# climextremes Documentation

***Release 0.3.1***

**Christopher Paciorek**

**Dec 13, 2023**

# CONTENTS:

climextremes.**calc_logReturnPeriod_fevd**(*fit=None*, *returnValue=None*, *covariates=None*, *upper=False*)

Calculates log return period and standard error given return value(s) of interest

**description**

Calculates log return period given return value(s) of interest, using model fit from extRemes::fevd . Standard error is obtained via the delta method. The return period is the average number of blocks expected to occur before the return value is exceeded and is equal to the inverse of the probability of exceeding the return value in a single block. For non-stationary models (those that include covariates for the location, scale, and/or shape parameters, log return periods and standard errors are returned for as many sets of covariates as provided.

**arguments**

fit: fitted object from extRemes fevd

returnValue: value(s) for which return period is desired

covariates: matrix of covariate values, each row a set of covariates for which the log return period is desired

upper: logical value indicating whether upper tail or lower tail is being considered

**details**

Results are calculated (and returned) on log scale as delta-method based standard errors are more accurate for the log period. Confidence intervals on the return period scale should be calculated by calculating a confidence interval for the log return period and exponentiating the endpoints of the interval.

climextremes.**calc_logReturnProbDiff_fevd**(*fit=None*, *returnValue=None*, *covariates1=None*, *covariates2=None*, *getSE=True*, *scaling=1.0*, *upper=False*)

Calculates log return probability difference for two sets of covariates and standard error of difference given return value(s) of interest

**description**

Calculates difference in log return probabilities for two sets of covariates given return value(s) of interest, using model fit from extRemes::fevd . Standard error is obtained via the delta method. The return probability is the probability of exceeding the return value in a single block. Differences and standard errors are returned for as many contrasts between covariate sets as provided.

**arguments**

fit: fitted object from extRemes fevd

returnValue: value(s) for which the log return probability difference is desired

covariates1: matrix of covariate values, each row a set of covariates for which the log return probability difference relative to the corresponding row of covariates2 is desired

covariates2: matrix of covariate values, each row a set of covariates

getSE: logical indicating whether standard error is desired, in addition to the point estimate

scaling: if returnValue is scaled for numerics, this allows names of output to be on original scale

upper: logical value indicating whether upper tail or lower tail is being considered

**details**

Results are calculated (and returned) on log scale as delta-method based standard errors are more accurate for the log probability. Confidence intervals for the ratio of return probabilities should be

calculated by calculating a confidence interval for the log probability difference and exponentiating the endpoints of the interval.

This is designed to calculate differences in log return probabilities and associated standard errors for different covariate values based on the same model fit. It is not designed for differences based on separate model fits, although it may be possible handle this case by' fit 'two models in a single model specification using dummy variables.

climextremes.**calc_logReturnProb_fevd**(*fit=None*, *returnValue=None*, *covariates=None*, *getSE=True*, *scaling=1.0*, *upper=False*)

Calculates log return probability and standard error given return value(s) of interest

**description**

Calculates log return probability given return value(s) of interest, using model fit from extRemes::fevd . Standard error is obtained via the delta method. The return probability is the probability of exceeding the return value in a single block. For non-stationary models (those that include covariates for the location, scale, and/or shape parameters, log probabilities and standard errors are returned for as many sets of covariates as provided.

**arguments**

fit: fitted object from extRemes fevd

returnValue: value(s) for which log return probability is desired

covariates: matrix of covariate values, each row a set of covariates for which the return probability is desired

getSE: logical indicating whether standard error is desired, in addition to the point estimate

scaling: if returnValue is scaled for numerics, this allows names of output to be on original scale

upper: logical value indicating whether upper tail or lower tail is being considered

**details**

Results are calculated (and returned) on log scale as delta-method based standard errors are more accurate for the log probability. Confidence intervals on the probability scale should be calculated by calculating a confidence interval for the log probability and exponentiating the endpoints of the interval.

climextremes.**calc_returnValueDiff_fevd**(*fit=None*, *returnPeriod=None*, *covariates1=None*, *covariates2=None*, *getSE=True*)

Calculates return value difference for two sets of covariates and standard error of difference given return period(s) of interest

**description**

Calculates difference in return values (also known as return levels) for two sets of covariates given return period(s) of interest, using model fit from extRemes::fevd . Standard error is obtained via the delta method. The return value is the value for which the expected number of blocks until an event that exceeds that value is equal to the return period. Differences and standard errors are returned for as many contrasts between covariate sets as provided.

**arguments**

fit: fitted object from extRemes fevd

returnPeriod: value(s) for which return value difference is desired

covariates1: matrix of covariate values, each row a set of covariates for which the return value difference relative to the corresponding row of covariates2 is desired

covariates2: matrix of covariate values, each row a set of covariates

getSE: logical indicating whether standard error is desired, in addition to the point estimate

**details**

This is designed to calculate differences in return values and associated standard errors for different covariate values based on the same model fit. It is not designed for differences based on separate model fits, although it may be possible handle this case by' fit 'two models in a single model specification using dummy variables.

climextremes.**calc_returnValue_fevd**(*fit=None*, *returnPeriod=None*, *covariates=None*)

Calculates return value and standard error given return period(s) of interest

**description**

Calculates return value (also known as the return level) given return period(s) of interest, using model fit from extRemes::fevd . Standard error is obtained via the delta method. The return value is the value for which the expected number of blocks until an event that exceeds that value is equal to the return period. For non-stationary models (those that include covariates for the location, scale, and/or shape parameters, return values and standard errors are returned for as many sets of covariates as provided.

**arguments**

fit: fitted object from extRemes fevd

returnPeriod: value(s) for which return value is desired

covariates: matrix of covariate values, each row a set of covariates for which the return value is desired

climextremes.**calc_riskRatio_binom**(*y=None*, *n=None*, *ciLevel=0.9*, *ciType=None*, *bootSE=None*, *bootControl=None*, *lrtControl=None*)

Compute risk ratio and uncertainty based on binomial models for counts of events relative to possible number of events

**description**

Compute risk ratio and uncertainty by fitting binomial models to counts of events relative to possible number of events. The risk ratio is the ratio of the probability of an event under the model fit to the first dataset to the probability under the model fit to the second dataset. Default standard errors are based on the usual MLE asymptotics using a delta-method-based approximation, but standard errors based on the nonparametric bootstrap and on a likelihood ratio procedure can also be computed.

**arguments**

y: numpy array of two values, the number of events in the two scenarios.

n: numpy array of two values, the number of samples (possible occurrences of events) in the two scenarios.

ciLevel: statistical confidence level for confidence intervals; in repeated experimentation, this proportion of confidence intervals should contain the true risk ratio. Note that if only one endpoint of the resulting interval is used, for example the lower bound, then the effective confidence level increases by half of one minus 'ciLevel'. For example, a two-sided 0.90 confidence interval corresponds to a one-sided 0.95 confidence interval.

ciType: string or numpy array of strings indicating which type of confidence intervals to compute. See 'Details'.

bootSE: boolean indicating whether to use the bootstrap to estimate the standard error of the risk ratio.

bootControl: dictionary of control parameters for the bootstrapping, used only when at least one bootstrap confidence interval is requested via 'ciType'. See 'Details'.

lrtControl: dictionary containing a single component, a numpy array named 'bounds', which sets the range inside which the algorithm searches for the endpoints of the likelihood ratio-based confidence interval. This

avoids numerical issues with endpoints converging to zero and infinity. If an endpoint is not found within the interval, it is set to 'nan'. Used only when ''lrt" is one of the 'ciType' values.

**details**

' ciType ' can include one or more of the following: 'delta' , 'koopman' , 'lrt' , 'boot_norm' , 'boot_perc' , 'boot_basic' , 'boot_stud' , 'boot_bca' . 'delta' uses the delta method to compute an asymptotic interval based on the standard error of the log risk ratio. 'koopman' uses the method described in Koopman (1984), following the implementation discussed in Fageland et al. (2015), including the calculation of Nam (1995). 'lrt' inverts a likelihood-ratio test. Bootstrap-based options are the normal-based interval using the bootstrap standard error ( 'boot_norm' ), the percentile bootstrap ( 'boot_perc' ), the basic bootstrap ( 'boot_basic' ), the bootstrap-t ( 'boot_stud' ), and the bootstrap BCA method ( 'boot_bca' ). See Paciorek et al. for more details.

See fit_pot for information on the ' bootControl ' argument.

**value**

The primary outputs of this function are as follows: the log of the risk ratio and standard error of that log risk ratio ('logRiskRatio' and 'se_logRiskRatio') as well the risk ratio itself ('riskRatio'). The standard error is based on the usual MLE asymptotics using a delta-method-based approximation. If requested via 'ciType', confidence intervals will be returned, as discussed in 'Details'.

**author**

Christopher J. Paciorek

**references**

Paciorek, C.J., D.A. Stone, and M.F. Wehner. 2018. Quantifying uncertainty in the attribution of human influence on severe weather. Weather and Climate Extremes 20:69-80. arXiv preprint <https://arxiv.org/abs/1706.03388>.

Koopman, P.A.R. 1984. Confidence intervals for the ratio of two binomial proportions. Biometrics 40: 513-517.

Fagerland, M.W., S. Lydersen, and P. Laake. 2015. Recommended confidence intervals for two independent binomial proportions. Statistical Methods in Medical Research 24: 224-254.

