xtremes Documentation

Version 0.3.0

Author: Erik Haufs

Fig. 1: Dall-E's interpretation of the Xtremes package

January 8, 2025

CHAPTERS

xtremes is a Python library for various utilities useful in Extreme Value Statistics. So far, you will find tools to fit more than one order statistics for parameter estimation as well as a Bootstrap device.

It is created within the ClimXtreme project and will provide supplementary code and simulations for the papers yet to come.

Check out the *Usage* section for further information, including how to *Installation* the project.

Note: This project is under active and heavy development.

CHAPTER

ONE

USAGE

1.1 Installation

To use xtremes, it is possible to install it via pip

```
(.venv) $ pip install xtremes
```

1.2 The modules

So far, three modules are implemented. The module `xtremes.miscellaneous` contains basic functionalities, whereas `xtremes.topt` is specialized on the influence of higher order statistics for Maximum Likelihood estimations. `xtremes.Bootstrap` provides a suitable Bootstrap device for block maxima.

For each module, there will be a tutorial and a subsection in the API reference.

1.2.1 Using the TimeSeries Class

The TimeSeries Class is used to generate realizations of specific time series and extract their blockmaxima

[1]:
```
import xtremes as xx
import xtremes.topt as hos
```

In this section, we create a `TimeSeries` object `TS` using the `xtremes.HigherOrderStatistics` module. The time series is generated with the following parameters:

- `n=1000`: The length of the time series.
- `distr='GEV'`: The distribution type, which is the Generalized Extreme Value distribution.
- `modelparams=(0.5, 0.1, 0.1)`: The parameters for the GEV distribution.

We then simulate 200 realizations of this time series using the `simulate` method.

Finally, we extract the block maxima from the simulated time series with a block size of 10 using the `get_blockmaxima` method:

[2]:
```
TS = hos.TimeSeries(n=1000, distr='GEV', modelparams=(0.5,0.1,0.1))
TS.simulate(rep=200)
```

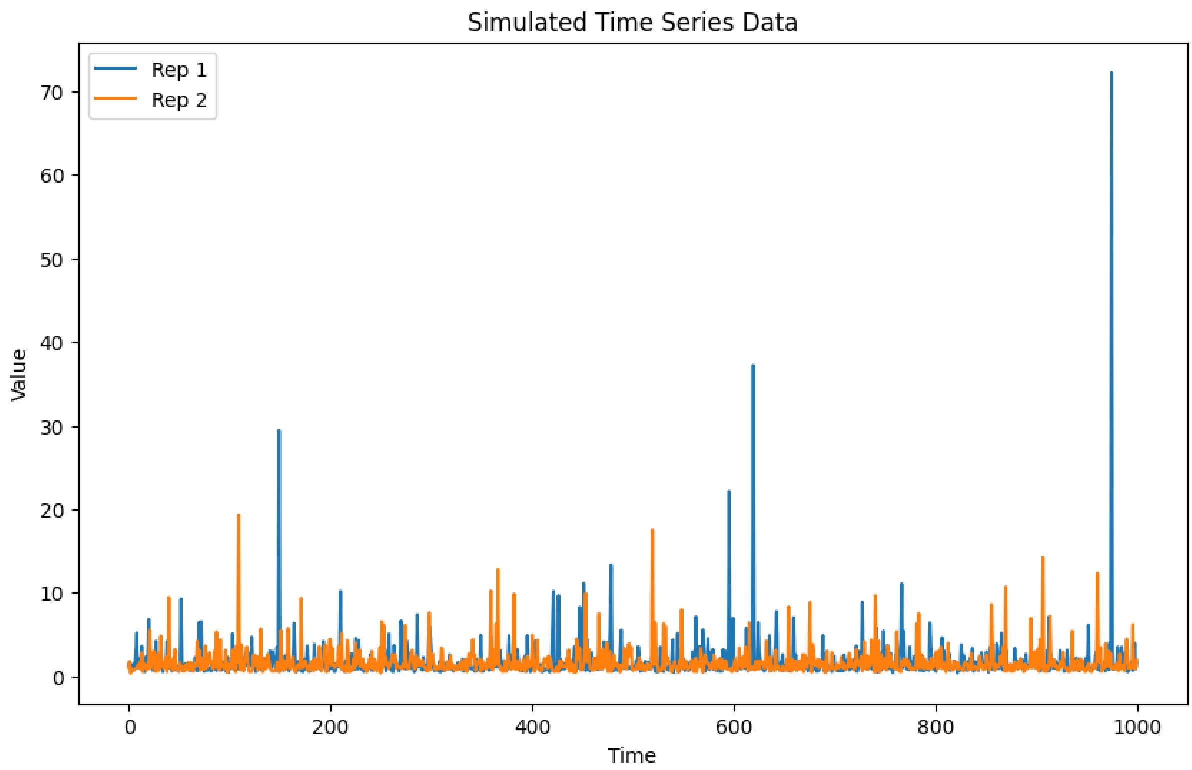
(continues on next page)

(continued from previous page)

```
TS.get_blockmaxima(block_size=10)
```

The `TS.plot(rep=[1,2])` method is used to plot specific realizations of the time series. In this case, the method will plot the first and second realizations of the time series `TS`. This is useful for visualizing the behavior of the time series over different realizations and understanding the variability in the data.

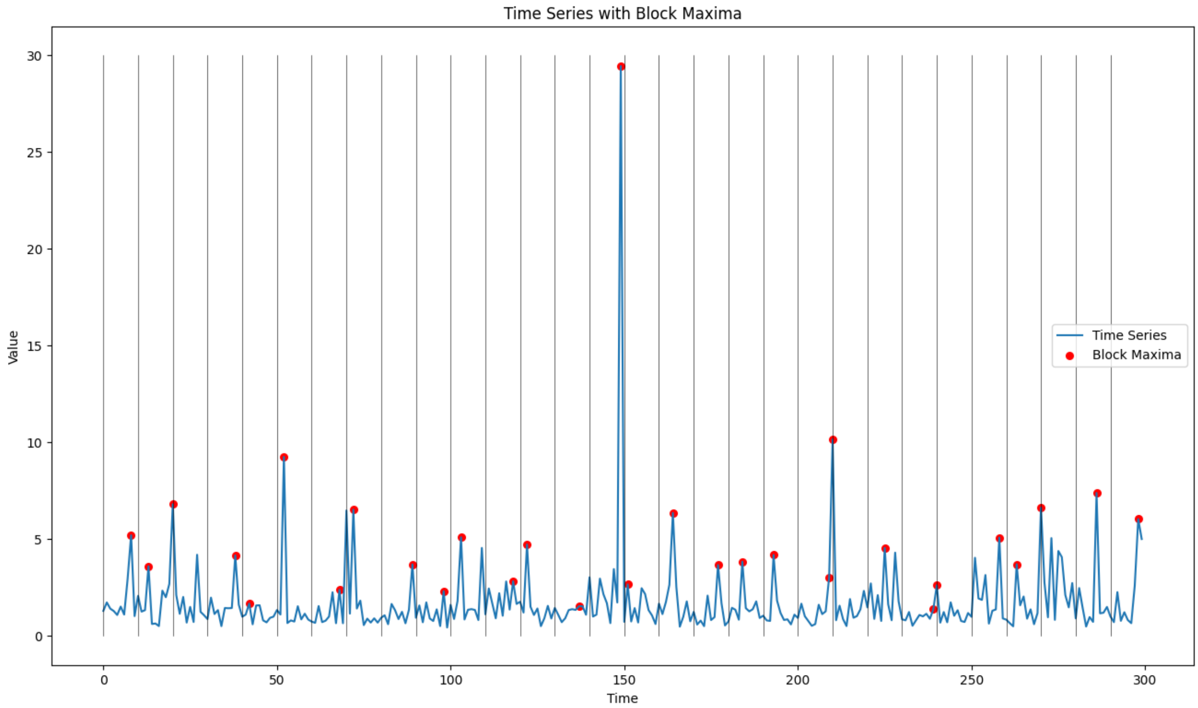
```
[3]: TS.plot(rep=[1,2])
```



The `TS.plot_with_blockmaxima()` method is used to visualize the time series along with its block maxima. This method plots the original time series data and overlays the block maxima on the same plot. This is useful for identifying extreme values within the time series and understanding how these extremes are distributed over time.

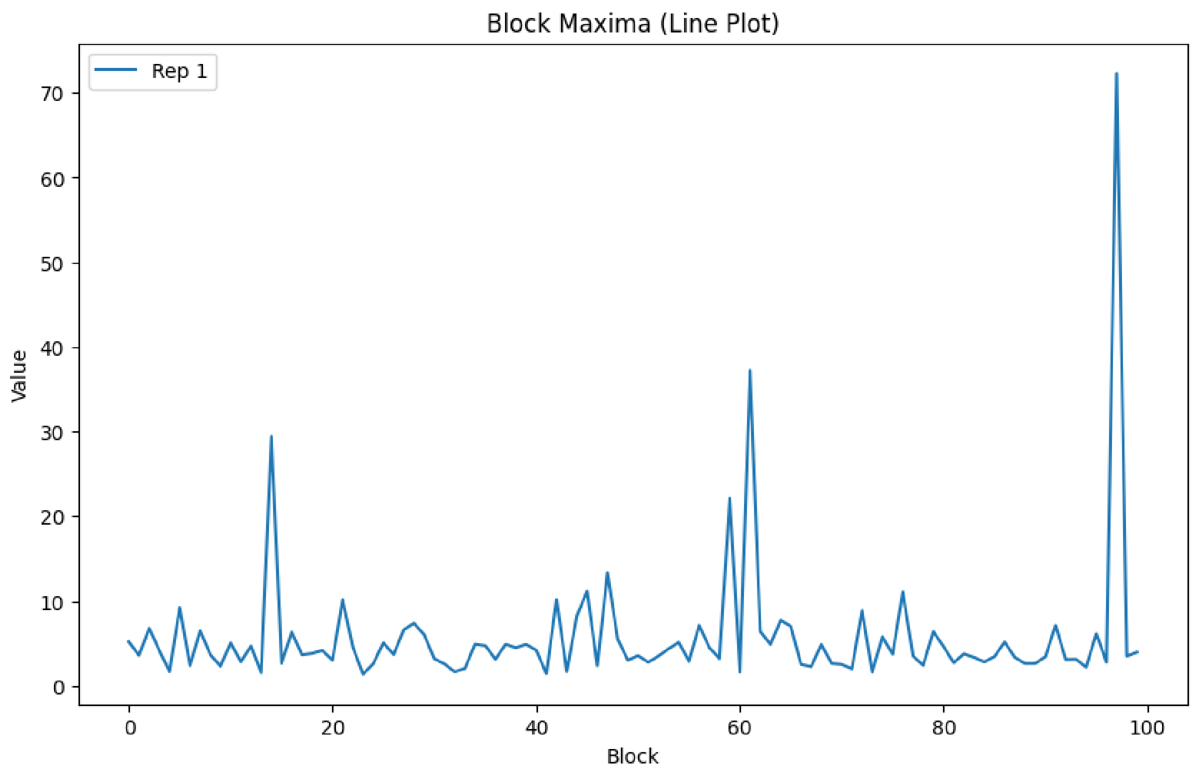
By visualizing both the time series and its block maxima, one can gain insights into the behavior of extreme events and their frequency.

```
[4]: TS.plot_with_blockmaxima()
```



The `TS.plot_blockmaxima(rep=1)` method is used to plot the block maxima of a specific realization of the time series. In this case, the method will plot the block maxima for the first realization (`rep=1`).

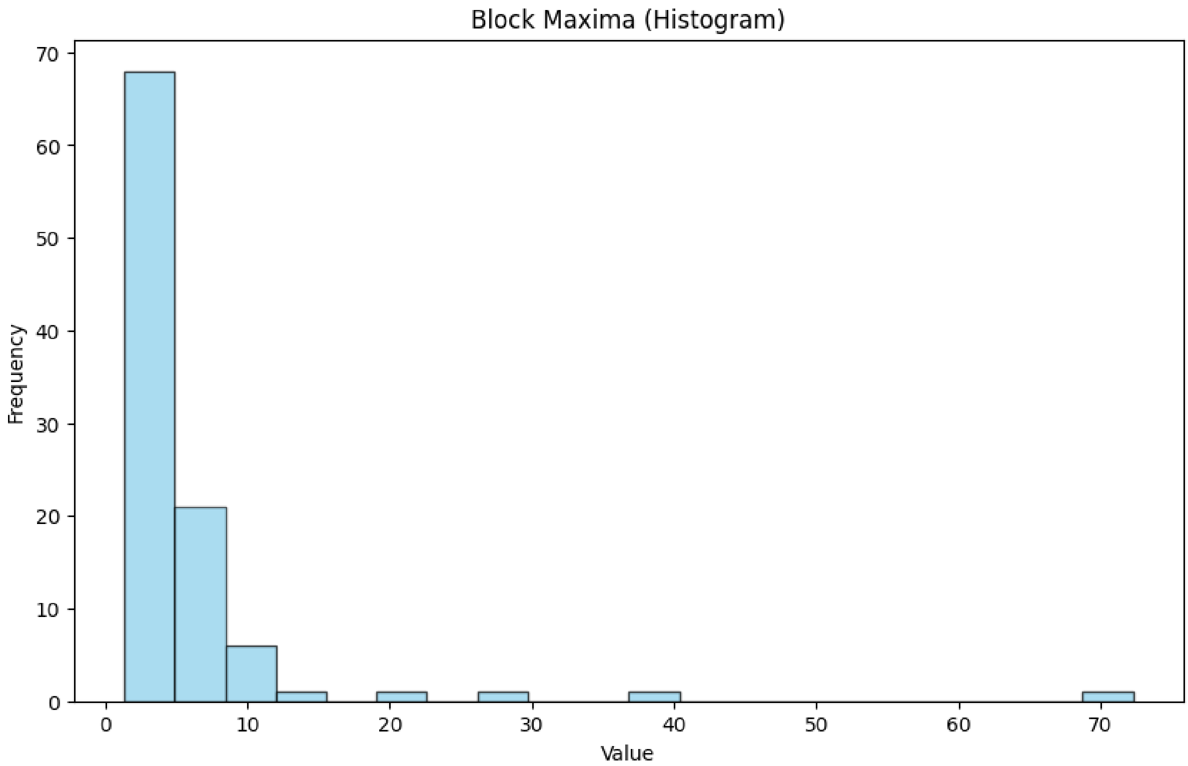
[5]:
```
TS.plot_blockmaxima(rep=1)
```



The `TS.plot_blockmaxima(rep=1, plot_type='hist')` method is used to plot the block maxima of a specific realization of the time series as a histogram. In this case, the method will plot the block maxima for the first realization (`rep=1`).

By setting the `plot_type` parameter to `'hist'`, the block maxima are visualized in the form of a histogram, which helps in understanding the distribution of the block maxima values. This is useful for analyzing the frequency and range of extreme values within the time series.

