Package 'RelValAnalysis'

February 19, 2015

Type Package

Title Relative Value Analysis

Version 1.0

Depends zoo

Date 2014-06-25

Maintainer Ting-Kam Leonard Wong <wongting@uw.edu>

Description Classes and functions for analyzing the performance of portfolios relative to a benchmark.

License GPL-2 | GPL-3

Author Ting-Kam Leonard Wong [aut, cre]

NeedsCompilation no

Repository CRAN

Date/Publication 2014-06-26 07:40:05

R topics documented:

RelValAnalysis-package . . . 2
applestarbucks . . . 3
AtlasModel . . . 3
capdist . . . 4
CapDistSlope . . . 5
ConstantPortfolio . . . 6
Diversity . . . 7
DiversityPortfolio . . . 8
EEControl . . . 9
EnergyEntropyDecomp . . . 10
EntropyPortfolio . . . 12
FernholzDecomp . . . 12
fgp . . . 14
FreeEnergy . . . 15
GeometricMean . . . 16
GetGroupWeight . . . 17
GetLambdaWeight . . . 18

GetNewLambdaWeight 19
GetWeight 20
Invest 21
marketmodel 23
ParetoCapDist 24
plot.capdist 25
plot.toymkt 26
print.capdist 27
print.fgp 27
print.marketmodel 28
print.toymkt 28
RelativeEntropy 29
RenyiEntropy 30
ShannonEntropy 31
SimMarketModel 32
toymkt 33
VolStabModel 35

Index **37**

`RelValAnalysis-package`

The RelValAnalysis Package

Description

Classes and functions for analyzing the performance of portfolios relative to a benchmark.

Details

Package:	RelValAnalysis
Type:	Package
Version:	1.0
Date:	2014-06-25
License:	GPL-2 | GPL-3

Author(s)

Ting-Kam Leonard Wong

Maintainer: Ting-Kam Leonard Wong <wongting@uw.edu>

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

Karatzas, I. and R. Fernholz (2009). Stochastic portfolio theory: an overview. *Handbook of numerical analysis 15*, 89-167.

Pal, S. and T.-K. L. Wong (2013). Energy, entropy, and arbitrage. *arXiv preprint arXiv:1308.5376*.

Pal, S. and T.-K. L. Wong (2014). The geometry of relative arbitrage. *arXiv preprint arXiv:1402.3720*.

`applestarbucks` *Monthly Stock Prices of Apple and Starbucks from Jan 1994 to Apr 2012*

Description

This data set gives the monthly stock prices of Apple and Starbucks, in US dollars, from Jan 1994 to Apr 2012.

Usage

```
applestarbucks
```

Format

A zoo object containing 220 observations.

`AtlasModel` *Atlas Model*

Description

The function `AtlasModel` is used to create a `marketmodel` object which represents an Atlas model with user-provided parameters.

Usage

```
AtlasModel(n, g, sigma)
```

Arguments

`n`	a positive integer representing the number of stocks in the market
`g`	a positive number. In this model, `n*g` is the growth rate of the smallest stock.
`sigma`	a positive number representing the common volatility of the stocks.

Details

The definition of the Atlas model follows Example 5.3.3 in Fernholz (2002). The stochastic differential equation of the market capitalizations X_i(t) for the i-th stock takes the form

dlog X_i(t) = gamma_i(t)dt + sigma dW_i(t), i = 1, ..., n,

where gamma_i(t) = ng if stock i is the smallest, and is 0 otherwise.

It is the simplest market model which exhibits an asymptotic Pareto-shaped capital distribution curve.

Value

A `marketmodel` object.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

`marketmodel`, `SimMarketModel`

Examples

```
# Create an Atlas model of 100 stocks
model <- AtlasModel(n = 100, g = 0.001, sigma = 0.2)

# Simulate the Atlas model to get 5 years of monthly data
market <- SimMarketModel(model, n.years = 5, frequency = 12)
```

---

capdist *Captial-distribution Objects*

Description

The function `capdist` is used to create capital-distribution objects.

Usage

```
capdist(x)
```

Arguments

`x`	a numeric vector storing the market weights of assets.

Details

The function `capdist` is used to create capital-distribution objects. These are vectors with class `"capdist"` which represents the distribution of capital in a market at a single moment.

When the function is called, it is checked whether the sum of weights is equal to 1 up to 0.1%. If not, a warning message will be displayed.

Capital distribution objects can be plotted. See `plot.capdist` for details.

Value

A `capdist` object.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

`plot.capdist`

Examples

```
# Create a random distribution
x <- runif(100)
x <- x/sum(x)
x <- capdist(x)
plot(x)
```

`CapDistSlope` *Slope of a Capital Distribution Curve*

Description

The function `CapDistSlope` computes the slope of a capital distribution curve.

Usage

```
## S3 method for class "capdist"

CapDistSlope(x, cut.end = 0.1)
```

Arguments

`x`	a `capdist` object.
`cut.end`	A number between 0 and 1. It is the proportion of the smallest assets that will be ignored when fitting a straight line to the capital distribution curve. The default value is 0.1.

Details

If the capital distribution curve is a straight line, we say that the capital distribution follows a Pareto distribution. Empirically, a lot of capital distribution curves are approximately Pareto shaped.

The slope is estimated by ordinary least squares after removing the smallest `cut.end` proportion of the weights.

Value

The slope.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

`capdist`

Examples

```
# Create a random distribution
x <- runif(100)
x <- x/sum(x)
x <- capdist(x)
CapDistSlope(x)
```

`ConstantPortfolio` *Constant-weighted Portfolio*

Description

The function `ConstantPortfolio` is creates an `fgp` object representing the constant-weighted portfolio with a given weight vector.