**examples**

```
>>> result = climextremes.calc_riskRatio_binom(numpy.array((40, 8)),
...                                            numpy.array((400, 400)),
...                                            ciType = numpy.array(('lrt', 'boot_
↪stud', 'koopman')))
... result['logRiskRatio']
... result['se_logRiskRatio']
... result['riskRatio']
... result['ci_riskRatio_lrt']
... result['ci_riskRatio_koopman']
... result['ci_riskRatio_boot_stud']
...
... # Koopman and LRT method can estimate lower confidence interval endpoint,
... # even if estimated risk ratio is infinity
... result = climextremes.calc_riskRatio_binom(numpy.array((4, 0)),
...                                            numpy.array((100, 100)),
...                                            ciType = numpy.array(('lrt', 'boot_
↪stud', 'koopman')))
... result['logRiskRatio']
... result['se_logRiskRatio']
```

(continues on next page)

(continued from previous page)

```
... result['riskRatio']
... result['ci_riskRatio_lrt']
... result['ci_riskRatio_koopman']
... result['ci_riskRatio_boot_stud']
...
...
```

climextremes.**calc_riskRatio_gev**(*returnValue=None*, *y1=None*, *y2=None*, *x1=None*, *x2=None*, *locationFun1=None*, *locationFun2=None*, *scaleFun1=None*, *scaleFun2=None*, *shapeFun1=None*, *shapeFun2=None*, *nReplicates1=1.0*, *nReplicates2=1.0*, *replicateIndex1=None*, *replicateIndex2=None*, *weights1=None*, *weights2=None*, *xNew1=None*, *xNew2=None*, *maxes=True*, *scaling1=1.0*, *scaling2=1.0*, *ciLevel=0.9*, *ciType=None*, *bootSE=None*, *bootControl=None*, *lrtControl=None*, *optimArgs=None*, *optimControl=None*, *initial1=None*, *initial2=None*, *logScale1=None*, *logScale2=None*, *getReturnCalcs=False*, *getParams=False*, *getFit=False*)

Compute risk ratio and uncertainty based on generalized extreme value model fit to block maxima or minima

**description**

Compute risk ratio and uncertainty by fitting generalized extreme value model, designed specifically for climate data, to exceedance-only data, using the point process approach. The risk ratio is the ratio of the probability of exceedance of a pre-specified value under the model fit to the first dataset to the probability under the model fit to the second dataset. Default standard errors are based on the usual MLE asymptotics using a delta-method-based approximation, but standard errors based on the nonparametric bootstrap and on a likelihood ratio procedure can also be computed.

**arguments**

returnValue: numeric value giving the value for which the risk ratio should be calculated, where the resulting period will be the average number of blocks until the value is exceeded and the probability the probability of exceeding the value in any single block.

y1: numpy array of observed maxima or minima values for the first dataset. See 'Details' for how the values of 'y1' should be ordered if there are multiple replicates and the values of 'x1' are identical for all replicates. For better optimization performance, it is recommended that the 'y1' have magnitude around one (see 'Details'), for which one can use 'scaling1'.

y2: numpy array of observed maxima or minima values for the second dataset. Analogous to 'y1'.

x1: numpy array or pandas data frame with columns corresponding to covariate/predictor/feature variables and each row containing the values of the variable for the corresponding observed maximum/minimum. The number of rows should either equal the length of 'y1' or (if there is more than one replicate) it can optionally equal the number of observations in a single replicate, in which case the values will be assumed to be the same for all replicates.

x2: analogous to 'x1' but for the second dataset

locationFun1: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the location parameter using columns from 'x1' for the first dataset. 'x1' must be supplied if this is anything other than 'None'.

locationFun2: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the location parameter using columns from 'x2' for the second dataset. 'x2' must be supplied if this is anything other than 'None'.

scaleFun1: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the (potentially transformed) scale parameter using columns from 'x1' for the first dataset. 'x1' must be

supplied if this is anything other than 'None'. 'logscale1' controls whether this determines the log of the scale or the scale directly.

scaleFun2: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the (potentially transformed) scale parameter using columns from 'x2' for the second dataset. 'x2' must be supplied if this is anything other than 'None'. 'logscale2' controls whether this determines the log of the scale or the scale directly.

shapeFun1: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the shape parameter using columns from 'x1' for the first dataset. 'x1' must be supplied if this is anything other than 'None'.

shapeFun2: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the shape parameter using columns from 'x2' for the first dataset. 'x2' must be supplied if this is anything other than 'None'.

nReplicates1: numeric value indicating the number of replicates for the first dataset.

nReplicates2: numeric value indicating the number of replicates for the second dataset.

replicateIndex1: numpy array providing the index of the replicate corresponding to each element of 'y1'. Used (and therefore required) only when using bootstrapping with the resampling by replicates based on the 'by' element of 'bootControl'.

replicateIndex2: numpy array providing the index of the replicate corresponding to each element of 'y2'. Analogous to 'replicateIndex1'.

weights1: numpy array providing the weights for each observation in the first dataset. When there is only one replicate or the weights do not vary by replicate, an array of length equal to the number of observations. When weights vary by replicate, this should be of equal length to 'y'. Likelihood contribution of each observation is multiplied by the corresponding weight.

weights2: numpy array providing the weights for each observation in the second dataset. Analogous to 'weights1'.

xNew1: object of the same form as 'x1', providing covariate/predictor/feature values for which one desires log risk ratios.

xNew2: object of the same form as 'x2', providing covariate/predictor/feature values for which log risk ratios are desired. Must provide the same number of covariate sets as 'xNew1' as the risk ratio is based on contrasting return probabilities under 'xNew1' and 'xNew2'.

maxes: boolean indicating whether analysis is for block maxima ('True') or block minima ('False'); in the latter case, the function works with the negative of the values, changing the sign of the resulting location parameters

scaling1: positive-valued scalar used to scale the data values of the first dataset for more robust optimization performance. When multiplied by the values, it should produce values with magnitude around 1.

scaling2: positive-valued scalar used to scale the data values of the second dataset for more robust optimization performance. When multiplied by the values, it should produce values with magnitude around 1.

ciLevel: statistical confidence level for confidence intervals; in repeated experimentation, this proportion of confidence intervals should contain the true risk ratio. Note that if only one endpoint of the resulting interval is used, for example the lower bound, then the effective confidence level increases by half of one minus 'ciLevel'. For example, a two-sided 0.90 confidence interval corresponds to a one-sided 0.95 confidence interval.

ciType: string or numpy array of strings indicating which type of confidence intervals to compute. See 'Details'.

bootSE: boolean indicating whether to use the bootstrap to estimate the standard error of the risk ratio.

bootControl: dictionary of control parameters for the bootstrapping, used only when at least one bootstrap confidence interval is requested via 'ciType'. See 'Details'.

lrtControl: dictionary containing a single component, a numpy array named 'bounds', which sets the range inside which the algorithm searches for the endpoints of the likelihood ratio-based confidence interval. This avoids numerical issues with endpoints converging to zero and infinity. If an endpoint is not found within the interval, it is set to 'nan'. Used only when ''lrt" is one of the 'ciType' values.

optimArgs: a dictionary with named components matching exactly any arguments that the user wishes to pass to R's 'optim' function. See 'help(optim)' in R for details. Of particular note, ''method" can be used to choose the optimization method used for maximizing the log-likelihood to fit the model (e.g., ''method" could be 'BFGS' instead of the default 'Nelder-Mead'). To specify the 'control' argument, use 'optimControl' rather than including 'control' here.

optimControl: a dictionary with named components matching exactly any elements that the user wishes to pass as the 'control' argument to R's 'optim' function. For example, 'control={'maxit': VALUE)' for a user-specified VALUE can be used to increase the number of iterations if the optimization is converging slowly.

initial1: a dictionary with components named ''location", ''scale", and ''shape" providing initial parameter values, intended for use in speeding up or enabling optimization when the default initial values are resulting in failure of the optimization; note that use of 'scaling', 'logScale' and '.normalizeX = True' cause numerical changes in some of the parameters. For example with 'logScale1 = True', initial value(s) for ''scale" should be specified on the log scale.

initial2: a dictionary of initial parameter values for the second dataset, analogous to 'initial1'.

logScale1: boolean indicating whether optimization for the scale parameter should be done on the log scale for the first dataset. By default this is 'False' when the scale is not a function of covariates and 'True' when the scale is a function of covariates (to ensure the scale is positive regardless of the regression coefficients).

logScale2: boolean indicating whether optimization for the scale parameter should be done on the log scale for the second dataset. By default this is 'False' when the scale is not a function of covariates and 'True' when the scale is a function of covariates (to ensure the scale is positive regardless of the regression coefficients).

getReturnCalcs: boolean indicating whether to return the estimated return values/probabilities/periods from the fitted models.

getParams: boolean indicating whether to return the fitted parameter values and their standard errors for the fitted models; WARNING: parameter values for models with covariates for the scale parameter must interpreted based on the value of 'logScale'.

getFit: boolean indicating whether to return the full fitted models (potentially useful for model evaluation and for understanding optimization problems); note that estimated parameters in the fit object for nonstationary models will not generally match the MLE provided when 'getParams' is 'True' because covariates are normalized before fitting and the fit object is based on the normalized covariates. Similarly, parameters will not match if 'scaling' is not 1.

