Package 'ForeCA'

January 20, 2025

Type Package

Title Forecastable Component Analysis

Version 0.2.7

Date 2020-06-21

URL https://github.com/gmgeorg/ForeCA

Description Implementation of Forecastable Component Analysis ('ForeCA'), including main algorithms and auxiliary function (summary, plotting, etc.) to apply 'ForeCA' to multivariate time series data. 'ForeCA' is a novel dimension reduction (DR) technique for temporally dependent signals. Contrary to other popular DR methods, such as 'PCA' or 'ICA', 'ForeCA' takes time dependency explicitly into account and searches for the most "forecastable" signal. The measure of forecastability is based on the Shannon entropy of the spectral density of the transformed signal.

Depends R (>= 3.5.0)

License GPL-2

Imports astsa (>= 1.10), MASS, graphics, reshape2 (>= 1.4.4), utils

Suggests psd, fBasics, knitr, markdown, mgcv, nlme (>= 3.1-64), testthat (>= 2.0.0), rSFA,

RoxygenNote 7.1.1

Encoding UTF-8

VignetteBuilder knitr

NeedsCompilation no

Author Georg M. Goerg [aut, cre]

Maintainer Georg M. Goerg <im@gmge.org>

Repository CRAN

Date/Publication 2020-06-29 12:40:42 UTC

Contents

`ForeCA-package` *Implementation of Forecastable Component Analysis (ForeCA)*

Description

Forecastable Component Analysis (ForeCA) is a novel dimension reduction technique for multivariate time series $\mathbf{X}_t$. ForeCA finds a linar combination $y_t = \mathbf{X}_t\mathbf{v}$ that is easy to forecast. The measure of forecastability $\Omega(y_t)$ (`Omega`) is based on the entropy of the spectral density $f_y(\lambda)$ of y_t: higher entropy means less forecastable, lower entropy is more forecastable.

The main function `foreca` runs ForeCA on a multivariate time series $\mathbf{X}_t$.

Consult `NEWS.md` for a history of release notes.

Author(s)

Author and maintainer: Georg M. Goerg <im@gmge.org>

References

Goerg, G. M. (2013). "Forecastable Component Analysis". Journal of Machine Learning Research (JMLR) W&CP 28 (2): 64-72, 2013. Available at http://jmlr.org/proceedings/papers/v28/goerg13.html.

Examples

```
XX <- ts(diff(log(EuStockMarkets)))
Omega(XX)

plot(log10(lynx))
Omega(log10(lynx))

## Not run:
ff <- foreca(XX, n.comp = 4)
ff
plot(ff)
summary(ff)

## End(Not run)
```

`common-arguments` *List of common arguments*

Description

Common arguments used in several functions in this package.

Arguments

`series`	a $T \times K$ array with `T` observations from the K-dimensional time series $\mathbf{X}_t$. Can be a `matrix`, `data.frame`, or a multivariate `ts` object.
`U`	a $T \times K$ array with `T` observations from the K-dimensional **whitened** (`whiten`) time series $\mathbf{U}_t$. Can be a `matrix`, `data.frame`, or a multivariate `ts` object.
`mvspectrum.output`	an object of class `"mvspectrum"` representing the multivariate spectrum of $\mathbf{X}_t$ (not necessarily `normalized`).
`f.U`	multivariate spectrum of class `'mvspectrum'` with `normalize = TRUE`.
`algorithm.control`	list; control settings for any *iterative* ForeCA algorithm. See `complete_algorithm_control` for details.
`entropy.control`	list; control settings for entropy estimation. See `complete_entropy_control` for details.
`spectrum.control`	list; control settings for spectrum estimation. See `complete_spectrum_control` for details.
`entropy.method`	string; method to estimate the entropy from discrete probabilities p_i; here *probabilities* are the spectral density evaluated at the Fourier frequencies, $\widehat{p}_i = \widehat{f}(\omega_i)$.
`spectrum.method`	string; method for spectrum estimation; see `method` argument in `mvspectrum`.

threshold	numeric; values of spectral density below threshold are set to 0; default threshold = 0.
smoothing	logical; if TRUE the spectrum will be smoothed with a nonparametric estimate using gam and an exponential family (with link = log). Only works for univariate spectrum. The smoothing parameter is chosen automatically using generalized cross-validation (see gam for details). Default: FALSE.
base	logarithm base; entropy is measured in "nats" for base = exp(1); in "bits" if base = 2 (default).

`complete-controls` *Completes several control settings*

Description

Completes algorithm, entropy, and spectrum control lists.

Usage

```
complete_algorithm_control(
 algorithm.control = list(max.iter = 50, num.starts = 10, tol = 0.001, type = "EM")
)

complete_entropy_control(
 entropy.control = list(base = NULL, method = "MLE", prior.probs = NULL, prior.weight
    = 0.001, threshold = 0),
  num.outcomes
)

complete_spectrum_control(
  spectrum.control = list(kernel = NULL, method = c("mvspec", "pspectrum", "ar",
    "pgram"), smoothing = FALSE)
)
```

Arguments

algorithm.control	list; control parameters for any *iterative* ForeCA algorithm.
entropy.control	list; control settings for entropy estimation.
num.outcomes	positive integer; number of outcomes for the discrete probability distribution. Must be specified (no default value).
spectrum.control	list; control settings for spectrum estimation.

Value

A list with fully specified algorithm, entropy, or spectrum controls. Default values are only added if the input `{spectrum,entropy,algorithm}.control` list does not already set this value.

`complete_algorithm_control` returns a list containing:

`max.iter`	maximum number of iterations; default: `50`.
`num.starts`	number of random starts to avoid local optima; default: `10`.
`tol`	tolerance for when convergence is reached in any *iterative* ForeCA algorithm; default: `1e-03`.
`type`	string; type of algorithm. Default: `'EM'`.

`complete_entropy_control` returns a list with:

`base`	logarithm base for the entropy.
`method`	string; method to estimate entropy; default: `"MLE"`.
`prior.probs`	prior distribution; default: uniform `rep(1 / num.outcomes, num.outcomes)`.
`prior.weight`	weight of the prior distribution; default: `1e-3`.
`threshold`	non-negative float; set probabilities below threshold to zero; default: `0`.

`complete_spectrum_control` returns a list containing:

`kernel`	R function; function to weigh each Fourier frequency λ; default: `NULL` (no re-weighting).
`method`	string; method to estimate the spectrum; default: `'mvspec'` if **sapa** is installed, `'mvspec'` if only **astsa** is installed, and `'pgram'` if neither is installed.
`smoothing`	logical; default: `FALSE`.

Available methods for spectrum estimation are (alphabetical order)

`"ar"`	autoregressive spectrum fit via `spec.ar`; only for univariate time series.
`"mvspec"`	smoothed estimate using `mvspec`; many tuning parameters are available – they can be passed as additional arguments (`...`) to `mvspectrum`.
`"pgram"`	raw periodogram using `spectrum`
`"pspectrum"`	advanced non-parametric estimation of a tapered power spectrum using `pspectrum`.

Setting `smoothing = TRUE` will smooth the estimated spectrum (again); this option is only available for univariate time series/spectra.

See Also

`mvspectrum`, `discrete_entropy`, `continuous_entropy`

`continuous_entropy` *Shannon entropy for a continuous pdf*

Description

Computes the Shannon entropy $\mathcal{H}(p)$ for a continuous probability density function (pdf) $p(x)$ using numerical integration.

