free variables. If `float` or `int`, value will be propagated to all variables.
- **ub** (`Union[ndarray, Iterable, int, float, complex]`) – Vector of upper bounds. If array, must be same length as number of free variables. If `float` or `int`, value will be propagated to all variables.

Returns Own instance

Return type *BaseOptimizer*

set_epsilon(*eps*)
Sets the step difference used when calculating the gradient for derivative based optimization algorithms. This can ignored if you use a derivative free algorithm or if you specify your gradient specifically.

Parameters **eps** (*float*) – The gradient step

Returns Own instance

Return type *BaseOptimizer*

set_epsilon_constraint(*eps*)
Sets the tolerance for the constraint functions

Parameters **eps** (*float*) – Tolerance

Returns Own instance

Return type *BaseOptimizer*

set_ftol_abs(*tol*)
Set absolute tolerance on objective function value

Parameters **tol** (*float*) – absolute tolerance of objective function value

Returns Own instance

Return type *BaseOptimizer*

set_ftol_rel(*tol*)
Set relative tolerance on objective function value

Parameters **tol** (*float*) – Absolute relative of objective function value

Returns Own instance

Return type *BaseOptimizer*

set_lower_bounds(*lb*)
Sets the lower bounds

Parameters **lb** (Union[ndarray, Iterable, int, float, complex]) – Vector of lower bounds. If vector, must be same length as number of free variables. If float or int, value will be propagated to all variables.

Returns Own instance

Return type *BaseOptimizer*

set_max_objective(*fn*, **args*)
Sets the optimizer to maximize the objective function. If gradient of the objective function is not set and the algorithm used to optimize is gradient-based, the optimizer will attempt to insert a smart numerical gradient for it.

Parameters

- **fn** (*Callable*) – Objective function
- **args** – Other arguments to pass to the objective function. This can be ignored in most cases

Returns Own instance

Return type *BaseOptimizer*

set_maxeval(*n*)
Sets maximum number of objective function evaluations.

After maximum number of evaluations, optimization will stop. Set 0 or negative for no limit.

Parameters **n** (*int*) – maximum number of evaluations

Returns Own instance

Return type *BaseOptimizer*

set_min_objective(*fn*, **args*)
Sets the optimizer to minimize the objective function. If gradient of the objective function is not set and the algorithm used to optimize is gradient-based, the optimizer will attempt to insert a smart numerical gradient for it.

Parameters

- **fn** (*Callable*) – Objective function
- **args** – Other arguments to pass to the objective function. This can be ignored in most cases

Returns Own instance

Return type *BaseOptimizer*

set_stopval(*stopval*)
Stop when an objective value of at least/most stopval is found depending on min or max objective

Parameters **stopval** (*float*) – Stopping value

Returns Own instance

Return type *BaseOptimizer*

set_upper_bounds(*ub*)
Sets the upper bound

Parameters **ub** (Union[ndarray, Iterable, int, float, complex]) – Vector of lower bounds. If vector, must be same length as number of free variables. If float or int, value will be propagated to all variables.

Returns Own instance

Return type *BaseOptimizer*

set_xtol_abs(*tol*)
Sets absolute tolerances on optimization parameters.

The tol input must be an array of length *n* specified in the initialization. Alternatively, pass a single number in order to set the same tolerance for all optimization parameters.

Parameters **tol** (*{float, ndarray}*) – Absolute tolerance for each of the free variables

Returns Own instance

Return type *BaseOptimizer*

set_xtol_rel(*tol*)
Sets relative tolerances on optimization parameters.

The tol input must be an array of length *n* specified in the initialization. Alternatively, pass a single number in order to set the same tolerance for all optimization parameters.

Parameters **tol** (*float or ndarray, optional*) – relative tolerance for each of the free variables

Returns Own instance

Return type *BaseOptimizer*

property summary
Prints a summary report of the optimizer

property upper_bounds
Upper bound of each variable

3.2 Portfolio Optimizer

The `PortfolioOptimizer` inherits the `BaseOptimizer` to add several convenience methods. These methods include common optimization programs which would be tedious to craft with the `BaseOptimizer` over and over again. Of course, as an extension, it can do anything that the `BaseOptimizer` can. However, if that's the goal, it would be better to stick with the `BaseOptimizer` to reduce confusion when reading the code.

Using the `PortfolioOptimizer` assumes that there is a returns stream from which all other asset classes are benchmarked against. This is the first index in the assets axis.

For example, if you have a benchmark (beta) returns stream, 9 other asset classes over 10000 trials and 40 periods, the simulation tensor will be 40 x 10000 x 10 with the first asset axis being the returns of the benchmark. In such a case, the active portfolio optimizer can be used to optimize the portfolio relative to the benchmark.

The `PortfolioOptimizer` houses the following convenience methods:

maximize_returns Maximize the returns of the portfolio. You may put in volatility or CVaR constraints for this procedure.

minimize_volatility Minimizes the total portfolio volatility

minimize_cvar Minimizes the conditional value at risk (expected shortfall of the portfolio)

maximize_sharpe_ratio Maximizes the Sharpe ratio of the portfolio.

class allopy.optimize.**PortfolioOptimizer**(*data*, *algorithm=40*, *cvar_data=None*, *rebalance=False*, *time_unit='quarterly'*, *sum_to_1=True*, **args*, ***kwargs*)

__init__(*data*, *algorithm=40*, *cvar_data=None*, *rebalance=False*, *time_unit='quarterly'*, *sum_to_1=True*, **args*, ***kwargs*)
PortfolioOptimizer houses several common pre-specified optimization routines.

PortfolioOptimizer assumes that the optimization model has no uncertainty. That is, the portfolio is expected to undergo a single fixed scenario in the future. By default, the PortfolioOptimizer will automatically add an equality constraint that forces the portfolio weights to sum to 1.

Parameters

- **data** (*{ndarray, OptData}*) – The data used for optimization
- **algorithm** (*{int, string}*) – The algorithm used for optimization. Default is Sequential Least Squares Programming
- **cvar_data** (*{ndarray, OptData}*) – The cvar_data data used as constraint during the optimization. If this is not set, will default to being a copy of the original data that is trimmed to the first 3 years. If an array like object is passed in, the data must be a 3D array with axis representing time, trials and assets respectively. In that instance, the horizon will not be cut at 3 years, rather it'll be left to the user.

- **rebalance** (*bool, optional*) – Whether the weights are rebalanced in every time instance. Defaults to False
- **time_unit** (*{int, 'monthly', 'quarterly', 'semi-annually', 'yearly'}, optional*) – Specifies how many units (first axis) is required to represent a year. For example, if each time period represents a month, set this to 12. If quarterly, set to 4. Defaults to 12 which means 1 period represents a month. Alternatively, specify one of 'monthly', 'quarterly', 'semi-annually' or 'yearly'
- **sum_to_1** – If True, portfolio weights must sum to 1.
- **args** – other arguments to pass to the *BaseOptimizer*
- **kwargs** – other keyword arguments to pass into `OptData` (if you passed in a numpy array for *data*) or into the *BaseOptimizer*

See also:

BaseOptimizer Base Optimizer

OptData Optimizer data wrapper

maximize_returns(*max_vol=None*, *max_cvar=None*, *x0=None*, ***, *percentile=5.0*, *tol=0.0*, *initial_solution='random'*, *random_state=None*)

Optimizes the expected returns of the portfolio subject to max volatility and/or cvar constraint. At least one of the tracking error or cvar constraint must be defined.

If *max_vol* is defined, the tracking error will be offset by that amount. Maximum tracking error is usually defined by a positive number. Meaning if you would like to cap tracking error to 3%, max_te should be set to 0.03.

Parameters

- **max_vol** (*scalar, optional*) – Maximum tracking error allowed
- **max_cvar** (*scalar, optional*) – Maximum cvar_data allowed
- **x0** (*ndarray*) – Initial vector. Starting position for free variables
- **percentile** (*float*) – The CVaR percentile value. This means to the expected shortfall will be calculated from values below this threshold
- **tol** (*float*) – A tolerance for the constraints in judging feasibility for the purposes of stopping the optimization
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

Returns Optimal weights

Return type ndarray

maximize_sharpe_ratio(*x0=None*, ***, *initial_solution='random'*, *random_state=None*)

Maximizes the sharpe ratio the portfolio.

Parameters

- **x0** (*array_like, optional*) – Initial vector. Starting position for free variables

- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`
- **initial_solution** – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** – Random seed. Applicable if `initial_solution` is not `None`

Returns Optimal weights

Return type ndarray

minimize_cvar(*min_ret=None*, *x0=None*, ***, *percentile=5.0*, *tol=0.0*, *initial_solution='random'*, *random_state=None*)

Maximizes the conditional value at risk of the portfolio. The present implementation actually minimizes the expected shortfall. Maximizing this value means you stand to lose less (or even make more) money during bad times

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float, optional*) – The minimum returns required for the portfolio
- **x0** (*ndarray*) – Initial vector. Starting position for free variables
- **percentile** (*float*) – The CVaR percentile value for the objective. This means to the expected shortfall will be calculated from values below this threshold
- **tol** (*float*) – A tolerance for the constraints in judging feasibility for the purposes of stopping the optimization
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

Returns Optimal weights

Return type ndarray

minimize_volatility(*min_ret=None*, *x0=None*, ***, *tol=0.0*, *initial_solution='random'*, *random_state=None*)

Minimizes the tracking error of the portfolio

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float, optional*) – The minimum returns required for the portfolio
- **x0** (*ndarray*) – Initial vector. Starting position for free variables
- **tol** (*float*) – A tolerance for the constraints in judging feasibility for the purposes of stopping the optimization

- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

Returns Optimal weights

Return type ndarray

3.3 Active Portfolio Optimizer

The `ActivePortfolioOptimizer` inherits the `BaseOptimizer` to add several convenience methods. These methods include common optimization programs which would be tedious to craft with the `BaseOptimizer` over and over again. Of course, as an extension, it can do anything that the `BaseOptimizer` can. However, if that's the goal, it would be better to stick with the `BaseOptimizer` to reduce confusion when reading the code.

`ActivePortfolioOptimizer` houses the following convenience methods:

maximize_eva Maximize the expected value added of the portfolio. The objective function is the same with maximize returns as it just maximizes the total returns. For risk constraints, this method by default will constrain on tracking error and total CVaR. That is the volatility (tracking error) is calculated with the first variable (usually a passive portion) set to 0. Total CVaR has no treatments done to it. You can override these defaults in the method itself.

minimize_tracking_error Minimizes the tracking error of the portfolio. Tracking error is calculated by setting the first variable to 0 whilst the rest are updated by the optimizer.

minimize_volatility Minimizes the total portfolio volatility

minimize_cvar Minimizes the conditional value at risk (expected shortfall of the portfolio)

maximize_info_ratio Maximizes the information ratio of the portfolio. The information ratio of the portfolio is calculated like the Sharpe ratio. The only difference is the first variable is set to 0.

maximize_sharpe_ratio Maximizes the Sharpe ratio of the portfolio.

class allopy.optimize.**ActivePortfolioOptimizer**(*data*, *algorithm=40*, *cvar_data=None*, *rebalance=False*, *time_unit='quarterly'*, *sum_to_1=False*, **args*, ***kwargs*)

__init__(*data*, *algorithm=40*, *cvar_data=None*, *rebalance=False*, *time_unit='quarterly'*, *sum_to_1=False*, **args*, ***kwargs*)

The ActivePortfolioOptimizer houses several common pre-specified optimization routines.

ActivePortfolioOptimizer assumes that the optimization model has no uncertainty. That is, the portfolio is expected to undergo a single fixed scenario in the future.

Notes

ActivePortfolioOptimizer is a special case of the PortfolioOptimizer where the goal is to determine the best mix of of the portfolio relative to ba benchmark. By convention, the first asset of the data is the benchmark returns stream. The remaining returns stream is then the over or under performance of the returns over the benchmark. In this way, the optimization has an intuitive meaning of allocating resources whilst taking account

For example, if you have a benchmark (beta) returns stream, 9 other asset classes over 10000 trials and 40 periods, the simulation tensor will be 40 x 10000 x 10 with the first asset axis being the returns of the benchmark. In such a case, the active portfolio optimizer can be used to optimize the portfolio relative to the benchmark.