**details**

> See fit_gev for more details on fitting the block maxima model for each dataset, including details on blocking and replication. Also see fit_gev for information on the ' bootControl ' argument.
>
> Optimization failures:
>
> It is not uncommon for maximization of the log-likelihood to fail for extreme value models. Please see the help information for fit_gev . Also note that if the probability in the denominator of the risk ratio is near one, one may achieve better numerical performance by swapping the two datasets and computing the risk ratio for the probability under dataset 2 relative to the probability under dataset 1.

' ciType ' can include one or more of the following: 'delta' , 'lrt' , 'boot_norm' , 'boot_perc' , 'boot_basic' , 'boot_stud' , 'boot_bca' . 'delta' uses the delta method to compute an asymptotic interval based on the standard error of the log risk ratio. 'lrt' inverts a likelihood-ratio test. Bootstrap-based options are the normal-based interval using the bootstrap standard error ( 'boot_norm' ), the percentile bootstrap ( 'boot_perc' ), the basic

bootstrap ( 'boot_basic' ), the bootstrap-t ( 'boot_stud' ), and the bootstrap BCA method ( 'boot_bca' ). See Paciorek et al. for more details.

See fit_pot for information on the ' bootControl ' argument.

**value**

The primary outputs of this function are as follows: the log of the risk ratio and standard error of that log risk ratio ('logRiskRatio' and 'se_logRiskRatio') as well the risk ratio itself ('riskRatio'). The standard error is based on the usual MLE asymptotics using a delta-method-based approximation. If requested via 'ciType', confidence intervals will be returned, as discussed in 'Details'.

**author**

Christopher J. Paciorek

**references**

Paciorek, C.J., D.A. Stone, and M.F. Wehner. 2018. Quantifying uncertainty in the attribution of human influence on severe weather. Weather and Climate Extremes 20:69-80. arXiv preprint <https://arxiv.org/abs/1706.03388>.

Jeon S., C.J. Paciorek, and M.F. Wehner. 2016. Quantile-based bias correction and uncertainty quantification of extreme event attribution statements. Weather and Climate Extremes 12: 24-32. <DOI:10.1016/j.wace.2016.02.001>. arXiv preprint: <http://arxiv.org/abs/1602.04139>.

**examples**

```
>>> Fort = climextremes.Fort
... earlyPeriod = numpy.array((1900, 1930))
... earlyYears = numpy.array(range(earlyPeriod[0], earlyPeriod[1]))
... latePeriod = numpy.array((1970, 2000))
... lateYears = numpy.array(range(latePeriod[0], latePeriod[1]))
...
... FortMax = Fort.groupby('year').max()[['Prec']]
... FortMax.reset_index(inplace=True)
...
... y1 = FortMax.Prec[FortMax.year < earlyPeriod[1]]
... y2 = FortMax.Prec[FortMax.year >= latePeriod[0]]
...
... # contrast late period with early period, assuming a nonstationary fit
... # within each time period and finding RR based on midpoint of each period
... result = climextremes.calc_riskRatio_gev(
...     returnValue = 3,
...     y1 = numpy.array(y1), y2 = numpy.array(y2),
...     x1 = earlyYears, x2 = lateYears,
...     locationFun1 = 1, locationFun2 = 1,
...     xNew1 = earlyYears.mean(), xNew2 = lateYears.mean(),
...     ciType = 'lrt')
...
... result['logRiskRatio']
... result['se_logRiskRatio']
... result['riskRatio']
... result['ci_riskRatio_lrt']
...
...
```

climextremes.**calc_riskRatio_pot**(*returnValue=None*, *y1=None*, *y2=None*, *x1=None*, *x2=None*, *threshold1=None*, *threshold2=None*, *locationFun1=None*, *locationFun2=None*, *scaleFun1=None*, *scaleFun2=None*, *shapeFun1=None*, *shapeFun2=None*, *nBlocks1=None*, *nBlocks2=None*, *blockIndex1=None*, *blockIndex2=None*, *firstBlock1=1.0*, *firstBlock2=1.0*, *index1=None*, *index2=None*, *nReplicates1=1.0*, *nReplicates2=1.0*, *replicateIndex1=None*, *replicateIndex2=None*, *weights1=None*, *weights2=None*, *proportionMissing1=None*, *proportionMissing2=None*, *xNew1=None*, *xNew2=None*, *declustering=None*, *upperTail=True*, *scaling1=1.0*, *scaling2=1.0*, *ciLevel=0.9*, *ciType=None*, *bootSE=None*, *bootControl=None*, *lrtControl=None*, *optimArgs=None*, *optimControl=None*, *initial1=None*, *initial2=None*, *logScale1=None*, *logScale2=None*, *getReturnCalcs=False*, *getParams=False*, *getFit=False*)

Compute risk ratio and uncertainty based on peaks-over-threshold models fit to exceedances over a threshold

**description**

Compute risk ratio and uncertainty by fitting peaks-over-threshold model, designed specifically for climate data, to exceedance-only data, using the point process approach. The risk ratio is the ratio of the probability of exceedance of a pre-specified value under the model fit to the first dataset to the probability under the model fit to the second dataset. Default standard errors are based on the usual MLE asymptotics using a delta-method-based approximation, but standard errors based on the nonparametric bootstrap and on a likelihood ratio procedure can also be computed.

**arguments**

returnValue: numeric value giving the value for which the risk ratio should be calculated, where the resulting period will be the average number of blocks until the value is exceeded and the probability the probability of exceeding the value in any single block.

y1: numpy array of exceedance values for the first dataset (values of the outcome variable above the threshold). For better optimization performance, it is recommended that the 'y' have magnitude around one (see 'Details'), for which one can use 'scaling1'.

y2: numpy array of exceedance values for the second dataset (values of the outcome variable above the threshold).

x1: numpy array or pandas data frame with columns corresponding to covariate/predictor/feature variables and each row containing the values of the variable for a block (e.g., often a year with climate data) for the first dataset. The number of rows must equal the number of blocks.

x2: analogous to 'x1' but for the second dataset

threshold1: a single numeric value for constant threshold or a numpy array with length equal to the number of blocks, indicating the threshold for each block for the first dataset.

threshold2: analogous to 'threshold1' but for the second dataset

locationFun1: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the location parameter using columns from 'x1' for the first dataset. 'x1' must be supplied if this is anything other than 'None'.

locationFun2: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the location parameter using columns from 'x2' for the second dataset. 'x2' must be supplied if this is anything other than 'None'.

scaleFun1: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the (potentially transformed) scale parameter using columns from 'x1' for the first dataset. 'x1' must be supplied if this is anything other than 'None'. 'logscale1' controls whether this determines the log of the scale or the scale directly.

scaleFun2: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the (potentially transformed) scale parameter using columns from 'x2' for the second dataset. 'x2' must be supplied if this is anything other than 'None'. 'logscale2' controls whether this determines the log of the scale or the scale directly.

shapeFun1: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the shape parameter using columns from 'x1' for the first dataset. 'x1' must be supplied if this is anything other than 'None'.

shapeFun2: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the shape parameter using columns from 'x2' for the first dataset. 'x2' must be supplied if this is anything other than 'None'.

nBlocks1: number of blocks (e.g., a block will often be a year with climate data) in first dataset; note this value determines the interpretation of return values/periods/probabilities; see 'returnPeriod' and 'returnValue'.

nBlocks2: number of blocks (e.g., a block will often be a year with climate data) in second dataset; note this value determines the interpretation of return values/periods/probabilities; see 'returnPeriod' and 'returnValue'.

blockIndex1: numpy array providing the index of the block corresponding to each element of 'y1'. Used only when 'x1' is provided to match exceedances to the covariate/predictor/feature value for the exceedance or when using bootstrapping with the resampling based on blocks based on the 'by' element of 'bootControl'. If 'firstBlock1' is not equal to one, then 'blockIndex1' need not have one as its smallest possible value.

blockIndex2: numpy array providing the index of the block corresponding to each element of 'y2'. Analogous to 'blockIndex1'.

firstBlock1: single numeric value indicating the numeric value of the first possible block of 'blockIndex1'. For example the values in 'blockIndex1' might indicate the year of each exceedance with the first year of data being 1969, in which case 'firstBlock1' would be 1969. Note that the first block may not have any exceedances so it may not be represented in 'blockIndex1'. Used only to adjust 'blockIndex1' so that the block indices start at one and therefore correspond to the rows of 'x1'.

firstBlock2: single numeric value indicating the numeric value of the first possible block of 'blockIndex2'. Analogous to 'firstBlock1'.

index1: (optional) numpy array providing the integer-valued index (e.g., julian day for daily climate data) corresponding to each element of 'y1'. For example if there are 10 original observations and the third, fourth, and seventh values are exceedances, then 'index1' would be the vector 3,4,7. Used only when 'declustering' is provided to determine which exceedances occur sequentially or within a contiguous set of values of a given length. The actual values are arbitrary; only the lags between the values are used.

index2: (optional) numpy array providing the integer-valued index (e.g., julian day for daily climate data) corresponding to each element of 'y2'. Analogous to 'index1'.

nReplicates1: numeric value indicating the number of replicates for the first dataset.

nReplicates2: numeric value indicating the number of replicates for the second dataset.