Usage

```
continuous_entropy(pdf, lower, upper, base = 2)
```

Arguments

`pdf`	R function for the pdf $p(x)$ of a RV $X \sim p(x)$. This function must be non-negative and integrate to 1 over the interval [`lower`, `upper`].
`lower`, `upper`	lower and upper integration limit. `pdf` must integrate to 1 on this interval.
`base`	logarithm base; entropy is measured in "nats" for `base = exp(1)`; in "bits" if `base = 2` (default).

Details

The Shannon entropy of a continuous random variable (RV) $X \sim p(x)$ is defined as

$$\mathcal{H}(p) = -\int_{-\infty}^{\infty} p(x) \log p(x) dx.$$

Contrary to discrete RVs, continuous RVs can have negative entropy (see Examples).

Value

scalar; entropy value (real).

Since `continuous_entropy` uses numerical integration (`integrate()`) convergence is not garantueed (even if integral in definition of $\mathcal{H}(p)$ exists). Issues a warning if `integrate()` does not converge.

See Also

`discrete_entropy`

Examples

```
# entropy of U(a, b) = log(b - a). Thus not necessarily positive anymore, e.g.
continuous_entropy(function(x) dunif(x, 0, 0.5), 0, 0.5) # log2(0.5)

# Same, but for U(-1, 1)
my_density <- function(x){
  dunif(x, -1, 1)
}
```

```
continuous_entropy(my_density, -1, 1) # = log(upper - lower)

# a 'triangle' distribution
continuous_entropy(function(x) x, 0, sqrt(2))
```

`discrete_entropy` *Shannon entropy for discrete pmf*

Description

Computes the Shannon entropy $\mathcal{H}(p) = -\sum_{i=1}^{n} p_i \log p_i$ of a discrete RV X taking values in $\{x_1, \ldots, x_n\}$ with probability mass function (pmf) $P(X = x_i) = p_i$ with $p_i \geq 0$ for all i and $\sum_{i=1}^{n} p_i = 1$.

Usage

```
discrete_entropy(
  probs,
  base = 2,
  method = c("MLE"),
  threshold = 0,
  prior.probs = NULL,
  prior.weight = 0
)
```

Arguments

`probs`	numeric; probabilities (empirical frequencies). Must be non-negative and add up to 1.
`base`	logarithm base; entropy is measured in "nats" for `base = exp(1)`; in "bits" if `base = 2` (default).
`method`	string; method to estimate entropy; see Details below.
`threshold`	numeric; frequencies below `threshold` are set to 0; default `threshold = 0`, i.e., no thresholding. If `prior.weight > 0` then thresholding will be done *before* smoothing.
`prior.probs`	optional; only used if `prior.weight > 0`. Add a prior probability distribution to `probs`. By default it uses a uniform distribution putting equal probability on each outcome.
`prior.weight`	numeric; how much weight does the prior distribution get in a mixture model between data and prior distribution? Must be between `0` and `1`. Default: `0` (no prior).

Details

`discrete_entropy` uses a plug-in estimator (`method = "MLE"`):

$$\widehat{\mathcal{H}}(p) = -\sum_{i=1}^{n} \widehat{p}_i \log \widehat{p}_i.$$

If `prior.weight > 0`, then it mixes the observed proportions $\widehat{p}_i$ with a prior distribution

$$\widehat{p}_i \leftarrow (1-\lambda)\cdot \widehat{p}_i + \lambda \cdot prior_i, \quad i = 1, \ldots, n,$$

where $\lambda \in [0, 1]$ is the `prior.weight` parameter. By default the prior is a uniform distribution, i.e., $prior_i = \frac{1}{n}$ for all i.

Note that this plugin estimator is biased. See References for an overview of alternative methods.

Value

numeric; non-negative real value.

References

Archer E., Park I. M., Pillow J.W. (2014). "Bayesian Entropy Estimation for Countable Discrete Distributions". Journal of Machine Learning Research (JMLR) 15, 2833-2868. Available at http://jmlr.org/papers/v15/archer14a.html.

See Also

`continuous_entropy`

Examples

```
probs.tmp <- rexp(5)
probs.tmp <- sort(probs.tmp / sum(probs.tmp))

unif.distr <- rep(1/length(probs.tmp), length(probs.tmp))

matplot(cbind(probs.tmp, unif.distr), pch = 19,
        ylab = "P(X = k)", xlab = "k")
matlines(cbind(probs.tmp, unif.distr))
legend("topleft", c("non-uniform", "uniform"), pch = 19,
       lty = 1:2, col = 1:2, box.lty = 0)

discrete_entropy(probs.tmp)
# uniform has largest entropy among all bounded discrete pmfs
# (here = log(5))
discrete_entropy(unif.distr)
# no uncertainty if one element occurs with probability 1
discrete_entropy(c(1, 0, 0))
```

foreca *Forecastable Component Analysis*

Description

foreca performs Forecastable Component Analysis (ForeCA) on $\mathbf{X}_t$ – a K-dimensional time series with T observations. Users should only call foreca, rather than foreca.one_weightvector or foreca.multiple_weightvectors.

foreca.one_weightvector is a wrapper around several algorithms that solve the ForeCA optimization problem for a single weightvector $\mathbf{w}_i$ and whitened time series $\mathbf{U}_t$.

foreca.multiple_weightvectors applies foreca.one_weightvector iteratively to $\mathbf{U}_t$ in order to obtain multiple weightvectors that yield most forecastable, uncorrelated signals.

Usage

```
foreca(series, n.comp = 2, algorithm.control = list(type = "EM"), ...)

foreca.one_weightvector(
  U,
  f.U = NULL,
  spectrum.control = list(),
  entropy.control = list(),
  algorithm.control = list(),
  keep.all.optima = FALSE,
  dewhitening = NULL,
  ...
)

foreca.multiple_weightvectors(
  U,
  spectrum.control = list(),
  entropy.control = list(),
  algorithm.control = list(),
  n.comp = 2,
  plot = FALSE,
  dewhitening = NULL,
  ...
)
```

Arguments

series	a $T \times K$ array with T observations from the K-dimensional time series $\mathbf{X}_t$. Can be a matrix, data.frame, or a multivariate ts object.
n.comp	positive integer; number of components to be extracted. Default: 2.

`algorithm.control`
: list; control settings for any *iterative* ForeCA algorithm. See `complete_algorithm_control` for details.

`...`
: additional arguments passed to available ForeCA algorithms.

`U`
: a $T \times K$ array with `T` observations from the K-dimensional **whitened** (`whiten`) time series $\mathbf{U}_t$. Can be a `matrix`, `data.frame`, or a multivariate `ts` object.

`f.U`
: multivariate spectrum of class `'mvspectrum'` with `normalize = TRUE`.

`spectrum.control`
: list; control settings for spectrum estimation. See `complete_spectrum_control` for details.

`entropy.control`
: list; control settings for entropy estimation. See `complete_entropy_control` for details.

`keep.all.optima`
: logical; if `TRUE`, it keeps the optimal solutions of each random start. Default: `FALSE` (only returns the best solution).

`dewhitening`
: optional; if provided (returned by `whiten`) then it uses the dewhitening transformation to obtain the original series $\mathbf{X}_t$ and it uses that vector (normalized) as the initial weightvector which corresponds to the series $\mathbf{X}_{t,i}$ with larges `Omega`.