[6]:
```
TS.plot_blockmaxima(rep=1, plot_type='hist')
```



1.2.2 Maximum Likelihood Estimation on High Order Statistics

In this notebook, we perform an analysis of time series data using the `xtremes` library. We utilize various estimators and statistical methods provided by the library to analyze and visualize the data.

The following variables are used in this notebook:

- **PWM**: A `PWM_estimators` object used for Probability Weighted Moments estimation.
- **MLE**: A `ML_estimators` object used for Maximum Likelihood estimation.

[1]:
```
import xtremes as xx
import xtremes.topt as hos
```

[2]:
```
TS = hos.TimeSeries(n=1000, modelparams=(0.5,0.1,0.1))
TS.simulate(rep=200)
TS.get_blockmaxima(block_size=10)
```

Probability Weighted Moments (PWM) Estimation

In this section, we perform Probability Weighted Moments (PWM) estimation on the time series data. The `PWM_estimators` object is used for this purpose. Below are the steps involved:

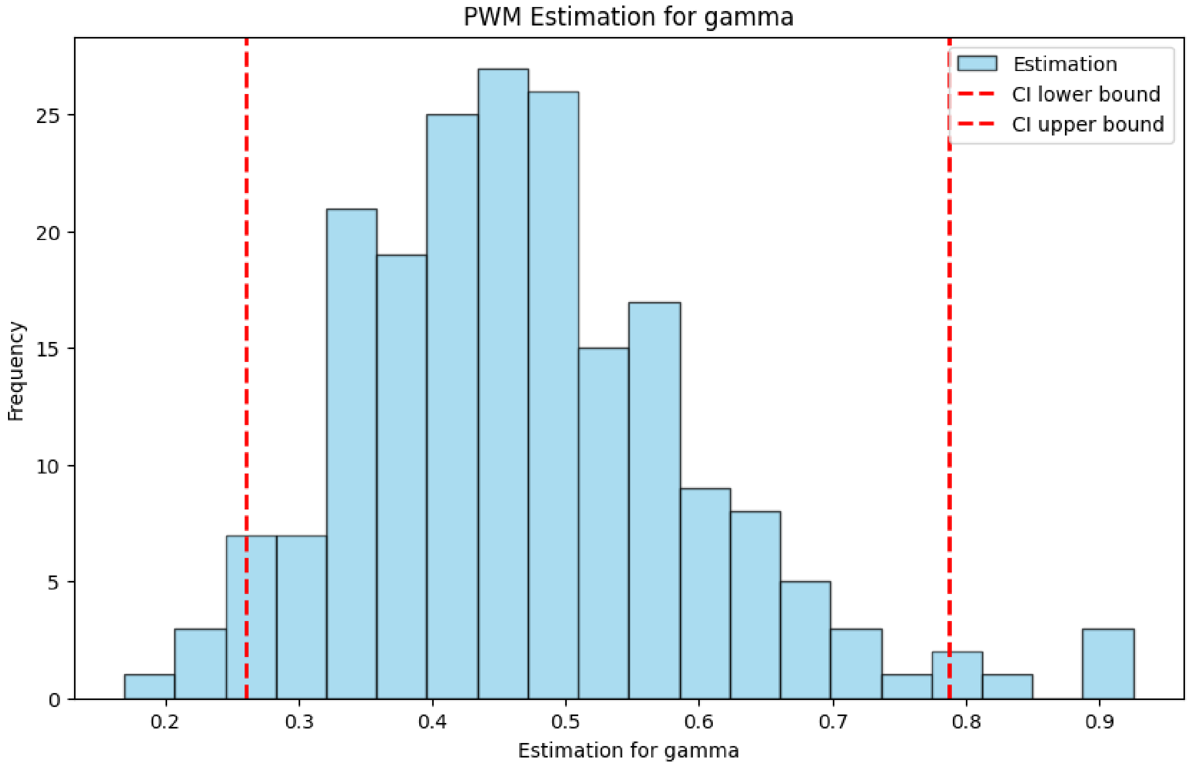
1. **Initialization**: `python PWM = hos.PWM_estimators(TS)` We initialize the `PWM_estimators` object with the time series data `TS`.
2. **PWM Estimation**: `python PWM.get_PWM_estimation()` This method computes the PWM estimates for the given time series data.
3. **Statistics Calculation**: `python PWM.get_statistics(gamma_true=0)` This method calculates the statistics based on the PWM estimates. Here, `gamma_true` is the true value of the shape parameter used for comparison.
4. **Confidence Intervals**: `python PWM.get_CIs()` This method computes the confidence intervals for the PWM estimates.
5. **View Statistics**: `python PWM.statistics` This attribute holds the computed statistics, which can be accessed for further analysis or visualization.

```
[3]: PWM = hos.PWM_estimators(TS)
     PWM.get_PWM_estimation()
     PWM.get_statistics(gamma_true=0.5)
     PWM.get_CIs()
     PWM.statistics
```

```
[3]: {'gamma_mean': np.float64(0.4708851765999638),
      'gamma_variance': np.float64(0.016709443122544274),
      'gamma_bias': np.float64(0.029187884552301O4),
      'gamma_mse': np.float64(0.017561375727182728),
      'mu_mean': np.float64(3.1883509998758033),
      'mu_variance': np.float64(0.041621701106587236),
      'sigma_mean': np.float64(1.5985219865770657),
      'sigma_variance': np.float64(0.052464586606184645),
      'gamma_CI': array([0.26048308, 0.7876679 ]),
      'mu_CI': array([2.85180707, 3.67041571]),
      'sigma_CI': array([1.24809831, 2.0891756 ])}
```

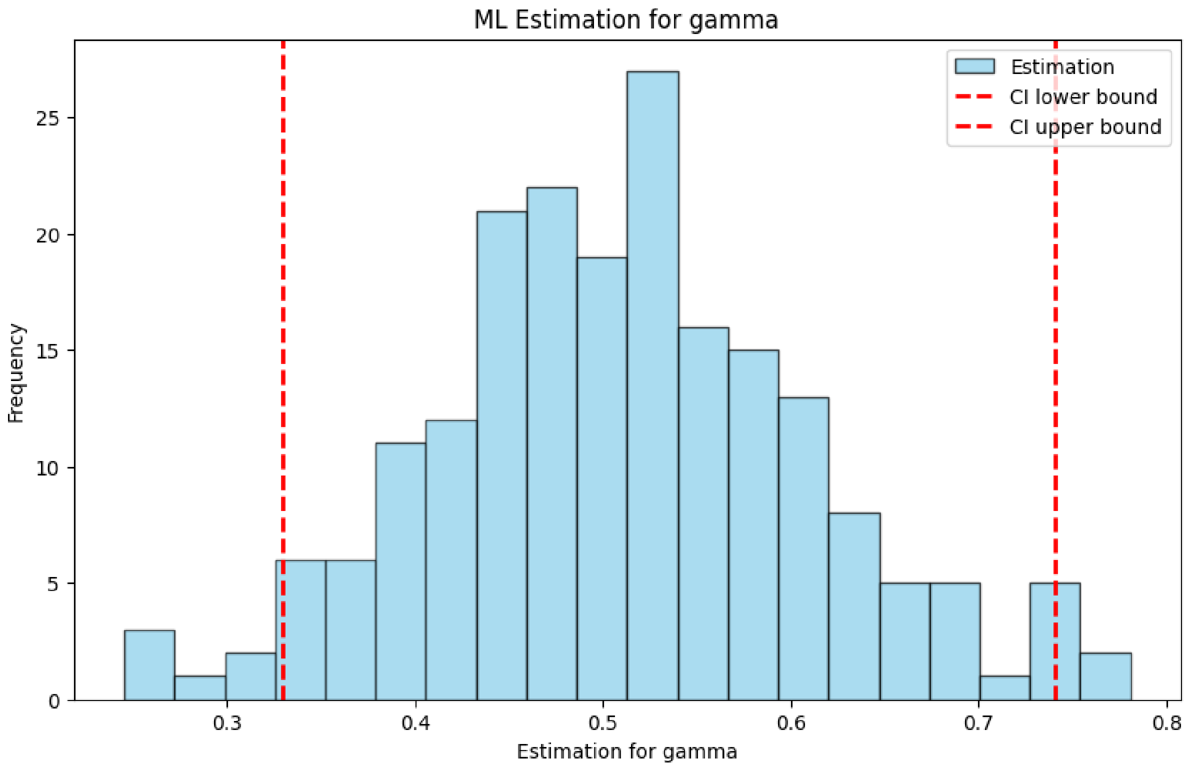
Plotting the distribution of the optained estimators together with a symmetrical CI is as easy as:

```
[4]: PWM.plot()
```



The same works analogously for MLE. We use the PWM estimators as an initial guess for otimization. Instead of symmetrical CIs (default) we can also plot CIs with minimal width

[5]:
```
MLE = hos.ML_estimators(TS)
MLE.get_ML_estimation(PWM_estimators= PWM)
MLE.get_statistics(gamma_true=0)
MLE.get_CIs(method='minimal_width')
MLE.plot()
```



Maximum Likelihood Estimation (MLE) with Frechet Distribution

In this section, we perform Maximum Likelihood Estimation (MLE) using the Frechet distribution on the time series data. The `Frechet_ML_estimators` object is used for this purpose. Below are the steps involved:

1. **Initialization**: `python MLE = hos.Frechet_ML_estimators(TS)` We initialize the `Frechet_ML_estimators` object with the time series data `TS`.
2. **MLE Estimation**: `python MLE.get_ML_estimation(PWM_estimators= PWM)` This method computes the MLE estimates for the given time series data using the PWM estimators as initial guesses for optimization.
3. **Statistics Calculation**: `python MLE.get_statistics(alpha_true=0)` This method calculates the statistics based on the MLE estimates. Here, `alpha_true` is the true value of the shape parameter used for comparison.
4. **Confidence Intervals**: `python MLE.get_CIs()` This method computes the confidence intervals for the MLE estimates.
5. **View Statistics**: `python MLE.plot()` This method plots the distribution of the obtained estimators together with the confidence intervals.

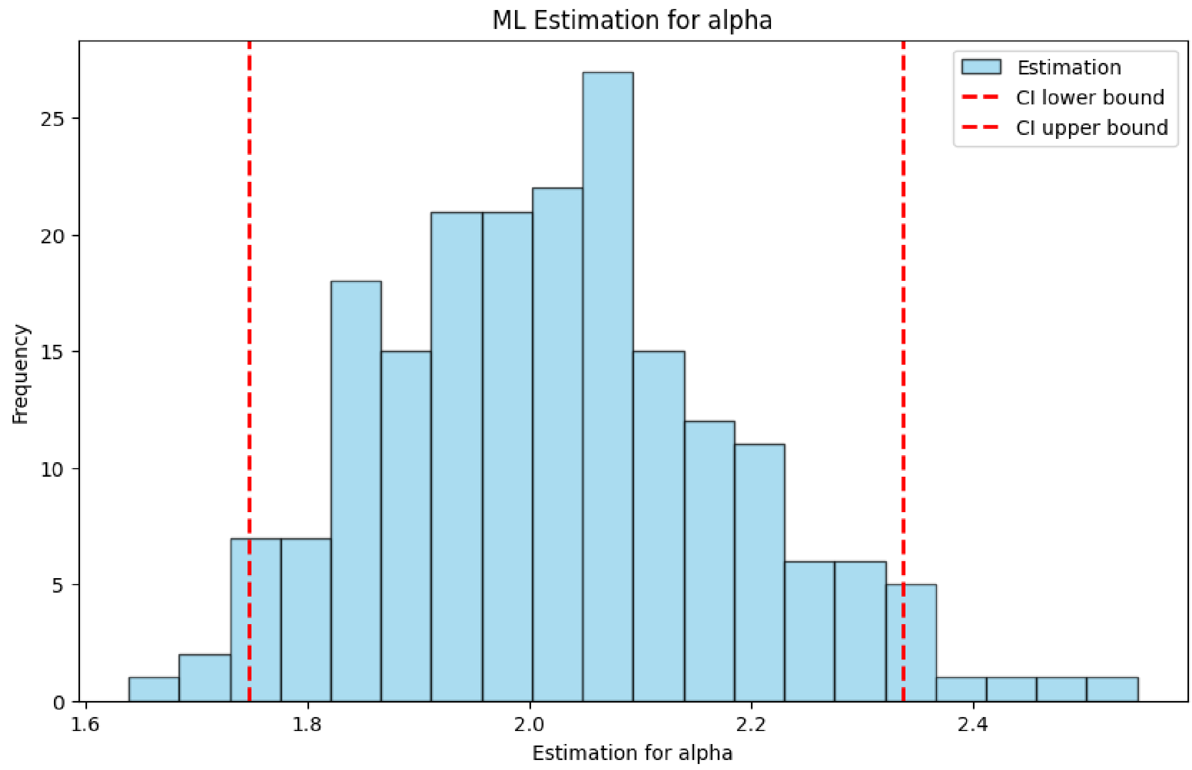
Difference between Frechet and GEV Distributions

The key difference between using the Frechet distribution and the Generalized Extreme Value (GEV) distribution lies in the type of tail behavior they model:

- **Frechet Distribution**: This distribution is used to model data with heavy tails. It is a special case of the GEV distribution with a positive shape parameter. The Frechet distribution is particularly useful for modeling extreme values that follow a power-law decay.
- **GEV Distribution**: The GEV distribution is a more general form that encompasses three types of distributions based on the shape parameter: Gumbel (light tails), Frechet (heavy tails), and Weibull (bounded tails). The GEV distribution provides more flexibility in modeling different types of extreme value behavior.

By using the `Frechet_ML_estimators`, we specifically focus on modeling heavy-tailed data, which is suitable for certain types of extreme value analysis.

[6]:
```
MLE = hos.Frechet_ML_estimators(TS)
MLE.get_ML_estimation(PWM_estimators= PWM)
MLE.get_statistics(alpha_true=0)
MLE.get_CIs()
MLE.plot()
```



1.2.3 Real Data

```
[1]: print('to be filled')
```

```
to be filled
```

1.2.4 The Bootstrap

```
[1]: print('to be filled')
```

```
to be filled
```

```
[ ]:
```

CHAPTER

TWO

TIME SERIES AND EXTREME VALUE ANALYSIS

In this tutorial, we will explore how to use the functionalities provided in *topt.py* to simulate time series data, extract block maxima, and perform Maximum Likelihood Estimation (MLE) for extreme value distributions. We will also see how to work with real data to extract high-order statistics and compute MLE.

We will cover: - Simulating time series data using the *TimeSeries* class. - Extracting block maxima and high-order statistics from the simulated data. - Estimating parameters for GEV and Frechet distributions using PWM and MLE. - Analyzing real-world datasets for extreme value statistics.

Let's walk through each of these steps with code examples.

2.1 Step 1: Simulating Time Series Data

The first step is to simulate a time series dataset using the *TimeSeries* class. You can specify the length of the series, the type of distribution (e.g., GEV), and whether to apply any correlation structure (e.g., IID or ARMAX).

Here's how you can simulate a GEV-distributed time series:

```
from topt import TimeSeries

# Create a time series object with 100 data points, GEV distribution, and␣
↪IID correlation
ts = TimeSeries(n=100, distr='GEV', correlation='IID', modelparams=[0.5])

# Simulate the time series with 10 repetitions
ts.simulate(rep=10)

# Print the simulated time series data
print(ts.values)
```

2.2 Step 2: Extracting Block Maxima

Once the time series is generated, the next step is to extract block maxima from the data. Block maxima are the largest values within blocks of a certain size in the time series.

Here's how to extract block maxima with a block size of 5:

```
# Extract block maxima with a block size of 5
ts.get_blockmaxima(block_size=5, stride='DBM')

# Print the extracted block maxima
print(ts.blockmaxima)
```

In this example, the *get_blockmaxima()* function divides the time series into blocks of size 5 and extracts the maximum value from each block. You can adjust the stride (e.g., 'DBM' for disjoint blocks or 'SBM' for sliding blocks).

2.3 Step 3: Extracting High Order Statistics

High-order statistics refer to the second, third, or higher-largest values within a block of data. You can extract these using the *get_HOS()* method.

Here's how to extract the top 2 largest values from each block:

```
# Extract the two highest values from each block
ts.get_HOS(orderstats=2, block_size=5, stride='DBM')

# Print the high-order statistics
print(ts.high_order_stats)
```

2.4 Step 4: Estimating Parameters with PWM

Once block maxima are extracted, you can estimate the parameters of the Generalized Extreme Value (GEV) distribution using Probability Weighted Moments (PWM). The *PWM_estimators* class handles this.

```
from topt import PWM_estimators

# Initialize PWM estimator with the time series data
pwm = PWM_estimators(ts)

# Compute PWM estimators for GEV parameters
pwm.get_PWM_estimation()

# Print the GEV parameter estimates
print(pwm.values)
```

2.5 Step 5: Maximum Likelihood Estimation (MLE)

To estimate the GEV or Frechet parameters using MLE, you can use the *ML_estimators* class. This method fits the distribution to the block maxima or high-order statistics.

Here's how to perform MLE for the GEV distribution:

```
from topt import ML_estimators

# Initialize MLE estimator with the time series data
ml = ML_estimators(ts)

# Perform MLE for the GEV distribution
ml.get_ML_estimation()

# Print the MLE results
print(ml.values)
```

2.6 Step 6: Analyzing Real Data

You can also work with real-world datasets using the *Data* class. This class allows you to extract block maxima and high-order statistics, and perform MLE on the dataset.

Here's how to analyze a real dataset:

```
from topt import Data

# Initialize the Data class with a real dataset
data = Data([2.5, 3.1, 1.7, 4.6, 5.3, 2.2, 6.0])

# Extract block maxima
data.get_blockmaxima(block_size=2, stride='DBM')

# Extract high-order statistics
data.get_HOS(orderstats=2, block_size=2, stride='DBM')

# Perform MLE on the dataset
data.get_ML_estimation(FrechetOrGEV='GEV')

# Print the MLE results
print(data.ML_estimators.values)
```

2.7 Conclusion

In this tutorial, we explored how to simulate time series data, extract block maxima and high-order statistics, and perform MLE for extreme value distributions. We also saw how to analyze real-world data for extreme value statistics using block maxima and MLE.

CHAPTER THREE

BOOTSTRAP AND MLE TUTORIAL

In this tutorial, we will explore how to work with block maxima extraction, bootstrap resampling, and Maximum Likelihood Estimation (MLE) for extreme value distributions using the *bootstrap.py* code.

We will cover: - Block maxima extraction using the *circmax()* function. - Generating bootstrap samples with *Bootstrap()*. - Aggregating bootstrap samples and estimating Fréchet and GEV parameters using *ML_Estimator*. - Running full bootstrapping procedures for MLE with *FullBootstrap*.

Let's walk through each of these steps with code examples.

3.1 Step 1: Block Maxima Extraction

The first step in working with extreme value statistics is often to extract block maxima from a dataset. The *circmax()* function allows you to do this using either disjoint blocks or sliding blocks.

Here's how you can extract block maxima from a sample:

```python
import numpy as np
from bootstrap import circmax

# Example dataset
sample = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Extract disjoint block maxima (DBM)
block_maxima_dbm = circmax(sample, bs=5, stride='DBM')
print("Disjoint Block Maxima (DBM):", block_maxima_dbm)

# Extract sliding block maxima (SBM)
block_maxima_sbm = circmax(sample, bs=3, stride='SBM')
print("Sliding Block Maxima (SBM):", block_maxima_sbm)
```

As you can see, *circmax()* allows you to specify the block size (*bs*) and the stride method (*DBM* or *SBM*). DBM extracts maxima from non-overlapping blocks, while SBM uses overlapping blocks, providing more block maxima.

3.2 Step 2: Bootstrapping a Sample

Next, we'll generate a bootstrap sample. Bootstrapping is a resampling technique used to estimate the variability of a statistic by randomly resampling the data with replacement.

Here's how you can generate a bootstrap sample from the block maxima we extracted earlier:

```
from bootstrap import Bootstrap

# Generate a bootstrap sample from the block maxima
boot_sample = Bootstrap(block_maxima_dbm)
print("Bootstrap Sample:", boot_sample)
```

In this case, the *Bootstrap()* function takes a list or array as input and returns a new sample of the same size, created by randomly selecting elements from the original data with replacement.

3.3 Step 3: Aggregating the Bootstrap Sample

Once you have a bootstrap sample, the next step is to aggregate the counts of unique values in the sample. This aggregation helps us prepare the data for MLE by summarizing the frequencies of the unique values.

Here's how you can aggregate the bootstrap sample:

```
from bootstrap import aggregate_boot

# Aggregate the counts of unique values in the bootstrap sample
aggregated_sample = aggregate_boot(boot_sample)
print("Aggregated Bootstrap Sample:", aggregated_sample)
```

Now, we have an aggregated sample that shows the unique values and their corresponding counts. This aggregated data will be used in the next step for MLE.

3.4 Step 4: Maximum Likelihood Estimation (MLE)

The *ML_Estimator* class allows us to perform MLE for either the Fréchet or GEV distribution using the aggregated bootstrap sample.

Let's first perform MLE for the Fréchet distribution:

```
from bootstrap import ML_Estimator

# Initialize the ML_Estimator with the aggregated bootstrap sample
estimator = ML_Estimator(aggregated_sample)

# Perform MLE for the Fréchet distribution
frechet_params = estimator.maximize_frechet()
print("Estimated Fréchet Parameters:", frechet_params)
```

Similarly, you can perform MLE for the GEV distribution:

```
# Perform MLE for the GEV distribution
gev_params = estimator.maximize_gev()
print("Estimated GEV Parameters:", gev_params)
```

With these methods, you can estimate the parameters (shape, scale, location) of both the Fréchet and GEV distributions using the MLE approach.

3.5 Step 5: Running Full Bootstrap for MLE

Finally, to estimate the variability of the MLE parameters, we can use the *FullBootstrap* class. This class applies the full bootstrapping procedure, including resampling, block maxima extraction, and MLE estimation, to obtain mean and standard deviation estimates for the parameters.

Here's how to run the full bootstrap procedure for the Fréchet distribution:

```
from bootstrap import FullBootstrap

# Example dataset
sample = np.random.rand(100)

# Initialize FullBootstrap with the dataset
bootstrap = FullBootstrap(sample, bs=10, stride='DBM', dist_type='Frechet')

# Run the bootstrap procedure
bootstrap.run_bootstrap(num_bootstraps=100)

# Print the mean and standard deviation of the estimates
print("Mean of Bootstrap Estimates:", bootstrap.statistics['mean'])
print("Standard Deviation of Bootstrap Estimates:", bootstrap.statistics[
↪'std'])
```

This process generates multiple bootstrap samples, applies MLE to each, and calculates the mean and standard deviation of the resulting estimates. You can also do this for the GEV distribution by setting *dist_type='GEV'*.

3.6 Conclusion

In this tutorial, we walked through the process of extracting block maxima, generating bootstrap samples, aggregating the data, and estimating parameters using MLE. We also saw how to apply the full bootstrap procedure to analyze the variability of the MLE estimates. This framework is essential when dealing with extreme value theory and understanding the uncertainty in parameter estimates.

CHAPTER

FOUR

UTILITY FUNCTIONS FOR EXTREME VALUE ANALYSIS

In this tutorial, we will explore the utility functions provided in *miscellaneous.py* that are used in the context of extreme value analysis. These functions include calculating probability-weighted moments, simulating time series, computing GEV distributions, and performing basic statistical tasks like sigmoid and inverse sigmoid transformations.

We will cover: - Basic mathematical functions like *sigmoid()* and *mse()*. - Functions related to Generalized Extreme Value (GEV) distributions such as *GEV_pdf()*, *GEV_cdf()*, and *PWM_estimation()*. - Simulating time series data using the *simulate_timeseries()* function.

Let's go through each function step by step.

4.1 Step 1: Using the Sigmoid and Inverse Sigmoid Functions

The sigmoid function is used to map real-valued numbers into the range (0, 1), and the inverse sigmoid performs the opposite transformation.

Here's how you can use the *sigmoid()* and *invsigmoid()* functions:

```
from miscellaneous import sigmoid, invsigmoid

# Apply sigmoid transformation
x = [-2, -1, 0, 1, 2]
sigmoid_values = sigmoid(x)
print(sigmoid_values)

# Apply inverse sigmoid transformation
y = [0.1, 0.5, 0.9]
invsigmoid_values = invsigmoid(y)
print(invsigmoid_values)
```

The sigmoid function maps any real-valued input to a value between 0 and 1, while the inverse sigmoid transforms probabilities back to their original values.

4.2 Step 2: Calculating Probability Weighted Moments (PWM)

In extreme value theory, PWMs are used to estimate parameters of the Generalized Extreme Value (GEV) distribution. The function *PWM_estimation()* computes the first three PWMs based on block maxima.

Here's how to compute PWM for a set of block maxima:

```
from miscellaneous import PWM_estimation

# Example block maxima data
maxima = [5, 8, 12, 15, 18]

# Compute PWM estimators
beta_0, beta_1, beta_2 = PWM_estimation(maxima)
print(f"β0: {beta_0}, β1: {beta_1}, β2: {beta_2}")
```

These PWMs can then be used to estimate GEV parameters using the *PWM2GEV()* function, which converts PWMs to GEV parameters (shape, location, and scale).

4.3 Step 3: Estimating GEV Parameters from PWM

The function *PWM2GEV()* converts the first three PWM moments into GEV distribution parameters: shape (γ), location (μ), and scale (σ).

Here's how to compute GEV parameters from PWM estimators:

```
from miscellaneous import PWM2GEV

# PWM estimators
b_0, b_1, b_2 = 11.6, 11.2, 39.2

# Compute GEV parameters
gamma, mu, sigma = PWM2GEV(b_0, b_1, b_2)
print(f"GEV Shape (γ): {gamma}, Location (μ): {mu}, Scale (σ): {sigma}")
```

The *PWM2GEV()* function allows you to estimate the GEV distribution parameters based on the computed PWM moments.

4.4 Step 4: Working with the GEV Distribution

The module provides several functions to compute properties of the Generalized Extreme Value (GEV) distribution, including: - *GEV_pdf()*: Computes the Probability Density Function (PDF). - *GEV_cdf()*: Computes the Cumulative Distribution Function (CDF). - *GEV_ll()*: Computes the log-likelihood of the GEV distribution.

Here's how to use these functions:

```
from miscellaneous import GEV_pdf, GEV_cdf, GEV_ll

# Example data
x = [1, 2, 3, 4, 5]
```

(continues on next page)

(continued from previous page)

```
# Compute GEV PDF
pdf_values = GEV_pdf(x, gamma=0.5, mu=2, sigma=1)
print("GEV PDF:", pdf_values)

# Compute GEV CDF
cdf_values = GEV_cdf(x, gamma=0.5, mu=2, sigma=1)
print("GEV CDF:", cdf_values)

# Compute GEV log-likelihood
ll_values = GEV_ll(x, gamma=0.5, mu=2, sigma=1)
print("GEV Log-Likelihood:", ll_values)
```

These functions allow you to work with GEV distributions for various tasks like computing probabilities, densities, or performing likelihood-based inference.

4.5 Step 5: Simulating Time Series Data

The *simulate_timeseries()* function is a powerful utility to generate time series data with different distributions and correlation structures. You can simulate IID (independent and identically distributed) data or time series with temporal dependence using ARMAX or AR models.

Here's how to simulate a time series:

```
from miscellaneous import simulate_timeseries

# Simulate an IID time series with GEV distribution
simulated_ts = simulate_timeseries(n=100, distr='GEV', correlation='IID',␣
↪modelparams=[0.5], seed=42)

# Print the first 10 values
print(simulated_ts[:10])
```

This function supports various distributions (e.g., GEV, Frechet, GPD) and allows you to introduce temporal dependence using ARMAX or AR models.

4.6 Conclusion

In this tutorial, we explored several utility functions provided in *miscellaneous.py* for extreme value analysis. These functions help in tasks ranging from basic mathematical transformations (like sigmoid) to more advanced operations like PWM estimation, GEV parameter estimation, and time series simulation.

CHAPTER

FIVE

REFERENCE: TOPT

This module is specialized in implementing Top-t based Maximum Likelihood Estimators.

5.1 Overview

The *xtremes.topt* module provides tools for analyzing higher order statistics and their influence on Maximum Likelihood estimations. It includes classes and functions for handling time series data, extracting block maxima, and performing statistical analysis.

5.2 Classes

5.3 The `TimeSeries` Class and its Functionalities

The *TimeSeries* class is used to handle and manipulate time series data. It provides methods for extracting block maxima and high order statistics.

class xtremes.topt.**TimeSeries**(*n*, *distr='GEV'*, *correlation='IID'*, *modelparams=[0]*, *ts=0*)

Bases: `object`

TimeSeries class for simulating and analyzing time series data with optional correlation structures.

This class is designed to simulate time series data based on specified distributions and correlation types. It also provides methods to extract block maxima and high order statistics from the simulated data.

5.3.1 Parameters

n

[int] The length of the time series.

distr

[str, optional] The distribution to simulate the time series data from. Default is 'GEV' (Generalized Extreme Value).

correlation

[str, optional] The type of correlation structure. Options include ['IID', 'ARMAX', 'AR']. Default is 'IID' (independent and identically distributed).

modelparams
[list, optional] The parameters of the specified distribution. Default is [0].

ts
[float, optional] A parameter for controlling the time series characteristics, particularly for correlated series (e.g., in AR models). Must be in the range [0, 1]. Default is 0.

5.3.2 Attributes

values
[list] A list to store the simulated time series data.

distr
[str] The type of distribution used for generating the time series data.

corr
[str] The type of correlation structure applied to the time series.

modelparams
[list] The parameters of the chosen distribution model.

ts
[float] A parameter controlling the correlation or time series structure, if applicable.

len
[int] The length of the time series.

blockmaxima
[list] A list storing block maxima extracted from the simulated data.

high_order_stats
[list] A list storing the high order statistics extracted from the simulated data.

5.3.3 Methods

simulate(rep=10, seeds='default'):
Simulates time series data based on the given distribution and correlation type.

get_blockmaxima(block_size=2, stride='DBM', rep=10):
Extracts block maxima from the simulated time series data.

get_HOS(orderstats=2, block_size=2, stride='DBM', rep=10):
Extracts high order statistics from the simulated time series data.

5.3.4 Examples

```
>>> # Create a TimeSeries object
>>> ts = TimeSeries(n=100, distr='GEV', correlation='ARMAX',␣
↪modelparams=[0.5], ts=0.6)
>>> # Simulate time series data with 5 repetitions and specific seeds
>>> ts.simulate(rep=5, seeds=[42, 123, 456, 789, 1011])
>>> # Extract block maxima using a block size of 5
>>> ts.get_blockmaxima(block_size=5, stride='DBM', rep=5)
>>> # Extract high order statistics (order 3) using the same block␣
```

(continues on next page)

(continued from previous page)

```
↪size and stride
>>> ts.get_HOS(orderstats=3, block_size=5, stride='DBM', rep=5)
```

get_HOS(*orderstats=2*, *block_size=2*, *stride='DBM'*, *rep=10*)

Extract high order statistics from simulated time series data.

High order statistics include values such as the second-largest value within each block of the time series.

Parameters

orderstats
: [int, optional] The order of statistics to extract. Default is 2 (i.e., second-highest value).

block_size
: [int, optional] The size of blocks from which to extract the statistics. Default is 2.

stride
: [str, optional] The stride or step type used for block extraction. Choose from ['SBM', 'DBM']. Default is 'DBM'.

rep
: [int, optional] The number of repetitions for extracting high order statistics. Should match the number of simulations. Default is 10.

get_blockmaxima(*block_size=2*, *stride='DBM'*, *rep=10*)

Extract block maxima from simulated time series data.

Block maxima are the maximum values extracted from blocks of the time series data.

Parameters

block_size
: [int, optional] The size of blocks from which to extract maxima. Default is 2.

stride
: [{str, int}, optional] The stride or step type used for block extraction. Choose from ['SBM' (Sliding Block Maxima), 'DBM' (Disjoint Block Maxima)] or specify an integer as a step size. Default is 'DBM'.

rep
: [int, optional] The number of repetitions for maxima extraction. This should match the number of simulations. Default is 10.

plot(*rep=1*, *filename=None*)

Plot the simulated time series data.

This method generates a plot showing the simulated time series data. The user can choose to display data from specific repetitions or all repetitions.

Parameters

rep
[int or list, optional] The repetition number(s) to plot. If 0, all repetitions are plotted. If a list is provided, only the specified repetitions are plotted. Default is 1.

filename
[str, optional] The name of the PNG file to save the plot. If None, the plot is displayed but not saved. Default is None.

plot_blockmaxima (*rep=1*, *filename=None*, *plot_type='line'*)

Plot the extracted block maxima.

This method generates a plot showing the extracted block maxima from the simulated time series data. The user can choose to display block maxima from specific repetitions or all repetitions. The plot can be either a line plot or a histogram.

Parameters

rep
[int or list, optional] The repetition number(s) to plot. If 0, all repetitions are plotted. If a list is provided, only the specified repetitions are plotted. Default is 1.

filename
[str, optional] The name of the PNG file to save the plot. If None, the plot is displayed but not saved. Default is None.

plot_type
[str, optional] The type of plot to generate. Options are 'line' for a line plot and 'hist' for a histogram. Default is 'line'.

plot_with_blockmaxima (*rep=1*, *plotlim=300*, *filename=None*)

Plot the simulated time series data along with block maxima.

This method generates a plot showing the simulated time series data along with the block maxima. The user can choose to display data from specific repetitions or all repetitions.

Parameters

rep
[int, optional] The repetition number to plot. Default is 1.

plotlim
[int, optional] The limit for the number of data points to plot. Default is 300.

filename
[str, optional] The name of the PNG file to save the plot. If None, the plot is displayed but not saved. Default is None.

simulate (*rep=10*, *seeds='default'*)

Simulate time series data.

This method generates time series data based on the specified distribution, correlation type, and other parameters.

Parameters

rep
[int, optional] The number of repetitions (simulations) to run. Default is 10.

seeds
[{str, list}, optional] Seed(s) for random number generation. If 'default', a sequence of seeds is automatically generated based on the number of repetitions. If a list of seeds is provided, it must match the number of repetitions (*rep*). Default is 'default'.

Raises

ValueError
If the number of provided seeds does not match the number of repetitions.

5.4 Examples

1. **Extract Block Maxima**:

```
from xtremes.topt import TimeSeries
import numpy as np

ts_data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
↪ 14, 15])
ts = TimeSeries(ts_data)
block_maxima = ts.extract_BM(block_size=5, stride='DBM')
print("Block Maxima:", block_maxima)
```

2. **Extract High Order Statistics**:

```
from xtremes.topt import TimeSeries
import numpy as np

ts_data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
↪ 14, 15])
ts = TimeSeries(ts_data)
high_order_stats = ts.extract_HOS(orderstats=3, block_size=5,␣
↪stride='DBM')
print("High Order Statistics:", high_order_stats)
```

5.5 The `HighOrderStats` Class and its Functionalities

The *HighOrderStats* class is used to compute and analyze higher order statistics from the time series data. It provides methods for calculating log-likelihoods and performing Maximum Likelihood Estimation (MLE).

class xtremes.topt.**HighOrderStats**(*TimeSeries*)

Bases: `object`

HighOrderStats class for calculating and analyzing high-order statistics of time series data.

5.5.1 Notes

This class provides functionality for calculating Probability Weighted Moment (PWM) estimators and Maximum Likelihood (ML) estimators from a given TimeSeries object.

5.5.2 Methods

get_PWM_estimation():
Calculate the PWM estimators for the time series data.

get_ML_estimation(initParams='auto', FrechetOrGEV = 'Frechet', option=1, estimate_pi=False):
Calculate the ML estimators for the time series data.

5.5.3 Attributes

TimeSeries
[TimeSeries] The TimeSeries object containing the time series data.

high_order_stats
[list] List of high-order statistics extracted from the TimeSeries object.

blockmaxima
[list] List of block maxima derived from the high-order statistics.

gamma_true
[float] True gamma parameter of the GEV distribution derived from the TimeSeries object.

PWM_estimators
[PWM_estimators] Instance of PWM_estimators class for calculating PWM estimators.

ML_estimators
[ML_estimators] Instance of ML_estimators class for calculating ML estimators.

5.5.4 Example

```
>>> # Create a TimeSeries object
>>> ts = TimeSeries(n=100, distr='GEV', correlation='ARMAX',␣
↪modelparams=[0.5], ts=0.6)
>>> # Simulate time series data
>>> ts.simulate(rep=5, seeds=[42, 123, 456, 789, 1011])
>>> # Initialize HighOrderStats object
>>> hos = HighOrderStats(ts)
>>> # Calculate PWM estimators
>>> hos.get_PWM_estimation()
>>> # Calculate ML estimators
>>> hos.get_ML_estimation(initParams='auto', r=1)
```

get_ML_estimation(*initParams='auto'*, *r=None*, *FrechetOrGEV='Frechet'*)

Calculate the Maximum Likelihood (ML) estimators.

Parameters

initParams
: [str or array-like, optional] Method for initializing parameters. Default is 'auto', which uses automatic parameter initialization.

r
: [int, optional] Number of order statistics to calculate the log-likelihood on. If not specified, use all provided.

FrechetOrGEV
: [str, optional] Whether to fit the Frechet or GEV distribution.

Notes

This function performs maximum likelihood estimation based on either the Frechet or GEV distribution.

get_PWM_estimation()

Calculate the Probability Weighted Moment (PWM) estimators.

5.6 Examples

1. **Log Likelihood**:

```
from xtremes.topt import HighOrderStats
import numpy as np

hos_data = np.array([[0.1, 0.2], [0.3, 0.4], [0.2, 0.5], [0.4,
↪ 0.6]])
hos = HighOrderStats(hos_data)
log_likelihood = hos.log_likelihood(gamma=0.5, mu=0, sigma=2,␣
↪r=2)
print("Log Likelihood:", log_likelihood)
```

2. **Frechet Log Likelihood**:

```
from xtremes.topt import HighOrderStats
import numpy as np

hos_data = np.array([[0.5, 1.0], [1.5, 2.0], [1.2, 2.2], [2.0,
↪ 3.0]])
hos = HighOrderStats(hos_data)
frechet_log_likelihood = hos.Frechet_log_likelihood(alpha=2,␣
↪sigma=1.5, r=2)
print("Frechet Log Likelihood:", frechet_log_likelihood)
```

5.7 The `Data` Class and its Functionalities

The *Data* class is used to handle and manipulate real data for analysis.

class xtremes.topt.**Data**(*values*)

Bases: `object`

A class for analyzing data with block maxima and high order statistics.

This class provides methods to extract block maxima and high order statistics from a given dataset. It also supports Maximum Likelihood (ML) estimation for the parameters of either the Frechet or GEV distributions.

5.7.1 Parameters

values
[list or numpy.ndarray] The dataset from which block maxima and high order statistics are extracted. This should be a 1D array or list of values representing the time series or data.

5.7.2 Attributes

values
[list or numpy.ndarray] The dataset on which operations are performed.

len
[int] The length of the dataset.

blockmaxima
[list] List to store the block maxima extracted from the dataset.

bm_indices
[list] List of indices corresponding to the positions of the block maxima in the original dataset.

high_order_stats
[list] List to store the high order statistics extracted from the dataset.

ML_estimators
[ML_estimators_data] Object that stores and handles the ML estimation results for the Frechet or GEV parameters.

5.7.3 Methods

get_blockmaxima(block_size=2, stride='DBM'):
Extracts block maxima from the dataset.

get_HOS(orderstats=2, block_size=2, stride='DBM'):
Extracts high order statistics from the dataset.

get_ML_estimation(r=None, FrechetOrGEV='GEV'):
Computes ML estimations for the Frechet or GEV parameters.

5.7.4 Example

```
>>> # Initialize the Data object with a dataset
>>> data = Data([2.5, 3.1, 1.7, 4.6, 5.3, 2.2, 6.0])
>>> # Extract block maxima using a block size of 2
>>> data.get_blockmaxima(block_size=2, stride='DBM')
>>> print(data.blockmaxima)
>>> # Extract second-highest order statistics (HOS) with the same␣
↪block size and stride
>>> data.get_HOS(orderstats=2, block_size=2, stride='DBM')
>>> print(data.high_order_stats)
>>> # Perform ML estimation using the extracted HOS, choosing between␣
↪Frechet and GEV
>>> data.get_ML_estimation(FrechetOrGEV='GEV')
>>> print(data.ML_estimators.values)
```

get_HOS(*orderstats=2*, *block_size=2*, *stride='DBM'*)

Extract high order statistics (HOS) from the dataset.

High order statistics refer to statistics of interest beyond the maximum (e.g., second-highest, third-highest). This method extracts these statistics from blocks of the dataset.

Parameters

orderstats
[int, optional] The order of the statistic to extract (e.g., 2 for second-highest). Default is 2.

block_size
[int, optional] The size of blocks from which to extract the statistics. Default is 2.

stride
[str or int, optional] The type of stride to use when extracting blocks. Choose from: - 'SBM': Sliding Block Maxima (step size 1) - 'DBM': Disjoint Block Maxima (non-overlapping blocks) - int: Specifies the step size directly. Default is 'DBM'.

Returns

None
The high order statistics are stored in the *high_order_stats* attribute.

get_ML_estimation(*r=None*, *FrechetOrGEV='GEV'*)

Compute Maximum Likelihood (ML) estimations for the Frechet or GEV parameters.

This method computes ML estimators for the parameters of either the Frechet or GEV distribution based on the high order statistics extracted from the data. If no high order statistics are available, it will first extract them.

Parameters

r
[int, optional] The number of order statistics to use in the ML estimation. If not specified, it uses all the extracted statistics.

FrechetOrGEV
[str, optional] The type of distribution to use for the ML estimation. Choose between 'Frechet' and 'GEV'. Default is 'GEV'.

Returns

None
The ML estimators are stored in the *ML_estimators* attribute.

get_blockmaxima(*block_size=2*, *stride='DBM'*)

Extract block maxima from the dataset.

Block maxima are the largest values in each block of the dataset, where the block size and the stride (step size) determine how the blocks are divided.

Parameters

block_size
[int, optional] The size of blocks for maxima extraction. Default is 2.

stride
[str or int, optional] The type of stride to use when extracting blocks. Choose from: - 'SBM': Sliding Block Maxima (step size 1) - 'DBM': Disjoint Block Maxima (non-overlapping blocks) - int: Specifies the step size directly. Default is 'DBM'.

Returns

None
The block maxima and their corresponding indices are stored in the *blockmaxima* and *bm_indices* attributes.

5.8 The `PWM_estimators` Class

The *PWM_estimators* class is used to compute Probability Weighted Moment (PWM) estimators.

class `xtremes.topt.`**`PWM_estimators`**(*TimeSeries*)

Bases: `object`

Calculates Probability Weighted Moment (PWM) estimators from block maxima and computes statistics and confidence intervals for the GEV parameters.

5.8.1 Notes

This class provides methods to compute the Probability Weighted Moment (PWM) estimators and convert them into Generalized Extreme Value (GEV) distribution parameters (gamma, mu, sigma). It also allows users to compute confidence intervals for the estimated parameters and assess the statistical properties (mean, variance, bias, mean squared error) of the estimators compared to a true gamma value. Visualization options are provided to plot the estimators and confidence intervals.

5.8.2 Parameters

param TimeSeries
TimeSeries object TimeSeries object containing block maxima or high order statistics.

5.8.3 Attributes

attribute blockmaxima
numpy.ndarray Array of block maxima extracted from the TimeSeries object.

attribute values
numpy.ndarray Array containing the PWM estimators (γ, μ, σ) for each block maxima series.

attribute statistics
dict Dictionary containing statistics (mean, variance, bias, mse) and confidence intervals of the PWM estimators.

5.8.4 Methods

method get_PWM_estimation()
Compute PWM estimators for each block maxima series and convert them into GEV parameters.

method get_statistics(gamma_true)
Compute statistics of the PWM estimators using a true gamma value.

method get_CIs(alpha=0.05, method='symmetric')
Compute confidence intervals (CIs) for the GEV parameters using either symmetric quantiles or minimal width intervals.

method plot(param='gamma', show_CI=True, show_true=True, filename=None)
Plot the PWM estimators for the GEV parameters (gamma, mu, sigma) with options to display confidence intervals and save the plot as an image.

5.8.5 Raises

raises ValueError
If block maxima or high order statistics are not available in the TimeSeries object.

5.8.6 Examples

```
>>> from TimeSeries import TimeSeries
>>> from PWM_estimators import PWM_estimators
>>> ts = TimeSeries(data)  # Initialize TimeSeries object with data
>>> ts.get_blockmaxima(block_size=10, stride='SBM')  # Extract block␣
↪maxima
>>> pwm = PWM_estimators(ts)  # Initialize PWM_estimators object
>>> pwm.get_PWM_estimation()  # Compute PWM estimators
>>> pwm.get_statistics(0.1)  # Compute statistics with true gamma␣
↪value 0.1
>>> pwm.get_CIs(alpha=0.05, method='minimal_width')  # Compute␣
↪confidence intervals
>>> pwm.plot(param='gamma', show_CI=True, show_true=True, filename=
↪'PWM_plot.png')  # Plot the results and save as image
>>> print(pwm.statistics)  # Print computed statistics
{'gamma_mean': 0.25, 'gamma_variance': 0.005, 'gamma_bias': 0.02,
↪'gamma_mse': 0.01,
 'mu_mean': 1.15, 'mu_variance': 0.04, 'sigma_mean': 0.9, 'sigma_
↪variance': 0.02,
 'gamma_CI': (0.2, 0.5), 'mu_CI': (1.1, 1.5), 'sigma_CI': (0.8, 1.0)}
```

get_CIs(*alpha=0.05*, *method='symmetric'*)

Compute confidence intervals (CIs) for the GEV parameters using different methods.

Notes

This function calculates confidence intervals (CIs) for the Generalized Extreme Value (GEV) parameters (gamma, mu, and sigma) estimated from Probability-Weighted Moments (PWM). The user can choose between two methods for computing the confidence intervals:

- **'symmetric'**: This method uses the quantiles of the distribution of parameter estimates to compute symmetric confidence intervals.
- **'minimal_width'**: This method finds the interval with minimal width that contains the desired proportion (1 - alpha) of the sorted parameter estimates.

For each block maxima series, confidence intervals for the GEV shape (gamma), location (mu), and scale (sigma) parameters are calculated. The results are stored in the *self.statistics* dictionary, with keys 'gamma_CI', 'mu_CI', and 'sigma_CI' corresponding to the computed confidence intervals.

Parameters

param alpha
float, optional Significance level for the confidence intervals (default is 0.05, for a 95% CI).

param method
str, optional Method for computing the confidence intervals. Options are: - 'symmetric': Uses quantiles to compute symmetric CIs (default). - 'minimal_width': Computes the minimal width interval containing (1 - alpha) of the estimates.

Example

```
>>> estimator = PWM_Estimators(timeseries_data)
>>> estimator.get_PWM_estimation()
>>> estimator.get_CIs(alpha=0.05, method='minimal_width')
>>> print(estimator.statistics)
{'gamma_CI': (0.2, 0.5), 'mu_CI': (1.1, 1.5), 'sigma_CI': (0.8, 1.
↪0)}
```

Returns

None
The results are stored in *self.statistics*, which contains the confidence intervals for each GEV parameter.

get_PWM_estimation()

Compute Probability-Weighted Moment (PWM) estimators and convert them into GEV parameters.

Notes

This function iterates over each block maxima series, computes the Probability-Weighted Moments (PWMs), and converts the PWMs into the Generalized Extreme Value (GEV) distribution parameters: shape (gamma), location (mu), and scale (sigma). The function utilizes the *misc.PWM_estimation* function to calculate the PWMs and then applies *misc.PWM2GEV* to convert these moments into GEV parameters.

The results for each block maxima series are stored in the *self.values* attribute, which is a NumPy array where each row corresponds to the GEV parameters [gamma, mu, sigma] for a specific block maxima series.

This function clears any previously stored values in *self.values* before appending new estimates.

Parameters

None

Example

```
>>> estimator = PWM_estimators(timeseries_data)
>>> estimator.get_PWM_estimation()
>>> print(estimator.values)
array([[0.2, 1.1, 0.8],
    [0.3, 1.2, 0.9]])
```

Returns

None
The results are stored in *self.values*, a NumPy array containing the estimated GEV parameters for each block maxima series.

get_statistics(*gamma_true*)

Compute statistics of the PWM estimators using the true gamma value.

Notes

This function calculates various statistical measures (mean, variance, bias, and mean squared error) for the estimated Generalized Extreme Value (GEV) shape parameter (gamma) compared to a provided true value (*gamma_true*). It also computes the mean and variance for the location (mu) and scale (sigma) parameters across all block maxima series.

- **Mean Squared Error (MSE)**: Measures the average of the squares of the differences between the estimated and true gamma values.
- **Bias**: Represents the systematic deviation of the estimated gamma values from the true gamma value.
- **Variance**: Describes the spread of the estimated gamma values around their mean.

If only one block maxima series is available, a warning is raised since variance cannot be computed.

The computed statistics are stored in the *self.statistics* dictionary with the following keys: - 'gamma_mean', 'gamma_variance', 'gamma_bias', 'gamma_mse' - 'mu_mean', 'mu_variance' - 'sigma_mean', 'sigma_variance'

Parameters

param gamma_true
float The true value of the GEV shape parameter (gamma) used to compute bias and MSE.

Example

```
>>> estimator = PWM_estimators(timeseries_data)
>>> estimator.get_PWM_estimation()
>>> estimator.get_statistics(gamma_true=0.2)
>>> print(estimator.statistics)
{'gamma_mean': 0.25, 'gamma_variance': 0.005, 'gamma_bias': 0.02,
↪'gamma_mse': 0.01,
'mu_mean': 1.15, 'mu_variance': 0.04, 'sigma_mean': 0.9, 'sigma_
↪variance': 0.02}
```

Returns

None
The results are stored in *self.statistics*, containing the calculated statistics for gamma, mu, and sigma.

plot (*param='gamma'*, *show_CI=True*, *show_true=True*, *filename=None*)

Plot the PWM estimators and confidence intervals for the GEV parameters.

Notes

This function generates a plot showing the Probability-Weighted Moment (PWM) estimators for the Generalized Extreme Value (GEV) parameters (gamma, mu, sigma) computed from block maxima. The user can choose to display confidence intervals (CIs) for each parameter.

The plot is saved as a PNG image if the *save* parameter is set to True.

Parameters

param param
str, optional GEV parameter to plot (default is 'gamma').

param show_CI
bool, optional Flag indicating whether to display confidence intervals (default is True).

param show
bool, optional Flag indicating whether to display the plot.

param save
bool, optional Flag indicating whether to save the plot as a PNG image (default is False).

param filename
str, optional Name of the PNG file to save the plot (default is None).

Example

```
>>> estimator = PWM_estimators(timeseries_data)
>>> estimator.get_PWM_estimation()
>>> estimator.get_CIs(alpha=0.05, method='minimal_width')
>>> estimator.plot(show_CI=True, show_true=True, save=True,␣
↪filename='PWM_estimation.png')
```

Returns

None
The plot is displayed in the console and saved as a PNG image if the *save* parameter is set to True.

`xtremes.topt.`**`automatic_parameter_initialization`**(*PWM_estimators*, *corr*, *ts=0.5*)

Automatic parameter initialization for ML estimation.

5.8.7 Notes

This function is designed for initializing parameters for maximum likelihood estimation (ML) in statistical models. It automatically computes the probability weighted moment (PWM) estimators and adjusts them based on the specified correlation type ('ARMAX', 'IID', etc.). The 'ts' parameter is used to control the strength of temporal dependence in the model.

5.8.8 Parameters

param PWM_estimators
list or numpy.array Probability weighted moment estimators.

param corr
str Correlation type for the model. Supported values are 'ARMAX', 'IID', etc.

param ts
float, optional Time series parameter controlling the strength of temporal dependence (default is 0.5).

return
numpy.ndarray Initial parameters for ML estimation.

5.8.9 See also

misc.PWM_estimation : Function for computing probability weighted moment estimators.

5.8.10 Examples

```
>>> from test_xtremes import misc
>>> PWM_est = misc.PWM_estimators(data)
>>> init_params = automatic_parameter_initialization(PWM_est, 'ARMAX',
↪ ts=0.8)
```

5.9 The ML_estimators, Frechet_ML_estimators and ML_estimators_data Classes

The *ML_estimators*, *Frechet_ML_estimators*, and *ML_estimators_data* classes are used for performing Maximum Likelihood Estimation (MLE) and handling the results.

class xtremes.topt.**ML_estimators**(*TimeSeries*)

Bases: object

Maximum Likelihood Estimators (MLE) for GEV parameters.

This class calculates Maximum Likelihood Estimators (MLE) for the parameters of the Generalized Extreme Value (GEV) distribution using the method of maximum likelihood estimation.

5.9.1 Parameters

TimeSeries
[TimeSeries] The TimeSeries object containing the data for which MLE estimators will be calculated.

5.9.2 Attributes

values
[numpy.ndarray] An array containing the MLE estimators for each set of high order statistics.

statistics
[dict] A dictionary containing statistics computed from the MLE estimators.

5.9.3 Methods

__len__()
Returns the number of MLE estimators calculated.

get_ML_estimation(PWM_estimators=None, initParams='auto', option=1, estimate_pi=False)
Computes the MLE estimators for the GEV parameters.

get_statistics(gamma_true)
Computes statistics from the MLE estimators.

5.9.4 Examples

```
>>> import numpy as np
>>> from hos import TimeSeries, ML_estimators, PWM_estimators
>>> blockmaxima_data = np.random.normal(loc=10, scale=2, size=100)
>>> high_order_stats_data = np.random.normal(loc=5, scale=1,␣
↪size=(100, 3))
>>> ts = TimeSeries(blockmaxima=blockmaxima_data, high_order_
↪stats=high_order_stats_data, corr='IID', ts=0.5)
>>> pwm = PWM_estimators(ts)
>>> pwm.get_PWM_estimation()
>>> ml = ML_estimators(ts)
>>> ml.get_ML_estimation(PWM_estimators=pwm)
>>> ml.get_statistics(gamma_true=0.1)
>>> print("ML Estimators:")
>>> print(ml.values)
>>> print("\nStatistics:")
>>> print(ml.statistics)
```

get_CIs(*alpha=0.05*, *method='symmetric'*)
Compute confidence intervals (CIs) for the GEV parameters using different methods.

Notes

This function calculates confidence intervals (CIs) for the Generalized Extreme Value (GEV) parameters (gamma, mu, and sigma) estimated from Maximum Likelihood Estimators (MLE). The user can choose between two methods for computing the confidence intervals:

- **'symmetric'**: This method uses the quantiles of the distribution of parameter estimates to compute symmetric confidence intervals.
- **'minimal_width'**: This method finds the interval with minimal width that contains the desired proportion (1 - alpha) of the sorted parameter estimates.

For each block maxima series, confidence intervals for the GEV shape (gamma), location (mu), and scale (sigma) parameters are calculated. The results are stored in the *self.statistics* dictionary, with keys 'gamma_CI', 'mu_CI', and 'sigma_CI' corresponding to the computed confidence intervals.

Parameters

param alpha
float, optional Significance level for the confidence intervals (default is 0.05, for a 95% CI).

param method
str, optional Method for computing the confidence intervals. Options are: - 'symmetric': Uses quantiles to compute symmetric CIs (default). - 'minimal_width': Computes the minimal width interval containing (1 - alpha) of the estimates.

Example

```
>>> estimator = ML_stimator(timeseries_data)
>>> estimator.get_PWM_estimation()
>>> estimator.get_CIs(alpha=0.05, method='minimal_width')
>>> print(estimator.statistics)
{'gamma_CI': (0.2, 0.5), 'mu_CI': (1.1, 1.5), 'sigma_CI': (0.8, 1.
↪0)}
```

Returns

None
The results are stored in *self.statistics*, which contains the confidence intervals for each GEV parameter.

get_ML_estimation(*PWM_estimators=None*, *initParams='auto'*, *r=None*)

Compute Maximum Likelihood (ML) estimators for each series of high order statistics within the *ML_estimators* class.

This method fits the Generalized Extreme Value (GEV) distribution to each series of high order statistics using Maximum Likelihood Estimation (MLE) by optimizing the log-likelihood function.

Parameters

PWM_estimators
[PWM_estimators object, optional] An object containing PWM (Probability Weighted Moments) estimators. This is used for initializing the parameters in the ML estimation if *initParams* is set to 'auto'. Required if *initParams* is 'auto'.

initParams
[str or numpy.ndarray, optional] Initial parameters for the ML estimation. If 'auto', the initial parameters will be computed automatically using the *PWM_estimators* object. If

a NumPy array is provided, these will be used as initial parameter values. Default is 'auto'.

r
[int, optional] The number of order statistics to calculate the log-likelihood on. If not specified, all provided order statistics will be used.

Returns

None
This method updates the *self.values* attribute of the *ML_estimators* object with the estimated parameters (gamma, mu, sigma) for each series of high order statistics.

Raises

ValueError
If *initParams* is set to 'auto' and no *PWM_estimators* are provided, a ValueError is raised.

Notes

- This method performs Maximum Likelihood Estimation (MLE) to fit the Generalized Extreme Value (GEV) distribution to the high order statistics within the *ML_estimators* class.
- The method uses optimization techniques such as Nelder-Mead (and optionally COBYLA) to minimize the negative log-likelihood.
- If *initParams* is set to 'auto', the initial parameters for the optimization are derived using the *PWM_estimators* object.
- The optimization results (gamma, mu, sigma) are stored in the *self.values* list for each series of high order statistics.

`get_statistics` (*gamma_true*)

Compute statistics of the ML estimators using a true γ value.

Parameters

param gamma_true
float True γ value for calculating statistics.

`plot` (*param='gamma'*, *show_CI=True*, *show_true=True*, *filename=None*)

Plot the ML estimators and confidence intervals for the GEV parameters.

Notes

This function generates a plot showing the Maximum Likelihood estimators for the Generalized Extreme Value (GEV) parameters (gamma, mu, sigma) computed from block maxima. The user can choose to display confidence intervals (CIs) for each parameter

The plot is saved as a PNG image if the *save* parameter is set to True.

Parameters

param param
: str, optional GEV parameter to plot (default is 'gamma').

param show_CI
: bool, optional Flag indicating whether to display confidence intervals (default is True).

param show
: bool, optional Flag indicating whether to display the plot.

param save
: bool, optional Flag indicating whether to save the plot as a PNG image (default is False).

param filename
: str, optional Name of the PNG file to save the plot (default is None).

Example

```
>>> estimator = ML_estimators(timeseries_data)
>>> estimator.get_ML_estimation()
>>> estimator.get_CIs(alpha=0.05, method='minimal_width')
>>> estimator.plot(show_CI=True, show_true=True, save=True,␣
↪filename='PWM_estimation.png')
```

Returns

None
: The plot is displayed in the console and saved as a PNG image if the *save* parameter is set to True.

class xtremes.topt.**Frechet_ML_estimators**(*TimeSeries*)

Bases: `object`

Maximum Likelihood Estimators (MLE) for Frechet parameters.

This class calculates Maximum Likelihood Estimators (MLE) for the parameters of the 2-parameter Frechet distribution using the method of maximum likelihood estimation on a series of high order statistics.

5.9.5 Parameters

TimeSeries
[TimeSeries] The TimeSeries object containing the data (high order statistics) for which MLE estimators will be calculated.

5.9.6 Attributes

values
[numpy.ndarray] An array containing the MLE estimators (alpha, sigma) for each set of high order statistics.

statistics
[dict] A dictionary containing computed statistics from the MLE estimators such as mean, variance, bias, and MSE.

5.9.7 Methods

__len__()
Returns the number of MLE estimators calculated.

get_ML_estimation(PWM_estimators=None, initParams='auto', r=None)
Computes the MLE estimators for the Frechet parameters (alpha, sigma).

get_statistics(alpha_true)
Computes statistics (mean, variance, bias, and MSE) of the MLE estimators using a true alpha value.

5.9.8 Examples

```
>>> import numpy as np
>>> from hos import TimeSeries, Frechet_ML_estimators, PWM_estimators
>>> blockmaxima_data = np.random.normal(loc=10, scale=2, size=100)
>>> high_order_stats_data = np.random.normal(loc=5, scale=1,␣
↪size=(100, 3))
>>> ts = TimeSeries(blockmaxima=blockmaxima_data, high_order_
↪stats=high_order_stats_data, corr='IID', ts=0.5)
>>> pwm = PWM_estimators(ts)
>>> pwm.get_PWM_estimation()
>>> ml = Frechet_ML_estimators(ts)
>>> ml.get_ML_estimation(PWM_estimators=pwm)
>>> ml.get_statistics(alpha_true=0.1)
>>> print("ML Estimators:")
>>> print(ml.values)
>>> print("\nStatistics:")
>>> print(ml.statistics)
```

get_CIs (*alpha=0.05*, *method='symmetric'*)

Compute confidence intervals (CIs) for the Frechet parameters using different methods.

Notes

This function calculates confidence intervals (CIs) for the Frechet parameters (alpha and sigma) estimated from Maximum Likelihood Estimators (MLE). The user can choose between two methods for computing the confidence intervals:

- **'symmetric'**: This method uses the quantiles of the distribution of parameter estimates to compute symmetric confidence intervals.
- **'minimal_width'**: This method finds the interval with minimal width that contains the desired proportion (1 - alpha) of the sorted parameter estimates.

For each block maxima series, confidence intervals for the shape (alpha) and scale (sigma) parameters are calculated. The results are stored in the *self.statistics* dictionary, with keys 'gamma_CI', 'mu_CI', and 'sigma_CI' corresponding to the computed confidence intervals.

Do not get confused with alpha being the significance level as well as the shape parameter of the Frechet distribution.

Parameters

param alpha
float, optional Significance level for the confidence intervals (default is 0.05, for a 95% CI).

param method
str, optional Method for computing the confidence intervals. Options are: - 'symmetric': Uses quantiles to compute symmetric CIs (default). - 'minimal_width': Computes the minimal width interval containing (1 - alpha) of the estimates.

Example

```
>>> estimator = ML_stimator(timeseries_data)
>>> estimator.get_PWM_estimation()
>>> estimator.get_CIs(alpha=0.05, method='minimal_width')
>>> print(estimator.statistics)
```

Returns

None
The results are stored in *self.statistics*, which contains the confidence intervals for each GEV parameter.

get_ML_estimation(*PWM_estimators=None*, *initParams='auto'*, *r=None*)

Compute ML estimators (alpha, sigma) for each high order statistics series using Frechet distribution.

Parameters

PWM_estimators
[PWM_estimators object, optional] PWM_estimators object containing PWM estimators for initializing parameters. Required if *initParams* is set to 'auto'.

initParams
[str or numpy.ndarray, optional] Initial parameters for ML estimation. 'auto' to calculate automatically using PWM estimators. If a numpy array is provided, these will be used as initial parameter values. Default is 'auto'.

r
[int, optional] Number of order statistics to calculate the log-likelihood on. If not specified, all provided order statistics will be used.

Returns

None
Updates the *self.values* attribute with the estimated parameters (alpha, sigma) for each series of high order statistics.

Raises

ValueError
If *initParams* is set to 'auto' and no *PWM_estimators* are provided.

`get_statistics`(*alpha_true*)

Compute statistics (mean, variance, bias, and MSE) of the ML estimators using the true α value.

Parameters

alpha_true
[float] The true value of α to calculate bias, MSE, and other statistics.

Returns

None
Updates the *self.statistics* dictionary with calculated statistics such as mean, variance, bias, and MSE for both α and σ.

Notes

The statistics include: - Mean and variance for the estimated alpha and sigma values. - Bias and mean squared error (MSE) for the alpha estimates.

plot (*param='alpha'*, *show_CI=True*, *show_true=True*, *filename=None*)

Plot the ML estimators and confidence intervals for the GEV parameters.

Notes

This function generates a plot showing the Maximum Likelihood estimators for the Generalized Extreme Value (GEV) parameters (gamma, mu, sigma) computed from block maxima. The user can choose to display confidence intervals (CIs) for each parameter

The plot is saved as a PNG image if the *save* parameter is set to True.

Parameters

param param
: str, optional GEV parameter to plot (default is 'gamma').

param show_CI
: bool, optional Flag indicating whether to display confidence intervals (default is True).

param show
: bool, optional Flag indicating whether to display the plot.

param save
: bool, optional Flag indicating whether to save the plot as a PNG image (default is False).

param filename
: str, optional Name of the PNG file to save the plot (default is None).

Example

```
>>> estimator = PWM_estimators(timeseries_data)
>>> estimator.get_PWM_estimation()
>>> estimator.get_CIs(alpha=0.05, method='minimal_width')
>>> estimator.plot(show_CI=True, show_true=True, save=True,␣
↪filename='PWM_estimation.png')
```

Returns

None
The plot is displayed in the console and saved as a PNG image if the *save* parameter is set to True.

`xtremes.topt.`**`log_likelihood`**(*high_order_statistics*, *gamma=0*, *mu=0*, *sigma=1*, *r=None*)
Calculate the GEV log likelihood based on the two highest order statistics in three different ways.

5.9.9 Parameters

high_order_statistics
[numpy.ndarray] A 2D array where each row contains the two highest order statistics for each observation.

gamma
[float, optional] The shape parameter (γ) for the Generalized Extreme Value (GEV) distribution. Default is 0.

mu
[float, optional] The location parameter (μ) for the GEV distribution. Default is 0.

sigma
[float, optional] The scale parameter (σ) for the GEV distribution. Must be positive. Default is 1.

r
[int, optional] The number of order statistics to calculate the log-likelihood on. If not specified, it uses all provided statistics.

5.9.10 Returns

float
The calculated log likelihood.

5.9.11 Notes

- This function computes the log likelihood using the two highest order statistics and supports both the classical Gumbel case ($\gamma = 0$) and the generalized case ($\gamma \neq 0$).
- The *high_order_statistics* array should be structured with the two highest order statistics per observation as rows.
- The shape parameter γ controls the tail behavior of the distribution. When $\gamma = 0$, the distribution becomes the Gumbel type.
- The *r* parameter controls how many order statistics are used for the likelihood calculation, typically *r=2* for two order statistics.

5.9.12 Example

```
>>> hos = np.array([[0.1, 0.2], [0.3, 0.4], [0.2, 0.5], [0.4, 0.6]])
>>> log_likelihood(hos, gamma=0.5, mu=0, sigma=2, r=2)
-7.494890426732856
```

xtremes.topt.**Frechet_log_likelihood**(*high_order_statistics*, *alpha=1*, *sigma=1*, *r=None*)

Calculate the 2-parameter Frechet log likelihood based on the highest order statistics. The calculation can be done using either the joint likelihood of the top two order statistics or the product of their marginals.

5.9.13 Parameters

high_order_statistics
[numpy.ndarray] A 2D array where each row contains the two highest order statistics for each observation.

alpha
[float, optional] The shape parameter (α) for the Frechet distribution. Default is 1. Controls the tail behavior of the distribution.

sigma
[float, optional] The scale parameter (σ) for the Frechet distribution. Must be positive. Default is 1.

r
[int, optional] The number of order statistics to calculate the log-likelihood on. If not specified, it uses all provided statistics.

5.9.14 Returns

float
The calculated log likelihood for the given data under the Frechet distribution.

5.9.15 Notes

- This function computes the log likelihood using the two highest order statistics from each observation, with a focus on the Frechet distribution.
- The shape parameter *alpha* determines the heaviness of the tail in the distribution, and the scale parameter *sigma* must be strictly positive.
- The *high_order_statistics* array should be structured such that each row represents the two highest order statistics for an observation.
- The function can either calculate the joint likelihood of the top two order statistics or consider the product of their marginals, depending on the values used.

5.9.16 Example

```
>>> hos = np.array([[0.5, 1.0], [1.5, 2.0], [1.2, 2.2], [2.0, 3.0]])
>>> Frechet_log_likelihood(hos, alpha=2, sigma=1.5, r=2)
-15.78467219003245
```

5.10 Running Extensive Simulations

The *xtremes.topt* module also provides functions for running extensive simulations and performing multiple MLEs.

`xtremes.topt.`**`run_ML_estimation`**(*file*, *corr='IID'*, *gamma_true=0*, *block_sizes=[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]*, *stride='SBM'*, *option=1*, *estimate_pi=False*)

Run maximum likelihood estimation (ML) for a given time series file.

5.10.1 Notes

This function reads a time series from a file and performs ML estimation for the specified correlation type, GEV shape parameter, block sizes, and stride type. It iterates over each block size, extracts block maxima, computes high order statistics, and performs ML estimation. The results are stored in a dictionary with the GEV shape parameter as the key and ML estimation results for each block size as the value.

5.10.2 Parameters

param file
str Path to the file containing the time series data.

param corr
str, optional Correlation type for the model (default is 'IID').

param gamma_true
float, optional True value of the GEV shape parameter (default is 0).

param block_sizes
list, optional List of block sizes for extracting block maxima (default is [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]).

param stride
str, optional Stride type for block maxima extraction. Options are 'SBM' (sliding block maxima) or 'DBM' (default block maxima) (default is 'SBM').

param r
int, optional Number of orderstatistics to calculate the log-likelihood on. If not specified, use all provided

param estimate_pi
bool, optional Flag indicating whether to estimate the pi parameter (default is False).

5.10.3 Example

```
>>> result = run_ML_estimation("timeseries_data.pkl", corr='ARMAX',␣
↪gamma_true=0.5, block_sizes=[10, 20, 30], stride='DBM', option=2,␣
↪estimate_pi=True)
>>> print(result)
{0.5: {10: HighOrderStats_object, 20: HighOrderStats_object, 30:␣
↪HighOrderStats_object}}
```

5.10.4 Returns

return
dict Dictionary containing ML estimation results for each block size.

`xtremes.topt.`**`run_multiple_ML_estimations`**(*file*, *corr='IID'*, *gamma_trues=array([-0.4, -0.3, -0.2, -0.1, 0., 0.1, 0.2, 0.3, 0.4])*, *block_sizes=[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]*, *stride='SBM'*, *option=1*, *estimate_pi=False*, *parallelize=False*)

Run multiple maximum likelihood estimations (ML) for a range of GEV shape parameter values.

5.10.5 Notes

This function performs ML estimation for a range of GEV shape parameter values specified in 'gamma_trues' and aggregates the results into a single dictionary. It iterates over each gamma value, calls the 'run_ML_estimation' function, and collects the results. If 'parallelize' is set to True, it runs the estimations concurrently using asyncio.

5.10.6 Parameters

param file
str Path to the file containing the time series data.

param corr
str, optional Correlation type for the model (default is 'IID').

param gamma_trues
numpy.ndarray, optional Array of GEV shape parameter values to perform ML estimation for (default is np.arange(-4, 5, 1)/10).

param block_sizes
list, optional List of block sizes for extracting block maxima (default is [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]).

param stride
str, optional Stride type for block maxima extraction. Options are 'SBM' (sliding block maxima) or 'DBM' (default block maxima) (default is 'SBM').

param r
int, optional Number of orderstatistics to calculate the log-likelihood on. If not specified, use all provided

param estimate_pi
bool, optional Flag indicating whether to estimate the pi parameter (default is False).

param parallelize
bool, optional Flag indicating whether to parallelize the ML estimations using asyncio (default is False).

5.10.7 Returns

return
dict Dictionary containing ML estimation results for each gamma value and block size.

5.11 Examples

1. **Run ML Estimation**:

```
from xtremes.topt import run_ML_estimation

result = run_ML_estimation("timeseries_data.pkl", corr='ARMAX
↪', gamma_true=0.5, block_sizes=[10, 20, 30], stride='DBM',␣
↪option=2, estimate_pi=True)
print(result)
```

2. **Run Multiple ML Estimations**:

```
from xtremes.topt import run_multiple_ML_estimations
import numpy as np

result = run_multiple_ML_estimations("timeseries_data.pkl",␣
↪corr='IID', gamma_trues=np.arange(-4, 5, 1)/10, block_
↪sizes=[10, 20, 30], stride='SBM', option=1, estimate_
↪pi=False, parallelize=True)
print(result)
```

CHAPTER

SIX

REFERENCE: BOOTSTRAP

This module computes a Bootstrap procedure on disjoint or sliding block maxima.

6.1 Overview

The *xtremes.bootstrap* module provides tools for performing bootstrap procedures on block maxima. It includes classes and functions for extracting block maxima, resampling, and estimating parameters using Maximum Likelihood Estimation (MLE).

6.2 Classes

6.3 The `FullBootstrap` Class

class xtremes.bootstrap.**FullBootstrap**(*initial_sample*, *bs=10*, *stride='DBM'*, *dist_type='Frechet'*)

Bases: `object`

A class to perform bootstrapping of Maximum Likelihood Estimates (MLE) for Fréchet or GEV distributions.

This class performs block maxima extraction from an initial sample using either disjoint or sliding blocks. It applies a bootstrap resampling procedure to estimate the variability of the MLE parameters for the specified distribution type (Fréchet or GEV). The bootstrap method is parallelized for efficiency and supports reproducibility through optional seed setting.

6.3.1 Parameters

initial_sample
: [list or numpy.ndarray] The initial dataset from which block maxima will be extracted and bootstrapped.

bs
: [int, optional] Block size for the block maxima extraction. Default is 10.

stride
: [{'DBM', 'SBM'}, optional] Stride type for block maxima extraction: - 'DBM' (Disjoint Block Maxima): Non-overlapping blocks. - 'SBM' (Sliding Block Maxima): Overlapping blocks. Default is 'DBM'.

dist_type
: [{'Frechet', 'GEV'}, optional] Distribution type to estimate the parameters for: - 'Frechet': Estimate parameters for the 2-parametric Fréchet distribution. - 'GEV': Estimate parameters for the 3-parametric Generalized Extreme Value (GEV) distribution. Default is 'Frechet'.

6.3.2 Attributes

circmaxs
: [list] The block maxima extracted from the initial sample using the specified block size and stride.

data
: [hos.Data] The *hos.Data* object containing the original dataset and its MLE results.

MLEvals
: [numpy.ndarray] The MLE estimates from the original dataset before bootstrapping.

values
: [numpy.ndarray] MLE estimates for each bootstrap sample after running the *run_bootstrap* method.

statistics
: [dict] Dictionary containing summary statistics (mean and standard deviation) of the bootstrap estimates.

6.3.3 Methods

run_bootstrap(num_bootstraps=100, set_seeds=False, max_workers=1)
: Runs the bootstrap procedure in parallel and calculates the MLE estimates for each bootstrap sample.

6.3.4 Example

```
>>> sample = np.random.rand(100)
>>> bootstrap = FullBootstrap(sample, bs=10, stride='DBM', dist_type=
↪'Frechet')
>>> bootstrap.run_bootstrap(num_bootstraps=100, set_seeds=True, max_
↪workers=4)
>>> bootstrap.statistics['mean']  # Mean of bootstrap estimates
>>> bootstrap.statistics['std']   # Standard deviation of bootstrap␣
↪estimates
```

`get_CI` (*alpha=0.05*, *method='bootstrap'*)
: Compute the confidence interval (CI) for the Maximum Likelihood Estimate (MLE) parameters based on bootstrap samples.

Parameters

alpha
: [float, optional] Significance level for the confidence interval. Default is 0.05, corresponding to a 95% confidence interval.

method
: [str, optional] Method to compute the confidence interval. Two options are available: - 'symmetric': The confidence interval is computed using the symmetric quantiles. - 'minimal_width': The confidence interval is computed by finding the minimal-width interval that contains (1 - alpha) proportion of the bootstrap distribution. The default is 'symmetric'.

Returns

numpy.ndarray
: A 2D array with shape (n_parameters, 2) containing the lower and upper bounds of the confidence interval for each parameter. The first column represents the lower bounds, and the second column represents the upper bounds.

Notes

The confidence intervals are based on bootstrap estimates of the MLE parameters, which means the confidence intervals are derived from the empirical distribution of the parameter estimates obtained from multiple bootstrap samples.

There are two methods available for calculating the confidence intervals: - 'symmetric': This method takes the alpha/2 and (1 - alpha/2) quantiles of the bootstrap distribution for each parameter. It is based on the assumption that the distribution is approximately symmetric and works well when the bootstrap distribution is roughly normal. - 'minimal_width': This method identifies the interval with the minimal width that contains (1 - alpha) proportion of the bootstrap samples. It is particularly useful when the bootstrap distribution is skewed or not symmetric.

plot_bootstrap(*param_idx=0*, *param_name='gamma'*, *bins=30*, *output_file=None*, *show=True*)

Plot the bootstrap distribution for a specified parameter.

Parameters

param_idx
: [int, optional] Index of the parameter to plot (0 for the first parameter, 1 for the second, etc.). Default is 0.

bins
: [int, optional] Number of bins to use for the histogram. Default is 30.

Notes

This method generates a histogram of the bootstrap estimates for the specified parameter and overlays the mean and confidence interval.

run_bootstrap(*num_bootstraps=100*, *set_seeds=False*, *max_workers=1*)

Run the bootstrap resampling procedure in parallel.

This method resamples the block maxima dataset, estimates the MLE parameters for each bootstrap sample, and computes summary statistics (mean and standard deviation) of the bootstrap estimates. The computation is parallelized using *ProcessPoolExecutor* with an adjustable number of worker processes.

Parameters

num_bootstraps
[int, optional] Number of bootstrap samples to generate. Default is 100.

set_seeds
[bool, optional] If True, sets the random seed for reproducibility in each bootstrap iteration. Default is False.

max_workers
[int, optional] Maximum number of worker processes to use for parallelization. Default is 1 (no parallelism). Set to *None* to use all available CPU cores.

Returns

None
Results are stored in the *values* attribute and summary statistics in the *statistics* attribute.

Example

```
>>> bootstrap.run_bootstrap(num_bootstraps=500, set_seeds=True,␣
↪max_workers=4)
>>> bootstrap.statistics['mean']  # Access the mean of bootstrap␣
↪estimates
>>> bootstrap.statistics['std']   # Access the standard deviation␣
↪of bootstrap estimates
```

6.4 Functions

6.5 The `circmax` Function

xtremes.bootstrap.**circmax**(*sample*, *bs=10*, *stride='DBM'*)

Extract the block maxima (BM) from a given sample using different stride methods.

6.5.1 Parameters

sample
[numpy.ndarray] A 1D array containing the sample from which block maxima will be extracted.

bs
[int, optional] The block size (number of observations per block) used to divide the sample for block maxima extraction. Default is 10.

stride
[{'DBM', 'SBM'}, optional] The stride method used for extracting block maxima: - 'DBM' (Disjoint Block Maxima): Extracts maxima from non-overlapping blocks. - 'SBM' (Sliding Block Maxima): Extracts maxima using overlapping blocks. Default is 'DBM'.

6.5.2 Returns

numpy.ndarray
A 1D or 2D array containing the block maxima extracted from the sample. The result depends on the stride method used: - For 'DBM', returns a 1D array of block maxima. - For 'SBM', returns a 2D array where each row contains the block maxima extracted from overlapping blocks.

6.5.3 Raises

ValueError
If an invalid stride method is specified.

6.5.4 Notes

- 'DBM' (Disjoint Block Maxima) extracts block maxima from non-overlapping blocks of size *bs*.
- 'SBM' (Sliding Block Maxima) creates overlapping blocks, effectively increasing the number of block maxima compared to 'DBM'.
- In the 'SBM' setting, the circmax() method introduced by Bücher and Staud 2024 is used.

6.5.5 References

Bücher, A., & Staud, T. (2024). Bootstrapping Estimators based on the Block Maxima Method. arXiv preprint arXiv:2409.05529.

6.5.6 Example

```
>>> sample = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> circmax(sample, bs=5, stride='DBM')
array([5, 10])
```

```
>>> circmax(sample, bs=3, stride='SBM')
array([[3, 6, 9],
       [4, 7, 10]])
```

6.6 The `uniquening` Function

`xtremes.bootstrap.`**`uniquening`**(*circmaxs*, *stride='DBM'*)

Identify unique values and their counts from a list of arrays.

6.6.1 Parameters

circmaxs

[numpy.ndarray] A NumPy array containing block maxima values extracted from a sample.

6.6.2 Returns

list of tuples

A list where each element is a tuple containing two NumPy arrays: - The first array contains the unique values from the corresponding row in *circmaxs*. - The second array contains the counts of each unique value.

6.7 The `Bootstrap` Function

`xtremes.bootstrap.`**`Bootstrap`**(*xx*)

Generate a bootstrap sample by resampling with replacement from the input data.

6.7.1 Parameters

xx

[list or numpy.ndarray] The input sample to resample from.

6.7.2 Returns

list

A new sample of the same size, created by randomly selecting elements from *xx* with replacement.

6.7.3 Notes

This function creates a bootstrap sample, which is commonly used in statistical resampling methods to estimate the variability of a statistic.

6.7.4 Example

```
>>> sample = [1, 2, 3, 4, 5]
>>> Bootstrap(sample)
[2, 5, 3, 1, 2]  # Example output, actual result may vary
```

6.8 The `aggregate_boot` Function

`xtremes.bootstrap.`**`aggregate_boot`**(*boot_samp*, *stride='DBM'*)

Aggregate counts of unique values from a list of tuples containing values and their counts.

6.8.1 Parameters

boot_samp

[list of tuples] Each tuple contains two arrays: the first with values and the second with corresponding counts.

6.8.2 Returns

numpy.ndarray

A 2D array with two columns: the first column contains unique values, and the second column contains the aggregated counts.

6.8.3 Example

```
>>> boot_samp = [(np.array([1, 2, 3]), np.array([1, 1, 2])), (np.
↪array([2, 3]), np.array([2, 1]))]
>>> aggregate_boot(boot_samp)
array([[1, 1],
       [2, 3],
       [3, 3]])
```

6.9 The `bootstrap_worker` Function

`xtremes.bootstrap.`**`bootstrap_worker`**`(`*args*`)`

Auxiliary function to perform a single bootstrap resampling and MLE estimation.

This function is designed to be used in parallelized bootstrap procedures. It takes arguments for a single bootstrap iteration, performs resampling on the given block maxima, estimates MLE parameters using the specified distribution type, and returns the results.

6.9.1 Parameters

args

[tuple] A tuple containing the following elements: - idx (int): The iteration index, used for setting the random seed if *set_seeds* is True. - set_seeds (bool): Whether to set the random seed for reproducibility. - circmaxs (list or numpy.ndarray): The block maxima dataset to be resampled. - aggregate_boot (callable): A function to aggregate the resampled data. - ML_estimators_data (callable): A function or class to compute MLE parameters on the aggregated data. - dist_type (str): The distribution type for MLE estimation ('Frechet' or 'GEV').

6.9.2 Returns

numpy.ndarray

The MLE parameter estimates for the current bootstrap sample.

6.9.3 Notes

- This function is designed to be compatible with *ProcessPoolExecutor* or other parallel processing tools.
- The random seed is set per iteration to ensure reproducibility when *set_seeds* is True.

6.9.4 Example

```
>>> args = (0, True, circmaxs, aggregate_boot, ML_estimators_data,
↪'GEV')
>>> bootstrap_worker(args)
array([param1, param2, param3])  # Example output for GEV distribution
```

6.10 Examples

Here are some examples of how to use the *xtremes.bootstrap* module:

1. **FullBootstrap Class**:

```
import numpy as np
from xtremes.bootstrap import FullBootstrap

sample = np.random.rand(100)
bootstrap = FullBootstrap(sample, bs=10, stride='DBM', dist_
↪type='Frechet')
bootstrap.run_bootstrap(num_bootstraps=100)
print("Mean of bootstrap estimates:", bootstrap.statistics[
↪'mean'])
print("Standard deviation of bootstrap estimates:", bootstrap.
↪statistics['std'])
```

2. **circmax Function**:

```
import numpy as np
from xtremes.bootstrap import circmax

sample = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
block_maxima = circmax(sample, bs=5, stride='DBM')
print("Block Maxima (DBM):", block_maxima)

block_maxima = circmax(sample, bs=3, stride='SBM')
print("Block Maxima (SBM):", block_maxima)
```

3. **uniquening Function**:

```
import numpy as np
from xtremes.bootstrap import uniquening

circmaxs = np.array([[1, 2, 2, 3], [2, 3, 3, 4]])
unique_values = uniquening(circmaxs)
print("Unique values and counts:", unique_values)
```

4. **Bootstrap Function**:

```
from xtremes.bootstrap import Bootstrap

sample = [1, 2, 3, 4, 5]
bootstrap_sample = Bootstrap(sample)
print("Bootstrap sample:", bootstrap_sample)
```

5. **aggregate_boot Function**:

```
import numpy as np
from xtremes.bootstrap import aggregate_boot

boot_samp = [(np.array([1, 2, 3]), np.array([1, 1, 2])), (np.
↪array([2, 3]), np.array([2, 1]))]
aggregated_counts = aggregate_boot(boot_samp)
print("Aggregated counts:", aggregated_counts)
```

6.11 References

- Bücher, A., & Staud, T. (2024). Bootstrapping Estimators based on the Block Maxima Method. arXiv preprint arXiv:2409.05529.

CHAPTER

SEVEN

REFERENCE: MISCELLANEOUS

This module is a collection for non-specialized, frequently used or basic functions.

7.1 Overview

The *xtremes.miscellaneous* module provides a variety of utility functions that can be used across different parts of your project. These functions are designed to be general-purpose and can help simplify common tasks.

7.2 Basic Functions

The following functions are basic utility functions provided by the *xtremes.miscellaneous* module:

`xtremes.miscellaneous.sigmoid(x)`

Compute the sigmoid function for the given input.

7.2.1 Parameters

param x
array_like The input value or array.

7.2.2 Returns

return
numpy.ndarray The sigmoid of the input array. It has the same shape as *x*.

7.2.3 Notes

- The sigmoid function is defined as:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

- It maps any real-valued number to the range (0, 1).

7.2.4 Example

```
>>> import numpy as np
>>> x = np.array([-2, -1, 0, 1, 2])
>>> sigmoid(x)
array([0.11920292, 0.26894142, 0.5       , 0.73105858, 0.88079708])
```

`xtremes.miscellaneous.`**`invsigmoid`**`(`*y*`)`

Compute the inverse sigmoid function for the given input.

7.2.5 Parameters

param y
float or array_like The input value or array, representing probabilities in the range [0, 1].

7.2.6 Returns

return
float or numpy.ndarray The inverse sigmoid of the input value or array.

7.2.7 Raises

ValueError
If the input value is outside the range [0, 1].

7.2.8 Notes

- The inverse sigmoid function finds the input value(s) that would produce the given output probability.
- It is the inverse of the sigmoid function: *invsigmoid(sigmoid(x))* = *x*.

7.2.9 Example

```
>>> import numpy as np
>>> y = np.array([0.1, 0.5, 0.9])
>>> invsigmoid(y)
array([-1.3652469 ,  0.        ,  1.3652469 ])
```

`xtremes.miscellaneous.`**`mse`**`(`*gammas*, *gamma_true*`)`

Compute Mean Squared Error, Variance, and Bias of estimators.

7.2.10 Notes

Computes the Mean Squared Error (MSE), Variance, and Bias of a set of estimators given the true (theoretical) value. This function is intended for estimating the GEV shape parameter γ, but works for other estimators as well.

$$\mathrm{MSE}(\hat{\gamma}) := \frac{1}{n-1}\sum_{i=1}^{n}(\hat{\gamma}_i - \gamma)^2$$

$$\mathrm{Var}(\hat{\gamma}) := \frac{1}{n-1}\sum_{i=1}^{n}(\hat{\gamma}_i - \overline{\gamma})^2$$

$$\mathrm{Bias}(\hat{\gamma}) := \frac{n}{n-1}(\gamma - \overline{\gamma})^2$$

Here, $\overline{\gamma}$ denotes the mean.

$$\overline{\gamma} := \frac{1}{n}\sum_{i=1}^{n}\gamma_i$$

Also note that:

$$\mathrm{MSE} := \mathrm{Bias} + \mathrm{Var}$$

7.2.11 Parameters

param gammas
array_like Estimated values.

param gamma_true
int or float True (theoretical) parameter.

7.2.12 Returns

return
tuple[float] MSE, variance, and bias.

7.2.13 Raises

raise test_xtremes.miscellaneous.warning
If `len(gammas)==1`. `NaN`s are returned.

7.3 Examples

1. **Sigmoid Function**:

```
import numpy as np
from xtremes.miscellaneous import sigmoid

x = np.array([-2, -1, 0, 1, 2])
result = sigmoid(x)
print("Sigmoid Result:", result)
```

2. **Inverse Sigmoid Function**:

```
import numpy as np
from xtremes.miscellaneous import invsigmoid

y = np.array([0.1, 0.5, 0.9])
result = invsigmoid(y)
print("Inverse Sigmoid Result:", result)
```

3. **Mean Squared Error (MSE)**:

```
from xtremes.miscellaneous import mse

gammas = [0.1, 0.2, 0.3]
gamma_true = 0.2
mse_value, variance, bias = mse(gammas, gamma_true)
print("MSE:", mse_value, "Variance:", variance, "Bias:", bias)
```

7.4 The GEV and its Likelihood

The following functions are related to the Generalized Extreme Value (GEV) distribution and its likelihood:

`xtremes.miscellaneous.`**`GEV_pdf`**`(`*x*, *gamma=0*, *mu=0*, *sigma=1*`)`

Compute the Probability Density Function (PDF) of the Generalized Extreme Value distribution.

7.4.1 Notes

Computes the probability density function of the Generalized Extreme Value distribution:

$$g(x) = \frac{\exp\left(-\left(1+\gamma\frac{x-\mu}{\sigma}\right)^{-1/\gamma}\right) \cdot \left(1+\gamma\frac{x-\mu}{\sigma}\right)^{-1-1/\gamma}}{\sigma}$$

7.4.2 Parameters

param x
int, float, list or numpy.ndarray GEV argument $x \in \mathbb{R}$.

param gamma
int, float, list or numpy.ndarray, optional GEV shape parameter $\gamma \in \mathbb{R}$. Default is 0.

param mu
int, float, list or numpy.ndarray, optional GEV location parameter $\mu \in \mathbb{R}$. Default is 0.

param sigma
int, float, list or numpy.ndarray, optional GEV scale parameter $\sigma > 0$. Default is 1.

7.4.3 Returns

return
numpy.ndarray or float The Probability Density Function values corresponding to the input *x*.

7.4.4 Example

```
>>> import numpy as np
>>> x = np.array([1, 2, 3, 4, 5])
>>> GEV_pdf(x, gamma=0.5, mu=2, sigma=1)
array([0.17603266, 0.23254416, 0.28556358, 0.33477888, 0.37960368])
```

`xtremes.miscellaneous.`**`GEV_cdf`**(*x*, *gamma=0*, *mu=0*, *sigma=1*, *theta=1*)

Compute the Cumulative Density Function (CDF) of the Generalized Extreme Value distribution.

7.4.5 Notes

Computes the cumulative density function of the Generalized Extreme Value distribution:

$$G_{\gamma,\mu,\sigma}(x) = \exp\left(-\left(1+\gamma\frac{x-\mu}{\sigma}\right)^{-1/\gamma}\right).$$

For $\gamma = 0$, the term can be interpreted as the limit $\lim_{\gamma\to 0}$:

$$G_{0,\mu,\sigma}(x) = \exp\left(-\exp\left(-\frac{x-\mu}{\sigma}\right)\right).$$

This function also allows the usage of an extremal index, another parameter relevant when dealing with stationary time series and its extreme values:

$$G_{\gamma,\mu,\sigma,\vartheta}(x) = \exp\left(-\vartheta\left(1+\gamma\frac{x-\mu}{\sigma}\right)^{-1/\gamma}\right).$$

7.4.6 Parameters

param x
int, float, list or numpy.ndarray GEV argument $x \in \mathbb{R}$.

param gamma
int, float, list or numpy.ndarray, optional GEV shape parameter $\gamma \in \mathbb{R}$. Default is 0.

param mu
int, float, list or numpy.ndarray, optional GEV location parameter $\mu \in \mathbb{R}$. Default is 0.

param sigma
int, float, list or numpy.ndarray, optional GEV scale parameter $\sigma > 0$. Default is 1.

param theta
int, float, list or numpy.ndarray, optional Extremal index $\vartheta \in [0, 1]$. Default is 1.

7.4.7 Returns

return
numpy.ndarray or float The Cumulative Density Function values corresponding to the input *x*.

7.4.8 Example

```
>>> import numpy as np
>>> x = np.array([1, 2, 3, 4, 5])
>>> GEV_cdf(x, gamma=0.5, mu=2, sigma=1, theta=0.8)
array([0.54610814, 0.62171922, 0.69703039, 0.77196099, 0.84644106])
```

xtremes.miscellaneous.**GEV_ll**(*x*, *gamma=0*, *mu=0*, *sigma=1*)
Compute the log-likelihood function of the Generalized Extreme Value distribution.

7.4.9 Notes

Computes the log-likelihood function of the Generalized Extreme Value distribution:

$$l(x) = -\left(1 + \gamma\frac{x-\mu}{\sigma}\right)^{-1/\gamma} - \frac{\gamma+1}{\gamma}\log\left(1 + \gamma\frac{x-\mu}{\sigma}\right) - \log\sigma$$

7.4.10 Parameters

param x
int, float, list or numpy.ndarray GEV argument $x \in \mathbb{R}$.

param gamma
int, float, list or numpy.ndarray, optional GEV shape parameter $\gamma \in \mathbb{R}$.

param mu
int, float, list or numpy.ndarray, optional GEV location parameter $\mu \in \mathbb{R}$.

param sigma
int, float, list or numpy.ndarray, optional GEV scale parameter $\sigma > 0$.

7.4.11 Returns

return
numpy.ndarray or float The log-likelihood values corresponding to the input *x*.

7.4.12 Example

```
>>> import numpy as np
>>> x = np.array([1, 2, 3, 4, 5])
>>> GEV_ll(x, gamma=0.5, mu=2, sigma=1)
array([-3.11578562, -2.23851549, -1.85551157, -1.57084383, -1.
↪33403504])
```

7.5 Examples

1. **GEV CDF**:

```
import numpy as np
from xtremes.miscellaneous import GEV_cdf

x = np.array([1, 2, 3, 4, 5])
result = GEV_cdf(x, gamma=0.5, mu=2, sigma=1, theta=0.8)
print("GEV CDF Result:", result)
```

2. **GEV PDF**:

```
import numpy as np
from xtremes.miscellaneous import GEV_pdf

x = np.array([1, 2, 3, 4, 5])
result = GEV_pdf(x, gamma=0.5, mu=2, sigma=1)
print("GEV PDF Result:", result)
```

3. **GEV Log-Likelihood**:

```
import numpy as np
from xtremes.miscellaneous import GEV_ll

x = np.array([1, 2, 3, 4, 5])
result = GEV_ll(x, gamma=0.5, mu=2, sigma=1)
print("GEV Log-Likelihood Result:", result)
```

7.6 Piece Wise Moment Estimation

The following functions are related to Probability Weighted Moment (PWM) estimation:

xtremes.miscellaneous.**PWM_estimation**(*maxima*)

PWM Estimation of GEV parameters.

7.6.1 Notes

Computes the first three Probability Weighted Moments $\beta_0, \beta_1, \beta_2$ on given block maxima, as introduced in Greenwood et al. (1979).

Let $M_{(1)} \leq M_{(2)} \leq \cdots \leq M_{(n)}$ be increasingly sorted block maxima. Then the first three PWMs are defined as:

$$\beta_0 := \frac{1}{n}\sum_{i=1}^{n} M_{(i)},$$

$$\beta_1 := \frac{1}{n(n-1)}\sum_{i=1}^{n}(i-1)M_{(i)},$$

$$\beta_2 := \frac{1}{n(n-1)(n-2)}\sum_{i=1}^{n}(i-1)(i-2)M_{(i)}.$$

7.6.2 Parameters

param maxima
list or numpy.array Sequence of maxima. *len(maxima)* must be greater than or equal to 3.

7.6.3 Returns

return
tuple of floats The first three PWMs: β_0, β_1, and β_2.

7.6.4 References

Greenwood, J. A., Landwehr, J. M., Matalas, N. C., & Wallis, J. R. (1979). Probability weighted moments: Definition and relation to parameters of several distributions expressible in inverse form. Water Resources Research, 15(5), 1049–1054. https://doi.org/10.1029/wr015i005p01049

7.6.5 Example

```
>>> import numpy as np
>>> maxima = np.array([5, 8, 12, 15, 18])
>>> PWM_estimation(maxima)
(11.6, 11.2, 39.2)
```

`xtremes.miscellaneous.`**`PWM2GEV`**`(`*b_0*, *b_1*, *b_2*`)`

Compute estimators for the parameters of the GEV distribution from given PWM estimators.

7.6.6 Notes

Computes estimators for the parameters of the Generalized Extreme Value (GEV) distribution from given Probability Weighted Moment (PWM) estimators. As shown in Hosking et al. (1985), they follow the relationship:

$$\gamma = g_1^{-1}\left(\frac{3\beta_2 - \beta_0}{2\beta_1 - \beta_0}\right),$$
$$\sigma = g_2(\gamma) \cdot (2\beta_1 - \beta_0),$$
$$\mu = \beta_0 + \sigma \cdot g_3(\gamma),$$

where:

$$g_1(\gamma) := \frac{3^\gamma - 1}{2^\gamma - 1},$$
$$g_2(\gamma) := \frac{\gamma}{\Gamma(1-\gamma)(2^\gamma - 1)},$$
$$g_3(\gamma) := \frac{1 - \Gamma(1-\gamma)}{\gamma}.$$

Note that Γ denotes the gamma function. The values for $g_\bullet(0)$ are defined by continuity to result in:

$$g_1(0) = \frac{\log 3}{\log 2}, \quad g_2(0) = \frac{1}{\log 2}, \quad g_3(0) = -\gamma_{\mathrm{EM}},$$

with γ_{EM} being the Euler-Mascheroni constant.

7.6.7 Parameters

param b_0
int or float PWM estimator for the first moment.

param b_1
int or float PWM estimator for the second moment.

param b_2
int or float PWM estimator for the third moment.

7.6.8 Returns

return
tuple of floats Estimators for the GEV parameters: shape parameter (γ), location parameter (μ), and scale parameter (σ).

7.6.9 References

Hosking, J. R. M., Wallis, J. R., & Wood, E. F. (1985). Estimation of the Generalized Extreme-Value Distribution by the Method of Probability-Weighted Moments. Technometrics, 27(3), 251–261. https://doi.org/10.1080/00401706.1985.10488049

7.6.10 Example

```
>>> b_0 = 10
>>> b_1 = 20
>>> b_2 = 30
>>> PWM2GEV(b_0, b_1, b_2)
(0.289510206281886, 10.446586187753782, 15.207496500178042)
```

7.7 Examples

1. **PWM Estimation**:

```
import numpy as np
from xtremes.miscellaneous import PWM_estimation

maxima = np.array([5, 8, 12, 15, 18])
result = PWM_estimation(maxima)
print("PWM Estimation Result:", result)
```

2. **PWM to GEV**:

```
from xtremes.miscellaneous import PWM2GEV

b_0 = 10
b_1 = 20
b_2 = 30
result = PWM2GEV(b_0, b_1, b_2)
print("PWM to GEV Result:", result)
```

7.8 Simulating Time Series

The following functions are related to simulating time series data:

xtremes.miscellaneous.**simulate_timeseries**(*n*, *distr='GEV'*, *correlation='IID'*, *modelparams=[0]*, *ts=0*, *seed=None*)

Simulate a Time Series for GEV.

7.8.1 Notes

This function allows simulating three different kinds of time series.

- The most basic time series is the IID (independent and identically distributed) case, where there is no temporal dependence. The distribution from which the random variables are drawn can be chosen via *distr*, and respective model parameters are passed via *modelparams*.

•For a stationary time series with temporal dependence, two models are available:

  - ARMAX model: The next value is computed as the maximum of two values: $ts * X_{i-1}$ and $(1 - ts) * Z_i$, where Z_i is drawn from a GPD (Generalized Pareto Distribution) specified by *modelparams*. The parameter *ts* controls the temporal dependence.
  - AR (Autoregressive) model: Similar to ARMAX, but Z_i is drawn from a Cauchy distribution.

7.8.2 Parameters

param n
: int Length of time series to simulate.

param distr
: str, optional Distribution to draw from. Default is 'GEV'.

param correlation
: str, optional Correlation type to specify, choose from ['IID', 'ARMAX', 'AR']. Default is 'IID'.

param modelparams
: list, optional Parameters belonging to *distr*. Default is [0].

param ts
: float, optional Time series parameter $\alpha \in [0, 1]$. Default is 0.

param seed
: int, optional Random seed for reproducibility. Default is None.

7.8.3 Returns

return
: numpy.ndarray[float] Simulated time series.

7.8.4 Raises

ValueError
: If an invalid model is specified.

7.8.5 See Also

Tutorial: Link to the associated tutorial.

7.8.6 Example

```
>>> import numpy as np
>>> simulated_ts = simulate_timeseries(n=100, distr='GPD',␣
↪correlation='ARMAX', modelparams=[0.5], ts=0.7, seed=42)
>>> simulated_ts[:10]
array([0.5791584 , 0.71057555, 0.55079821, 0.61636037, 0.63544943,
       0.74830653, 0.66283107, 0.69929097, 0.75026675, 0.69214764])
```

`xtremes.miscellaneous.`**`stride2int`**(*stride*, *block_size*)

Integer from Stride.

7.8.7 Notes

This function is a utility when handling Block maxima (disjoint, sliding, striding). Apart from giving the stride directly, it is handy to have the additional options 'SBM' and 'DBM' for sliding and disjoint BM, respectively. This function converts exactly this.

7.8.8 Parameters

param stride
int or str Stride to be converted.

param block_size
int Block size for conversion, ignored if stride=='SBM'.

return
int The converted stride.

7.8.9 Raises

TypeError
If stride is not an integer or string.

7.8.10 Examples

```
>>> stride2int(2, 10)
2
>>> stride2int('SBM', 10)
1
>>> stride2int('DBM', 10)
10
```

`xtremes.miscellaneous.`**`modelparams2gamma_true`**(*distr*, *correllation*, *modelparams*)

Extract gamma_true from model parameters.

7.8.11 Notes

For some models, it is possible to extract the true value of gamma theoretically. Whenever this is possible, the conversion should be subject to this function.

7.8.12 Parameters

param distr
: str Valid distribution type.

param correlation
: str Valid correlation type, currently ['IID', 'ARMAX', 'AR'].

param modelparams
: list Valid model parameters.

return
: numpy.ndarray[float] Gamma_true, if applicable. Returns None if gamma_true cannot be extracted.

7.8.13 Raises

ValueError
: If the distribution or correlation type is not supported.

7.8.14 Examples

```
>>> modelparams2gamma_true('GEV', 'IID', [0.5])
0.5
>>> modelparams2gamma_true('GPD', 'ARMAX', [0.3])
0.3
>>> modelparams2gamma_true('Normal', 'IID', [0.5])
Traceback (most recent call last):
    ...
ValueError: Distribution type 'Normal' is not supported.
```

7.9 Examples

1. **Simulate Time Series**:

```
from xtremes.miscellaneous import simulate_timeseries

simulated_ts = simulate_timeseries(n=100, distr='GEV',␣
↪correlation='IID', modelparams=[0.5], ts=0.7, seed=42)
print("Simulated Time Series:", simulated_ts[:10])
```

2. **Stride to Integer**:

```
from xtremes.miscellaneous import stride2int

stride = 'DBM'
block_size = 10
result = stride2int(stride, block_size)
print("Stride to Integer Result:", result)
```

3. **Model Parameters to Gamma True**:

```
from xtremes.miscellaneous import modelparams2gamma_true

distr = 'GEV'
correlation = 'IID'
modelparams = [0.5]
result = modelparams2gamma_true(distr, correlation,␣
↪modelparams)
print("Model Parameters to Gamma True Result:", result)
```