replicateIndex1: numpy array providing the index of the replicate corresponding to each element of 'y1'. Used for three purposes: (1) when using bootstrapping with the resampling based on replicates based on the 'by' element of 'bootControl', (2) to avoid treating values in different replicates as potentially being sequential or within a short interval when removing values based on 'declustering', and (3) to match outcomes to 'weights' or 'proportionMissing' when either vary by replicate.

replicateIndex2: numpy array providing the index of the replicate corresponding to each element of 'y2'. Analogous to 'replicateIndex1'.

weights1: numpy array providing the weights by block for the first dataset. When there is only one replicate or the weights do not vary by replicate, a one-dimensional array of length equal to the number of blocks. When

weights vary by replicate, a two-dimensional array with rows corresponding to blocks and columns to replicates. Likelihood contribution of each block is multiplied by the corresponding weight.

weights2: numpy array providing the weights by block for the second dataset. Analogous to 'weights1'.

proportionMissing1: a numeric value or numpy array indicating the proportion of missing values in the original first dataset before exceedances were selected. When the proportion missing is the same for all blocks and replicates, a single value. When there is only one replicate or the weights do not vary by replicate, a one-dimensional array of length equal to the number of blocks. When weights vary by replicate, a two-dimensional array with rows corresponding to blocks and columns to replicates.

proportionMissing2: a numeric value or numpy array indicating the proportion of missing values in the original second dataset before exceedances were selected. Analogous to 'proportionMissing1'.

xNew1: object of the same form as 'x1', providing covariate/predictor/feature values for which log risk ratios are desired.

xNew2: object of the same form as 'x2', providing covariate/predictor/feature values for which log risk ratios are desired. Must provide the same number of covariate sets as 'xNew1' as the risk ratio is based on contrasting return probabilities under 'xNew1' and 'xNew2'.

declustering: one of 'None', ''noruns", or a number. If ''noruns" is specified, only the maximum (or minimum if 'upperTail = False') value within a set of exceedances corresponding to successive indices is included. If a number, this should indicate the size of the interval (which will be used with the 'index' argument) within which to allow only the largest (or smallest if 'upperTail = False') value.

upperTail: boolean indicating whether one is working with exceedances over a high threshold ('True') or exceedances under a low threshold ('False'); in the latter case, the function works with the negative of the values and the threshold, changing the sign of the resulting location parameters.

scaling1: positive-valued scalar used to scale the data values of the first dataset for more robust optimization performance. When multiplied by the values, it should produce values with magnitude around 1.

scaling2: positive-valued scalar used to scale the data values of the second dataset for more robust optimization performance. When multiplied by the values, it should produce values with magnitude around 1.

ciLevel: statistical confidence level for confidence intervals; in repeated experimentation, this proportion of confidence intervals should contain the true risk ratio. Note that if only one endpoint of the resulting interval is used, for example the lower bound, then the effective confidence level increases by half of one minus 'ciLevel'. For example, a two-sided 0.90 confidence interval corresponds to a one-sided 0.95 confidence interval.

ciType: numpy string or numpy array of strings indicating which type of confidence intervals to compute. See 'Details'.

bootSE: boolean indicating whether to use the bootstrap to estimate the standard error of the risk ratio.

bootControl: dictionary of control parameters for the bootstrapping, used only when at least one bootstrap confidence interval is requested via 'ciType'. See 'Details'.

lrtControl: dictionary containing a single component, a numpy array named 'bounds', which sets the range inside which the algorithm searches for the endpoints of the likelihood ratio-based confidence interval. This avoids numerical issues with endpoints converging to zero and infinity. If an endpoint is not found within the interval, it is set to 'nan'. Used only when ''lrt" is one of the 'ciType' values.

optimArgs: a dictionary with named components matching exactly any arguments that the user wishes to pass to R's 'optim' function. See 'help(optim)' in R for details. Of particular note, ''method" can be used to choose the optimization method used for maximizing the log-likelihood to fit the model (e.g., ''method" could be 'BFGS' instead of the default 'Nelder-Mead'). To specify the 'control' argument, use 'optimControl' rather than including 'control' here.

optimControl: a dictionary with named components matching exactly any elements that the user wishes to pass as the 'control' argument to R's 'optim' function. For example, 'control={'maxit': VALUE)' for a user-specified

VALUE can be used to increase the number of iterations if the optimization is converging slowly.

initial1: a dictionary with components named ‘’location”, ‘’scale”, and ‘’shape” providing initial parameter values, intended for use in speeding up or enabling optimization when the default initial values are resulting in failure of the optimization; note that use of ‘scaling’, ‘logScale’ and ‘.normalizeX = True’ cause numerical changes in some of the parameters. For example with ‘logScale1 = True’, initial value(s) for ‘’scale” should be specified on the log scale.

initial2: a dictionary of initial parameter values for the second dataset, analogous to ‘initial1’.

logScale1: boolean indicating whether optimization for the scale parameter should be done on the log scale for the first dataset. By default this is ‘False’ when the scale is not a function of covariates and ‘True’ when the scale is a function of covariates (to ensure the scale is positive regardless of the regression coefficients).

logScale2: boolean indicating whether optimization for the scale parameter should be done on the log scale for the second dataset. By default this is ‘False’ when the scale is not a function of covariates and ‘True’ when the scale is a function of covariates (to ensure the scale is positive regardless of the regression coefficients).

getReturnCalcs: boolean indicating whether to return the estimated return values/probabilities/periods from the fitted models.

getParams: boolean indicating whether to return the fitted parameter values and their standard errors for the fitted models; WARNING: parameter values for models with covariates for the scale parameter must interpreted based on the value of ‘logScale’.

getFit: boolean indicating whether to return the full fitted models (potentially useful for model evaluation and for understanding optimization problems); note that estimated parameters in the fit object for nonstationary models will not generally match the MLE provided when ‘getParams’ is ‘True’ because covariates are normalized before fitting and the fit object is based on the normalized covariates. Similarly, parameters will not match if ‘scaling’ is not 1.

**details**

See fit_pot for more details on fitting the peaks-over-threshold model for each dataset, including details on blocking and replication. Also see fit_pot for information on the ‘ bootControl ‘ argument.

Optimization failures:

It is not uncommon for maximization of the log-likelihood to fail for extreme value models. Please see the help information for fit_pot . Also note that if the probability in the denominator of the risk ratio is near one, one may achieve better numerical performance by swapping the two datasets and computing the risk ratio for the probability under dataset 2 relative to the probability under dataset 1.

‘ ciType ‘ can include one or more of the following: ‘delta’ , ‘lrt’ , ‘boot_norm’ , ‘boot_perc’ , ‘boot_basic’ , ‘boot_stud’ , ‘boot_bca’ . ‘delta’ uses the delta method to compute an asymptotic interval based on the standard error of the log risk ratio. ‘lrt’ inverts a likelihood-ratio test. Bootstrap-based options are the normal-based interval using the bootstrap standard error ( ‘boot_norm’ ), the percentile bootstrap ( ‘boot_perc’ ), the basic bootstrap ( ‘boot_basic’ ), the bootstrap-t ( ‘boot_stud’ ), and the bootstrap BCA method ( ‘boot_bca’ ). See Paciorek et al. for more details.

See fit_pot for information on the ‘ bootControl ‘ argument.

**value**

The primary outputs of this function are as follows: the log of the risk ratio and standard error of that log risk ratio (‘logRiskRatio’ and ‘se_logRiskRatio’) as well the risk ratio itself (‘riskRatio’). The standard error is based on the usual MLE asymptotics using a delta-method-based approximation. If requested via ‘ciType’, confidence intervals will be returned, as discussed in ‘Details’.

**author**

Christopher J. Paciorek

**references**

Paciorek, C.J., D.A. Stone, and M.F. Wehner. 2018. Quantifying uncertainty in the attribution of human influence on severe weather. Weather and Climate Extremes 20:69-80. arXiv preprint <https://arxiv.org/abs/1706.03388>.

Jeon S., C.J. Paciorek, and M.F. Wehner. 2016. Quantile-based bias correction and uncertainty quantification of extreme event attribution statements. Weather and Climate Extremes 12: 24-32. <DOI:10.1016/j.wace.2016.02.001>. arXiv preprint: <http://arxiv.org/abs/1602.04139>.