Parameters

- **data** (*{ndarray, OptData}*) – The data used for optimization
- **algorithm** (*{int, string}*) – The algorithm used for optimization. Default is Sequential Least Squares Programming
- **cvar_data** (*{ndarray, OptData}*) – The cvar_data data used as constraint during the optimization. If this is not set, will default to being a copy of the original data that is trimmed to the first 3 years. If an array like object is passed in, the data must be a 3D array with axis representing time, trials and assets respectively. In that instance, the horizon will not be cut at 3 years, rather it'll be left to the user.
- **rebalance** (*bool, optional*) – Whether the weights are rebalanced in every time instance. Defaults to False
- **time_unit** (*{int, 'monthly', 'quarterly', 'semi-annually', 'yearly'}, optional*) – Specifies how many units (first axis) is required to represent a year. For example, if each time period represents a month, set this to 12. If quarterly, set to 4. Defaults to 12 which means 1 period represents a month. Alternatively, specify one of 'monthly', 'quarterly', 'semi-annually' or 'yearly'
- **sum_to_1** (*bool*) – If False, the weights do not need to sum to 1. This should be False for active optimizer.
- **args** – other arguments to pass to the *BaseOptimizer*
- **kwargs** – other keyword arguments to pass into `OptData` (if you passed in a numpy array for *data*) or into the *BaseOptimizer*

See also:

BaseOptimizer Base Optimizer

OptData Optimizer data wrapper

maximize_eva(*max_vol=None*, *max_cvar=None*, *percentile=5.0*, *x0=None*, ***, *as_tracking_error=True*, *as_active_cvar=False*, *tol=0.0*)

Optimizes the expected value added of the portfolio subject to max tracking error and/or cvar constraint. At least one of the tracking error or cvar constraint must be defined.

If *max_te* is defined, the tracking error will be offset by that amount. Maximum tracking error is usually defined by a positive number. Meaning if you would like to cap tracking error to 3%, max_te should be set to 0.03.

Parameters

- **max_vol** (*float, optional*) – Maximum tracking error allowed

- **max_cvar** (*float, optional*) – Maximum cvar_data allowed
- **percentile** (*float*) – The CVaR percentile value. This means to the expected shortfall will be calculated from values below this threshold
- **x0** (*ndarray*) – Initial vector. Starting position for free variables
- **as_active_cvar** (*bool*) – If True, the cvar constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the cvar constraint is calculated using the entire weight vector.
- **as_tracking_error** (*bool*) – If True, the volatility constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the volatility constraint is calculated using the entire weight vector. This is also known as tracking error.
- **tol** (*float*) – A tolerance for the constraints in judging feasibility for the purposes of stopping the optimization

Returns Optimal weights

Return type ndarray

maximize_info_ratio(*x0=None*)
Maximizes the information ratio the portfolio.

Parameters **x0** (*array_like, optional*) – initial vector. Starting position for free variables

Returns Optimal weights

Return type ndarray

maximize_sharpe_ratio(*x0=None*)
Maximizes the Sharpe ratio the portfolio.

Parameters **x0** (*array_like, optional*) – initial vector. Starting position for free variables

Returns Optimal weights

Return type ndarray

minimize_cvar(*min_ret=None*, *x0=None*, ***, *percentile=5.0*, *as_active_cvar=False*, *as_active_return=False*, *tol=0.0*)
Minimizes the conditional value at risk of the portfolio. The present implementation actually minimizes the expected shortfall.

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float, optional*) – The minimum returns required for the portfolio
- **x0** (*ndarray*) – Initial vector. Starting position for free variables
- **percentile** (*float*) – The CVaR percentile value for the objective. This means to the expected shortfall will be calculated from values below this threshold
- **as_active_cvar** (*bool, optional*) – If True, minimizes the active cvar instead of the entire portfolio cvar. If False, minimizes the entire portfolio's cvar
- **as_active_return** (*bool, optional*) – If True, the returns constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the returns constraint is calculated using the entire weight vector.

- **tol** (*float*) – A tolerance for the constraints in judging feasibility for the purposes of stopping the optimization

Returns Optimal weights

Return type ndarray

minimize_tracking_error(*min_ret=None*, *x0=None*, ***, *as_active_return=False*, *tol=0.0*)
Minimizes the tracking error of the portfolio

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float, optional*) – The minimum returns required for the portfolio
- **x0** (*ndarray*) – Initial vector. Starting position for free variables
- **as_active_return** (*boolean, optional*) – If True, the returns constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the returns constraint is calculated using the entire weight vector.
- **tol** (*float*) – A tolerance for the constraints in judging feasibility for the purposes of stopping the optimization

Returns Optimal weights

Return type ndarray

minimize_volatility(*min_ret=None*, *x0=None*, ***, *as_active_return=False*, *tol=0.0*)
Minimizes the volatility of the portfolio

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float, optional*) – The minimum returns required for the portfolio
- **x0** (*ndarray*) – Initial vector. Starting position for free variables

as_active_return: boolean, optional If True, the returns constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the returns constraint is calculated using the entire weight vector.

tol: float A tolerance for the constraints in judging feasibility for the purposes of stopping the optimization

Returns Optimal weights

Return type ndarray

3.4 Regret Optimizer

The `RegretOptimizer` inherits the `DiscreteUncertaintyOptimizer`. The minimum regret optimization is a technique under decision theory on making decisions under uncertainty.

class allopy.optimize.**RegretOptimizer**(*num_assets*, *num_scenarios*, *prob=None*, *algorithm=40*, *c_eps=None*, *xtol_abs=None*, *xtol_rel=None*, *ftol_abs=None*, *ftol_rel=None*, *max_eval=None*, *verbose=False*, *sum_to_1=True*, *max_attempts=5*)

__init__(*num_assets*, *num_scenarios*, *prob=None*, *algorithm=40*, *c_eps=None*, *xtol_abs=None*, *xtol_rel=None*, *ftol_abs=None*, *ftol_rel=None*, *max_eval=None*, *verbose=False*, *sum_to_1=True*, *max_attempts=5*)

The RegretOptimizer is a convenience class for scenario based optimization.

Notes

The term regret refers to the instance where after having decided on one alternative, the choice of a different alternative would have led to a more optimal (better) outcome when the eventual scenario transpires.

The RegretOptimizer employs a 2 stage optimization process. In the first step, the optimizer calculates the optimal weights for each scenario. In the second stage, the optimizer minimizes the regret function to give the final optimal portfolio weights.

Assuming the objective is to maximize returns subject to some volatility constraints, the first stage optimization will be as listed

where $R_s(\cdot)$ is the returns function for scenario s, $\sigma_s(\cdot)$ is the volatility function for scenario s and Σ is the volatility threshold. Subsequently, to minimize the regret across all scenarios, S,

Where $D(\cdot)$ is a distance function (usually quadratic) and p_s is the discrete probability of scenario s occurring.

Parameters

- **num_assets** (*int*) – Number of assets
- **num_scenarios** (*int*) – Number of scenarios
- **prob** (Union[Iterable[float], Iterable[int], ndarray, None]) – Vector containing probability of each scenario occurring
- **algorithm** – The optimization algorithm. Default algorithm is Sequential Least Squares Quadratic Programming.
- **c_eps** (*float, optional*) – Constraint epsilon is the tolerance for the inequality and equality constraints functions. Any value that is less than the constraint epsilon is considered to be within the boundary.
- **xtol_abs** (*float or np.ndarray, optional*) – Set absolute tolerances on optimization parameters. `tol` is an array giving the tolerances: stop when an optimization step (or an estimate of the optimum) changes every parameter `x[i]` by less than `tol[i]`. For convenience, if a scalar `tol` is given, it will be used to set the absolute tolerances in all n optimization parameters to the same value. Criterion is disabled if tol is non-positive.
- **xtol_rel** (*float or np.ndarray, optional*) – Set relative tolerance on optimization parameters: stop when an optimization step (or an estimate of the optimum) causes a relative change the parameters `x` by less than `tol`, i.e. $\|\Delta x\|_w < tol \cdot \|x\|_w$ measured by a weighted L_1 norm $\|x\|_w = \sum_i w_i |x_i|$, where the weights w_i default to 1. (If there is any chance that the optimal $\|x\|$ is close to zero, you might want to set an absolute tolerance with *code:`xtol_abs* as well.) Criterion is disabled if tol is non-positive.
- **ftol_abs** (*float, optional*) – Set absolute tolerance on function value: stop when an optimization step (or an estimate of the optimum) changes the function value by less than `tol`. Criterion is disabled if tol is non-positive.

- **ftol_rel** (*float, optional*) – Set relative tolerance on function value: stop when an optimization step (or an estimate of the optimum) changes the objective function value by less than `tol` multiplied by the absolute value of the function value. (If there is any chance that your optimum function value is close to zero, you might want to set an absolute tolerance with `ftol_abs` as well.) Criterion is disabled if tol is non-positive.
- **max_eval** (*int, optional*) – Stop when the number of function evaluations exceeds `maxeval`. (This is not a strict maximum: the number of function evaluations may exceed `maxeval` slightly, depending upon the algorithm.) Criterion is disabled if maxeval is non-positive.
- **verbose** (*bool*) – If True, the optimizer will report its operations
- **sum_to_1** (*bool*) – If true, the optimal weights for each first level scenario must sum to 1.
- **max_attempts** (*int*) – Number of times to retry optimization. This is useful when optimization is in a highly unstable or non-convex space.

See also:

DiscreteUncertaintyOptimizer Discrete Uncertainty Optimizer

add_equality_constraint (*functions*)

Adds the equality constraint function in standard form, A = b. If the gradient of the constraint function is not specified and the algorithm used is a gradient-based one, the optimizer will attempt to insert a smart numerical gradient for it.

The list of functions needs to match the number of scenarios. The function at index 0 will be assigned as a constraint function to the first optimization regime.

Parameters **functions** (*Callable or List of Callable*) – Constraint function. The function signature should be such that the first argument takes in a weight vector and outputs a numeric (float). The second argument is optional and contains the gradient. If given, the user must put adjust the gradients inplace. If only a single function is given, the same function will be used for all the scenarios

add_equality_matrix_constraint (*Aeq*, *beq*)

Sets equality constraints in standard matrix form.

For equality, $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$

Parameters

- **Aeq** (*{iterable float, ndarray}*) – Equality matrix. Must be 2 dimensional
- **beq** (*{scalar, ndarray}*) – Equality vector or scalar. If scalar, it will be propagated

add_inequality_constraint (*functions*)

Adds the equality constraint function in standard form, A <= b. If the gradient of the constraint function is not specified and the algorithm used is a gradient-based one, the optimizer will attempt to insert a smart numerical gradient for it.

The list of functions needs to match the number of scenarios. The function at index 0 will be assigned as a constraint function to the first optimization regime.

Parameters **functions** (*Callable or List of Callable*) – Constraint functions. The function signature should be such that the first argument takes in a weight vector and outputs a numeric (float). The second argument is optional and contains the gradient. If given, the user must put adjust the gradients inplace. If only a single function is given, the same function will be used for all the scenarios

add_inequality_matrix_constraint(*A*, *b*)
Sets inequality constraints in standard matrix form.

For inequality, $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$

Parameters

- **A** (*{iterable float, ndarray}*) – Inequality matrix. Must be 2 dimensional.
- **b** (*{scalar, ndarray}*) – Inequality vector or scalar. If scalar, it will be propagated.

property lower_bounds
Lower bound of each variable for all the optimization models in the first and second stages

optimize(*x0_first_level=None*, *x0_prop=None*, *initial_solution='random'*, *approx=True*, *dist_func=<ufunc 'square'>*, *random_state=None*)
Finds the minimal regret solution across the range of scenarios

Notes

The exact (actual) objective function to minimize regret is given below,

However, given certain problem formulations where the objective and constraint functions are linear and convex, the problem can be transformed to

where W is a matrix where each rows represents a single scenario, s and each column represents an asset class. This formulation solves for a which represents the proportion of each scenario that contributes to the final portfolio weights. Thus if there are 3 scenarios and a is `[0.3, 0.5, 0.2]`, it means that the final portfolio took 30% from scenario 1, 50% from scenario 2 and 20% from scenario 3.

This formulation makes a strong assumption that the final minimal regret portfolio is a linear combination of the weights from each scenario's optimal.

The following lists the options for finding an initial solution for the optimization problem. It is best if the user supplies an initial value instead of using the heuristics provided if the user already knows the region to search.

random Randomly generates "bound-feasible" starting points for the decision variables. Note that these variables may not fulfil the other constraints. For problems where the bounds have been tightly defined, this often yields a good solution.

min_constraint_norm Solves the optimization problem listed below. The objective is to minimize the L_2 norm of the constraint functions while keeping the decision variables bounded by the original problem's bounds.

$$\begin{aligned} &\min |constraint|^2 \\ &s.t. \\ &LB \leq x \leq UB \end{aligned}$$

Parameters

- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.

- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied. See notes on Initial Solution for more information
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal. See Notes.
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is "random"

Returns Regret optimal solution weights

Return type np.ndarray

property prob
Vector containing probability of each scenario occurring

set_bounds(*lb*, *ub*)
Sets the lower and upper bound

Parameters

- **lb** (*{int, float, ndarray}*) – Vector of lower bounds. If array, must be same length as number of free variables. If `float` or `int`, value will be propagated to all variables.
- **ub** (*{int, float, ndarray}*) – Vector of upper bounds. If array, must be same length as number of free variables. If `float` or `int`, value will be propagated to all variables.

set_epsilon_constraint(*eps*)
Sets the tolerance for the constraint functions

Parameters **eps** (*float*) – Tolerance

set_ftol_abs(*tol*)
Set absolute tolerance on objective function value.

The absolute tolerance on function value: stop when an optimization step (or an estimate of the optimum) changes the function value by less than `tol`. Criterion is disabled if tol is non-positive.

Parameters **tol** (*float*) – absolute tolerance of objective function value

set_ftol_rel(*tol*)
Set relative tolerance on objective function value.

Set relative tolerance on function value: stop when an optimization step (or an estimate of the optimum) changes the objective function value by less than `tol` multiplied by the absolute value of the function value. (If there is any chance that your optimum function value is close to zero, you might want to set an absolute tolerance with `ftol_abs` as well.) Criterion is disabled if tol is non-positive.

Parameters **tol** (*float, optional*) – Absolute relative of objective function value

set_max_objective(*functions*)
Sets the optimizer to maximize the objective function. If gradient of the objective function is not set and

the algorithm used to optimize is gradient-based, the optimizer will attempt to insert a smart numerical gradient for it.

The list of functions needs to match the number of scenarios. The function at index 0 will be assigned as the objective function to the first optimization regime.

Parameters **functions** (*Callable or List of Callable*) – Objective function. The function signature should be such that the first argument takes in a weight vector and outputs a numeric (float). The second argument is optional and contains the gradient. If given, the user must put adjust the gradients inplace. If only a single function is given, the same function will be used for all the scenarios

set_maxeval(*n*)

Sets maximum number of objective function evaluations.

Stop when the number of function evaluations exceeds `maxeval`. (This is not a strict maximum: the number of function evaluations may exceed `maxeval` slightly, depending upon the algorithm.) Criterion is disabled if maxeval is non-positive.

Parameters **n** (*int*) – maximum number of evaluations

set_meta(*, *assets=None*, *scenarios=None*)

Sets meta data which will be used for result summary

Parameters

- **assets** (*list of str, optional*) – Names of each asset class
- **scenarios** (*list of str, optional*) – Names of each scenario

set_min_objective(*functions*)

Sets the optimizer to minimize the objective function. If gradient of the objective function is not set and the algorithm used to optimize is gradient-based, the optimizer will attempt to insert a smart numerical gradient for it.

The list of functions needs to match the number of scenarios. The function at index 0 will be assigned as the objective function to the first optimization regime.

Parameters **functions** (*Callable or List of Callable*) – Objective function. The function signature should be such that the first argument takes in a weight vector and outputs a numeric (float). The second argument is optional and contains the gradient. If given, the user must put adjust the gradients inplace. If only a single function is given, the same function will be used for all the scenarios

set_xtol_abs(*tol*)

Sets absolute tolerances on optimization parameters.

The absolute tolerances on optimization parameters. `tol` is an array giving the tolerances: stop when an optimization step (or an estimate of the optimum) changes every parameter `x[i]` by less than `tol[i]`. For convenience, if a scalar `tol` is given, it will be used to set the absolute tolerances in all n optimization parameters to the same value. Criterion is disabled if tol is non-positive.

Parameters **tol** (*float or np.ndarray*) – Absolute tolerance for each of the free variables

set_xtol_rel(*tol*)

Sets relative tolerances on optimization parameters.

Set relative tolerance on optimization parameters: stop when an optimization step (or an estimate of the optimum) causes a relative change the parameters `x` by less than `tol`, i.e. $\|\Delta x\|_w < tol \cdot \|x\|_w$ measured by a weighted L_1 norm $\|x\|_w = \sum_i w_i |x_i|$, where the weights w_i default to 1. (If there is any chance that

the optimal $\|x\|$ is close to zero, you might want to set an absolute tolerance with *code:`xtol_abs* as well.) Criterion is disabled if tol is non-positive.

Parameters **tol** (*float or np.ndarray*) – relative tolerance for each of the free variables

property upper_bounds
Upper bound of each variable for all the optimization models in the first and second stages

3.5 Portfolio Regret Optimizer

The `PortfolioRegretOptimizer` inherits the `RegretOptimizer`. The minimum regret optimization is a technique under decision theory on making decisions under uncertainty.

The methods in the `PortfolioRegretOptimizer` are only applied at the first stage of the procedure. The `PortfolioRegretOptimizer` houses the following convenience methods:

maximize_returns Maximize the returns of the portfolio. You may put in volatility or CVaR constraints for this procedure.

minimize_volatility Minimizes the total portfolio volatility

minimize_cvar Minimizes the conditional value at risk (expected shortfall of the portfolio)

maximize_sharpe_ratio Maximizes the Sharpe ratio of the portfolio.

class allopy.optimize.**PortfolioRegretOptimizer**(*data*, *cvar_data=None*, *prob=None*, *rebalance=True*, *sum_to_1=True*, *time_unit='quarterly'*, ***kwargs*)

__init__(*data*, *cvar_data=None*, *prob=None*, *rebalance=True*, *sum_to_1=True*, *time_unit='quarterly'*, ***kwargs*)
PortfolioRegretOptimizer houses several common pre-specified regret optimization routines. Regret optimization is a scenario based optimization.

Notes

The term regret refers to the instance where after having decided on one alternative, the choice of a different alternative would have led to a more optimal (better) outcome when the eventual scenario transpires.

The RegretOptimizer employs a 2 stage optimization process. In the first step, the optimizer calculates the optimal weights for each scenario. In the second stage, the optimizer minimizes the regret function to give the final optimal portfolio weights.

Assuming the objective is to maximize returns subject to some volatility constraints, the first stage optimization will be as listed

where $R_s(\cdot)$ is the returns function for scenario s, $\sigma_s(\cdot)$ is the volatility function for scenario s and Σ is the volatility threshold. Subsequently, to minimize the regret across all scenarios, S,

Where $D(\cdot)$ is a distance function (usually quadratic) and p_s is the discrete probability of scenario s occurring.

Parameters

- **data** (List[Union[OptData, ndarray]]) – Scenario data. Each data must be a 3 dimensional tensor. Thus data will be a 4-D tensor.
- **cvar_data** (*optional*) – CVaR scenario data. Each data must be a 3 dimensional tensor. Thus data will be a 4-D tensor.
- **prob** (Union[Iterable[float], Iterable[int], ndarray, None]) – Vector containing probability of each scenario occurring
- **rebalance** (*bool, optional*) – Whether the weights are rebalanced in every time instance. Defaults to True
- **sum_to_1** – If True, portfolio weights must sum to 1. Defaults to True
- **time_unit** (*{int, 'monthly', 'quarterly', 'semi-annually', 'yearly'}, optional*) – Specifies how many units (first axis) is required to represent a year. For example, if each time period represents a month, set this to 12. If quarterly, set to 4. Defaults to 12 which means 1 period represents a month. Alternatively, specify one of 'monthly', 'quarterly', 'semi-annually' or 'yearly'
- **kwargs** – Other keyword arguments to pass to the *RegretOptimizer* base class

See also:

RegretOptimizer RegretOptimizer

maximize_returns(*max_vol=None*, *max_cvar=None*, *percentile=5.0*, ***, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Optimizes the expected returns of the portfolio subject to max volatility and/or cvar constraint. At least one of the tracking error or cvar constraint must be defined.

If *max_vol* is defined, the tracking error will be offset by that amount. Maximum tracking error is usually defined by a positive number. Meaning if you would like to cap tracking error to 3%, max_te should be set to 0.03.

Parameters

- **max_vol** (*float or list of floats, optional*) – Maximum tracking error allowed. If a scalar, the same value will be used for each scenario optimization.
- **max_cvar** (*float or list of floats, optional*) – Maximum cvar_data allowed. If a scalar, the same value will be used for each scenario optimization.
- **percentile** (*float*) – The CVaR percentile value. This means to the expected shortfall will be calculated from values below this threshold
- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.
- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if approx is False or the scenario proportion otherwise.
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.

- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

maximize_sharpe_ratio(*, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Maximizes the sharpe ratio the portfolio.

Parameters

- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.
- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

Returns Optimal weights

Return type ndarray

minimize_cvar(*min_ret=None*, *percentile=5.0*, *, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Minimizes the conditional value at risk of the portfolio. The present implementation actually minimizes the expected shortfall.

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float or list of floats, optional*) – The minimum returns required for the portfolio. If a scalar, the same value will be used for each scenario optimization.
- **percentile** (*float*) – The CVaR percentile value for the objective. This is the average expected shortfall from values below this threshold
- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.
- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.

- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

Returns Optimal weights

Return type ndarray

minimize_volatility(*min_ret=None*, ***, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Minimizes the tracking error of the portfolio

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float or list of floats, optional*) – The minimum returns required for the portfolio. If a scalar, the same value will be used for each scenario optimization.
- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.
- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

3.6 Acitve Portfolio Regret Optimizer

The `ActivePortfolioRegretOptimizer` inherits the `RegretOptimizer`. The minimum regret optimization is a technique under decision theory on making decisions under uncertainty.

The methods in the `ActivePortfolioRegretOptimizer` are only applied at the first stage of the procedure. `ActivePortfolioRegretOptimizer` houses the following convenience methods:

maximize_eva Maximize the expected value added of the portfolio. The objective function is the same with maximize returns as it just maximizes the total returns. For risk constraints, this method by default will constrain on tracking error and total CVaR. That is the volatility (tracking error) is calculated with the first variable (usually a passive portion) set to 0. Total CVaR has no treatments done to it. You can override these defaults in the method itself.

minimize_tracking_error Minimizes the tracking error of the portfolio. Tracking error is calculated by setting the first variable to 0 whilst the rest are updated by the optimizer.

minimize_volatility Minimizes the total portfolio volatility

minimize_cvar Minimizes the conditional value at risk (expected shortfall of the portfolio)

maximize_info_ratio Maximizes the information ratio of the portfolio. The information ratio of the portfolio is calculated like the Sharpe ratio. The only difference is the first variable is set to 0.

maximize_sharpe_ratio Maximizes the Sharpe ratio of the portfolio.

class allopy.optimize.**ActivePortfolioRegretOptimizer**(*data*, *cvar_data=None*, *prob=None*, *rebalance=False*, *sum_to_1=False*, *time_unit='quarterly'*, ***kwargs*)

__init__(*data*, *cvar_data=None*, *prob=None*, *rebalance=False*, *sum_to_1=False*, *time_unit='quarterly'*, ***kwargs*)

PortfolioRegretOptimizer houses several common pre-specified regret optimization routines. Regret optimization is a scenario based optimization.

Notes

The term regret refers to the instance where after having decided on one alternative, the choice of a different alternative would have led to a more optimal (better) outcome when the eventual scenario transpires.

The RegretOptimizer employs a 2 stage optimization process. In the first step, the optimizer calculates the optimal weights for each scenario. In the second stage, the optimizer minimizes the regret function to give the final optimal portfolio weights.

Assuming the objective is to maximize returns subject to some volatility constraints, the first stage optimization will be as listed

where $R_s(\cdot)$ is the returns function for scenario s, $\sigma_s(\cdot)$ is the volatility function for scenario s and Σ is the volatility threshold. Subsequently, to minimize the regret across all scenarios, S,

Where $D(\cdot)$ is a distance function (usually quadratic) and p_s is the discrete probability of scenario s occurring.

Parameters

- **data** (List[Union[OptData, ndarray]]) – Scenario data. Each data must be a 3 dimensional tensor. Thus data will be a 4-D tensor.
- **cvar_data** (*optional*) – CVaR scenario data. Each data must be a 3 dimensional tensor. Thus data will be a 4-D tensor.
- **prob** (Union[Iterable[float], Iterable[int], ndarray, None]) – Vector containing probability of each scenario occurring
- **rebalance** (*bool, optional*) – Whether the weights are rebalanced in every time instance. Defaults to False
- **sum_to_1** – If True, portfolio weights must sum to 1. Defaults to False
- **time_unit** ({*int, 'monthly', 'quarterly', 'semi-annually', 'yearly'}, optional*) – Specifies how many units (first axis) is required to represent a year. For example, if each time period represents a month, set this to 12. If quarterly, set to 4. Defaults to 12 which means 1 period represents a month. Alternatively, specify one of 'monthly', 'quarterly', 'semi-annually' or 'yearly'
- **kwargs** – Other keyword arguments to pass to the *RegretOptimizer* base class

See also:

RegretOptimizer RegretOptimizer

maximize_eva(*max_vol=None*, *max_cvar=None*, *percentile=5.0*, ***, *as_tracking_error=True*, *as_active_cvar=False*, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Optimizes the expected value added of the actively managed portfolio subject to max volatility and/or cvar constraint. At least one of the tracking error or cvar constraint must be defined.

If *max_vol* is defined, the tracking error will be offset by that amount. Maximum tracking error is usually defined by a positive number. Meaning if you would like to cap tracking error to 3%, max_te should be set to 0.03.

Parameters

- **max_vol** (*float or list of floats, optional*) – Maximum tracking error allowed. If a scalar, the same value will be used for each scenario optimization.
- **max_cvar** (*float or list of floats, optional*) – Maximum cvar_data allowed. If a scalar, the same value will be used for each scenario optimization.
- **percentile** (*float*) – The CVaR percentile value. This means to the expected shortfall will be calculated from values below this threshold
- **as_active_cvar** (*bool*) – If True, the cvar constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the cvar constraint is calculated using the entire weight vector.
- **as_tracking_error** (*bool*) – If True, the volatility constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the volatility constraint is calculated using the entire weight vector. This is also known as tracking error.
- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.

- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

maximize_info_ratio(*, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Maximizes the information ratio the portfolio.

Parameters

- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.
- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

Returns Optimal weights

Return type ndarray

maximize_sharpe_ratio(*, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Maximizes the sharpe ratio the portfolio.

Parameters

- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.
- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal

- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

Returns Optimal weights

Return type ndarray

minimize_cvar(*min_ret=None*, *percentile=5.0*, ***, *as_active_cvar=False*, *as_active_return=False*, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Minimizes the conditional value at risk of the portfolio. The present implementation actually minimizes the expected shortfall.

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float or list of floats, optional*) – The minimum returns required for the portfolio. If a scalar, the same value will be used for each scenario optimization.
- **percentile** (*float*) – The CVaR percentile value for the objective. This is the average expected shortfall from values below this threshold
- **as_active_cvar** (*bool, optional*) – If True, minimizes the active cvar instead of the entire portfolio cvar. If False, minimizes the entire portfolio's cvar
- **as_active_return** (*bool, optional*) – If True, the returns constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the returns constraint is calculated using the entire weight vector.
- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.
- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

Returns Optimal weights

Return type ndarray

minimize_tracking_error(*min_ret=None*, ***, *as_active_return=False*, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Minimizes the tracking error of the portfolio

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float or list of floats, optional*) – The minimum returns required for the portfolio. If a scalar, the same value will be used for each scenario optimization.
- **as_active_return** (*boolean, optional*) – If True, the returns constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the returns constraint is calculated using the entire weight vector.
- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.
- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

minimize_volatility(*min_ret=None*, ***, *as_active_return=False*, *x0_first_level=None*, *x0_prop=None*, *approx=True*, *dist_func=<ufunc 'square'>*, *initial_solution='random'*, *random_state=None*)

Minimizes the volatility of the portfolio

If the *min_ret* is specified, the optimizer will search for an optimal portfolio where the returns are at least as large as the value specified (if possible).

Parameters

- **min_ret** (*float or list of floats, optional*) – The minimum returns required for the portfolio. If a scalar, the same value will be used for each scenario optimization.
- **as_active_return** (*boolean, optional*) – If True, the returns constraint is calculated using the active portion of the weights. That is, the first value is forced to 0. If False, the returns constraint is calculated using the entire weight vector.
- **x0_first_level** (*list of list of floats or ndarray, optional*) – List of initial solution vector for each scenario optimization. If provided, the list must have the same length at the first dimension as the number of solutions.

- **x0_prop** (*list of floats, optional*) – Initial solution vector for the regret optimization (2nd level). This can either be the final optimization weights if `approx` is `False` or the scenario proportion otherwise.
- **approx** (*bool*) – If True, a linear approximation will be used to calculate the regret optimal
- **dist_func** (*Callable*) – A callable function that will be applied as a distance metric for the regret function. The default is a quadratic function. See Notes.
- **initial_solution** (*str, optional*) – The method to find the initial solution if the initial vector `x0` is not specified. Set as `None` to disable. However, if disabled, the initial vector must be supplied.
- **random_state** (*int, optional*) – Random seed. Applicable if `initial_solution` is not `None`

3.7 Algorithms

All algorithms in `allopy` follows a particular naming pattern. Specifically, the names are of the form `{G,L}{N,D}_xxx` where `G`/`L` denotes if the algorithm is **global** or **local** optimization, `N`/`D` denotes if the algorithm is a **derivative-free** or **gradient-based** algorithm and `xxx` is the name of the algorithm.

For example, `LD_SLSQP` refers to the Sequential Quadratic Least Squares Programming. This is a local and derivative free algorithm.

Many algorithms have variants and some, such as `AUGLAG`, may not have the construct described above. In using any of these algorithms, consult a textbook or Wikipedia to understand if it is fit for your purpose.

A list of the algorithms is given below. Check out the docs at nlopt for more information.

- GN_DIRECT_L
- GN_DIRECT_L_RAND
- GN_DIRECT_NOSCAL
- GN_DIRECT_L_NOSCAL
- GN_DIRECT_L_RAND_NOSCAL
- GN_ORIG_DIRECT
- GN_ORIG_DIRECT_L
- GD_STOGO
- GD_STOGO_RAND
- LD_LBFGS_NOCEDAL
- LD_LBFGS
- LN_PRAXIS
- LD_VAR1
- LD_VAR2
- LD_TNEWTON
- LD_TNEWTON_RESTART
- LD_TNEWTON_PRECOND

- LD_TNEWTON_PRECOND_RESTART
- GN_CRS2_LM
- GN_MLSL
- GD_MLSL
- GN_MLSL_LDS
- GD_MLSL_LDS
- LD_MMA
- LN_COBYLA
- LN_NEWUOA
- LN_NEWUOA_BOUND
- LN_NELDERMEAD
- LN_SBPLX
- LN_AUGLAG
- LD_AUGLAG
- LN_AUGLAG_EQ
- LD_AUGLAG_EQ
- LN_BOBYQA
- GN_ISRES
- AUGLAG
- AUGLAG_EQ
- G_MLSL
- G_MLSL_LDS
- LD_SLSQP
- LD_CCSAQ
- GN_ESCH
- GN_AGS

CHAPTER

FOUR

PENALTY

The penalties are classes that are added to the Portfolio Optimizer family of optimizers (i.e. `PortfolioOptimizer`, `ActivePortfolioOptimizer`) to impose a penalty to the particular asset weight based on the amount of uncertainty present for that asset class.

Uncertainty in this instance does not mean the risk (or variance). Rather, it signifies how uncertain we are of those estimates. For example, it represents how uncertain we are of the returns (mean) and volatility (standard deviation) estimates we have projected for the asset class.

4.1 NoPenalty

class allopy.penalty.**NoPenalty**(*dim*)

__init__(*dim*)

No penalty is a no-op penalty function. Essentially it applies no penalty to the objective function when applied to the objective function.

Parameters **dim** (int) – Number of assets

cost(_)

Calculates the penalty to apply

$$p(w) = 0$$

Return type float

4.2 UncertaintyPenalty

class allopy.penalty.**UncertaintyPenalty**(*uncertainty*, *alpha=0.95*, *method='direct'*, *dim=None*)

__init__(*uncertainty*, *alpha=0.95*, *method='direct'*, *dim=None*)

The uncertainty penalty. It penalizes the objective function relative to the level of uncertainty for the given asset

Notes

Given an initial maximizing objective, this penalty will change the objective to

$$f(w) - \lambda\sqrt{w^T \Phi w}$$

where Φ represent the uncertainty matrix. $\lambda = 0$ or a 0-matrix is a special case where there are no uncertainty in the projections.

If using χ^2 method, the λ value is given by

$$\lambda = \frac{1}{\chi^2_{n-1}(\alpha)}$$

where n is the number of asset classes and α is the confidence interval. Otherwise the "direct" method will have $\lambda = \alpha$.

Parameters

- **uncertainty** (Union[Iterable[Union[int, float]], ndarray]) – A 1D vector or 2D matrix representing the uncertainty for the given asset class. If a 1D vector is provided, it will be converted to a diagonal matrix
- **alpha** (float) – A constant controlling the intensity of the penalty
- **method** (*"chi2" or "direct"*) – Method used to construct the lambda parameter. If "direct", the exact value specified by the *alpha* parameter is used. If "chi2", the value is determined using the inverse of the chi-square quantile function. In that instance, the *alpha* parameter will be the confidence level. See Notes.
- **dim** (*int*) – If provided, it will override the default dimension of the penalty which is determined by the length of the uncertainty vector/matrix provided

cost(*w*)

Calculates the penalty to apply

$$p(w) = \lambda\sqrt{w^T \Phi w}$$

Return type float

CHAPTER

FIVE

DATASETS

To aid development, the allopy package houses a few test dataset. The data could be large so they will be downloaded for the first time you request them and saved in your home folder.

5.1 Load Index

allopy.datasets.**load_index**(*, *download=False*)

Dataset contains the index value of 7 asset classes from 01 Jan 1985 to 01 Oct 2017.

This dataset is usually used only for demonstration purposes. As such, the values have been fudged slightly.

Parameters **download** (*bool*) – If True, forces the data to be downloaded again from the repository. Otherwise, loads the data from the stash folder

Returns A data frame containing the index of the 7 policy asset classes

Return type DataFrame

5.2 Load Monte Carlo

allopy.datasets.**load_monte_carlo**(*, *download=False*, *total=False*)

Loads a data set containing a mock Monte Carlo simulation of asset class returns.

The Monte Carlo tensor has axis represents time, trials and asset respectively. For the non-total cube, the shape is 80 x 10000 x 9 meaning there are 80 time periods over 10000 trials and 9 asset classes.

The total Monte Carlo tensor's shape is 60 x 10000 x 36

Parameters

- **download** (*bool*) – If True, forces the data to be downloaded again from the repository. Otherwise, loads the data from the stash folder
- **total** (*bool*) – If True, loads the monte carlo simulation with the total set of asset classes to simulate a big portfolio

Returns A Monte Carlo tensor

Return type ndarray

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

L

M

N

O

P

R

S

U