Usage

```
ConstantPortfolio(weight)
```

Arguments

`weight`	a numeric probability vector.

Details

The constant-weighetd portfolio is a functionally generated portfolio generated by the geometric mean (see `GeometricMean`). One example is the equal-weighted portfolio. The portfolio maintains the same weight in every period.

Value

An fgp object.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

GeometricMean

Examples

```
# Define the constant-weighted portfolio (0.2, 0.3, 0.5) for 3 stocks
portfolio <- ConstantPortfolio(c(0.2, 0.3, 0.5))
```

Diversity *Diversity*

Description

The function Diversity computes the diversity of a probability distribution.

Usage

```
Diversity(x, p = 0.5)
```

Arguments

x	a numeric vector with non-negative entries.
p	a parameter between 0 and 1. The default value is 0.5.

Details

The diversity function is used in stochastic portfolio theory as a measure of market diversity. It is the generating function of the diversity-weighted portfolio (see DiversityPortfolio). See Example 3.4.4 of Fernholz (2002) for more information.

Value

A number.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

DiversityPortfolio

Examples

```
x <- c(0.3, 0.2, 0.5)
Diversity(x, p = 0.7)
```

DiversityPortfolio *Diversity-weighted Portfolio*

Description

The function DiversityPortfolio is creates an fgp object representing the diversity-weighted portfolio with a given parameter.

Usage

```
DiversityPortfolio(p = 0.5)
```

Arguments

p	a number between 0 and 1. The default value is 0.5.

Details

The diversity-weighetd portfolio is a functionally generated portfolio, see Example 3.4.4 of Fernholz (2002). It is generated by the diversity function (see Diversity). It can be regarded as an interpolation between the market portfolio (p = 1) and the equal-weighted portfolio (p = 0).

Value

An fgp object.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

Diversity

Examples

```
# Create a diversity-weighted portfolio with p = 0.7
portfolio <- DiversityPortfolio(p = 0.7)
```

EEControl *Control Term in the Energy-entropy Decomposition*

Description

The function computes the control term of the energy-entropy decomposition.

Usage

```
EEControl(pi.current, pi.next, nu.next, nu.implied = nu.next)
```

Arguments

pi.current	a numeric vector of the current portfolio weights.
pi.next	a numeric vector of the portfolio weights for the next period.
nu.next	a numeric vector of the benchmark weights for the next period.
nu.implied	a numeric vector of the implied benchmark weights. The default value is nu.next (in this case the benchmark is buy-and-hold).

Details

The control term measures how much the portfolio moves towards the current market weights. For details, see Section 2 of Pal and Wong (2013). Here the formula is modified slightly so that the energy-entropy decomposition holds identically whether the market is buy-and-hold or not.

Value

A number.

References

Pal, S. and T.-K. L. Wong (2013). Energy, entropy, and arbitrage. *arXiv preprint arXiv:1308.5376*.

See Also

FreeEnergy, RelativeEntropy, EnergyEntropyDecomp

Examples

```
pi.current <- c(0.2, 0.3, 0.5)
pi.new <- c(0.3, 0.3, 0.4)
mu.new <- c(0.5, 0.3, 0.2)

EEControl(pi.current, pi.new, mu.new)
```

EnergyEntropyDecomp *Energy-entropy Decomposition*

Description

The function EnergyEntropyDecomp computes and plots the energy-energy decomposition of any portfolio relative to the benchmark portfolio.

Usage

```
EnergyEntropyDecomp(market, weight, grouping = NULL, plot = TRUE)
```

Arguments

market	an object of class toymkt.
weight	a matrix or dataframe of portfolio weights. Each row represents a vector of portfolio weights.
grouping	a numeric vector of positive integers taking values from 1 to m, where m is the number of groups. An example is c(1, 1, 2, 3, 2, 3, 1), where m is 3. If grouping is given, the hierachical energy-entropy decomposition will be performed.
plot	TRUE or FALSE. If TRUE, the energy-entropy decomposition will be plotted. The default value is TRUE.

Details

The energy-entropy decomposition decomposes the excess log return of a portfolio (with respect to the benchmark) into three terms: free energy, control and change in relative entropy. See Section 2 of Pal and Wong (2013) for details. It is important to note that Pal and Wong (2013) assumes that the benchmark is a buy-and-hold portfolio, so that market$buy.and.hold is TRUE. The decomposition is modified so that an identity holds even when the market is not buy-and-hold. However in that case the control term in the decomposition is harder to interpret.

A portfolio can sometimes be thought of as a portfolio of portfolios, and the energy-entropy decomposition has a corresponding hierchical decomposition, see Section 3 of Pal and Wong (2013). If grouping is provided, the hierachical decomposition will be performed and plotted. An example of grouping is a label for sectors (say 1: financial, 2: utility, 3:energy, etc). For more details see the supplementary files available on the author's website.

Value

If grouping is not provided, it is a zoo object with the followign columns. The definitions of the terms can be found in Section 2 of Pal and Wong (2013). Each term represents an increment for the period.

Excess log return
: relative log return.

`Free energy` free energy.

`Relative entropy`
minus of the change of relative entropy.

`Control` control.

`Drift` drift. It equals free energy + control.

If `grouping` is provided, it is a list containing several zoo objects:

`dlogV` relative log return.

`free.energies` free energy and its decomposition.

`relative.entropies`
relative entropy and its decomposition.

`control` control and its decomposition.

References

Pal, S. and T.-K. L. Wong (2013). Energy, entropy, and arbitrage. *arXiv preprint arXiv:1308.5376*.