**examples**

```
>>> Fort = climextremes.Fort
... threshold = 0.395
... FortExc = Fort[Fort.Prec > threshold]
...
... earlyPeriod = numpy.array((1900, 1930))
... earlyYears = numpy.array(range(earlyPeriod[0], earlyPeriod[1]))
... latePeriod = numpy.array((1970, 2000))
... lateYears = numpy.array(range(latePeriod[0], latePeriod[1]))
...
... y1 = FortExc.Prec[FortExc.year < earlyPeriod[1]]
... y2 = FortExc.Prec[FortExc.year >= latePeriod[0]]
... block1 = FortExc.year[FortExc.year < earlyPeriod[1]]
... block2 = FortExc.year[FortExc.year >= latePeriod[0]]
...
... # contrast late period with early period, assuming a nonstationary fit
... # within each time period and finding RR based on midpoint of each period
... result = climextremes.calc_riskRatio_pot(
...     returnValue = 3,
...     y1 = numpy.array(y1), y2 = numpy.array(y2),
...     x1 = earlyYears, x2 = lateYears,
...     threshold1 = threshold, threshold2 = threshold,
...     locationFun1 = 1, locationFun2 = 1,
...     xNew1 = earlyYears.mean(), xNew2 = lateYears.mean(),
...     blockIndex1 = numpy.array(block1), blockIndex2 = numpy.array(block2),
...     firstBlock1 = earlyYears[0], firstBlock2 = lateYears[0],
...     ciType = 'lrt')
...
... result['logRiskRatio']
... result['se_logRiskRatio']
... result['riskRatio']
... result['ci_riskRatio_lrt']
...
...
```

climextremes.**compute_input_map**(*input_arg_map*)

climextremes.**fit_gev**(*y=None*, *x=None*, *locationFun=None*, *scaleFun=None*, *shapeFun=None*, *nReplicates=1.0*, *replicateIndex=None*, *weights=None*, *returnPeriod=None*, *returnValue=None*, *getParams=False*, *getFit=False*, *xNew=None*, *xContrast=None*, *maxes=True*, *scaling=1.0*, *bootSE=False*, *bootControl=None*, *optimArgs=None*, *optimControl=None*, *missingFlag=None*, *initial=None*, *logScale=None*, *_normalizeX=True*, *_getInputs=False*, *_allowNoInt=True*)

Fit a generalized extreme value model to block maxima or minima

**description**

Fit a generalized extreme value model, designed specifically for climate data. It includes options for variable weights (useful for local likelihood), as well as for bootstrapping to estimate uncertainties. Results can be returned in terms of parameter values, return values, return periods, return probabilities, and differences in either return values or log return probabilities (i.e., log risk ratios).

**arguments**

y: a numpy array of observed maxima or minima values. See 'Details' for how the values of 'y' should be ordered if there are multiple replicates and the values of 'x' are identical for all replicates. For better optimization performance, it is recommended that the 'y' have magnitude around one (see 'Details'), for which one can use 'scaling'.

x: a numpy array or pandas data frame with columns corresponding to covariate/predictor/feature variables and each row containing the values of the variable for the corresponding observed maximum/minimum. The number of rows should either equal the length of 'y' or (if there is more than one replicate) it can optionally equal the number of observations in a single replicate, in which case the values will be assumed to be the same for all replicates.

locationFun: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the location parameter using columns from 'x'. 'x' must be supplied if this is anything other than 'None'.

scaleFun: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the (potentially transformed) scale parameter using columns from 'x'. 'x' must be supplied if this is anything other than 'None'. 'logscale' controls whether this determines the log of the scale or the scale directly.

shapeFun: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the shape parameter using columns from 'x'. 'x' must be supplied if this is anything other than 'None'.

nReplicates: numeric value indicating the number of replicates.

replicateIndex: numpy array providing the index of the replicate corresponding to each element of 'y'. Used (and therefore required) only when using bootstrapping with the resampling by replicates based on the 'by' element of 'bootControl'.

weights: a numpy array providing the weights for each observation. When there is only one replicate or the weights do not vary by replicate, an array of length equal to the number of observations. When weights vary by replicate, this should be of equal length to 'y'. Likelihood contribution of each observation is multiplied by the corresponding weight.

returnPeriod: numeric value giving the number of blocks for which return values should be calculated. For example a returnPeriod of 20 corresponds to the value of an event that occurs with probability 1/20 in any block and therefore occurs on average every 20 blocks. Often blocks will correspond to years.

returnValue: numeric value giving the value for which return probabilities/periods should be calculated, where the period would be the average number of blocks until the value is exceeded and the probability the probability of exceeding the value in any single block.

getParams: boolean indicating whether to return the fitted parameter values and their standard errors; WARNING: parameter values for models with covariates for the scale parameter must interpreted based on the value of 'logScale'.

getFit: boolean indicating whether to return the full fitted model (potentially useful for model evaluation and for understanding optimization problems); note that estimated parameters in the fit object for nonstationary models will not generally match the MLE provided when 'getParams' is 'True' because covariates are normalized before fitting and the fit object is based on the normalized covariates. Similarly, parameters will not match if 'scaling' is not 1.

xNew: object of the same form as 'x', providing covariate/predictor/feature values for which return values/periods/probabilities are desired.

xContrast: object of the same form and dimensions as 'xNew', providing covariate/predictor/feature values for which to calculate the differences of the return values and/or log return probabilities relative to the values in 'xNew'. This provides a way to estimate the difference in return value or log return probabilities (i.e., log risk ratios).

maxes: boolean indicating whether analysis is for block maxima ('True') or block minima ('False'); in the latter case, the function works with the negative of the values, changing the sign of the resulting location parameters

scaling: positive-valued number used to scale the data values for more robust optimization performance. When multiplied by the values, it should produce values with magnitude around 1.

bootSE: boolean indicating whether to use the bootstrap to estimate standard errors.

bootControl: a dictionary of control parameters for the bootstrapping. See 'Details'.

optimArgs: a dictionary with named components matching exactly any arguments that the user wishes to pass to R's 'optim' function. See 'help(optim)' in R for details. Of particular note, ''method" can be used to choose the optimization method used for maximizing the log-likelihood to fit the model (e.g., ''method" could be 'BFGS' instead of the default 'Nelder-Mead'). To specify the 'control' argument, use 'optimControl' rather than including 'control' here.

optimControl: a dictionary with named components matching exactly any elements that the user wishes to pass as the 'control' argument to R's 'optim' function. For example, 'control={'maxit': VALUE)' for a user-specified VALUE can be used to increase the number of iterations if the optimization is converging slowly.

missingFlag: optional value to be interpreted as missing values (instead of default 'numpy.nan')

initial: a dictionary with components named ''location", ''scale", and ''shape" providing initial parameter values, intended for use in speeding up or enabling optimization when the default initial values are resulting in failure of the optimization; note that use of 'scaling', 'logScale', and '.normalizeX = True' cause numerical changes in some of the parameters. For example with 'logScale = True', initial value(s) for ''scale" should be specified on the log scale.

logScale: boolean indicating whether optimization for the scale parameter should be done on the log scale. By default this is 'False' when the scale is not a function of covariates and 'True' when the scale is a function of covariates (to ensure the scale is positive regardless of the regression coefficients).

.normalizeX: boolean indicating whether to normalize 'x' values for better numerical performance; default is 'True'.

.getInputs: boolean indicating whether to return intermediate objects used in fitting. Defaults to 'False' and intended for internal use only

**details**

This function allows one to fit stationary or nonstationary block maxima/minima models using the generalized extreme value distribution. The function can return parameter estimates, return value/level for a given return period (number of blocks), and return probabilities/periods for a given return value/level. The function provides standard errors based on the usual MLE asymptotics, with delta-method-based standard errors for functionals of the parameters, but also standard errors based on the nonparametric bootstrap, either resampling by block or by replicate or both.

Replicates:

Replicates are repeated datasets, each with the same structure, including the same number of block maxima/minima. The additional observations in multiple replicates could simply be treated as additional blocks without replication (see next paragraph), but when the covariate values and' weights 'are the same across replicates, it is simpler to make use of ' nReplicates ' and ' replicateIndex '.

When using multiple replicates (e.g., multiple members of a climate model' initial 'condition ensemble), the standard input format is to append observations for additional replicates to the ' y ' argument and indicate the replicate ID for each value via ' replicateIndex ', which would be of the form 1,1,1,...2,2,2,...3,3,3,... etc. The values for each replicate should be grouped together and in the same order within replicate so that ' x ' can be correctly matched to the ' y ' values when ' x ' is only supplied for the first replicate. In other words, ' y ' should first contain all the values for the first replicate, then all the values for the second replicate in the same block order as for the first replicate, and so forth. Note that if ' y ' is provided as a matrix with the number of rows equal to the number of observations in each replicate and the columns corresponding to replicates, this ordering will occur naturally.

However, if one has different covariate values for different replicates, then one needs to treat the additional replicates as providing additional blocks, with only a single replicate (and ' nReplicates ' set to 1). The covariate values can then be included as additional rows in ' x '. Similarly, if there is a varying number of replicates by block, then all block-replicate pairs should be treated as individual blocks with a corresponding row in ' x ' (and ' nReplicates ' set to 1).

' bootControl ' arguments:

The ' bootControl ' argument is a list (or dictionary when calling from Python) that can supply any of the following components:

seed. Value of the random number seed as a single value, or in the form of .Random.seed , to set before doing resampling. Defaults to 1 . n. Number of bootstrap samples. Defaults to 250 . by. Character string, one of 'block' , 'replicate' , or 'joint' , indicating the basis for the resampling. If 'block' , resampled datasets will consist of blocks drawn at random from the original set of blocks; if there are replicates, each replicate will occur once for every resampled block. If 'replicate' , resampled datasets will consist of replicates drawn at random from the original set of replicates; all blocks from a replicate will occur in each resampled replicate. Note that this preserves any dependence across blocks rather than assuming independence between blocks. If 'joint' resampled datasets will consist of block-replicate pairs drawn at random from the original set of block-replicate pairs. Defaults to 'block' . getSample. Logical/boolean indicating whether the user wants the full bootstrap sample of parameter estimates and/or return value/period/probability information returned for use in subsequent calculations; if False (the default), only the bootstrap-based estimated standard errors are returned.