`plot`
: logical; if `TRUE` a plot of the current optimal solution $\mathbf{w}_i^*$ will be shown and updated for each iteration $i = 1, ...,$ `n.comp` of any iterative algorithm. Default: `FALSE`.

Value

An object of class `foreca`, which is similar to the output from `princomp`, with the following components (amongst others):

- `center`: sample mean $\widehat{\mu}_X$ of each `series`,
- `whitening`: whitening matrix of size $K \times K$ from `whiten`: $\mathbf{U}_t = (\mathbf{X}_t - \widehat{\mu}_X) \cdot whitening$; note that $\mathbf{X}_t$ is centered prior to the whitening transformation,
- `weightvectors`: orthonormal matrix of size $K \times n.comp$, which converts whitened data to `n.comp` forecastable components (ForeCs) $\mathbf{F}_t = \mathbf{U}_t \cdot weightvectors$,
- `loadings`: combination of whitening $\times$ weightvectors to obtain the final loadings for the original data: $\mathbf{F}_t = (\mathbf{X}_t - \widehat{\mu}_X) \cdot whitening \cdot weightvectors$; again, it centers $\mathbf{X}_t$ first,
- `loadings.normalized`: normalized loadings (unit norm). Note though that if you use these normalized loadings the resulting signals do not have variance 1 anymore.
- `scores`: `n.comp` forecastable components $\mathbf{F}_t$. They have mean 0, variance 1, and are uncorrelated.
- `Omega`: forecastability score of each ForeC of $\mathbf{F}_t$.

ForeCs are ordered from most to least forecastable (according to `Omega`).

Warning

Estimating Omega directly from the ForeCs $\mathbf{F}_t$ can be different to the reported `$Omega` estimates from `foreca`. Here is why:

In theory $f_y(\lambda)$ of a linear combination $y_t = \mathbf{X}_t\mathbf{w}$ can be analytically computed from the multivariate spectrum $f_{\mathbf{X}}(\lambda)$ by the quadratic form $f_y(\lambda) = \mathbf{w}' f_{\mathbf{X}}(\lambda)\mathbf{w}$ for all λ (see `spectrum_of_linear_combination`).

In practice, however, this identity does not hold always exactly since (often data-driven) control setting for spectrum estimation are not identical for the high-dimensional, noisy $\mathbf{X}_t$ and the combined univariate time series y_t (which is usually more smooth, less variable). Thus estimating $\widehat{f}_y$ directly from y_t can give slightly different estimates to computing it as $\mathbf{w}'\widehat{f}_{\mathbf{X}}\mathbf{w}$. Consequently also `Omega` estimates can be different.

In general, these differences are small and have no relevant implications for estimating ForeCs. However, in rare occasions the obtained ForeCs can have smaller `Omega` than the maximum `Omega` across all original series. In such a case users should not re-estimate Ω from the resulting ForeCs $\mathbf{F}_t$, but access them via `$Omega` provided by `'foreca'` output (the univariate estimates are stored in `$Omega.univ`).

References

Goerg, G. M. (2013). "Forecastable Component Analysis". Journal of Machine Learning Research (JMLR) W&CP 28 (2): 64-72, 2013. Available at http://jmlr.org/proceedings/papers/v28/goerg13.html.

Examples

```
XX <- diff(log(EuStockMarkets)) * 100
plot(ts(XX))
## Not run:
ff <- foreca(XX[,1:4], n.comp = 4, plot = TRUE, spectrum.control=list(method="pspectrum"))
ff
summary(ff)
plot(ff)

## End(Not run)


## Not run:
PW <- whiten(XX)
one.weight.em <- foreca.one_weightvector(U = PW$U,
                                         dewhitening = PW$dewhitening,
                                         algorithm.control =
                                           list(num.starts = 2,
                                                type = "EM"),
                                         spectrum.control =
                                           list(method = "mvspec"))
plot(one.weight.em)

## End(Not run)
## Not run:
```

```
PW <- whiten(XX)
ff <- foreca.multiple_weightvectors(PW$U, n.comp = 2,
                                    dewhitening = PW$dewhitening)
ff
plot(ff$scores)

## End(Not run)
```

`foreca-utils` *Plot, summary, and print methods for class 'foreca'*

Description

A collection of S3 methods for estimated ForeCA results (class `"foreca"`).

`summary.foreca` computes summary statistics.

`print.foreca` prints a human-readable summary in the console.

`biplot.foreca` shows a biplot of the ForeCA loadings (wrapper around `biplot.princomp`).

`plot.foreca` shows biplots, screeplots, and white noise tests.

Usage

```
## S3 method for class 'foreca'
summary(object, lag = 10, alpha = 0.05, ...)

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

## S3 method for class 'foreca'
biplot(x, ...)

## S3 method for class 'foreca'
plot(x, lag = 10, alpha = 0.05, ...)
```

Arguments

`lag`	integer; how many lags to test in `Box.test`; default: 10.
`alpha`	significance level for testing white noise in `Box.test`; default: 0.05.
`...`	additional arguments passed to `biplot.princomp`, `biplot`, `plot`, or `summary`.
`x, object`	an object of class `"foreca"`.

Examples

```
# see examples in 'foreca'
```

foreca.EM-aux *ForeCA EM auxiliary functions*

Description

`foreca.EM.one_weightvector` relies on several auxiliary functions:

`foreca.EM.E_step` computes the spectral density of $y_t = \mathbf{U}_t\mathbf{w}$ given the weightvector $\mathbf{w}$ and the normalized spectrum estimate $f_{\mathbf{U}}$. A wrapper around `spectrum_of_linear_combination`.

`foreca.EM.M_step` computes the minimizing eigenvector ($\to \widehat{\mathbf{w}}_{i+1}$) of the weighted covariance matrix, where the weights equal the negative logarithm of the spectral density at the current $\widehat{\mathbf{w}}_i$.

`foreca.EM.E_and_M_step` is a wrapper around `foreca.EM.E_step` followed by `foreca.EM.M_step`.

`foreca.EM.h` evaluates (an upper bound of) the entropy of the spectral density as a function of $\mathbf{w}_i$ (or $\mathbf{w}_{i+1}$). This is the objective funcion that should be `minimized`.

Usage

```
foreca.EM.E_step(f.U, weightvector)

foreca.EM.M_step(f.U, f.current, minimize = TRUE, entropy.control = list())

foreca.EM.E_and_M_step(
  weightvector,
  f.U,
  minimize = TRUE,
  entropy.control = list()
)

foreca.EM.h(
  weightvector.new,
  f.U,
  weightvector.current = weightvector.new,
  f.current = NULL,
  entropy.control = list(),
  return.negative = FALSE
)
```

Arguments

`f.U`	multivariate spectrum of class `'mvspectrum'` with `normalize = TRUE`.
`weightvector`	numeric; weights $\mathbf{w}$ for $y_t = \mathbf{U}_t\mathbf{w}$. Must have unit norm in ℓ^2.
`f.current`	numeric; spectral density estimate of $y_t = \mathbf{U}_t\mathbf{w}$ for the current estimate $\widehat{\mathbf{w}}_i$ (required for `foreca.EM.M_step`; optional for `foreca.EM.h`).
`minimize`	logical; if `TRUE` (default) it returns the eigenvector corresponding to the smallest eigenvalue; otherwise to the largest eigenvalue.

`entropy.control`
: list; control settings for entropy estimation. See `complete_entropy_control` for details.

`weightvector.new`
: weightvector $\widehat{\mathbf{w}}_{i+1}$ of the new iteration (i+1).

`weightvector.current`
: weightvector $\widehat{\mathbf{w}}_i$ of the current iteration (i).

`return.negative`
: logical; if `TRUE` it returns the negative spectral entropy. This is useful when maximizing forecastibility which is equivalent (up to an additive constant) to maximizing negative entropy. Default: `FALSE`.

Value

`foreca.EM.E_step` returns the normalized univariate spectral density (normalized such that its `sum` equals 0.5).

`foreca.EM.M_step` returns a list with three elements:

- `matrix`: weighted covariance matrix, where the weights are the negative log of the spectral density. If density is estimated by discrete probabilities, then this `matrix` is positive semi-definite, since $-\log(p) \geq 0$ for $p \in [0, 1]$. See `weightvector2entropy_wcov`.
- `vector`: minimizing (or maximizing if `minimize` = `FALSE`) eigenvector of `matrix`,
- `value`: corresponding eigenvalue.

Contrary to `foreca.EM.M_step`, `foreca.EM.E_and_M_step` only returns the optimal weightvector as a numeric.

`foreca.EM.h` returns non-negative real value (see References for details):

- entropy, if `weightvector.new` = `weightvector.current`,
- an upper bound of that entropy for `weightvector.new`, otherwise.

See Also

`weightvector2entropy_wcov`

Examples

```
## Not run:
XX <- diff(log(EuStockMarkets)) * 100
UU <- whiten(XX)$U
ff <- mvspectrum(UU, 'mvspec', normalize = TRUE)

ww0 <- initialize_weightvector(num.series = ncol(XX), method = 'rnorm')

f.ww0 <- foreca.EM.E_step(ff, ww0)
plot(f.ww0, type = "l")

## End(Not run)
## Not run:
```

```
one.step <- foreca.EM.M_step(ff, f.ww0,
                             entropy.control = list(prior.weight = 0.1))
image(one.step$matrix)

requireNamespace(LICORS)
# if you have the 'LICORS' package use
LICORS::image2(one.step$matrix)

ww1 <- one.step$vector
f.ww1 <- foreca.EM.E_step(ff, ww1)

layout(matrix(1:2, ncol = 2))
matplot(seq(0, pi, length = length(f.ww0)), cbind(f.ww0, f.ww1),
        type = "l", lwd =2, xlab = "omega_j", ylab = "f(omega_j)")
plot(f.ww0, f.ww1, pch = ".", cex = 3, xlab = "iteration 0",
     ylab = "iteration 1", main = "Spectral density")
abline(0, 1, col = 'blue', lty = 2, lwd = 2)

Omega(mvspectrum.output = f.ww0) # start
Omega(mvspectrum.output = f.ww1) # improved after one iteration

## End(Not run)
## Not run:
ww0 <- initialize_weightvector(NULL, ff, method = "rnorm")
ww1 <- foreca.EM.E_and_M_step(ww0, ff)
ww0
ww1
barplot(rbind(ww0, ww1), beside = TRUE)
abline(h = 0, col = "blue", lty = 2)

## End(Not run)
## Not run:
foreca.EM.h(ww0, ff)       # iteration 0
foreca.EM.h(ww1, ff, ww0)  # min eigenvalue inequality
foreca.EM.h(ww1, ff)       # KL divergence inequality
one.step$value

# by definition of Omega, they should equal 1 (modulo rounding errors)
Omega(mvspectrum.output = f.ww0) / 100 + foreca.EM.h(ww0, ff)
Omega(mvspectrum.output = f.ww1) / 100 + foreca.EM.h(ww1, ff)

## End(Not run)
```

`foreca.EM.one_weightvector` *EM-like algorithm to estimate optimal ForeCA transformation*

Description

`foreca.EM.one_weightvector` finds the optimal weightvector $\mathbf{w}^*$ that gives the most forecastable signal $y_t^* = \mathbf{U}_t \mathbf{w}^*$ using an EM-like algorithm (see References).

Usage

```
foreca.EM.one_weightvector(
  U,
  f.U = NULL,
  spectrum.control = list(),
  entropy.control = list(),
  algorithm.control = list(),
 init.weightvector = initialize_weightvector(num.series = ncol(U), method = "rnorm"),
  ...
)
```

Arguments

U	a $T \times K$ array with T observations from the K-dimensional **whitened** (whiten) time series $\mathbf{U}_t$. Can be a matrix, data.frame, or a multivariate ts object.
f.U	multivariate spectrum of class 'mvspectrum' with normalize = TRUE.
spectrum.control	list; control settings for spectrum estimation. See complete_spectrum_control for details.
entropy.control	list; control settings for entropy estimation. See complete_entropy_control for details.
algorithm.control	list; control settings for any *iterative* ForeCA algorithm. See complete_algorithm_control for details.
init.weightvector	numeric; starting point $\mathbf{w}_0$ for several iterative algorithms. By default it uses a (normalized) random vector from a standard Normal distribution (see initialize_weightvector).
...	other arguments passed to mvspectrum

Value

A list with useful quantities like the optimal weighvector, the corresponding signal, and its forecastability.

See Also

foreca.one_weightvector, foreca.EM-aux

Examples

```
## Not run:
XX <- diff(log(EuStockMarkets)[100:200,]) * 100
one.weight <- foreca.EM.one_weightvector(whiten(XX)$U,
                                         spectrum.control =
                                            list(method = "mvspec"))

## End(Not run)
```

foreca.one_weightvector-utils

Plot, summary, and print methods for class 'foreca.one_weightvector'

Description

S3 methods for the one weightvector optimization in ForeCA (class `"foreca.one_weightvector"`).

`summary.foreca.one_weightvector` computes summary statistics.

`plot.foreca.one_weightvector` shows the results of an (iterative) algorithm that obtained the i-th optimal a weightvector $\mathbf{w}_i^*$. It shows trace plots of the objective function and the weightvector, and a time series plot of the transformed signal y_t^* along with its spectral density estimate $\widehat{f_y}(\omega_j)$.

Usage

```
## S3 method for class 'foreca.one_weightvector'
summary(object, lag = 10, alpha = 0.05, ...)

## S3 method for class 'foreca.one_weightvector'
plot(x, main = "", cex.lab = 1.1, ...)
```

Arguments

`lag`	integer; how many lags to test in `Box.test`; default: 10.
`alpha`	significance level for testing white noise in `Box.test`; default: 0.05.
`...`	additional arguments passed to `plot`, or `summary`.
`x`, `object`	an object of class `"foreca.one_weightvector"`.
`main`	an overall title for the plot: see `title`.
`cex.lab`	size of the axes labels.

Examples

```
# see examples in 'foreca.one_weightvector'
```

initialize_weightvector

Initialize weightvector for iterative ForeCA algorithms

Description

`initialize_weightvector` returns a unit norm (in ℓ^2) vector $\mathbf{w}_0 \in R^K$ that can be used as the starting point for any iterative ForeCA algorithm, e.g., `foreca.EM.one_weightvector`. Several quickly computable heuristics are available via the `method` argument.

Usage

```
initialize_weightvector(
  U = NULL,
  f.U = NULL,
  num.series = ncol(U),
 method = c("rnorm", "max", "SFA", "PCA", "rcauchy", "runif", "SFA.slow", "SFA.fast",
    "PCA.large", "PCA.small"),
  seed = sample(1e+06, 1),
  ...
)
```

Arguments

U	a $T \times K$ array with T observations from the K-dimensional **whitened** (whiten) time series $\mathbf{U}_t$. Can be a matrix, data.frame, or a multivariate ts object.
f.U	multivariate spectrum of class 'mvspectrum' with normalize = TRUE.
num.series	positive integer; number of time series K (determines the length of the weightvector). If num.series = 1 it simply returns a 1×1 array equal to 1.
method	string; which heuristics should be used to generate a good starting $\mathbf{w}_0$? Default: "rnorm"; see Details.
seed	non-negative integer; seed for random initialization which will be returned for reproducibility. By default it sets a random seed.
...	additional arguments

Details

The method argument specifies the heuristics that is used to get a good starting vector $\mathbf{w}_0$:

- "max" vector with all 0s, but a 1 at the position of the maximum forecastable series in U.
- "rcauchy" random start using rcauchy(k).
- "rnorm" random start using rnorm(k, 0, 1).
- "runif" random start using runif(k, -1, 1).
- "PCA.large" first eigenvector of PCA (largest variance signal).
- "PCA.small" last eigenvector of PCA (smallest variance signal).
- "PCA" checks both small and large, and chooses the one with higher forecastability as computed by Omega..
- "SFA.fast" last eigenvector of SFA (fastest signal).
- "SFA.slow" first eigenvector of SFA (slowest signal).
- "SFA" checks both slow and fast, and chooses the one with higher forecastability as computed by Omega.

Each vector has length K and is automatically normalized to have unit norm in ℓ^2.

For the 'SFA*' methods see sfa. Note that maximizing (or minimizing) the lag 1 auto-correlation does not necessarily yield the most forecastable signal, but it's a good start.

Value

numeric; a vector of length K with unit norm in ℓ^2.

Examples

```
XX <- diff(log(EuStockMarkets))
## Not run:
initialize_weightvector(U = XX, method = "SFA")

## End(Not run)
initialize_weightvector(num.series = ncol(XX), method = "rnorm")
```

`mvspectrum` *Estimates spectrum of multivariate time series*

Description

The spectrum of a multivariate time series is a matrix-valued function of the frequency $\lambda \in [-\pi, \pi]$, which is symmetric/Hermitian around $\lambda = 0$.

`mvspectrum` estimates it and returns a 3D array of dimension $num.freqs \times K \times K$. Since the spectrum is symmetric/Hermitian around $\lambda = 0$ it is sufficient to store only positive frequencies. In the implementation in this package we thus usually consider only positive frequencies (omitting 0); `num.freqs` refers to the number of positive frequencies only.

`normalize_mvspectrum` normalizes the spectrum such that it adds up to 0.5 over all positive frequencies (by symmetry it will add up to 1 over the whole range – thus the name *normalize*).

For a K-dimensional time series it adds up to a Hermitian $K \times K$ matrix with 0.5 in the diagonal and imaginary elements (real parts equal to 0) in the off-diagonal. Since it is Hermitian the mvspectrum will add up to the identity matrix over the whole range of frequencies, since the off-diagonal elements are purely imaginary (real part equals 0) and thus add up to 0.

`check_mvspectrum_normalized` checks if the spectrum is normalized (see `normalize_mvspectrum` for the requirements).

`mvpgram` computes the multivariate periodogram estimate using bare-bone multivariate fft (`mvfft`). Use `mvspectrum(..., method = 'pgram')` instead of `mvpgram` directly.

This function is merely included to have one method that does not require the **astsa** nor the **sapa** R packages. However, it is strongly encouraged to install either one of them to get (much) better estimates. See Details.

`get_spectrum_from_mvspectrum` extracts the spectrum of one time series from an `"mvspectrum"` object by taking the i-th diagonal entry for each frequency.

`spectrum_of_linear_combination` computes the spectrum of the linear combination $\mathbf{y}_t = \mathbf{X}_t\boldsymbol{\beta}$ of K time series $\mathbf{X}_t$. This can be efficiently computed by the quadratic form

$$f_y(\lambda) = \boldsymbol{\beta}' f_{\mathbf{X}}(\lambda)\boldsymbol{\beta} \geq 0,$$

for each λ. This holds for any $\boldsymbol{\beta}$ (even $\boldsymbol{\beta} = \mathbf{0}$ – not only for $||\boldsymbol{\beta}||_2 = 1$. For $\boldsymbol{\beta} = \boldsymbol{e}_i$ (the i-th basis vector) this is equivalent to `get_spectrum_from_mvspectrum(..., which = i)`.

Usage

```
mvspectrum(
  series,
  method = c("mvspec", "pgram", "pspectrum", "ar"),
  normalize = FALSE,
  smoothing = FALSE,
  ...
)

normalize_mvspectrum(mvspectrum.output)

check_mvspectrum_normalized(f.U, check.attribute.only = TRUE)

mvpgram(series)

get_spectrum_from_mvspectrum(
  mvspectrum.output,
  which = seq_len(dim(mvspectrum.output)[2])
)

spectrum_of_linear_combination(mvspectrum.output, beta)
```

Arguments

series	a $T \times K$ array with T observations from the K-dimensional time series $\mathbf{X}_t$. Can be a matrix, data.frame, or a multivariate ts object.
method	string; method for spectrum estimation: use "pspectrum" for pspectrum; use "mvspec" to use mvspec (**astsa** package); or use "pgram" to use spec.pgram.
normalize	logical; if TRUE the spectrum will be normalized (see Value below for details).
smoothing	logical; if TRUE the spectrum will be smoothed with a nonparametric estimate using gam and an exponential family (with link = log). Only works for univariate spectrum. The smoothing parameter is chosen automatically using generalized cross-validation (see gam for details). Default: FALSE.
...	additional arguments passed to pspectrum or mvspec (e.g., taper)
mvspectrum.output	an object of class "mvspectrum" representing the multivariate spectrum of $\mathbf{X}_t$ (not necessarily normalized).
f.U	multivariate spectrum of class 'mvspectrum' with normalize = TRUE.
check.attribute.only	logical; if TRUE it checks the attribute only. This is much faster (it just needs to look up one attribute value), but it might not surface silent bugs. For sake of performance the package uses the attribute version by default. However, for testing/debugging the full computational version can be used.
which	integer(s); the spectrum of which series whould be extracted. By default, it returns all univariate spectra as a matrix (frequencies in rows).
beta	numeric; vector $\boldsymbol{\beta}$ that defines the linear combination.

Details

For an orthonormal time series $\mathbf{U}_t$ the raw periodogram adds up to I_K over all (negative and positive) frequencies. Since we only consider positive frequencies, the normalized multivariate spectrum should add up to $0.5 \cdot I_K$ plus a Hermitian imaginary matrix (which will add up to zero when combined with its symmetric counterpart.) As we often use non-parametric smoothing for less variance, the spectrum estimates do not satisfy this identity exactly. `normalize_mvspectrum` thus adjust the estimates so they satisfy it again exactly.

`mvpgram` has no options for improving spectrum estimation whatsoever. It thus yields very noisy (in fact, inconsistent) estimates of the multivariate spectrum $f_{\mathbf{X}}(\lambda)$. If you want to obtain better estimates then please use other `methods` in `mvspectrum` (this is highly recommended to obtain more reasonable/stable estimates).

Value

`mvspectrum` returns a 3D array of dimension $num.freqs \times K \times K$, where

- num.freqs is the number of frequencies
- K is the number of series (columns in `series`).

It also has an `"normalized"` attribute, which is `FALSE` if `normalize = FALSE`; otherwise `TRUE`. See `normalize_mvspectrum` for details.

`normalize_mvspectrum` returns a normalized spectrum over positive frequencies, which:

univariate: adds up to 0.5,

multivariate: adds up to Hermitian $K \times K$ matrix with 0.5 in the diagonal and purely imaginary elements in the off-diagonal.

`check_mvspectrum_normalized` throws an error if spectrum is not normalized correctly.

`get_spectrum_from_mvspectrum` returns either a matrix of all univariate spectra, or one single column (if `which` is specified.)

`spectrum_of_linear_combination` returns a vector with length equal to the number of rows of `mvspectrum.output`.

References

See References in `spectrum`, `pspectrum`, `mvspec`.

Examples

```
set.seed(1)
XX <- cbind(rnorm(100), arima.sim(n = 100, list(ar = 0.9)))
ss3d <- mvspectrum(XX)
dim(ss3d)

ss3d[2,,] # at omega_1; in general complex-valued, but Hermitian
identical(ss3d[2,,], Conj(t(ss3d[2,,]))) # is Hermitian
## Not run:
  ss <- mvspectrum(XX[, 1], method="pspectrum", smoothing = TRUE)
  mvspectrum(XX, normalize = TRUE)
```

```
## End(Not run)
ss <- mvspectrum(whiten(XX)$U, normalize = TRUE)

xx <- scale(rnorm(100), center = TRUE, scale = FALSE)
var(xx)
sum(mvspectrum(xx, normalize = FALSE, method = "pgram")) * 2
sum(mvspectrum(xx, normalize = FALSE, method = "mvspec")) * 2
## Not run:
  sum(mvspectrum(xx, normalize = FALSE, method = "pspectrum")) * 2

## End(Not run)
## Not run:
xx <- scale(rnorm(100), center = TRUE, scale = FALSE)
ss <- mvspectrum(xx)
ss.n <- normalize_mvspectrum(ss)
sum(ss.n)
# multivariate
UU <- whiten(matrix(rnorm(40), ncol = 2))$U
S.U <- mvspectrum(UU, method = "mvspec")
mvspectrum2wcov(normalize_mvspectrum(S.U))

## End(Not run)

XX <- matrix(rnorm(1000), ncol = 2)
SS <- mvspectrum(XX, "mvspec")
ss1 <- mvspectrum(XX[, 1], "mvspec")

SS.1 <- get_spectrum_from_mvspectrum(SS, 1)
plot.default(ss1, SS.1)
abline(0, 1, lty = 2, col = "blue")


XX <- matrix(arima.sim(n = 1000, list(ar = 0.9)), ncol = 4)
beta.tmp <- rbind(1, -1, 2, 0)
yy <- XX %*% beta.tmp

SS <- mvspectrum(XX, "mvspec")
ss.yy.comb <- spectrum_of_linear_combination(SS, beta.tmp)
ss.yy <- mvspectrum(yy, "mvspec")

plot(ss.yy, log = TRUE) # using plot.mvspectrum()
lines(ss.yy.comb, col = "red", lty = 1, lwd = 2)
```

`mvspectrum-utils` *S3 methods for class 'mvspectrum'*

Description

S3 methods for multivariate spectrum estimation.

plot.mvspectrum plots all univariate spectra. Analogouos to spectrum when plot = TRUE.

Usage

```
## S3 method for class 'mvspectrum'
plot(x, log = TRUE, ...)
```

Arguments

x	an object of class "foreca.one_weightvector".
log	logical; if TRUE (default), it plots the spectra on log-scale.
...	additional arguments passed to matplot.

See Also

get_spectrum_from_mvspectrum

Examples

```
# see examples in 'mvspectrum'

SS <- mvspectrum(diff(log(EuStockMarkets)) * 100,
                 spectrum.control = list(method = "mvspec"))
plot(SS, log = FALSE)
```

---

mvspectrum2wcov *Compute (weighted) covariance matrix from frequency spectrum*

Description

mvspectrum2wcov computes a (weighted) covariance matrix estimate from the frequency spectrum (see Details).

weightvector2entropy_wcov computes the weighted covariance matrix using the negative entropy of the univariate spectrum (given the weightvector) as kernel weights. This matrix is the objective matrix for many foreca.* algorithms.

Usage

```
mvspectrum2wcov(mvspectrum.output, kernel.weights = 1)

weightvector2entropy_wcov(
  weightvector = NULL,
  f.U,
  f.current = NULL,
  entropy.control = list()
)
```

Arguments

`mvspectrum.output`
: an object of class `"mvspectrum"` representing the multivariate spectrum of $\mathbf{X}_t$ (not necessarily `normalized`).

`kernel.weights`
: numeric; weights for each frequency. By default uses weights that average out to 1.

`weightvector`
: numeric; weights $\mathbf{w}$ for $y_t = \mathbf{U}_t\mathbf{w}$. Must have unit norm in ℓ^2.

`f.U`
: multivariate spectrum of class `'mvspectrum'` with `normalize = TRUE`.

`f.current`
: numeric; spectral density estimate of $y_t = \mathbf{U}_t\mathbf{w}$ for the current estimate $\widehat{\mathbf{w}}_i$ (required for `foreca.EM.M_step`; optional for `foreca.EM.h`).

`entropy.control`
: list; control settings for entropy estimation. See `complete_entropy_control` for details.

Details

The covariance matrix of a multivariate time series satisfies the identity

$$\Sigma_X \equiv \int_{-\pi}^{\pi} S_X(\lambda) d\lambda.$$

A generalized covariance matrix estimate can thus be obtained using a weighted average

$$\tilde{\Sigma}_X = \int_{-\pi}^{\pi} K(\lambda) S_X(\lambda) d\lambda,$$

where $K(\lambda)$ is a kernel symmetric around 0 which averages out to 1 over the interval $[-\pi, \pi]$, i.e., $\frac{1}{2\pi}\int_{-\pi}^{\pi} K(\lambda)d\lambda = 1$. This allows one to remove or amplify specific frequencies in the covariance matrix estimation.

For ForeCA `mvspectrum2wcov` is especially important as we use

$$K(\lambda) = -\log f_y(\lambda),$$

as the *weights* (their average is not 1!). This particular kernel weight is implemented as a wrapper in `weightvector2entropy_wcov`.

Value

A symmetric $n \times n$ matrix.

If `kernel.weights` ≥ 0, then it is positive semi-definite; otherwise, it is symmetric but not necessarily positive semi-definite.

See Also

`mvspectrum`

Examples

```
nn <- 50
YY <- cbind(rnorm(nn), arima.sim(n = nn, list(ar = 0.9)), rnorm(nn))
XX <- YY %*% matrix(rnorm(9), ncol = 3)  # random mix
XX <- scale(XX, scale = FALSE, center = TRUE)

# sample estimate of covariance matrix
Sigma.hat <- cov(XX)
dimnames(Sigma.hat) <- NULL

# using the frequency spectrum
SS <- mvspectrum(XX, "mvspec")
Sigma.hat.freq <- mvspectrum2wcov(SS)

layout(matrix(1:4, ncol = 2))
par(mar = c(2, 2, 1, 1))
plot(c(Sigma.hat/Sigma.hat.freq))
abline(h = 1)

image(Sigma.hat)
image(Sigma.hat.freq)
image(Sigma.hat / Sigma.hat.freq)

# examples for entropy wcov
XX <- diff(log(EuStockMarkets)) * 100
UU <- whiten(XX)$U
ff <- mvspectrum(UU, "mvspec", normalize = TRUE)

ww0 <- initialize_weightvector(num.series = ncol(XX), method = 'rnorm')

weightvector2entropy_wcov(ww0, ff,
                          entropy.control =
                            list(prior.weight = 0.1))
```

`Omega` *Estimate forecastability of a time series*

Description

An estimator for the forecastability $\Omega(x_t)$ of a univariate time series x_t. Currently it uses a discrete plug-in estimator given the empirical spectrum (periodogram).

Usage

```
Omega(
  series = NULL,
  spectrum.control = list(),
  entropy.control = list(),
  mvspectrum.output = NULL
)
```

Arguments

`series`	a univariate time series; if it is multivariate, then `Omega` works component-wise (i.e., same as `apply(series, 2, Omega)`).
`spectrum.control`	list; control settings for spectrum estimation. See `complete_spectrum_control` for details.
`entropy.control`	list; control settings for entropy estimation. See `complete_entropy_control` for details.
`mvspectrum.output`	an object of class `"mvspectrum"` representing the multivariate spectrum of $\mathbf{X}_t$ (not necessarily `normalized`).

Details

The *forecastability* of a stationary process x_t is defined as (see References)

$$\Omega(x_t) = 1 - \frac{-\int_{-\pi}^{\pi} f_x(\lambda) \log f_x(\lambda) d\lambda}{\log 2\pi} \in [0, 1]$$

where $f_x(\lambda)$ is the normalized spectral *density* of x_t. In particular $\int_{-\pi}^{\pi} f_x(\lambda) d\lambda = 1$.

For white noise ε_t forecastability $\Omega(\varepsilon_t) = 0$; for a sum of sinusoids it equals 100 %. However, empirically it reaches 100% only if the estimated spectrum has exactly one peak at some ω_j and $\widehat{f}(\omega_k) = 0$ for all $k \neq j$.

In practice, a time series of length `T` has T Fourier frequencies which represent a discrete probability distribution. Hence entropy of $f_x(\lambda)$ must be normalized by $\log T$, not by $\log 2\pi$.

Also we can use several smoothing techniques to obtain a less variance estimate of $f_x(\lambda)$.

Value

A real-value between 0 and 100 (%). 0 means not forecastable (white noise); 100 means perfectly forecastable (a sinusoid).

References

Goerg, G. M. (2013). "Forecastable Component Analysis". Journal of Machine Learning Research (JMLR) W&CP 28 (2): 64-72, 2013. Available at http://jmlr.org/proceedings/papers/v28/goerg13.html.

See Also

`spectral_entropy`, `discrete_entropy`, `continuous_entropy`

Examples

```
nn <- 100
eps <- rnorm(nn)  # white noise has Omega() = 0 in theory
Omega(eps, spectrum.control = list(method = "pgram"))
# smoothing makes it closer to 0
Omega(eps, spectrum.control = list(method = "mvspec"))

xx <- sin(seq_len(nn) * pi / 10)
Omega(xx, spectrum.control = list(method = "pgram"))
Omega(xx, entropy.control = list(threshold = 1/40))
Omega(xx, spectrum.control = list(method = "mvspec"),
      entropy.control = list(threshold = 1/20))

# an AR(1) with phi = 0.5
yy <- arima.sim(n = nn, model = list(ar = 0.5))
Omega(yy, spectrum.control = list(method = "mvspec"))

# an AR(1) with phi = 0.9 is more forecastable
yy <- arima.sim(n = nn, model = list(ar = 0.9))
Omega(yy, spectrum.control = list(method = "mvspec"))
```

---

quadratic_form *Computes quadratic form x' A x*

Description

`quadratic_form` computes the quadratic form $\mathbf{x}'\mathbf{A}\mathbf{x}$ for an $n \times n$ matrix $\mathbf{A}$ and an n-dimensional vector $\mathbf{x}$, i.e., a wrapper for `t(x) %*% A %*% x`.

`fill_symmetric` and `quadratic_form` work with real and complex valued matrices/vectors.

`fill_hermitian` fills up the lower triangular part (`NA`) of an upper triangular matrix to its Hermitian (symmetric if real matrix) version, such that it satisfies $\mathbf{A} = \bar{\mathbf{A}}'$, where $\bar{z}$ is the complex conjugate of z. If the matrix is real-valued this makes it simply symmetric.

Note that the input matrix must have a **real-valued** diagonal and `NA`s in the lower triangular part.

Usage

```
quadratic_form(mat, vec)

fill_hermitian(mat)
```

Arguments

`mat`	numeric; $n \times n$ matrix (real or complex).
`vec`	numeric; $n \times 1$ vector (real or complex).

Value

A real/complex value $\mathbf{x}'\mathbf{A}\mathbf{x}$.

Examples

```
## Not run:
 set.seed(1)
 AA <- matrix(1:4, ncol = 2)
 bb <- matrix(rnorm(2))
 t(bb) %*% AA %*% bb
 quadratic_form(AA, bb)

## End(Not run)


AA <- matrix(1:16, ncol = 4)
AA[lower.tri(AA)] <- NA
AA

fill_hermitian(AA)
```

---

sfa *Slow Feature Analysis*

Description

`sfa` performs Slow Feature Analysis (SFA) on a K-dimensional time series with T observations.

Important: This implementation of SFA is just the most basic version; it is merely included here for convenience in `initialize_weightvector`. If you want to actually use full functionality of SFA in R use the **rSFA** package, which has a much more advanced and efficient implementations. `sfa()` here corresponds to `sfa1`.

Usage

```
sfa(series, ...)
```

Arguments

`series`	a $T \times K$ array with `T` observations from the K-dimensional time series $\mathbf{X}_t$. Can be a `matrix`, `data.frame`, or a multivariate `ts` object.
`...`	additional arguments

Details

Slow Feature Analysis (SFA) finds *slow* signals (see References below). The problem has an analytic solution and thus can be computed quickly using generalized eigen-value solvers. For ForeCA it is important to know that SFA is equivalent to finding a linear combination signal with largest lag 1 autocorrelation.

The disadvantage of SFA for forecasting is that, e.g., white noise (WN) is ranked higher than an AR(1) with negative autocorrelation coefficient $\rho_1 < 0$. While it is true that WN is slower, it is not more forecastable. Thus we are also interested in the fastest signal, i.e., the last eigenvector. The so obtained fastest signal corresponds to minimizing the lag 1 auto-correlation (possibly $\rho_1 < 0$).

Note though that maximizing (or minimizing) the lag 1 auto-correlation does not necessarily yield the most forecastable signal (as measured by `Omega`), but it is a good start.

Value

An object of class `sfa` which inherits methods from `princomp`. Signals are ordered from slowest to fastest.

References

Laurenz Wiskott and Terrence J. Sejnowski (2002). "Slow Feature Analysis: Unsupervised Learning of Invariances", Neural Computation 14:4, 715-770.

See Also

`initialize_weightvector`

Examples

```
XX <- diff(log(EuStockMarkets[-c(1:100),])) * 100
plot(ts(XX))
ss <- sfa(XX[,1:4])

summary(ss)
plot(ss)
plot(ts(ss$scores))
apply(ss$scores, 2, function(x) acf(x, plot = FALSE)$acf[2])
biplot(ss)
```

`spectral_entropy` *Estimates spectral entropy of a time series*

Description

Estimates *spectral entropy* from a univariate (or multivariate) normalized spectral density.

Usage

```
spectral_entropy(
  series = NULL,
  spectrum.control = list(),
  entropy.control = list(),
  mvspectrum.output = NULL,
  ...
)
```

Arguments

`series`	univariate time series of length T. In the rare case that users want to call this for a multivariate time `series`, note that the estimated spectrum is in general *not* normalized for the computation. Only if the original data is whitened, then it is normalized.
`spectrum.control`	list; control settings for spectrum estimation. See `complete_spectrum_control` for details.
`entropy.control`	list; control settings for entropy estimation. See `complete_entropy_control` for details.
`mvspectrum.output`	optional; one can directly provide an estimate of the spectrum of `series`. Usually the output of `mvspectrum`.
`...`	additional arguments passed to `mvspectrum`.

Details

The *spectral entropy* equals the Shannon entropy of the spectral density $f_x(\lambda)$ of a stationary process x_t:

$$H_s(x_t) = -\int_{-\pi}^{\pi} f_x(\lambda) \log f_x(\lambda) d\lambda,$$

where the density is normalized such that $\int_{-\pi}^{\pi} f_x(\lambda) d\lambda = 1$. An estimate of $f(\lambda)$ can be obtained by the (smoothed) periodogram (see `mvspectrum`); thus using discrete, and not continuous entropy.

Value

A non-negative real value for the spectral entropy $H_s(x_t)$.

References

Jerry D. Gibson and Jaewoo Jung (2006). "The Interpretation of Spectral Entropy Based Upon Rate Distortion Functions". IEEE International Symposium on Information Theory, pp. 277-281.

L. L. Campbell, "Minimum coefficient rate for stationary random processes", Information and Control, vol. 3, no. 4, pp. 360 - 371, 1960.

See Also

`Omega`, `discrete_entropy`

Examples

```
set.seed(1)
eps <- rnorm(100)
spectral_entropy(eps)

phi.v <- seq(-0.95, 0.95, by = 0.1)
kMethods <- c("mvspec", "pgram")
SE <- matrix(NA, ncol = length(kMethods), nrow = length(phi.v))
for (ii in seq_along(phi.v)) {
  xx.tmp <- arima.sim(n = 200, list(ar = phi.v[ii]))
  for (mm in seq_along(kMethods)) {
    SE[ii, mm] <- spectral_entropy(xx.tmp, spectrum.control =
                                     list(method = kMethods[mm]))
  }
}

matplot(phi.v, SE, type = "l", col = seq_along(kMethods))
legend("bottom", kMethods, lty = seq_along(kMethods),
       col = seq_along(kMethods))

# AR vs MA
SE.arma <- matrix(NA, ncol = 2, nrow = length(phi.v))
SE.arma[, 1] <- SE[, 2]

for (ii in seq_along(phi.v)){
  yy.temp <- arima.sim(n = 1000, list(ma = phi.v[ii]))
  SE.arma[ii, 2] <-
    spectral_entropy(yy.temp, spectrum.control = list(method = "mvspec"))
}

matplot(phi.v, SE.arma, type = "l", col = 1:2, xlab = "parameter (phi or theta)",
        ylab = "Spectral entropy")
abline(v = 0, col = "blue", lty = 3)
legend("bottom", c("AR(1)", "MA(1)"), lty = 1:2, col = 1:2)
```

`whiten` *whitens multivariate data*

Description

`whiten` transforms a multivariate K-dimensional signal $\mathbf{X}$ with mean $\boldsymbol{\mu}_X$ and covariance matrix Σ_X to a *whitened* signal $\mathbf{U}$ with mean $\mathbf{0}$ and $\Sigma_U = I_K$. Thus it centers the signal and makes it contemporaneously uncorrelated. See Details.

`check_whitened` checks if data has been whitened; i.e., if it has zero mean, unit variance, and is uncorrelated.

`sqrt_matrix` computes the square root $\mathbf{B}$ of a square matrix $\mathbf{A}$. The matrix $\mathbf{B}$ satisfies $\mathbf{BB} = \mathbf{A}$.

Usage

```
whiten(data)

check_whitened(data, check.attribute.only = TRUE)

sqrt_matrix(mat, return.sqrt.only = TRUE, symmetric = FALSE)
```

Arguments

data	$n \times K$ array representing n observations of K variables.
check.attribute.only	logical; if TRUE it checks the attribute only. This is much faster (it just needs to look up one attribute value), but it might not surface silent bugs. For sake of performance the package uses the attribute version by default. However, for testing/debugging the full computational version can be used.
mat	a square $K \times K$ matrix.
return.sqrt.only	logical; if TRUE (default) it returns only the square root matrix; if FALSE it returns other auxiliary results (eigenvectors and eigenvalues, and inverse of the square root matrix).
symmetric	logical; if TRUE the eigen-solver assumes that the matrix is symmetric (which makes it much faster). This is in particular useful for a covariance matrix (which is used in whiten). Default: FALSE.

Details

whiten uses zero component analysis (ZCA) (aka zero-phase whitening filters) to whiten the data; i.e., it uses the inverse square root of the covariance matrix of $\mathbf{X}$ (see sqrt_matrix) as the whitening transformation. This means that on top of PCA, the uncorrelated principal components are back-transformed to the original space using the transpose of the eigenvectors. The advantage is that this makes them comparable to the original $\mathbf{X}$. See References for details.

The *square root* of a quadratic $n \times n$ matrix $\mathbf{A}$ can be computed by using the eigen-decomposition of $\mathbf{A}$

$$\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}',$$

where Λ is an $n \times n$ matrix with the eigenvalues $\lambda_1, \ldots, \lambda_n$ in the diagonal. The square root is simply $\mathbf{B} = \mathbf{V}\Lambda^{1/2}\mathbf{V}'$ where $\Lambda^{1/2} = diag(\lambda_1^{1/2}, \ldots, \lambda_n^{1/2})$.

Similarly, the *inverse square root* is defined as $\mathbf{A}^{-1/2} = \mathbf{V}\Lambda^{-1/2}\mathbf{V}'$, where $\Lambda^{-1/2} = diag(\lambda_1^{-1/2}, \ldots, \lambda_n^{-1/2})$ (provided that $\lambda_i \neq 0$).

Value

whiten returns a list with the whitened data, the transformation, and other useful quantities.

check_whitened throws an error if the input is not whitened, and returns (invisibly) the data with an attribute 'whitened' equal to TRUE. This allows to simply update data to have the attribute and thus only check it once on the actual data (slow) but then use the attribute lookup (fast).

`sqrt_matrix` returns an $n \times n$ matrix. If $\mathbf{A}$ is not semi-positive definite it returns a complex-valued $\mathbf{B}$ (since square root of negative eigenvalues are complex).

If `return.sqrt.only` = `FALSE` then it returns a list with:

`values`	eigenvalues of $\mathbf{A}$,
`vectors`	eigenvectors of $\mathbf{A}$,
`sqrt`	square root matrix $\mathbf{B}$,
`sqrt.inverse`	inverse of $\mathbf{B}$.

References

See appendix in http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

See http://ufldl.stanford.edu/wiki/index.php/Implementing_PCA/Whitening.

Examples

```
## Not run:
XX <- matrix(rnorm(100), ncol = 2) %*% matrix(runif(4), ncol = 2)
cov(XX)
UU <- whiten(XX)$U
cov(UU)

## End(Not run)
```

Index