Author's website: http://www.math.washington.edu/~wongting/

See Also

FreeEnergy, RelativeEntropy, EEControl,

Examples

```
# Example 1
# Energy-entropy decomposition for the entropy-weighted portfolio
data(applestarbucks)
market <- toymkt(applestarbucks, initial.weight = c(0.5, 0.5))
weight <- GetWeight(market, EntropyPortfolio$weight.function)
decomp <- EnergyEntropyDecomp(market, weight, plot = TRUE)


# Example 2
# Example of a hierchical decomposition of the entropy-weighted
# portfolio in the Atlas market model
model <- AtlasModel(n = 6, g = 0.1, sigma = 0.2)
market <- SimMarketModel(model)  # default settings
grouping <- c(1, 1, 2, 2, 2, 2)
weight <- GetWeight(market, EntropyPortfolio$weight.function)
decomp <- EnergyEntropyDecomp(market, weight, grouping, plot = TRUE)
```

EntropyPortfolio *Entropy-weighted Portfolio*

Description

EntropyPortfolio is a predefiend fgp object representing the entropy-weighted portfolio.

Usage

```
EntropyPortfolio
```

Details

The entropy-weighted portfolio is a functionally generated portfolio generated by the Shannon entropy (see ShannonEntropy). See Example 3.1.2 of Fernholz (2002).

Value

An fgp object.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

ShannonEntropy

FernholzDecomp *Fernholz's Decomposition for a Buy-and-hold Toy Market*

Description

The function FernholzDecomp computes the Fernholz decomposition of a functionally generated portfolio.

Usage

```
FernholzDecomp(market, portfolio, plot = TRUE)
```

Arguments

market	a toymkt object where market$buy.and.hold is TRUE.
portfolio	an fgp object represents a functionally generated portfolio.
plot	TRUE or FALSE. If TRUE, the Fernholz decomposition will be plotted. The default value is TRUE.

Details

Functionally generated portfolios (see `fgp`) and Fernholz's decomposition play an important role in stochastic portfolio theory.

In this context, the benchmark portfolio is a buy-and-hold market portfolio (modeled as a `toymkt` object in this package). The portfolio weights of the benchmark is represented by the market weight $\mu(t)$. The portfolio manager chooses portfolio weights $\pi(t)$.

Let $V(t)$ be the growth of \$1 of the manager's portfolio divided by that of the benchmark. Initially $V = 1$ and it increases when the manager outperforms the market. The quantity $log(V(t))$ is called the relative log return.

For a functionally generated portfolio, the portfolio weights $\pi(t)$ are deterministic functions of the current market weight $\mu(t)$. More precisely, the portfolio weights $\pi(t)$ are given by the derivatives of $log(\Phi)$ at $\mu(t)$, where Φ is called the generating function. See the references for more details (the notation here follows that of Pal and Wong (2014)).

In this case, Fernholz's decomposition states that the relative log return has the decomposition

$$log(V(t)) = [log(\Phi(\mu(t))) - log(\Phi(\mu(0)))] + \Theta(t),$$

where $\Theta(t)$ is called the drift process. Note that the first term depends only on the current and initial positions of the market weights. The drift process is determined by the portfolio and the cumulative amount of market volatility. This is the decomposition implemented by the function `FernholzDecomp`. See also Chapter 6 of Fernholz (2002).

Value

A list containing the following components.

`portfolio`	growth of \$1 of the portfolio.
`benchmark`	growth of \$1 of the market portfolio.
`relative.return`	cumulative excess log return with respect to the market portfolio.
`generating function`	change of the log of the generating function.
`drift`	drift process.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

Pal, S. and T.-K. L. Wong (2014). The geometry of relative arbitrage. *arXiv preprint arXiv:1402.3720*.

See Also

`fgp`

Examples

```
# Plot the Fernholz decomposition for the entropy-weighted portfolio
data(applestarbucks)
market <- toymkt(applestarbucks)
output <- FernholzDecomp(market, EntropyPortfolio, plot = TRUE)
```

fgp *Functionally Generated Portfolio Objects*

Description

The function fgport is used to create functionally generated portfolio objects.

Usage

```
fgp(name, gen.function, weight.function)
```

Arguments

name	a string which is the name of the portfolio (e.g. "Equal-weighted portfolio"")
gen.function	the generating function. It is a smooth function of the market weights (see Theorem 3.1.5 of Fernholz (2002)).
weight.function	the weight function of the portfolio. It must be related to gen.function via (3.1.7) of Fernholz (2002).

Details

The function fgp is used to create functionally generated portfolio objects. Detailed treatments of functionally generated portfolios can be found in Chapter 3 of Fernholz (2002) as well as Pal and Wong (2014). In brief, a functionally generated portfolio is determined by a generating function (gen.function) and the weight function (weight.function).

Some common functionally generated portfolios are provided in the package. For example, EntropyPortfolio is the entropy-weighted, DiversityPortfolio is a constructs fgp objects for the diversity-weigthed portfolio, and ConstantPortfolio defines constant-weighted portfolios.

The most important operation concerning fgp objects is Fernholz's decomposition. See FernholzDecomp for more details.

Value

name	a character string, representing the name of the portfolio
gen.funtion	a function object, representating the generating function
weight.function	a function object, representing the weight function

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

Pal, S. and T.-K. L. Wong (2014). The geometry of relative arbitrage. *arXiv preprint arXiv:1402.3720*.

See Also

FernholzDecomp

Examples

```
# Example 1: The diversity-weighted portfolio with p = 0.5
# This has the same effect has DiversityPortfolio(p = 0.5)
my.portfolio <- fgp("Diversity-Weighted Portfolio, p = 0.5",
                    function(x) (sum(sqrt(x)))^2,
                    function(x) sqrt(x) / sum(sqrt(x)))


# Example 2: A quadratic Gini coefficient
# See Example 3.4.7 of Fernholz (2002)

# Generating function
gen.function <- function(x) {
  n <- length(x)
  return(1 - sum((x - 1/n)^2)/2)
}

# Weight function
weight.function <- function(x) {
  n <- length(x)
  S <- gen.function(x)
  return(((1/n - x)/S + 1 - sum(x*(1/n - x)/S))*x)
}

# Define fgp object
my.portfolio <- fgp("Quadratic Gini",
                    gen.function, weight.function)

# Its performance in the apple-starbucks market
data(applestarbucks)
market <- toymkt(applestarbucks)
result <- FernholzDecomp(market, my.portfolio, plot = TRUE)
```

FreeEnergy *Free Energy*

Description

The function FreeEnergy computes the free energy of a portfolio given the simple returns of the individual assets.

Usage

```
FreeEnergy(pi, R, group.index = NULL)
```

Arguments

pi	a numeric vector of portfolio weights (a probability vector).
R	a numeric vector of simple returns.
group.index	if provided, the free energy will be decomposed according to the chain rule (see below for more details). The default is NULL. For the format of group.index see the example in GetGroupWeight.

Details

The free energy equals the portfolio log return minus the weighted average log return of the individual assets, see Definition 2.2 of Pal and Wong (2013). It is a weighted measure of the cross volatility of the market.

If group.index is provided the free energy will be decompoesd using the chain rule stated in Lemma 3.1(ii) of Pal and Wong (2013), see equation (24) there. In this case the output has 1 + 1 + m components, where m is the number of groups defined by group.index. The first component is the left-hand-side of (24). The second component is the first term on the right-hand-side of (24). The other m components are the terms in the sum on the right-hand-side of (24).

Value

A non-negative number or +Inf if group.index is not given. A numeric vector if group.index is given.

References

Pal, S. and T.-K. L. Wong (2013). Energy, entropy, and arbitrage. *arXiv preprint arXiv:1308.5376*.

See Also

EnergyEntropyDecomp

Examples

```
pi <- c(1/3, 1/3, 1/3)  # portfolio weights
R <- c(0.1, 0.02, -0.05)  # simple returns

FreeEnergy(pi, R)
```

GeometricMean *Geometric Mean*

Description

The function GeometricMean computes the geometric mean of non-negative numbers with a given weight.

Usage

```
GeometricMean(x, weight = rep(1/length(x), length(x)))
```

Arguments

x	a numeric vector with non-negative entries.
weight	a numeric probability vector. The default value is the vector with equal weights.

Details

The geometric mean is used in stochastic portfolio theory as the generating function of the constant-weighted portfolio. See Example 3.1.6 of Fernholz (2002).

Value

A number.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

ConstantPortfolio

Examples

```
x <- c(0.3, 0.7)
weight <- c(0.6, 0.4)
GeometricMean(x, weight)
```

GetGroupWeight *Computing First and Second Level Portfolio Weights*

Description

Given a portfolio weight vector and a grouping, the function GetGroupWeight decompose the portfolio into a portfolio of portfolios. See the decription below for more details.

Usage

```
GetGroupWeight(pi, group.index)
```

Arguments

pi	a portfolio weight vector.
group.index	a list whose components form a partition of 1, 2, ..., n, where n is the length of pi.

Details

This function is mainly intended to be used internally in EnergyEntropyDecomp.

We explain the main idea with an example. For more details, see Section 3 of Pal and Wong (2013). Consider the portfolio pi = (0.1, 0.2, 0.3, 0.4) for four stocks. To fix ideas, suppose the first two stocks are financial stocks, and the other two are utility stocks. The first two stocks can be regarded to form a "sub-portfolio" of financial stocks, and the other two form a sub-portfolio of utility stocks. Hence, the portfolio pi can be regarded as a portfolio of two portfolios.

The portfolio puts 0.3 weight into financial stocks and 0.7 weight into utility stocks. Hence the first level portfolio weights are 0.3 and 0.7. Restricting to the finanical sector, the second level portfolio weights are 0.1/(0.1 + 0.2) and 0.2/(0.1 + 0.2) respectively. Hence the (normalized) weights for the financial sector portfolio are 1/3 and 2/3. Similarly, the weights for the utility sector portfolio are 3/7 and 4/7. The function GetGroupWeight performs this decomposition.

Value

A list containing the following components.

group.weight	a vector of first level portfolio weights.
cond.weight	a list whose components are the second level portfolio weights.

References

Pal, S. and T.-K. L. Wong (2013). Energy, entropy, and arbitrage. *arXiv preprint arXiv:1308.5376*.

See Also

FreeEnergy, RelativeEntropy

Examples

```
# The example described above
pi <- c(0.1, 0.2, 0.3, 0.4)

group.index <- list()
group.index[[1]] <- c(1, 2)
group.index[[2]] <- c(3, 4)

GetGroupWeight(pi, group.index)
```

GetLambdaWeight *Portfolio Weights of the Lambda-strategy*

Description

Given a toymkt and an initial weight vector, the function GetLambdaWeight computes a matrix of portfolio weights following the lambda strategy.

Usage

```
GetLambdaWeight(market, initial.weight = market$benchmark.weight[1, ], lambda)
```

Arguments

market	a toymkt object with toymkt$buy.and.hold = TRUE.
initial.weight	the initial portfolio weights. The default is the initial benchmark weights.
lambda	a number between 0 and 1.

Details

For details see GetNewLambdaWeight and Section 6.1 of Pal and Wong (2013). The purpose of this function is analogous to that of GetWeight, but here the strategy depends on the entire history of market weights.

Value

A matrix of portfolio weights.

References

Pal, S. and T.-K. L. Wong (2013). Energy, entropy, and arbitrage. *arXiv preprint arXiv:1308.5376*.

See Also

GetNewLambdaWeight, EnergyEntropyDecomp

Examples

```
data(applestarbucks)
market <- toymkt(applestarbucks)
weight <- GetLambdaWeight(market, initial.weight = c(0.5, 0.5), lambda = 0.2)
decomp <- EnergyEntropyDecomp(market, weight, plot = TRUE)
```

GetNewLambdaWeight *Portfolio Weights of the Lambda-strategy*

Description

The function computes the portfolio weights of the lambda strategy.

Usage

```
GetNewLambdaWeight(pi.current, mu.next, energy, lambda = 0.5)
```

Arguments

pi.current	current portfolio weight.
mu.next	market weights for the next period.
energy	a non-negative number. The free energy of the previous period.
lambda	a number between 0 and 1. The default is 0.5.

Details

The lambda-strategy is a simple energy-entropy strategy that depends on a parameter lambda between 0 and 1, see Section 6.1 of Pal and Wong (2013) for more details. Given the portfolio weights of the previous time period, the portfolio weights of the next period is a convex combination of the previous weights and the current market weights, chosen in such a way that the drift term is a constant proportion of the free energy.

In this implementation an SDE approximation is used. Hence the function is accurate only when lambda is small (say <= 0.5).

Value

A portfolio weight vector.

References

Pal, S. and T.-K. L. Wong (2013). Energy, entropy, and arbitrage. *arXiv preprint arXiv:1308.5376*.

See Also

GetLambdaWeight

Examples

```
pi.previous <- c(1/3, 1/3, 1/3)
mu.next <- c(0.5, 0.3, 0.2)
energy <- 0.05
lambda <- 0.3

GetNewLambdaWeight(pi.previous, mu.next, energy, lambda)
```

GetWeight *Computing Portfolio Weights from a Weight Function*

Description

Given the benchmark weights (say, derived from a toy market, see toymkt) and a function which maps benchmark weights to portfolio weights, the function GetWeight computes a matrix of portfolio weights.

Usage

```
GetWeight(market, weight.function, ...)
```

Arguments

market	a toymkt object or a matrix/dataframe of benchmarket weights.
weight.function	a function object representing the weight function. The weight function depends only on the current market weights.
...	Additional parameters of weight.function. See the example below.

Details

A probability vector is a numeric vector with non-negative entries summing to one. weight.function is a function which maps probability vectors to probability vectors. It represents a portfolio whose weights are deterministic functions of the current benchmark weights.

Value

A matrix where each row represents the portfolio weights for a period.

See Also

Invest

Examples

```
data(applestarbucks)
market <- toymkt(applestarbucks)

# This is the diversity-weighted portfolio
weight.function <- function(x, p) {
  return(x^p / sum(x^p))
}

weight <- GetWeight(market, weight.function, p = 0.7)
Invest(market, weight)
```

Invest *Investing in a Toy Market*

Description

Given the portfolio weights and a toy market, the function Invest simulates the growth of $1 of the corresponding portfolio and that of the benchmark.

Usage

```
Invest(market, weight, plot = TRUE)
```

Arguments

market	a toymkt object.
weight	the portfolio weights. The portfolio weights must be non-negative and sum to one (full investment with no short sales). It can be a zoo object or matrix/dataframe whose number of rows is at least as large as that of market$R. If the number of rows of weight is larger than required, only the initial rows will be used. weight can also be a numeric vector whose length is equal to the number of columns of market$R (the number of assets). In the latter case the portfolio is assumed to be constant-weighted through out.
plot	TRUE or FALSE. If TRUE, the growth of $1 of the portfolio will be plotted together with the growth of $1 of the market portfolio. The default value is TRUE.

Details

The relative value in the second plot is the ratio of the growth of $1 of the portfolio to that of the benchmark. It is called relative because the value is normalized by the value of the benchmark portfolio.

Value

A list containing the following components.

growth	a zoo object representing the growth of $1 of the portfolio and the benchmark.
R	a zoo object of simple returns of the two portfolios.
r	a zoo object of log returns of the two portfolios.

See Also

toymkt

Examples

```
# Performance of the equal-weighted portfolio
data(applestarbucks)
market <- toymkt(applestarbucks)
weight <- c(0.5, 0.5)  # equal-weighted portfolio
result <- Invest(market, weight, plot = TRUE)
```

`marketmodel` *Market Model Objects*

Description

The function `marketmodel` is used to create market model objects.

Usage

```
marketmodel(name, n, gamma, diffmatrix, diag = FALSE)
```

Arguments

`name`	a string which is the name of the market model (e.g. Atlas model).
`n`	an integer which is the number of assets in the model.
`gamma`	represents the instantaneous growth rates. It can be a function or a numeric vector. In the latter case the growth rates are constant.
`diffmatrix`	represents the diffusion matrix of the model. There are four possibilities: (i) a numeric vector, (ii) a matrix, (iii) a vector-valued function, (iv) a matrix-valued function. In cases (i) and (iii), the matrix is diagonal. The dimensions of the matrix are `n` times `n`. In the first two cases the diffusion matrix is constant.
`diag`	if `TRUE`, the diffusion matrix is diagonal. The default value is `FALSE`.

Details

A market model is defined by a system of stochastic differential equations (SDE) of the form

dlog X_i(t) = gamma_i(t)dt + Sigma(t) dW(t), i = 1, ..., n,

where gamma_i(t) are the growth rates (represented by `gamma`), Sigma is the n times n diffusion matrix (represented by `diffmatrix`), and W(t) is an n-dimensional standard Brownian motion. The unit of time is annual. We assume that the coefficients are functions of the X_i's, i.e., the system is Markovian. Note that the SDE are stated in terms of the logarithms of the X_i's.

The Atlas model (see `AtlasModel`) and the volatility-stablized market model (see `VolStabModel`) are predefined in the package. More general rank-based models can be constructed as `marketmodel` objects using this function.

The function `SimMarketModel` is used to simulate a market model.

At present no effort is put to check the consistency among the inputs (e.g. inconsistent dimensions).

Value

`name`	the name of the model.
`gamma`	the growth rate function.
`diffmatrix`	the diffusion matrix which can be a constant or a function (see above for the details).
`n`	the number of stocks in the model.
`diag`	`TRUE` or `FALSE` as supplied by the user.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

AtlasModel, SimMarketModel, VolStabModel

Examples

```
# Create a model of two independent geometric Brownian motions
model1 <- marketmodel(name = "GBM", n = 2,
                      gamma = c(0.1, 0.05),
                      diffmatrix = c(0.1, 0.2),
                      diag = TRUE)

# Create an Atlas model of 100 stocks
model2 <- AtlasModel(n = 100, g = 0.0001, sigma = 0.1)

# Create a Volatility stabilized market of 10 stocks
model3 <- VolStabModel(n = 10, alpha = 0.001, sigma = 0.01)
```

ParetoCapDist *Generating a Pareto Capital Distribution*

Description

The function ParetoCapDist is used to generate a capital-distribution object which follows a Pareto distribution with user-defined slope parameter.

Usage

```
ParetoCapDist(n, index = 1)
```

Arguments

n	the number of assets.
index	a positive number. The slope of the capital distribution curve (i.e., the log-log curve of market weights against ranks) is -index.

Details

A capital distribution is said to follow a Pareto distribution if the log-log curve of market weights against ranks is linear. This function creates a hypothetitical market distribution given the slope of the curve.

Value

a capdist object.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

capdist

Examples

```
x <- ParetoCapDist(n = 100, index = 1.1)

plot(x)
```

`plot.capdist` *Plotting Captial Distribution Objects*

Description

The function `plot.capdist` plots the capital distribution curve.

Usage

```
## S3 method for class 'capdist'
plot(x, draw.line = TRUE, cut.end = 0.1, ...)
```

Arguments

`x`	a `capdist` object.
`draw.line`	`TRUE` or `FALSE`. If true, a straight line is fitted to the capital distribution curve. The default value is `TRUE`.
`cut.end`	A number between 0 and 1. It is the proportion of the smallest assets that will be ignored when fitting a straight line to the capital distribution curve. The default value is 0.1.
`...`	further arguments such as `main`.

Details

A capital distribution curve is the log-log curve of the market weights against the rank. Empirically, capital distribution curves are approximately Pareto-shaped with some concavity at the lower end.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

capdist

Examples

```
# Create a random distribution
x <- runif(100)
x <- x/sum(x)
x <- capdist(x)
plot(x)
```

`plot.toymkt` *Plotting Toy Market Objects*

Description

The function `plot.toymkt` produces several plots of a `toymkt` object.

Usage

```
## S3 method for class 'toymkt'
plot(x, ...)
```

Arguments

`x`	a `toymkt` object.
`...`	further arguments such as `main`.

Details

Four plots are produced: (i) time series of prices, (ii) time series of log returns, (iii) time series of benchmark weights, and (iv) time series of Shannon entropy (see `ShannonEntropy`) of the benchmark weights. Default settings are used for these plots, so they may not be informative if the number of assets is large.

See Also

`toymkt`

Examples

```
data(EuStockMarkets)
market <- toymkt(price = EuStockMarkets)
plot(market)
```

print.capdist *Printing Capital Distribution Objects*

Description

The function `print.capdist` prints a capital distribution objects.

Usage

```
## S3 method for class 'capdist'
print(x, m = 5L, cut.end = 0.1, ...)
```

Arguments

`x`	a `capdist` object.
`m`	a positive integer which is at most half of the length of `x`. The largest and smallest `m` assets will be printed. The default value is 5.
`cut.end`	a number between 0 and 1. It is the proportion of the smallest assets that will be ignored when fitting a straight line to the capital distribution curve. The default value is 0.1.
`...`	further arguments.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

`capdist`

print.fgp *Printing Functionally Generated Portfolio Objects*

Description

The function `print.fgp` prints a functionally generated portfolio object.

Usage

```
## S3 method for class 'fgp'
print(x, ...)
```

Arguments

`x`	an `fgp` object.
`...`	futher arguments.

Details

The function prints the name of the portfolio and the names of the components.

See Also

fgp

`print.marketmodel` *Printing Market Model Object*

Description

The function `print.marketmodel` prints a `marketmodel` object.

Usage

```
## S3 method for class 'marketmodel'
print(x, ...)
```

Arguments

`x`	a `marketmodel` object.
`...`	further arguments.

Details

The function prints the name, number of assets, and the names of the components.

See Also

marketmodel

`print.toymkt` *Printing Toy Market Object*

Description

The function `print.toymkt` prints a `toymkt` object.

Usage

```
## S3 method for class 'toymkt'
print(x, ...)
```

Arguments

x	a toymkt object.
...	further arguments.

Details

The number of assets is printed together with their names. This is followed by the components of the object.

See Also

toymkt

Examples

```
data(EuStockMarkets)
market <- toymkt(EuStockMarkets)
market
```

RelativeEntropy *Relative Entropy*

Description

The function RelativeEntropy is used to compute the relative entropy between two probability distributions.

Usage

```
RelativeEntropy(p, q, group.index = NULL)
```

Arguments

p	a numeric vector representing a probability distribution.
q	a numeric vector representing a probability distribution. p and q must have the same length.
group.index	if provided, the relative entropy will be decomposed according to the chain rule (see below for more details). The default is NULL. For the format of group.index see the example in GetGroupWeight.

Details

Relative entropy can be thought of as a measure of distance between two probability distributions. It is also known as the Kullback-Leibler divergence and is usually denoted by H(p|q). It is not a metric as it is not symmetric and it does not satisfy the triangle inequality.

If there is an index `i` where `q[i] == 0` but `p[i] > 0`, then the relative entropy is `Inf`. Mathematically, this happens when `p` is not absolutely continuous with respect to `q`.

If `group.index` is provided the relative entropy will be decompoesd using the chain rule stated in Lemma 3.1(i) of Pal and Wong (2013), see equation (23) there. In this case the output has `1 + 1 + m` components, where `m` is the number of groups defined by `group.index`. The first component is the left-hand-side of (23). The second component is the first term on the right-hand-side of (23). The other `m` components are the terms in the sum on the right-hand-side of (23).

Value

A non-negative number or `+Inf` if `group.index` is not given. A numeric vector if `group.index` is given.

References

Pal, S. and T.-K. L. Wong (2013). Energy, entropy, and arbitrage. *arXiv preprint arXiv:1308.5376*.

See Also

ShannonEntropy

Examples

```
p <- c(0.3, 0.3, 0.4)
q <- c(0.5, 0.3, 0.2)

RelativeEntropy(p, q)
RelativeEntropy(q, p)  # relative entropy is not symmetric
```

RenyiEntropy *Renyi Entropy*

Description

The function `Renyi` computes the Renyi entropy of a probability vector.

Usage

```
RenyiEntropy(x, p)
```

Arguments

`x`	a numeric vector representing a probability distribution.
`p`	a number not equal to 1. The default is 0.5.

Details

The Renyi entropy is a generalization of Shannon entropy. See Example 3.4.8 of Fernholz (2002) for more details.

Value

A number.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

ShannonEntropy, Diversity

Examples

```
# compute the Renyi entropy of the country weights.
x <- c(0.2, 0.3, 0.5)
RenyiEntropy(x, p = 0.5)
```

ShannonEntropy *Shannon Entropy*

Description

The function ShannonEntropy computes the Shannon entropy of a probability vector.

Usage

```
ShannonEntropy(x)
```

Arguments

x	a numeric probability vector (a vector with non-negative entries summing to one).

Details

Shannon entropy is a measure of uncertainty. It is maximized when the distribution is uniform, and is zero if it is a point mass. It is used in stochastic portfolio theory as a measure of market diversity. It is also the generating function of the entropy-weighted portfolio (see EntropyPortfolio). See Examples 3.1.2 and 3.4.3 of Fernholz (2002) for more information.

It will be checked whether the input x is reasonably close to a probability vector. If some entries are negative or the sum of the entries is not close enough to 1 (the error margin is 0.01), an error message will be displayed.

Value

A number.

References

Fernholz, E. R. (2002) *Stochastic portfolio theory*. Springer.

See Also

EntropyPortfolio, RelativeEntropy

Examples

```
x <- c(1/3, 1/3, 1/3)
ShannonEntropy(x)  # equals log(3)
```

SimMarketModel *Simulating a Market Model*

Description

The function SimMarketModel is used to simulate a market model defined by a marketmodel object. The output is a toymkt object with which further analysis can be performed.

Usage

```
SimMarketModel(model, n.years = 10, frequency = 12,
               initial.weight = rep(1/model$n, model$n),
               sub.freq = 1)
```

Arguments

model	a marketmodel object.
n.years	an integer which is the number of years of data to be simulated. The default value is 10.
frequency	an integer which is the number of periods for each year. The default value is 12, i.e., monthly data is generated.
initial.weight	a numeric vector of positive numbers representing the initial weights of each stock. The default value is rep(1/model$n, model$n), i.e., the market is equal-weighted initially.
sub.freq	a positive integer, which is the number of subperiods within each period. The default value is 1. This is included to allow more accurate simulation of models defined by stochastic differential equations.

Details

The function SimMarketModel simulates a given market model with user-defined parameters. See marketmodel for the definition of a market model.

The option sub.freq does the following. Suppose we set frequency = 12 and sub.freq = 4. Although the output is monthly prices, during the simulation, for each month the algorithm divides in month into 4 subperiods (say 4 weeks). In other words, the actual time step in the simulation is

dt = 1 / frequency * subfreq,

and the output shows only the prices sampled monthly. This feature allows more accurate simulation of rank-based models, where multiple changes in rankings can happen within each sampling period.

When the aim is to simulate the evolution of capital distribution curves, the initial market weights play an important role. The default option is equal-weighting. To start the market at stationarity, there are three ways to proceed:

1) Remove an initial segment of the output.

2) Perform two simulations, where in the second simulation the market is started at the ending distribution of the first simulation.

3) Use the long term distribution (say Pareto with a certain slope parameter) directly if it is known. A possibility is to use ParetoCapDist.

Value

A toymkt object containing the simulated market. For this object buy.and.hold is TRUE.

See Also

AtlasModel, marketmodel, VolStabModel

Examples

```
# Create an Atlas model of 5 stocks
model <- AtlasModel(n = 5, g = 0.05, sigma = 0.1)

# Simulate the model to get 20 years of monthly data
# with initial weights c(0.1, 0.2, 0.2, 0.2, 0.3)
market <- SimMarketModel(model, n.years = 20,
                         initial.weight = c(0.1, 0.2, 0.2, 0.2, 0.3),
                         frequency = 12)
plot(market)
```

---

toymkt *Toy Market Objects*

Description

The function toymkt is used to create toy market objects.

Usage

```
toymkt(price = NULL, R = NULL,
       benchmark.weight = NULL, initial.weight = NULL,
       buy.and.hold = TRUE)
```

Arguments

price	A zoo object containing the price or market captitalizations of the assets. Matrix and dataframes are also accepted.
R	Simple returns of the assets (same format as price). At least one of price and R must be supplied.
benchmark.weight	Portfolio weights of the benchmark (same format as price).
initial.weight	a numeric vector of initial benchmark weights. All entries of initial.weight must be non-negative. If initial.weight is not supplied and buy.and.hold is TRUE, the initial weights will be equal-weighted by default. If buy.and.hold is FALSE, initial.weight will not be used.
buy.and.hold	If TRUE, the benchmark is a buy-and-hold portfolio, and benchmark.weight will be ignored even if it is supplied. If FALSE, benchmark.weight must be supplied. The default value is TRUE.

Details

The function toymkt is used to create toy market objects from prices, returns and/or benchmark weights. The universe consists of (say) n assets. For each asset, we have a time series of returns. A benchmark portfolio is given by a fixed set of portfolio weights, and we want to study the performances of portfolios with respect to this benchmark. The portfolio weights of the benchmark are called the benchmark weights.

If buy.and.hold is TRUE, the toy market is an idealized market where the benchmark is a buy-and-hold portfolio. In this case all data in the output is derived from price and initial.weight, or R and initial.weight. If buy.and.hold is TRUE, initial.weight is not given but benchmark.weight is given, the first row of benchmark.weight will be used as the initial weight vector.

In the case buy.and.hold is FALSE the benchmark portfolio can be quite arbitrary. It is defined in terms of the beginning benchmark weights for each period and the returns for each asset.

Value

A list containing the following components:

growth	a zoo object containing the growth of $1 for each asset.
R	a zoo object containing the simple returns.
r	a zoo object containing the log returns.
benchmark.weight	a zoo object containing the benchmark weights of the assets. If buy.and.hold = TRUE, these are derived from the normalized market capitalizations.
n	number of assets in the market.
buy.and.hold	TRUE if the market is buy-and-hold.

See Also

print.toymkt, plot.toymkt

Examples

```
# We use the EuStockMarkets dataset in the datasets package
data(EuStockMarkets)

# Example 1: Minimal specifications
# The market will be equal-weighted initially.
market <- toymkt(price = EuStockMarkets)
print(market)
plot(market)  # several plots


# Example 2: Generate a market from simulated log-normal returns.
n.periods <- 60
n.stocks <- 5
mu <- 0
sigma <- 0.1
R <- matrix(0, nrow = n.periods, ncol = n.stocks)
for (j in 1:n.stocks) {
  R[, j] <- exp(rnorm(n.periods, mean = mu, sd = sigma)) - 1
}
initial.weight <- c(0.1, 0.2, 0.3, 0.2, 0.2)  # specify initial weights
market <- toymkt(R = R, initial.weight = initial.weight,
                 buy.and.hold = TRUE)
plot(market)
```

VolStabModel *Volatility-Stabilized Model*

Description

The function VolStabModel is used to create a marketmodel object which represents an volatility-stabilized market model with user-provided parameters.

Usage

```
VolStabModel(n, alpha, sigma)
```

Arguments

n	a positive integer representing the number of stocks in the market.
alpha	a non-negative number. It is a growth rate parameter.
sigma	a positive number. It is a volatility parameter.

Details

The definition of the volatility-stabilized model is taken from Section 12 of Fernholz and Karatzas (2009). The stochastic differential equation of the market capitalization X_i(t) of the i-th stock takes the form

dlog X_i(t) = gamma_i(t)dt + sigma_i(t) dW_i(t), i = 1, ..., n,

where

gamma_i(t) = alpha / 2mu_i(t)

and

sigma_i(t) = 1 / sqrt(mu_i(t)).

This is a model which captures the idea that smaller stocks have larger growth rates and are more volatile.

Value

A `marketmodel` object.

References

Karatzas, I. and R. Fernholz (2009). Stochastic portfolio theory: an overview. *Handbook of numerical analysis 15*, 89-167.

See Also

`marketmodel`, `SimMarketModel`

Examples

```
# Create a Volatility stabilized market of 10 stocks
model <- VolStabModel(n = 10, alpha = 0.1, sigma = 0.1)
```

Index

∗Topic **datasets**
applestarbucks, 3
∗Topic **package**
RelValAnalysis-package, 2

applestarbucks, 3
AtlasModel, 3, *23, 24*, *33*

capdist, 4, *6*, *25*, *27*
CapDistSlope, 5
ConstantPortfolio, 6, *14*, *17*

Diversity, 7, *8*, *31*
DiversityPortfolio, *7, 8*, 8, *14*

EEControl, 9, *11*
EnergyEntropyDecomp, *9*, 10, *16*, *18, 19*
EntropyPortfolio, 12, *14*, *31, 32*

FernholzDecomp, 12, *14, 15*
fgp, *13*, 14, *28*
FreeEnergy, *9*, *11*, 15, *18*

GeometricMean, *6, 7*, 16
GetGroupWeight, *16*, 17, *29*
GetLambdaWeight, 18, *20*
GetNewLambdaWeight, *19*, 19
GetWeight, *19*, 20

Invest, *21*, 21

marketmodel, *4*, 23, *28*, *33*, *36*

ParetoCapDist, 24
plot.capdist, *5*, 25
plot.toymkt, 26, *35*
print.capdist, 27
print.fgp, 27
print.marketmodel, 28
print.toymkt, 28, *35*

RelativeEntropy, *9*, *11*, *18*, 29, *32*
RelValAnalysis
(RelValAnalysis-package), 2
RelValAnalysis-package, 2
RenyiEntropy, 30

ShannonEntropy, *12*, *26*, *30, 31*, 31
SimMarketModel, *4*, *23, 24*, 32, *36*

toymkt, *20*, *22*, *26*, *29*, 33

VolStabModel, *23, 24*, *33*, 35