Optimization failures:

It is not uncommon for maximization of the log-likelihood to fail for extreme value models. Users should carefully check the info element of the return object to ensure that the optimization converged. For better optimization performance, it is recommended that the observations be scaled to have magnitude around one (e.g., converting precipitation from mm to cm). When there is a convergence failure, one can try a different optimization method, more iterations, or different starting values – see ' optimArgs ' and ' initial '. In particular, the Nelder-Mead method is used; users may want to try the BFGS method by setting ' optimArgs '= list(method = 'BFGS') (or ' optimArgs '= {'method': 'BFGS'} when calling from Python).

When using the bootstrap, users should check that the number of convergence failures when fitting to the boostrapped datasets is small, as it is not clear how to interpret the bootstrap results when there are convergence failures for some bootstrapped datasets.

**value**

The primary outputs of this function are as follows, depending on what is requested via 'returnPeriod', 'returnValue', 'getParams' and 'xContrast':

when 'returnPeriod' is given: for the period given in 'returnPeriod' the return value(s) ('returnValue') and its corresponding asymptotic standard error ('se_returnValue') and, when 'bootSE=True', also the

bootstrapped standard error ('se_returnValue_boot'). For nonstationary models, these correspond to the covariate values given in 'x'.

when 'returnValue' is given: for the value given in 'returnValue', the log exceedance probability ('logReturnProb') and the corresponding asymptotic standard error ('se_logReturnProb') and, when 'bootSE=True', also the bootstrapped standard error ('se_logReturnProb_boot'). This exceedance probability is the probability of exceedance for a single block. Also returned are the log return period ('logReturnPeriod') and its corresponding asymptotic standard error ('se_logReturnPeriod') and, when 'bootSE=True', also the bootstrapped standard error ('se_logReturnPeriod_boot'). For nonstationary models, these correspond to the covariate values given in 'x'. Note that results are on the log scale as probabilities and return times are likely to be closer to normally distributed on the log scale and therefore standard errors are more naturally given on this scale. Confidence intervals for return probabilities/periods can be obtained by exponentiating the interval obtained from plus/minus twice the standard error of the log probabilities/periods.

when 'getParams=True': the MLE for the model parameters ('mle') and corresponding asymptotic standard error ('se_mle') and, when 'bootSE=True', also the bootstrapped standard error ('se_mle_boot').

when 'xContrast' is specified for nonstationary models: the difference in return values ('returnValueDiff') and its corresponding asymptotic standard error ('se_returnValueDiff') and, when 'bootSE=True', bootstrapped standard error ('se_returnValueDiff_boot'). These differences correspond to the differences when contrasting each row in 'x' with the corresponding row in 'xContrast'. Also returned are the difference in log return probabilities (i.e., the log risk ratio) ('logReturnProbDiff') and its corresponding asymptotic standard error ('se_logReturnProbDiff') and, when 'bootSE=True', bootstrapped standard error ('se_logReturnProbDiff_boot').

**author**

Christopher J. Paciorek

**references**

Coles, S. 2001. An Introduction to Statistical Modeling of Extreme Values. Springer.

Paciorek, C.J., D.A. Stone, and M.F. Wehner. 2018. Quantifying uncertainty in the attribution of human influence on severe weather. Weather and Climate Extremes 20:69-80. arXiv preprint <https://arxiv.org/abs/1706.03388>.

**examples**

```
>>> Fort = climextremes.Fort
...
... FortMax = Fort.groupby('year').max()[['Prec']]
... FortMax.reset_index(inplace=True)
...
... # stationary fit
... result = climextremes.fit_gev(numpy.array(FortMax.Prec), returnPeriod = 20,␣
↪returnValue = 3.5, getParams = True, bootSE = True)
... result['returnValue']
... result['se_returnValue']       # return value standard error (asymptotic)
... result['se_returnValue_boot']  # return value standard error (bootstrapping)
... result['logReturnProb']        # log of probability of exceeding 'returnValue'
... result['mle']                  # MLE array
... result['mle_names']            # names for MLE array
... result['mle'][2]               # MLE for shape parameter
...
... result['numBootFailures']      # number of bootstrap datasets for which the␣
```

(continues on next page)

(continued from previous page)

```
↪model could not be fit; if this is non-negligible relative to the number of␣
↪bootstrap samples (default of 250), interpret the bootstrap results with caution
...
... # modifying the bootstrapping specifications
... result = climextremes.fit_gev(numpy.array(FortMax.Prec), returnPeriod = 20,␣
↪returnValue = 3.5, getParams = True, bootSE = True, bootControl = {'n': 100, 'seed
↪': 3})
... result['se_returnValue_boot']  # return value standard error (bootstrapping)
...
... yrsToPred = numpy.array([min(Fort.year), max(Fort.year)])
...
... # nonstationary fit with location linear in year and two return values requested
... result_ns = climextremes.fit_gev(numpy.array(FortMax.Prec), numpy.array(FortMax.
↪year), locationFun = 1, returnPeriod = numpy.array([20, 30]), returnValue = 3.5,␣
↪xNew = yrsToPred, getParams = True, bootSE = False)
... result_ns['returnValue']
... result_ns['se_returnValue']
...
...
...
```

climextremes.**fit_pot**(*y=None*, *x=None*, *threshold=None*, *locationFun=None*, *scaleFun=None*, *shapeFun=None*, *nBlocks=None*, *blockIndex=None*, *firstBlock=1.0*, *index=None*, *nReplicates=1.0*, *replicateIndex=None*, *weights=None*, *proportionMissing=None*, *returnPeriod=None*, *returnValue=None*, *getParams=False*, *getFit=False*, *xNew=None*, *xContrast=None*, *declustering=None*, *upperTail=True*, *scaling=1.0*, *bootSE=False*, *bootControl=None*, *optimArgs=None*, *optimControl=None*, *initial=None*, *logScale=None*, *_normalizeX=True*, *_getInputs=False*, *_allowNoInt=True*)

Fit a peaks-over-threshold model to exceedances over a threshold

**description**

Fit a peaks-over-threshold model, designed specifically for climate data, to exceedance-only data, using the point process approach. Any covariates/predictors/features assumed to vary only between and not within blocks of observations. It includes options for variable weights (useful for local likelihood) and variable proportions of missing data, as well as for bootstrapping to estimate uncertainties. Results can be returned in terms of parameter values, return values, return periods, return probabilities, and differences in either return values or log return probabilities (i.e., log risk ratios).

**arguments**

y: a numpy array of exceedance values (values of the outcome variable above the threshold). For better optimization performance, it is recommended that the 'y' have magnitude around one (see 'Details'), for which one can use 'scaling'.

x: a numpy array or pandas data frame with columns corresponding to covariate/predictor/feature variables and each row containing the values of the variable for a block (e.g., often a year with climate data). The number of rows must equal the number of blocks.

threshold: a single numeric value for constant threshold or a numpy array with length equal to the number of blocks, indicating the threshold for each block.

locationFun: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the location parameter using columns from 'x'. 'x' must be supplied if this is anything other than 'None'.

scaleFun: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the (potentially transformed) scale parameter using columns from 'x'. 'x' must be supplied if this is anything other than 'None'. 'logscale' controls whether this determines the log of the scale or the scale directly.

shapeFun: numpy array of either character strings or indices describing a linear model (i.e., regression function) for the shape parameter using columns from 'x'. 'x' must be supplied if this is anything other than 'None'.

nBlocks: number of blocks (e.g., a block will often be a year with climate data); note this value determines the interpretation of return values/periods/probabilities; see 'returnPeriod' and 'returnValue'.

blockIndex: numpy array providing the index of the block corresponding to each element of 'y'. Used only when 'x' is provided to match exceedances to the covariate/predictor/feature value for the exceedance or when using bootstrapping with the resampling based on blocks based on the 'by' element of 'bootControl'. If 'firstBlock' is not equal to one, then 'blockIndex' need not have one as its smallest possible value.

firstBlock: single numeric value indicating the numeric value of the first possible block of 'blockIndex'. For example the values in 'blockIndex' might indicate the year of each exceedance with the first year of data being 1969, in which case 'firstBlock' would be 1969. Note that the first block may not have any exceedances so it may not be represented in 'blockIndex'. Used only to adjust 'blockIndex' so that the block indices start at one and therefore correspond to the rows of 'x'.

index: (optional) numpy array providing the integer-valued index (e.g., julian day for daily climate data) corresponding to each element of 'y'. For example if there are 10 original observations and the third, fourth, and seventh values are exceedances, then 'index' would be the vector 3,4,7. Used only when 'declustering' is provided to determine which exceedances occur sequentially or within a contiguous set of values of a given length. The actual values are arbitrary; only the lags between the values are used.

nReplicates: numeric value indicating the number of replicates.

replicateIndex: numpy array providing the index of the replicate corresponding to each element of 'y'. Used for three purposes: (1) when using bootstrapping with the resampling based on replicates based on the 'by' element of 'bootControl', (2) to avoid treating values in different replicates as potentially being sequential or within a short interval when removing values based on 'declustering', and (3) to match outcomes to 'weights' or 'proportionMissing' when either vary by replicate.

weights: a numpy array providing the weights by block. When there is only one replicate or the weights do not vary by replicate, a one-dimensional array of length equal to the number of blocks. When weights vary by replicate, a two-dimensional array with rows corresponding to blocks and columns to replicates. Likelihood contribution of each block is multiplied by the corresponding weight.

proportionMissing: a numeric value or numpy array indicating the proportion of missing values in the original dataset before exceedances were selected. When the proportion missing is the same for all blocks and replicates, a single value. When there is only one replicate or the weights do not vary by replicate, a one-dimensional array of length equal to the number of blocks. When weights vary by replicate, a two-dimensional array with rows corresponding to blocks and columns to replicates.

returnPeriod: numeric value giving the number of blocks for which return values should be calculated. For example a 'returnPeriod' equal to 20 will result in calculation of the value of an event that occurs with probability 1/20 in any block and therefore occurs on average every 20 blocks. Often blocks will correspond to years.

returnValue: numeric value giving the value for which return probabilities/periods should be calculated, where the resulting period will be the average number of blocks until the value is exceeded and the probability the probability of exceeding the value in any single block.

getParams: boolean indicating whether to return the fitted parameter values and their standard errors; WARNING: parameter values for models with covariates for the scale parameter must interpreted based on the value of 'logScale'.

getFit: boolean indicating whether to return the full fitted model (potentially useful for model evaluation and for understanding optimization problems); note that estimated parameters in the fit object for nonstationary models

will not generally match the MLE provided when 'getParams' is 'True' because covariates are normalized before fitting and the fit object is based on the normalized covariates. Similarly, parameters will not match if 'scaling' is not 1.

xNew: object of the same form as 'x', providing covariate/predictor/feature values for which return values/periods/probabilities are desired.

xContrast: object of the same form and dimensions as 'xNew', providing covariate/predictor/feature values for which to calculate the differences of the return values and/or log return probabilities relative to the values in 'xNew'. This provides a way to estimate differences in return value or log return probabilities (i.e., log risk ratios).

declustering: one of 'None', ''noruns", or a number. If ''noruns" is specified, only the maximum (or minimum if 'upperTail = False') value within a set of exceedances corresponding to successive indices is included. If a number, this should indicate the size of the interval (which will be used with the 'index' argument) within which to allow only the largest (or smallest if 'upperTail = False') value.

upperTail: boolean indicating whether one is working with exceedances over a high threshold ('True') or exceedances under a low threshold ('False'); in the latter case, the function works with the negative of the values and the threshold, changing the sign of the resulting location parameters.

scaling: positive-valued scalar used to scale the data values for more robust optimization performance. When multiplied by the values, it should produce values with magnitude around 1.

bootSE: boolean indicating whether to use the bootstrap to estimate standard errors.

bootControl: a dictionary of control parameters for the bootstrapping. See 'Details'.

optimArgs: a dictionary with named components matching exactly any arguments that the user wishes to pass to R's 'optim' function. See 'help(optim)' in R for details. Of particular note, ''method" can be used to choose the optimization method used for maximizing the log-likelihood to fit the model (e.g., ''method" could be 'BFGS' instead of the default 'Nelder-Mead'). To specify the 'control' argument, use 'optimControl' rather than including 'control' here.

optimControl: a dictionary with named components matching exactly any elements that the user wishes to pass as the 'control' argument to R's 'optim' function. For example, 'control={'maxit': VALUE)' for a user-specified VALUE can be used to increase the number of iterations if the optimization is converging slowly.

initial: a dictionary with components named ''location", ''scale", and ''shape" providing initial parameter values, intended for use in speeding up or enabling optimization when the default initial values are resulting in failure of the optimization; note that use of 'scaling', 'logScale' and '.normalizeX = True' cause numerical changes in some of the parameters. For example with 'logScale = True', initial value(s) for ''scale" should be specified on the log scale.

logScale: boolean indicating whether optimization for the scale parameter should be done on the log scale. By default this is 'False' when the scale is not a function of covariates and 'True' when the scale is a function of covariates (to ensure the scale is positive regardless of the regression coefficients).

.normalizeX: boolean indicating whether to normalize 'x' values for better numerical performance; default is 'True'.

.getInputs: boolean indicating whether to return intermediate objects used in fitting. Defaults to 'False' and intended for internal use only

**details**

This function allows one to fit stationary or nonstationary peaks-over-threshold models using the point process approach. The function can return parameter estimates (which are asymptotically equivalent to GEV model parameters for block maxima data), return value/level for a given return period (number of blocks), and return probabilities/periods for a given return value/level. The function provides standard errors based on the usual MLE asymptotics, with delta-method-based standard errors for

functionals of the parameters, but also standard errors based on the nonparametric bootstrap, either resampling by block or by replicate or both.

Analyzing aggregated observations, such as yearly averages:

Aggregated average or summed data such as yearly or seasonal averages can be fit using this function. The best way to do this is to specify ‘ nBlocks ‘ to be the number of observations (i.e., the length of the observation period, not the number of exceedances). Then any return probabilities can be interpreted as the probabilities for a single block (e.g., a year). If instead ‘ nBlocks ‘ were one (i.e., a single block) then probabilities would be interpreted as the probability of occurrence in a multi-year block.

Blocks and replicates:

Note that blocks and replicates are related concepts. Blocks are the grouping of values such that return values and return periods are calculated based on the equivalent block maxima (or minima) generalized extreme value model. For example if a block is a year, the return period is the average number of years before the given value is seen, while the return value when ‘ returnPeriod ‘ is, say, 20, is the value exceeded on average once in 20 years. A given dataset will generally have multiple blocks. In some cases a block may contain only a single value, such as when analyzing yearly sums or averages.

Replicates are repeated datasets, each with the same structure, including the same number of blocks. The additional blocks in multiple replicates could simply be treated as additional blocks without replication, but when the predictor variables and’ weights ‘are the same across replicates, it is simpler to make use of ‘ nReplicates ‘ and ‘ replicateIndex ‘ (see next paragraph). A given replicate might only contain a single block, such as with an ensemble of short climate model runs that are run only for the length of a single block (e.g., a single year). In this case ‘ nBlocks ‘ should be set to one.

When using multiple replicates (e.g., multiple members of a climate model’ initial ‘condition ensemble), the standard input format is to append outcome values for additional replicates to the ‘ y ‘ argument and indicate the replicate ID for each exceedance in ‘ replicateIndex ‘. However, if one has different covariate values or thresholds for different replicates, then one needs to treat the additional replicates as providing additional blocks, with only a single replicate. The covariate values can then be included as additional rows in ‘ x ‘, with ‘ nBlocks ‘ reflecting the product of the number of blocks per replicate and the number of replicates and ‘ nReplicates ‘ set to 1. In this situation, if ‘ declustering ‘ is specified, make sure to set ‘ index ‘ such that the values for separate replicates do not overlap with each other, to avoid treating exceedances from different replicates as being sequential or from a contiguous set of values. Similarly, if there is a varying number of replicates by block, then all block-replicate pairs should be treated as individual blocks with a corresponding row in ‘ x ‘.

‘ bootControl ‘ arguments:

The ‘ bootControl ‘ argument is a list (or dictionary when calling from Python) that can supply any of the following components:

seed. Value of the random number seed as a single value, or in the form of .Random.seed , to set before doing resampling. Defaults to 1 . n. Number of bootstrap samples. Defaults to 250 . by. Character string, one of ‘block’ , ‘replicate’ , or ‘joint’ , indicating the basis for the resampling. If ‘block’ , resampled datasets will consist of blocks drawn at random from the original set of blocks; if there are replicates, each replicate will occur once for every resampled block. If ‘replicate’ , resampled datasets will consist of replicates drawn at random from the original set of replicates; all blocks from a replicate will occur in each resampled replicate. Note that this preserves any dependence across blocks rather than assuming independence between blocks. If ‘joint’ resampled datasets will consist of block-replicate pairs drawn at random from the original set of block-replicate pairs. Defaults to ‘block’ . getSample. Logical/boolean indicating whether the user wants the full bootstrap sample of parameter estimates and/or return value/period/probability information provided for use in subsequent calculations; if False (the default), only the bootstrap-based estimated standard

errors are returned.

Optimization failures:

It is not uncommon for maximization of the log-likelihood to fail for extreme value models. Users should carefully check the info element of the return object to ensure that the optimization converged. For better optimization performance, it is recommended that the observations be scaled to have magnitude around one (e.g., converting precipitation from mm to cm). When there is a convergence failure, one can try a different optimization method, more iterations, or different starting values – see ' optimArgs ' and ' initial '. In particular, the Nelder-Mead method is used; users may want to try the BFGS method by setting ' optimArgs '= list(method = 'BFGS') (or ' optimArgs '= {'method': 'BFGS'} if calling from Python).

When using the bootstrap, users should check that the number of convergence failures when fitting to the boostrapped datasets is small, as it is not clear how to interpret the bootstrap results when there are convergence failures for some bootstrapped datasets.

**value**

The primary outputs of this function are as follows, depending on what is requested via 'returnPeriod', 'returnValue', 'getParams' and 'xContrast':

when 'returnPeriod' is given: for the period given in 'returnPeriod' the return value(s) ('returnValue') and its corresponding asymptotic standard error ('se_returnValue') and, when 'bootSE=True', also the bootstrapped standard error ('se_returnValue_boot'). For nonstationary models, these correspond to the covariate values given in 'x'.

when 'returnValue' is given: for the value given in 'returnValue', the log exceedance probability ('logReturnProb') and the corresponding asymptotic standard error ('se_logReturnProb') and, when 'bootSE=True', also the bootstrapped standard error ('se_logReturnProb_boot'). This exceedance probability is the probability of exceedance for a single block. Also returned are the log return period ('logReturnPeriod') and its corresponding asymptotic standard error ('se_logReturnPeriod') and, when 'bootSE=True', also the bootstrapped standard error ('se_logReturnPeriod_boot'). For nonstationary models, these correspond to the covariate values given in 'x'. Note that results are on the log scale as probabilities and return times are likely to be closer to normally distributed on the log scale and therefore standard errors are more naturally given on this scale. Confidence intervals for return probabilities/periods can be obtained by exponentiating the interval obtained from plus/minus twice the standard error of the log probabilities/periods.

when 'getParams=True': the MLE for the model parameters ('mle') and corresponding asymptotic standard error ('se_mle') and, when 'bootSE=True', also the bootstrapped standard error ('se_mle_boot').

when 'xContrast' is specified for nonstationary models: the difference in return values ('returnValueDiff') and its corresponding asymptotic standard error ('se_returnValueDiff') and, when 'bootSE=True', bootstrapped standard error ('se_returnValueDiff_boot'). These differences correspond to the differences when contrasting each row in 'x' with the corresponding row in 'xContrast'. Also returned are the difference in log return probabilities (i.e., the log risk ratio) ('logReturnProbDiff') and its corresponding asymptotic standard error ('se_logReturnProbDiff') and, when 'bootSE=True', bootstrapped standard error ('se_logReturnProbDiff_boot').

**author**

Christopher J. Paciorek

**references**

Coles, S. 2001. An Introduction to Statistical Modeling of Extreme Values. Springer.

Paciorek, C.J., D.A. Stone, and M.F. Wehner. 2018. Quantifying uncertainty in the attribution of human influence on severe weather. Weather and Climate Extremes 20:69-80. arXiv preprint <https:

//arxiv.org/abs/1706.03388>.

**examples**

```
>>> Fort = climextremes.Fort
...
... firstYr = min(Fort.year)
... yrs = numpy.array(range(int(firstYr), int(max(Fort.year)+1)))
... nYrs = len(yrs)
... yrsToPred = numpy.array([min(Fort.year), max(Fort.year)])
...
... threshold = 0.395
...
... FortExc = Fort[Fort.Prec > threshold]
...
... # stationary fit
... result = climextremes.fit_pot(numpy.array(FortExc.Prec), nBlocks = nYrs,␣
↪threshold = threshold, firstBlock = firstYr, blockIndex = numpy.array(FortExc.
↪year), getParams = True, returnPeriod = 20, returnValue = 3.5, bootSE = True)
... result['returnValue']
... result['se_returnValue']       # return value standard error (asymptotic)
... result['se_returnValue_boot']  # return value standard error (bootstrapping)
... result['logReturnProb']        # log of probability of exceeding 'returnValue'
... result['mle']                  # MLE array
... result['mle_names']            # names for MLE array
... result['mle'][2]               # MLE for shape parameter
...
... result['numBootFailures']      # number of bootstrap datasets for which the␣
↪model could not be fit; if this is non-negligible relative to the number of␣
↪bootstrap samples (default of 250), interpret the bootstrap results with caution
...
... # nonstationary fit with location linear in year and two return values requested
... result = climextremes.fit_pot(numpy.array(FortExc.Prec), x = yrs, firstBlock =␣
↪firstYr, nBlocks = nYrs, threshold = threshold, blockIndex = numpy.array(FortExc.
↪year), locationFun = 1, getParams = True, returnPeriod = numpy.array([20, 30]),␣
↪returnValue = 3.5, xNew = yrsToPred, bootSE = False)
... result['returnValue']
... result['se_returnValue']
...
... # fit with location a function of two covariates
... # here I'll use year and a random vector just to illustrate syntax
... # make 'x' be a 2-column numpy array, each column a covariate
... # 'xNew' also needs to have 2 columns, each row is a different set of covariate␣
↪values
... tmp = numpy.random.rand(nYrs)
... covByBlock = numpy.c_[yrs, numpy.random.rand(nYrs)]
... result = climextremes.fit_pot(numpy.array(FortExc.Prec), x = covByBlock,␣
↪firstBlock = firstYr, nBlocks = nYrs, threshold = threshold, blockIndex = numpy.
↪array(FortExc.year), locationFun = numpy.array([1,2]), getParams = True,␣
↪returnPeriod = 20, returnValue = 3.5, xNew = numpy.array([[min(Fort.year), 0],␣
↪[max(Fort.year), 0]]), bootSE = False)
...
... # with declustering (using max of exceedances on contiguous days)
... result = climextremes.fit_pot(numpy.array(FortExc.Prec), x = yrs, firstBlock =␣
```

(continues on next page)

(continued from previous page)

```
↪firstYr, nBlocks = nYrs, threshold = threshold, blockIndex = numpy.array(FortExc.
↪year), index = numpy.array(FortExc.obs), locationFun = 1, declustering = "noruns",
↪ getParams = True, returnPeriod = 20, returnValue = 3.5, xNew = yrsToPred, bootSE␣
↪= False)
... result['returnValue']
... result['se_returnValue']
...
... # with declustering (consider sequential blocks of 5 days and only use the max␣
↪of any exceedances within a block)
... result = climextremes.fit_pot(numpy.array(FortExc.Prec), x = yrs, firstBlock =␣
↪firstYr, nBlocks = nYrs, threshold = threshold, blockIndex = numpy.array(FortExc.
↪year), index = numpy.array(FortExc.obs), locationFun = 1, declustering = 5,␣
↪getParams = True, returnPeriod = 20, returnValue = 3.5, xNew = yrsToPred, bootSE␣
↪= False)
... result['returnValue']
... result['se_returnValue']
...
... # with replicates; for illustration here, I'll just duplicate the Fort data
... result = climextremes.fit_pot(numpy.append(numpy.array(FortExc.Prec), numpy.
↪array(FortExc.Prec)), x = yrs, firstBlock = firstYr, nBlocks = nYrs, nReplicates␣
↪= 2, threshold = threshold, blockIndex = numpy.append(FortExc.year, FortExc.year),
↪ locationFun = 1, getParams = True, returnPeriod = 20, returnValue = 3.5, xNew =␣
↪yrsToPred, bootSE = False)
... result['returnValue']
... result['se_returnValue']
...
... # analysis of seasonal total precipitation
... tmp = Fort[numpy.logical_and(Fort['month'] < 9, Fort['month'] > 5)]
... FortSummerTotal = tmp.groupby('year').sum()[['Prec']]
... FortSummerTotal.reset_index(inplace=True)
... threshold = numpy.percentile(FortSummerTotal.Prec, 80)
... FortSummerTotalExc = FortSummerTotal[FortSummerTotal.Prec > threshold]
...
... result = climextremes.fit_pot(numpy.array(FortSummerTotalExc.Prec), x = yrs,␣
↪firstBlock = firstYr, nBlocks = nYrs, blockIndex = numpy.array(FortSummerTotalExc.
↪year), locationFun = 1, threshold = threshold, getParams = True, returnPeriod =␣
↪20, returnValue = 10, xNew = yrsToPred, bootSE = False)
... result['returnValue']
... result['se_returnValue']
...
... # modifying control arguments and seeing more information on the optimization
... result = climextremes.fit_pot(numpy.array(FortSummerTotalExc.Prec), x = yrs,␣
↪firstBlock = firstYr, nBlocks = nYrs, blockIndex = numpy.array(FortSummerTotalExc.
↪year), locationFun = 1, threshold = threshold, getParams = True, returnPeriod =␣
↪20, returnValue = 10, xNew = yrsToPred, bootSE = True, bootControl = {'n':150,
↪'seed':3}, getFit = True)
... result['info']   # information on the optimization
... result['info']['counts']  # number of evaluations in the optimization
... result['info']['counts_names'] # names to interpret 'counts'
... result['numBootFailures']      # number of bootstrap datasets for which the␣
↪model could not be fit; if this is non-negligible relative to the number of␣
↪bootstrap samples (default of 250), interpret the bootstrap results with caution
```

(continues on next page)

(continued from previous page)

```
...
... # result['fit']  # voluminous output from the R function that does the fitting
...
```

climextremes.**normalize**(*vec=None*, *shift=None*, *lower=None*, *upper=None*)

Normalize a vector

**description**

Normalize a vector by subtracting off central point and dividing by range

**arguments**

vec: numpy array of values

shift: optional central point (if not provided, uses the mean of 'vec')

lower: optional lower end point of range (if not provided uses min of 'vec')

upper: optional upper end point of range (if not provided uses max of 'vec')

climextremes.**reinstall_climextremes**(*force*, *version*)

climextremes.**remove_runs**(*y=None*, *index=None*, *upperTail=True*)

Remove consecutive exceedances from a vector

**description**

Remove runs, i.e., consecutive exceedances, from a vector of values and associated indices (days); for use in declustering

**arguments**

y: numpy array of values

index: numpy array of indices, one per value, that indicate which elements of 'y' are consecutive

upperTail: boolean indicating whether values of 'y' are upper (right) tail values (True) or lower (left) tail values (False)

climextremes.**screen_within_block**(*y=None*, *index=None*, *blockLength=10.0*)

Remove multiple exceedances within non-overlapping blocks of fixed length

**description**

Remove multiple exceedances within non-overlapping blocks of fixed lengths, for use in declustering

**arguments**

y: numpy array of values #' index: numpy array of indices, one per value, that indicate which elements of 'y' are consecutive #' blockLength: length of block within which to remove all but the most extreme value

CHAPTER

ONE

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX