# tsbootstrap

*Release 0.1.1*

**Sankalp Gilda**

**May 01, 2024**

# CONTENTS:

CHAPTER

# ONE

# BASE BOOTSTRAP

class tsbootstrap.base_bootstrap.**BaseDistributionBootstrap**(*n_bootstraps: Integral = 10*, *distribution: str = 'normal'*, *refit: bool = False*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params=None*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, *save_models: bool = False*, *rng=None*, ***kwargs*)

Implementation of the Distribution Bootstrap (DB) method for time series data.

The DB method is a non-parametric method that generates bootstrapped samples by fitting a distribution to the residuals and then generating new residuals from the fitted distribution. The new residuals are then added to the fitted values to create the bootstrapped samples.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **distribution** (*str, default='normal'*) – The distribution to use for generating the bootstrapped samples. Must be one of 'poisson', 'exponential', 'normal', 'gamma', 'beta', 'lognormal', 'weibull', 'pareto', 'geometric', or 'uniform'.
- **refit** (*bool, default=False*) – Whether to refit the distribution to the resampled residuals for each bootstrap. If False, the distribution is fit once to the residuals and the same distribution is used for all bootstraps.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*Integral or list or tuple, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of (p, o, q) for ARIMA and (p, d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.
- **save_models** (*bool, default=False*) – Whether to save the fitted models.

- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*)
  – The random number generator or seed used to generate the bootstrap samples.

**resids_dist**

The distribution object used to generate the bootstrapped samples. If None, the distribution has not been fit yet.

**Type**
scipy.stats.rv_continuous or None

**resids_dist_params**

The parameters of the distribution used to generate the bootstrapped samples. If None, the distribution has not been fit yet.

**Type**
tuple or None

**__init__ : Initialize the BaseDistributionBootstrap class.**

**fit_distribution**(*resids: np.ndarray*) → tuple[rv_continuous, tuple]

Fit the specified distribution to the residuals and return the distribution object and the parameters of the distribution.

### Notes

The DB method is defined as:

$$hatX_t = hat mu + epsilon_t$$

where
$epsilon_t$
$simF_{hatepsilon}$ is a random variable sampled from the distribution $F_{hatepsilon}$ fitted to the residuals $hatepsilon$.

### References

**class** tsbootstrap.base_bootstrap.**BaseMarkovBootstrap**(*n_bootstraps: Integral = 10*, *method: Literal['first', 'middle', 'last', 'mean', 'mode', 'median', 'kmeans', 'kmedians', 'kmedoids'] = 'middle'*, *apply_pca_flag: bool = False*, *pca=None*, *n_iter_hmm: Integral = 10*, *n_fits_hmm: Integral = 1*, *blocks_as_hidden_states_flag: bool = False*, *n_states: Integral = 2*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params=None*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, *save_models: bool = False*, *rng=None*, ***kwargs*)

Base class for Markov bootstrap.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **method** (*str, default="middle"*) – The method to use for compressing the blocks. Must be one of "first", "middle", "last", "mean", "mode", "median", "kmeans", "kmedians", "kmedoids".
- **apply_pca_flag** (*bool, default=False*) – Whether to apply PCA to the residuals before fitting the HMM.
- **pca** (*PCA, default=None*) – The PCA object to use for applying PCA to the residuals.
- **n_iter_hmm** (*Integral, default=10*) – Number of iterations for fitting the HMM.
- **n_fits_hmm** (*Integral, default=1*) – Number of times to fit the HMM.
- **blocks_as_hidden_states_flag** (*bool, default=False*) – Whether to use blocks as hidden states.
- **n_states** (*Integral, default=2*) – Number of states for the HMM.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*Integral or list or tuple, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of (p, o, q) for ARIMA and (p, d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.
- **save_models** (*bool, default=False*) – Whether to save the fitted models.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

**hmm_object**

The MarkovSampler object used for sampling.

**Type**
MarkovSampler or None

**__init__ : Initialize the Markov bootstrap.**

### Notes

Fitting Markov models is expensive, hence we do not allow re-fititng. We instead fit once to the residuals and generate new samples by changing the random_seed.

class tsbootstrap.base_bootstrap.**BaseResidualBootstrap**(*n_bootstraps: Integral = 10*, *rng=None*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params=None*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, *save_models: bool = False*)

Base class for residual bootstrap.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*Integral or list or tuple, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of (p, o, q) for ARIMA and (p, d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.
- **save_models** (*bool, default=False*) – Whether to save the fitted models.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

**fit_model**

The fitted model.

**Type**
TSFitBestLag

**resids**

The residuals of the fitted model.

**Type**
np.ndarray

**X_fitted**

The fitted values of the fitted model.

**Type**
np.ndarray

**coefs**

The coefficients of the fitted model.

**Type**
np.ndarray

**__init__ : Initialize self.**

**_fit_model : Fits the model to the data and stores the residuals.**

class tsbootstrap.base_bootstrap.**BaseSieveBootstrap**(*n_bootstraps: Integral = 10*, *rng=None*, *resids_model_type: Literal['ar', 'arima', 'sarima', 'var', 'arch'] = 'ar'*, *resids_order=None*, *save_resids_models: bool = False*, *kwargs_base_sieve=None*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params=None*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, ***kwargs_base_residual*)

Base class for Sieve bootstrap.

This class provides the core functionalities for implementing the Sieve bootstrap method, allowing for the fitting of various models to the residuals and generation of bootstrapped samples. The Sieve bootstrap is a parametric method that generates bootstrapped samples by fitting a model to the residuals and then generating new residuals from the fitted model. The new residuals are then added to the fitted values to create the bootstrapped samples.

**Parameters**

- **resids_model_type** (*str, default="ar"*) – The model type to use for fitting the residuals. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **resids_order** (*Integral or list or tuple, default=None*) – The order of the model to use for fitting the residuals. If None, the order is automatically determined.
- **save_resids_models** (*bool, default=False*) – Whether to save the fitted models for the residuals.
- **kwargs_base_sieve** (*dict, default=None*) – Keyword arguments to pass to the SieveBootstrap class.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*Integral or list or tuple, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of (p, o, q) for ARIMA and (p, d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.

**resids_coefs**

Coefficients of the fitted residual model. Replace "type" with the specific type if known.

**Type**
type or None

**resids_fit_model**

Fitted residual model object. Replace "type" with the specific type if known.

**Type**
type or None

**__init__ : Initialize the BaseSieveBootstrap class.**

**_fit_resids_model : Fit the residual model to the residuals.**

class tsbootstrap.base_bootstrap.**BaseStatisticPreservingBootstrap**(*n_bootstraps: Integral = 10*, *statistic: Callable | None = None*, *statistic_axis: Integral = 0*, *statistic_keepdims: bool = False*, *rng=None*)

Bootstrap class that generates bootstrapped samples preserving a specific statistic.

This class generates bootstrapped time series data, preserving a given statistic (such as mean, median, etc.) The statistic is calculated from the original data and then used as a parameter for generating the bootstrapped samples. For example, if the statistic is np.mean, then the mean of the original data is calculated and then used as a parameter for generating the bootstrapped samples.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **statistic** (*Callable, default=np.mean*) – A callable function to compute the statistic that should be preserved.
- **statistic_axis** (*Integral, default=0*) – The axis along which the statistic should be computed.
- **statistic_keepdims** (*bool, default=False*) – Whether to keep the dimensions of the statistic or not.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

**statistic_X**

The statistic calculated from the original data. This is used as a parameter for generating the bootstrapped samples.

**Type**
np.ndarray, default=None

**__init__ : Initialize the BaseStatisticPreservingBootstrap class.**

**_calculate_statistic**(*X: np.ndarray*) → np.ndarray : Calculate the statistic from the input data.

class tsbootstrap.base_bootstrap.**BaseTimeSeriesBootstrap**(*n_bootstraps: Integral = 10*, *rng=None*)

Base class for time series bootstrapping.

**Raises**
**ValueError** – If n_bootstraps is not greater than 0.

**bootstrap**(*X: ndarray*, *return_indices: bool = False*, *y=None*, *test_ratio: float | None = None*)

Generate indices to split data into training and test set.

**Parameters**

- **X** (*2D array-like of shape (n_timepoints, n_features)*) – The endogenous time series to bootstrap. Dimension 0 is assumed to be the time dimension, ordered

- **return_indices** (*bool, default=False*) – If True, a second output is retured, integer locations of index references for the bootstrap sample, in reference to original indices. Indexed values do are not necessarily identical with bootstrapped values.
- **y** (*array-like of shape (n_timepoints, n_features_exog), default=None*) – Exogenous time series to use in bootstrapping.
- **test_ratio** (*float, default=0.0*) – The ratio of test samples to total samples. If provided, test_ratio fraction the data (rounded up) is removed from the end before applying the bootstrap logic.

**Yields**

- **X_boot_i** (*2D np.ndarray-like of shape (n_timepoints_boot_i, n_features)*) – i-th bootstrapped sample of X.
- **indices_i** (*1D np.nparray of shape (n_timepoints_boot_i,) integer values,*) – only returned if return_indices=True. Index references for the i-th bootstrapped sample of X. Indexed values do are not necessarily identical with bootstrapped values.

**get_n_bootstraps**(*X=None*, *y=None*) → int

Returns the number of bootstrap instances produced by the bootstrap.

**Parameters**

- **X** (*2D array-like of shape (n_timepoints, n_features)*) – The endogenous time series to bootstrap. Dimension 0 is assumed to be the time dimension, ordered
- **y** (*array-like of shape (n_timepoints, n_features_exog), default=None*) – Exogenous time series to use in bootstrapping.

**Returns**
**int**

**Return type**
The number of bootstrap instances produced by the bootstrap.

CHAPTER

# TWO

# BLOCK BOOTSTRAP

class tsbootstrap.block_bootstrap.**BartlettsBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *bootstrap_type: str = 'moving'*, *rng=None*, ***kwargs*)

Bartlett's Bootstrap class for time series data.

This class is a specialized bootstrapping class that uses Bartlett's window for tapered weights.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.
- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.
- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.
- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.
- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

### Notes

The Bartlett window is defined as:

$$w(n) = 1 - \\ frac|n - (N-1)/2|(N-1)/2$$

where $N$ is the block length.

### References

**class** tsbootstrap.block_bootstrap.**BaseBlockBootstrap**(*bootstrap_type: str = 'moving'*, ***kwargs*)

Base class for block bootstrapping.

**Parameters**

- **bootstrap_type** (*str, default="moving"*) – The type of block bootstrap to use. Must be one of "nonoverlapping", "moving", "stationary", or "circular".
- **kwargs** – Additional keyword arguments to pass to the BaseBlockBootstrapConfig class. See the documentation for BaseBlockBootstrapConfig for more information.

**class** tsbootstrap.block_bootstrap.**BlackmanBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *bootstrap_type: str = 'moving'*, *rng=None*, ***kwargs*)

Blackman Bootstrap class for time series data.

This class is a specialized bootstrapping class that uses Blackman window for tapered weights.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.
- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.
- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.
- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.

- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

### Notes

The Blackman window is defined as:

$$w(n) = 0.42 - 0.5 \\ cos \\ left( \\ frac2 \\ pinN - 1 \\ right) + 0.08 \\ cos \\ left( \\ frac4 \\ pinN - 1 \\ right)$$

where $N$ is the block length.

### References

**class** tsbootstrap.block_bootstrap.**BlockBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *rng=None*)

Block Bootstrap base class for time series data.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is automatically set to the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.
- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.

- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.
- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.
- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

**Raises**

**ValueError** – If block_length is not greater than 0.

class tsbootstrap.block_bootstrap.**CircularBlockBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *rng=None*, ***kwargs*)

Circular Block Bootstrap class for time series data.

This class functions similarly to the base *BlockBootstrap* class, with the following modifications to the default behavior: * *overlap_flag* is always set to True, meaning that blocks can overlap. * *wrap_around_flag* is always set to True, meaning that the data will wrap around when generating blocks. * *block_length_distribution* is always None, meaning that the block length distribution is not utilized. * *combine_generation_and_sampling_flag* is always False, meaning that the block generation and resampling are performed separately.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.
- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.
- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.
- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.

- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

### Notes

The Circular Block Bootstrap is defined as:

$$
\begin{aligned}
hatX_t = \\
frac1L \\
sum_{i=1}^{L} X_{t+} \\
lfloor U_i \\
rfloor
\end{aligned}
$$

where $L$ is the block length, $U_i$ is a uniform random variable on $[0, 1]$, and
$lfloor$
$cdot$
$rfloor$ is the floor function.

### References

**class** tsbootstrap.block_bootstrap.**HammingBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *bootstrap_type: str = 'moving'*, *rng=None*, ***kwargs*)

Hamming Bootstrap class for time series data.

This class is a specialized bootstrapping class that uses Hamming window for tapered weights.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.
- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.

- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.
- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.
- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

### Notes

The Hamming window is defined as:

$$w(n) = 0.54 - 0.46$$
$$cos$$
$$left($$
$$frac2$$
$$pinN - 1$$
$$right)$$

where $N$ is the block length.

### References

class tsbootstrap.block_bootstrap.**HanningBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *bootstrap_type: str = 'moving'*, *rng=None*, ***kwargs*)

Hanning Bootstrap class for time series data.

This class is a specialized bootstrapping class that uses Hanning window for tapered weights.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.

- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.
- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.
- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.
- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **bootstrap_type** (*str, default="moving"*) – The type of block bootstrap to use. Must be one of "nonoverlapping", "moving", "stationary", or "circular".
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

### Notes

The Hanning window is defined as:

$$w(n) = 0.5 - 0.5 \\ cos \\ left( \\ frac2 \\ pinN - 1 \\ right)$$

where $N$ is the block length.

### References

**class** tsbootstrap.block_bootstrap.**MovingBlockBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *rng=None*, ***kwargs*)

Moving Block Bootstrap class for time series data.

This class functions similarly to the base *BlockBootstrap* class, with the following modifications to the default behavior: * *overlap_flag* is always set to True, meaning that blocks can overlap. * *wrap_around_flag* is always set

to False, meaning that the data will not wrap around when generating blocks. * *block_length_distribution* is always None, meaning that the block length distribution is not utilized. * *combine_generation_and_sampling_flag* is always False, meaning that the block generation and resampling are performed separately.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.
- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.
- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.
- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.
- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

### Notes

The Moving Block Bootstrap is defined as:

$$
\begin{aligned}
hatX_t = \\
frac1L \\
sum_{i=1}^{L} X_{t+} \\
lfloorU_i \\
rfloor
\end{aligned}
$$

where $L$ is the block length, $U_i$ is a uniform random variable on $[0, 1]$, and
$lfloor$
$cdot$
$rfloor$ is the floor function.

### References

class tsbootstrap.block_bootstrap.**NonOverlappingBlockBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *rng=None*, ***kwargs*)

Non-Overlapping Block Bootstrap class for time series data.

This class functions similarly to the base *BlockBootstrap* class, with the following modifications to the default behavior: * *overlap_flag* is always set to False, meaning that blocks cannot overlap. * *wrap_around_flag* is always set to False, meaning that the data will not wrap around when generating blocks. * *block_length_distribution* is always None, meaning that the block length distribution is not utilized. * *combine_generation_and_sampling_flag* is always False, meaning that the block generation and resampling are performed separately.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.
- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.
- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.
- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.
- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

**Raises**

**ValueError** – If block_length is not greater than 0.

### Notes

The Non-Overlapping Block Bootstrap is defined as:

$$hatX_t =$$
$$frac1L$$
$$sum_{i=1}^{L} X_{t+i}$$

where $L$ is the block length.

### References

class tsbootstrap.block_bootstrap.**StationaryBlockBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *rng=None*, ***kwargs*)

Stationary Block Bootstrap class for time series data.

This class functions similarly to the base *BlockBootstrap* class, with the following modifications to the default behavior: * *overlap_flag* is always set to True, meaning that blocks can overlap. * *wrap_around_flag* is always set to False, meaning that the data will not wrap around when generating blocks. * *block_length_distribution* is always "geometric", meaning that the block length distribution is geometrically distributed. * *combine_generation_and_sampling_flag* is always False, meaning that the block generation and resampling are performed separately.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.
- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.
- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.
- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.

- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

### Notes

The Stationary Block Bootstrap is defined as:

$$
\begin{aligned}
hatX_t = \\
frac1L \\
sum_{i=1}^{L} X_{t+} \\
lfloor U_i \\
rfloor
\end{aligned}
$$

where $L$ is the block length, $U_i$ is a uniform random variable on $[0, 1]$, and
$lfloor$
$cdot$
$rfloor$ is the floor function.

### References

**class** tsbootstrap.block_bootstrap.**TukeyBootstrap**(*n_bootstraps: Integral = 10*, *block_length: Integral | None = None*, *block_length_distribution: str | None = None*, *wrap_around_flag: bool = False*, *overlap_flag: bool = False*, *combine_generation_and_sampling_flag: bool = False*, *block_weights=None*, *tapered_weights: Callable | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*, *bootstrap_type: str = 'moving'*, *rng=None*, ***kwargs*)

Tukey Bootstrap class for time series data.

This class is a specialized bootstrapping class that uses Tukey window for tapered weights.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **block_length** (*Integral, default=None*) – The length of the blocks to sample. If None, the block length is the square root of the number of observations.
- **block_length_distribution** (*str, default=None*) – The block length distribution function to use. If None, the block length distribution is not utilized.
- **wrap_around_flag** (*bool, default=False*) – Whether to wrap around the data when generating blocks.
- **overlap_flag** (*bool, default=False*) – Whether to allow blocks to overlap.
- **combine_generation_and_sampling_flag** (*bool, default=False*) – Whether to combine the block generation and sampling steps.

- **block_weights** (*array-like of shape (n_blocks,), default=None*) – The weights to use when sampling blocks.
- **tapered_weights** (*callable, default=None*) – The tapered weights to use when sampling blocks.
- **overlap_length** (*Integral, default=None*) – The length of the overlap between blocks.
- **min_block_length** (*Integral, default=None*) – The minimum length of the blocks.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

### Notes

The Tukey window is defined as:

$$
\begin{aligned}
w(n) = \\
begincases0.5 \\
left[1+ \\
cos \\
left( \\
frac2 \\
pin \\
alpha(N-1) \\
right) \\
right], \\
textifn < \\
frac \\
alpha(N-1)2 \\
\\
1, \\
textif \\
frac \\
alpha(N-1)2 \\
leqn \\
leq(N-1) \\
left(1- \\
frac \\
alpha2 \\
right) \\
\\
0.5 \\
left[1+ \\
cos \\
left( \\
frac2 \\
pin \\
alpha(N-1) \\
right) \\
right], \\
textifn > (N-1) \\
left(1- \\
frac \\
alpha2 \\
right) \\
endcases
\end{aligned}
$$

where $N$ is the block length and
$alpha$ is the parameter controlling the shape of the window.

### References

CHAPTER

# THREE

# BLOCK GENERATOR

class tsbootstrap.block_generator.**BlockGenerator**(*block_length_sampler: BlockLengthSampler*, *input_length: Integral*, *wrap_around_flag: bool = False*, *rng: Generator | None = None*, *overlap_length: Integral | None = None*, *min_block_length: Integral | None = None*)

A class that generates blocks of indices.

**__init__**()

Initialize the BlockGenerator with the given parameters.

**generate_non_overlapping_blocks**()

Generate non-overlapping block indices.

**generate_overlapping_blocks**()

Generate overlapping block indices.

**generate_blocks**(*overlap_flag=False*)

Generate block indices.

**property block_length_sampler: BlockLengthSampler**

The block length sampler.

**generate_blocks**(*overlap_flag: bool = False*)

Generate block indices.

This method is a general entry point to generate either overlapping or non-overlapping blocks based on the given flag.

**Parameters**

**overlap_flag** (*bool, optional*) – A flag indicating whether to generate overlapping blocks, by default False.

**Returns**

A list of numpy arrays where each array represents the indices of a block in the time series.

**Return type**

List[np.ndarray]

**generate_non_overlapping_blocks**()

Generate non-overlapping block indices in the time series.

**Returns**

List of numpy arrays containing the indices for each non-overlapping block.

**Return type**

List[np.ndarray]

### Example

```
>>> block_generator = BlockGenerator(input_length=100, block_length_
↪sampler=UniformBlockLengthSampler())
>>> non_overlapping_blocks = block_generator.generate_non_overlapping_blocks()
>>> len(non_overlapping_blocks)
10
```

**generate_overlapping_blocks**()

Generate overlapping block indices in the time series.

**Returns**
List of numpy arrays containing the indices for each overlapping block.

**Return type**
List[np.ndarray]

### Example

```
>>> block_generator = BlockGenerator(input_length=100, block_length_
↪sampler=UniformBlockLengthSampler(), overlap_length=5)
>>> overlapping_blocks = block_generator.generate_overlapping_blocks()
>>> len(overlapping_blocks)
15
```

**property input_length: Integral**

The length of the input time series.

**property min_block_length: Integral**

The minimum length of a block.

**property overlap_length: Integral**

The length of overlap between consecutive blocks.

**property rng: Generator**

The random number generator.

**property wrap_around_flag: bool**

A flag indicating whether to allow wrap-around in the block sampling.

CHAPTER

# FOUR

# BLOCK LENGTH SAMPLER

class tsbootstrap.block_length_sampler.**BlockLengthSampler**(*avg_block_length: Integral = 2*, *block_length_distribution: str | None = None*, *rng: Generator | Integral | None = None*)

A class for sampling block lengths for the random block length bootstrap.

**sample_block_length**()

Sample a block length from the selected distribution.

**property avg_block_length**

Getter for avg_block_length.

**property block_length_distribution: str**

Getter for block_length_distribution.

**property rng: Generator**

Getter for rng.

**sample_block_length**() → int

Sample a block length from the selected distribution.

**Returns**
A sampled block length.

**Return type**
int

CHAPTER

# FIVE

# BLOCK RESAMPLER

class tsbootstrap.block_resampler.**BlockResampler**(*blocks: List[ndarray]*, *X: ndarray*, *block_weights: Callable | ndarray | None = None*, *tapered_weights: Callable | ndarray | None = None*, *rng: Generator | Integral | None = None*)

A class to perform block resampling.

**resample_blocks**()

Resamples blocks and their corresponding tapered_weights with replacement to create a new list of blocks and tapered_weights with total length equal to n.

**resample_block_indices_and_data**()

Generate block indices and corresponding data for the input data array X.

**property X: ndarray**

The input data array.

**property block_weights: ndarray**

An array of normalized block_weights.

**property blocks: List[ndarray]**

A list of numpy arrays where each array represents the indices of a block in the time series.

**resample_block_indices_and_data**()

Generate block indices and corresponding data for the input data array X.

**Returns**

A tuple containing a list of block indices and a list of corresponding modified data blocks.

**Return type**

Tuple[List[np.ndarray], List[np.ndarray]]

### Example

```
>>> block_resampler = BlockResampler(blocks=blocks, X=data)
>>> block_indices, block_data = block_resampler.resample_block_indices_and_
↪data()
>>> len(block_indices) == len(data)
True
```

### Notes

The block indices are generated using the following steps: 1. Generate block weights using the block_weights argument. 2. Resample blocks with replacement to create a new list of blocks with total length equal to n. 3. Apply tapered_weights to the data within the blocks if provided.

**resample_blocks**()

Resample blocks and corresponding tapered weights with replacement to create a new list of blocks and tapered weights with total length equal to n.

**Returns**

The newly generated list of blocks and their corresponding tapered_weights with total length equal to n.

**Return type**

Tuple[list of ndarray, list of ndarray]

### Example

```
>>> block_resampler = BlockResampler(blocks=blocks, X=data)
>>> new_blocks, new_tapered_weights = block_resampler.resample_blocks()
>>> len(new_blocks) == len(data)
True
```

**property rng:** Generator

Generator for reproducibility.

**property tapered_weights**

A list of normalized weights.

CHAPTER

# SIX

# BOOTSTRAP

class tsbootstrap.bootstrap.**BlockDistributionBootstrap**(*n_bootstraps: Integral = 10*, *block_bootstrap: BlockBootstrap | None = None*, *distribution: str = 'normal'*, *refit: bool = False*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params: dict | None = None*, *order=None*, *save_models: bool = False*, *rng: Generator | Integral | None = None*)

Block Distribution Bootstrap class for time series data.

This class applies distribution bootstrapping to blocks of the time series. The residuals are fit to a distribution, then resampled using the specified block structure. Then new residuals are generated from the fitted distribution and added to the fitted values to generate new samples.

**Parameters**

- **block_bootstrap** (*BlockBootstrap, default=MovingBlockBootstrap()*) – The block bootstrap algorithm.
- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **distribution** (*str, default='normal'*) – The distribution to use for generating the bootstrapped samples. Must be one of 'poisson', 'exponential', 'normal', 'gamma', 'beta', 'lognormal', 'weibull', 'pareto', 'geometric', or 'uniform'.
- **refit** (*bool, default=False*) – Whether to refit the distribution to the resampled residuals for each bootstrap. If False, the distribution is fit once to the residuals and the same distribution is used for all bootstraps.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*Integral or list or tuple, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of (p, o, q) for ARIMA and (p, d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.
- **save_models** (*bool, default=False*) – Whether to save the fitted models.

- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

**resids_dist**

The distribution object used to generate the bootstrapped samples. If None, the distribution has not been fit yet.

**Type**
scipy.stats.rv_continuous or None

**resids_dist_params**

The parameters of the distribution used to generate the bootstrapped samples. If None, the distribution has not been fit yet.

**Type**
tuple or None

**__init__ : Initialize self.**

**_generate_samples_single_bootstrap : Generate a single bootstrap sample.**

### Notes

We either fit the distribution to the residuals once and generate new samples from the fitted distribution with a new random seed, or resample the residuals once and fit the distribution to the resampled residuals, then generate new samples from the fitted distribution with the same random seed n_bootstrap times.

**classmethod get_test_params**(*parameter_set='default'*)

Return testing parameter settings for the estimator.

**Parameters**
**parameter_set** (*str, default="default"*) – Name of the set of test parameters to return, for use in tests. If no special parameters are defined for a value, will return *"default"* set.

**Returns**
**params** – Parameters to create testing instances of the class Each dict are parameters to construct an "interesting" test instance, i.e., *MyClass(**params)* or *MyClass(**params[i])* creates a valid test instance. *create_test_instance* uses the first (or only) dictionary in *params*

**Return type**
dict or list of dict, default = {}

**class** tsbootstrap.bootstrap.**BlockMarkovBootstrap**(*n_bootstraps: Integral = 10*, *block_bootstrap: BlockBootstrap | None = None*, *method: Literal['first', 'middle', 'last', 'mean', 'mode', 'median', 'kmeans', 'kmedians', 'kmedoids'] = 'middle'*, *apply_pca_flag: bool = False*, *pca=None*, *n_iter_hmm: Integral = 10*, *n_fits_hmm: Integral = 1*, *blocks_as_hidden_states_flag: bool = False*, *n_states: Integral = 2*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params: dict | None = None*, *order=None*, *save_models: bool = False*, *rng: Generator | Integral | None = None*)

Block Markov Bootstrap class for time series data.

This class applies Markov bootstrapping to blocks of the time series. The residuals are fit to a Markov model, then resampled using the specified block structure. The resampled residuals are added to the fitted values to

generate new samples. This class is a combination of the *BlockResidualBootstrap* and *WholeMarkovBootstrap* classes.

**Parameters**

- **block_bootstrap** (*BlockBootstrap, default=MovingBlockBootstrap()*) – The block bootstrap algorithm.
- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **method** (*str, default="middle"*) – The method to use for compressing the blocks. Must be one of "first", "middle", "last", "mean", "mode", "median", "kmeans", "kmedians", "kmedoids".
- **apply_pca_flag** (*bool, default=False*) – Whether to apply PCA to the residuals before fitting the HMM.
- **pca** (*PCA, default=None*) – The PCA object to use for applying PCA to the residuals.
- **n_iter_hmm** (*Integral, default=10*) – Number of iterations for fitting the HMM.
- **n_fits_hmm** (*Integral, default=1*) – Number of times to fit the HMM.
- **blocks_as_hidden_states_flag** (*bool, default=False*) – Whether to use blocks as hidden states.
- **n_states** (*Integral, default=2*) – Number of states for the HMM.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*Integral or list or tuple, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of (p, o, q) for ARIMA and (p, d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.
- **save_models** (*bool, default=False*) – Whether to save the fitted models.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

**__init__ : Initialize self.**

**_generate_samples_single_bootstrap : Generate a single bootstrap sample.**

### Notes

Fitting Markov models is expensive, hence we do not allow re-fititng. We instead fit once to the residuals, resample using blocks once, and generate new samples by changing the random_seed.

**classmethod get_test_params**(*parameter_set='default'*)

Return testing parameter settings for the estimator.

**Parameters**

**parameter_set** (*str, default="default"*) – Name of the set of test parameters to return, for use in tests. If no special parameters are defined for a value, will return *"default"* set.

**Returns**

**params** – Parameters to create testing instances of the class Each dict are parameters to construct an "interesting" test instance, i.e., *MyClass(**params)* or *MyClass(**params[i])* creates a valid test instance. *create_test_instance* uses the first (or only) dictionary in *params*

**Return type**

dict or list of dict, default = { }

**class** tsbootstrap.bootstrap.**BlockResidualBootstrap**(*n_bootstraps: Integral = 10*, *block_bootstrap: BlockBootstrap | None = None*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params: dict | None = None*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, *save_models: bool = False*, *rng: Generator | Integral | None = None*)

Block Residual Bootstrap class for time series data.

This class applies residual bootstrapping to blocks of the time series. The residuals are bootstrapped using the specified block structure and added to the fitted values to generate new samples.

**Parameters**

- **block_bootstrap** (*BlockBootstrap, default=MovingBlockBootstrap()*) – The block bootstrap algorithm.
- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*OrderTypes, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of (p, o, q) for ARIMA and (p, d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.
- **save_models** (*bool, default=False*) – Whether to save the fitted models.
- **rng** (*RngTypes, default=None*) – The random number generator or seed used to generate the bootstrap samples.

**__init__ : Initialize self.**

**_generate_samples_single_bootstrap : Generate a single bootstrap sample.**

**classmethod get_test_params**(*parameter_set='default'*)

Return testing parameter settings for the estimator.

**Parameters**

**parameter_set** (*str, default="default"*) – Name of the set of test parameters to return, for use in tests. If no special parameters are defined for a value, will return *"default"* set.

**Returns**

**params** – Parameters to create testing instances of the class Each dict are parameters to construct an "interesting" test instance, i.e., *MyClass(**params)* or *MyClass(**params[i])* creates a valid test instance. *create_test_instance* uses the first (or only) dictionary in *params*

**Return type**

dict or list of dict, default = {}

**class** tsbootstrap.bootstrap.**BlockSieveBootstrap**(*n_bootstraps: Integral = 10*, *block_bootstrap: BlockBootstrap | None = None*, *resids_model_type: Literal['ar', 'arima', 'sarima', 'var', 'arch'] = 'ar'*, *resids_order=None*, *save_resids_models: bool = False*, *kwargs_base_sieve=None*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params: dict | None = None*, *order=None*, *save_models: bool = False*, *rng: Generator | Integral | None = None*)

Implementation of the Sieve bootstrap method for time series data.

This class applies Sieve bootstrapping to blocks of the time series. The residuals are fit to a second model, then resampled using the specified block structure. The new residuals are then added to the fitted values to generate new samples.

**Parameters**

- **block_bootstrap** (*BlockBootstrap, default=MovingBlockBootstrap()*) – The block bootstrap algorithm.
- **resids_model_type** (*str, default="ar"*) – The model type to use for fitting the residuals. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **resids_order** (*Integral or list or tuple, default=None*) – The order of the model to use for fitting the residuals. If None, the order is automatically determined.
- **save_resids_models** (*bool, default=False*) – Whether to save the fitted models for the residuals.
- **kwargs_base_sieve** (*dict, default=None*) – Keyword arguments to pass to the SieveBootstrap class.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*Integral or list or tuple, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of

(p, o, q) for ARIMA and (p, d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.

- **save_models** (*bool, default=False*) – Whether to save the fitted models.

**_init_ : Initialize self.**

**_generate_samples_single_bootstrap : Generate a single bootstrapped sample.**

**classmethod get_test_params**(*parameter_set='default'*)

Return testing parameter settings for the estimator.

**Parameters**

**parameter_set** (*str, default="default"*) – Name of the set of test parameters to return, for use in tests. If no special parameters are defined for a value, will return *"default"* set.

**Returns**

**params** – Parameters to create testing instances of the class Each dict are parameters to construct an "interesting" test instance, i.e., *MyClass(**params)* or *MyClass(**params[i])* creates a valid test instance. *create_test_instance* uses the first (or only) dictionary in *params*

**Return type**

dict or list of dict, default = { }

**class** tsbootstrap.bootstrap.**BlockStatisticPreservingBootstrap**(*n_bootstraps: Integral = 10*, *block_bootstrap: BlockBootstrap | None = None*, *statistic=None*, *statistic_axis: Integral = 0*, *statistic_keepdims: bool = False*, *rng: Generator | Integral | None = None*)

Block Statistic Preserving Bootstrap class for time series data.

This class applies statistic-preserving bootstrapping to blocks of the time series. The residuals are resampled using the specified block structure and added to the fitted values to generate new samples.

**Parameters**

- **block_bootstrap** (*BlockBootstrap, default=MovingBlockBootstrap()*) – The block bootstrap algorithm.
- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **statistic** (*Callable, default=np.mean*) – A callable function to compute the statistic that should be preserved.
- **statistic_axis** (*Integral, default=0*) – The axis along which the statistic should be computed.
- **statistic_keepdims** (*bool, default=False*) – Whether to keep the dimensions of the statistic or not.
- **rng** (*Integral or np.random.Generator, default=np.random.default_rng()*) – The random number generator or seed used to generate the bootstrap samples.

**statistic_X**

The statistic calculated from the original data. This is used as a parameter for generating the bootstrapped samples.

**Type**
np.ndarray, default=None

**__init__ : Initialize self.**

**_generate_samples_single_bootstrap : Generate a single bootstrap sample.**

**classmethod get_test_params**(*parameter_set='default'*)

Return testing parameter settings for the estimator.

**Parameters**
**parameter_set** (*str, default="default"*) – Name of the set of test parameters to return, for use in tests. If no special parameters are defined for a value, will return *"default"* set.

**Returns**
**params** – Parameters to create testing instances of the class Each dict are parameters to construct an "interesting" test instance, i.e., *MyClass(**params)* or *MyClass(**params[i])* creates a valid test instance. *create_test_instance* uses the first (or only) dictionary in *params*

**Return type**
dict or list of dict, default = {}

**class** tsbootstrap.bootstrap.**WholeDistributionBootstrap**(*n_bootstraps: Integral = 10*, *distribution: str = 'normal'*, *refit: bool = False*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params=None*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, *save_models: bool = False*, *rng=None*, ***kwargs*)

Whole Distribution Bootstrap class for time series data.

This class applies distribution bootstrapping to the entire time series, without any block structure. This is the most basic form of distribution bootstrapping. The residuals are fit to a distribution, and then resampled using the distribution. The resampled residuals are added to the fitted values to generate new samples.

**resids_dist**

The distribution object used to generate the bootstrapped samples. If None, the distribution has not been fit yet.

**Type**
scipy.stats.rv_continuous or None

**resids_dist_params**

The parameters of the distribution used to generate the bootstrapped samples. If None, the distribution has not been fit yet.

**Type**
tuple or None

**__init__ : Initialize self.**

**_generate_samples_single_bootstrap : Generate a single bootstrap sample.**

### Notes

We either fit the distribution to the residuals once and generate new samples from the fitted distribution with a new random seed, or resample the residuals once and fit the distribution to the resampled residuals, then generate new samples from the fitted distribution with the same random seed n_bootstrap times.

class tsbootstrap.bootstrap.**WholeMarkovBootstrap**(*n_bootstraps: Integral = 10*, *method: Literal['first', 'middle', 'last', 'mean', 'mode', 'median', 'kmeans', 'kmedians', 'kmedoids'] = 'middle'*, *apply_pca_flag: bool = False*, *pca=None*, *n_iter_hmm: Integral = 10*, *n_fits_hmm: Integral = 1*, *blocks_as_hidden_states_flag: bool = False*, *n_states: Integral = 2*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params=None*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, *save_models: bool = False*, *rng=None*, ***kwargs*)

Whole Markov Bootstrap class for time series data.

This class applies Markov bootstrapping to the entire time series, without any block structure. This is the most basic form of Markov bootstrapping. The residuals are fit to a Markov model, and then resampled using the Markov model. The resampled residuals are added to the fitted values to generate new samples.

**_generate_samples_single_bootstrap : Generate a single bootstrap sample.**

### Notes

Fitting Markov models is expensive, hence we do not allow re-fititng. We instead fit once to the residuals and generate new samples by changing the random_seed.

class tsbootstrap.bootstrap.**WholeResidualBootstrap**(*n_bootstraps: Integral = 10*, *rng: Generator | Integral | None = None*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params: dict | None = None*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, *save_models: bool = False*)

Whole Residual Bootstrap class for time series data.

This class applies residual bootstrapping to the entire time series, without any block structure. This is the most basic form of residual bootstrapping. The residuals are resampled with replacement and added to the fitted values to generate new samples.

**Parameters**

- **n_bootstraps** (*Integral, default=10*) – The number of bootstrap samples to create.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*OrderTypes, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of (p, o, q) for ARIMA and (p,

d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.

- **save_models** (*bool, default=False*) – Whether to save the fitted models.
- **rng** (*RngTypes, default=None*) – The random number generator or seed used to generate the bootstrap samples.

**__init__ : Initialize self.**

**_generate_samples_single_bootstrap : Generate a single bootstrap sample.**

class tsbootstrap.bootstrap.**WholeSieveBootstrap**(*n_bootstraps: Integral = 10*, *rng=None*, *resids_model_type: Literal['ar', 'arima', 'sarima', 'var', 'arch'] = 'ar'*, *resids_order=None*, *save_resids_models: bool = False*, *kwargs_base_sieve=None*, *model_type: Literal['ar', 'arima', 'sarima', 'var'] = 'ar'*, *model_params=None*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, ***kwargs_base_residual*)

Implementation of the Sieve bootstrap method for time series data.

This class applies Sieve bootstrapping to the entire time series, without any block structure. This is the most basic form of Sieve bootstrapping. The residuals are fit to a second model, and then new samples are generated by adding the new residuals to the fitted values.

**Parameters**

- **resids_model_type** (*str, default="ar"*) – The model type to use for fitting the residuals. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **resids_order** (*Integral or list or tuple, default=None*) – The order of the model to use for fitting the residuals. If None, the order is automatically determined.
- **save_resids_models** (*bool, default=False*) – Whether to save the fitted models for the residuals.
- **kwargs_base_sieve** (*dict, default=None*) – Keyword arguments to pass to the SieveBootstrap class.
- **model_type** (*str, default="ar"*) – The model type to use. Must be one of "ar", "arima", "sarima", "var", or "arch".
- **model_params** (*dict, default=None*) – Additional keyword arguments to pass to the TSFit model.
- **order** (*Integral or list or tuple, default=None*) – The order of the model. If None, the best order is chosen via TSFitBestLag. If Integral, it is the lag order for AR, ARIMA, and SARIMA, and the lag order for ARCH. If list or tuple, the order is a tuple of (p, o, q) for ARIMA and (p, d, q, s) for SARIMAX. It is either a single Integral or a list of non-consecutive ints for AR, and an Integral for VAR and ARCH. If None, the best order is chosen via TSFitBestLag. Do note that TSFitBestLag only chooses the best lag, not the best order, so for the tuple values, it only chooses the best p, not the best (p, o, q) or (p, d, q, s). The rest of the values are set to 0.

**_generate_samples_single_bootstrap : Generate a single bootstrapped sample.**

class tsbootstrap.bootstrap.**WholeStatisticPreservingBootstrap**(*n_bootstraps: Integral = 10*, *statistic: Callable | None = None*, *statistic_axis: Integral = 0*, *statistic_keepdims: bool = False*, *rng=None*)

Whole Statistic Preserving Bootstrap class for time series data.

This class applies statistic-preserving bootstrapping to the entire time series, without any block structure. This is the most basic form of statistic-preserving bootstrapping. The residuals are resampled with replacement and added to the fitted values to generate new samples.

**_generate_samples_single_bootstrap : Generate a single bootstrap sample.**

CHAPTER

# SEVEN

# MARKOV SAMPLER

class tsbootstrap.markov_sampler.**BlockCompressor**(*method: Literal['first', 'middle', 'last', 'mean', 'mode', 'median', 'kmeans', 'kmedians', 'kmedoids'] = 'middle'*, *apply_pca_flag: bool = False*, *pca: PCA | None = None*, *random_seed: Integral | None = None*)

BlockCompressor class provides the functionality to compress blocks of data using different techniques.

**__init__**(*method: BlockCompressorTypes = 'middle'*, *apply_pca_flag: bool = False*, *pca: PCA | None = None*, *random_seed: Integral | None = None*) → None

Initialize the BlockCompressor instance.

**_pca_compression**(*block: np.ndarray*, *summary: np.ndarray*) → np.ndarray

Summarize a block of data using PCA.

**_summarize_block**(*block: np.ndarray*) → np.ndarray

Summarize a block using a specified method.

**summarize_blocks**(*blocks*) → np.ndarray

Summarize each block in the input list of blocks using the specified method.

**property apply_pca_flag: bool**

Getter for apply_pca_flag.

**classmethod get_test_params**(*parameter_set='default'*)

Return testing parameter settings for the estimator.

**Parameters**

**parameter_set** (*str, default="default"*) – Name of the set of test parameters to return, for use in tests. If no special parameters are defined for a value, will return *"default"* set.

**Returns**

**params** – Parameters to create testing instances of the class Each dict are parameters to construct an "interesting" test instance, i.e., *MyClass(**params)* or *MyClass(**params[i])* creates a valid test instance. *create_test_instance* uses the first (or only) dictionary in *params*

**Return type**

dict or list of dict, default = {}

**property method: str**

Getter for method.

**property pca: PCA**

Getter for pca.

**summarize_blocks**(*blocks*) → ndarray

Summarize each block in the input list of blocks using the specified method.

**Parameters**
**blocks** (*List[np.ndarray]*) – List of numpy arrays representing the blocks to be summarized.

**Returns**
Numpy array containing the summarized blocks.

**Return type**
np.ndarray

### Example

```
>>> compressor = BlockCompressor(method='middle')
>>> blocks = [np.array([1, 2, 3]), np.array([4, 5, 6])]
>>> summarized_blocks = compressor.summarize_blocks(blocks)
>>> summarized_blocks
array([2, 5])
```

**class** tsbootstrap.markov_sampler.**MarkovSampler**(*method: Literal['first', 'middle', 'last', 'mean', 'mode', 'median', 'kmeans', 'kmedians', 'kmedoids'] = 'middle', apply_pca_flag: bool = False, pca: PCA | None = None, n_iter_hmm: Integral = 100, n_fits_hmm: Integral = 10, blocks_as_hidden_states_flag: bool = False, random_seed: Integral | None = None*)

A class for sampling from a Markov chain with given transition probabilities.

This class allows for the combination of block-based bootstrapping and Hidden Markov Model (HMM) fitting.

**transition_matrix_calculator**

An instance of MarkovTransitionMatrixCalculator to calculate transition probabilities.

**Type**
MarkovTransitionMatrixCalculator

**block_compressor**

An instance of BlockCompressor to perform block summarization/compression.

**Type**
BlockCompressor

**__init__**(*method: str = 'mean', apply_pca_flag: bool = False, pca: PCA | None = None, n_iter_hmm: Integral = 100, n_fits_hmm: Integral = 10, blocks_as_hidden_states_flag: bool = False, random_seed: Integral | None = None*) → None

Initialize the MarkovSampler instance.

**_validate_n_states**(*n_states: Integral*, *blocks*) → Integral

Validate the number of states.

**_validate_n_iter_hmm**(*n_iter_hmm: Integral*) → Integral

Validate the number of iterations for the HMM.

**_validate_n_fits_hmm**(*n_fits_hmm: Integral*) → Integral

Validate the number of fits for the HMM.

**_validate_blocks_as_hidden_states_flag**(*blocks_as_hidden_states_flag: bool*) → bool

Validate the blocks_as_hidden_states_flag.

**_validate_random_seed**(*random_seed: Integral | None*) → Integral | None

Validate the random seed.

**fit_hidden_markov_model**(*blocks*, *n_states: Integral = 5*) → hmm.GaussianHMM

Fit a Hidden Markov Model (HMM) to the input blocks.

**fit**(*blocks*, *n_states: Integral = 5*) → MarkovSampler

Fit the MarkovSampler instance to the input blocks.

**sample**(*blocks*, *n_states: Integral = 5*) → Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]

Sample from the MarkovSampler instance.

### Examples

```
>>> sampler = MarkovSampler(n_iter_hmm=200, n_fits_hmm=20)
>>> blocks = [np.random.rand(10, 5) for _ in range(50)]
>>> start_probs, trans_probs, centers, covariances, assignments = sampler.
↪sample(blocks, n_states=5, blocks_as_hidden_states_flag=True)
```

**property blocks_as_hidden_states_flag: bool**

Getter for blocks_as_hidden_states_flag.

**fit**(*blocks*, *n_states: Integral = 5*) → MarkovSampler

Sample from a Markov chain with given transition probabilities.

**Parameters**

- **blocks** (*List[np.ndarray] or np.ndarray*) – A list of 2D NumPy arrays, each representing a block of data, or a 2D NumPy array, where each row represents a row of raw data.
- **n_states** (*Integral, optional*) – The number of states in the hidden Markov model. Default is 5.

**Returns**

Current instance of the MarkovSampler class, with the model trained.

**Return type**

MarkovSampler

### Examples

```
>>> blocks = [np.random.rand(10, 5) for _ in range(50)]
>>> sampler.fit(blocks, n_states=5)
```

**fit_hidden_markov_model**(*X: ndarray*, *n_states: Integral = 5*, *transmat_init: ndarray | None = None*, *means_init: ndarray | None = None*, *lengths: ndarray | None = None*)

Fit a Gaussian Hidden Markov Model on the input data.

**Parameters**

- **X** (*np.ndarray*) – A 2D NumPy array, where each row represents a summarized block of data.

- **n_states** (*Integral, optional*) – The number of states in the hidden Markov model. By default 5.

**Returns**
The trained Gaussian Hidden Markov Model.

**Return type**
hmm.GaussianHMM

**property n_fits_hmm: Integral**

Getter for n_fits_hmm.

**property n_iter_hmm: Integral**

Getter for n_iter_hmm.

**property random_seed**

Getter for random_seed.

**sample**(*X: ndarray | None = None*, *random_seed: Integral | None = None*)

Sample from a Markov chain with given transition probabilities.

**Parameters**

- **X** (*Optional[np.ndarray]*) – A 2D NumPy array, where each row represents a summarized block of data. If not provided, the model will be sampled using the data used to fit the model.
- **random_seed** (*Optional[Integral]*) – The seed for the random number generator. If not provided, the random seed used to fit the model will be used.

**Returns**
A tuple containing the start probabilities and transition probabilities of the Markov chain.

**Return type**
Tuple[np.ndarray, np.ndarray]

**class** tsbootstrap.markov_sampler.**MarkovTransitionMatrixCalculator**

MarkovTransitionMatrixCalculator class provides the functionality to calculate the transition matrix for a set of data blocks based on their DTW distances between consecutive blocks.

The transition matrix is normalized to obtain transition probabilities. The underlying assumption is that the data blocks are generated from a Markov chain. In other words, the next block is generated based on the current block and not on any previous blocks.

**__init__**() → None

Initialize the MarkovTransitionMatrixCalculator instance.

**_calculate_dtw_distances**(*blocks*, *eps: float = 1e-5*) → np.ndarray

Calculate the DTW distances between all pairs of blocks.

**calculate_transition_probabilities**(*blocks*) → np.ndarray

Calculate the transition probability matrix based on DTW distances between all pairs of blocks.

### Examples

```
>>> calculator = MarkovTransitionMatrixCalculator()
>>> blocks = [np.random.rand(10, 5) for _ in range(50)]
>>> transition_matrix = calculator.calculate_transition_probabilities(blocks)
```

**static calculate_transition_probabilities**(*blocks*) → ndarray

Calculate the transition probability matrix based on DTW distances between all pairs of blocks.

**Parameters**

**blocks** (*List[np.ndarray]*) – A list of numpy arrays, each of shape (num_timestamps, num_features), representing the time series data blocks.

**Returns**

A transition probability matrix of shape (len(blocks), len(blocks)).

**Return type**

np.ndarray

CHAPTER

# EIGHT

# TIME SERIES MODEL

class tsbootstrap.time_series_model.**TimeSeriesModel**(*X: ndarray*, *y: ndarray | None = None*, *model_type: Literal['ar', 'arima', 'sarima', 'var', 'arch'] = 'ar'*, *verbose: bool = True*)

A class for fitting time series models to data.

property **X: ndarray**

The input data.

**fit**(*order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, ***kwargs*)

Fits a time series model to the input data.

**Parameters**

- **order** (*OrderTypes, optional*) – The order of the model. If not specified, the default order for the selected model type is used.
- ****kwargs** – Additional keyword arguments for the model.

**Return type**
The fitted time series model.

**Raises**
**ValueError** – If an invalid order is specified for the model type.

**fit_ar**(*order=None*, ***kwargs*)

Fits an AR model to the input data.

**Parameters**

- **order** (*Union[int, List[int]], optional*) – The order of the AR model or a list of order to include.
- ****kwargs** –

  **Additional keyword arguments for the AutoReg model, including:**

  - seasonal (bool): Whether to include seasonal terms in the model.
  - period (int): The seasonal period, if using seasonal terms.
  - trend (str): The trend component to include in the model.

**Returns**
The fitted AR model.

**Return type**
AutoRegResultsWrapper

**Raises**
**ValueError** – If an invalid period is specified for seasonal terms or if the maximum allowed lag value is exceeded.

**fit_arch**(*order: int | None = None*, *p: int = 1*, *q: int = 1*, *arch_model_type: Literal['GARCH', 'EGARCH', 'TARCH', 'AGARCH'] = 'GARCH'*, *mean_type: Literal['zero', 'AR'] = 'zero'*, ***kwargs*)

Fits a GARCH, GARCH-M, EGARCH, TARCH, or AGARCH model to the input data.

**Parameters**

- **order** (*int, optional*) – The number of order to include in the AR part of the model.
- **p** (*int, default 1*) – The number of order in the GARCH part of the model.
- **q** (*int, default 1*) – The number of order in the ARCH part of the model.
- **arch_model_type** (*Literal["GARCH", "EGARCH", "TARCH", "AGARCH"], default "GARCH"*) – The type of GARCH model to fit.
- **mean_type** (*Literal["zero", "AR"], default "zero"*) – The type of mean model to use.
- ****kwargs** – Additional keyword arguments for the ARCH model.

**Return type**
The fitted GARCH model.

**Raises**
**ValueError** – If the maximum allowed lag value is exceeded or if an invalid arch_model_type is specified.

**fit_arima**(*order=None*, ***kwargs*)

Fits an ARIMA model to the input data.

**Parameters**

- **order** (*Tuple[int, int, int], optional*) – The order of the ARIMA model (p, d, q).
- ****kwargs** –

  **Additional keyword arguments for the ARIMA model, including:**

  - seasonal (bool): Whether to include seasonal terms in the model.
  - period (int): The seasonal period, if using seasonal terms.
  - trend (str): The trend component to include in the model.

**Returns**
The fitted ARIMA model.

**Return type**
ARIMAResultsWrapper

**Raises**
**ValueError** – If an invalid period is specified for seasonal terms or if the maximum allowed lag value is exceeded.

### Notes

The ARIMA model is fit using the statsmodels implementation. The default solver is 'lbfgs' and the default optimization method is 'css'. The default maximum number of iterations is 50. These values can be changed by passing the appropriate keyword arguments to the fit method.

**fit_sarima**(*order=None*, *arima_order=None*, ***kwargs*)

Fits a SARIMA model to the input data.

**Parameters**

- **order** (*Tuple[int, int, int, int], optional*) – The order of the SARIMA model (p, d, q, s).
- **arima_order** (*Tuple[int, int, int], optional*) – The order of the ARIMA model (p, d, q). If not specified, the first three elements of order are used.
- ****kwargs** –

  **Additional keyword arguments for the SARIMA model, including:**

  - seasonal (bool): Whether to include seasonal terms in the model.
  - period (int): The seasonal period, if using seasonal terms.
  - trend (str): The trend component to include in the model.

**Returns**
The fitted SARIMA model.

**Return type**
SARIMAXResultsWrapper

**Raises**
**ValueError** – If an invalid period is specified for seasonal terms or if the maximum allowed lag value is exceeded.

### Notes

The SARIMA model is fit using the statsmodels implementation. The default solver is 'lbfgs' and the default optimization method is 'css'. The default maximum number of iterations is 50. These values can be changed by passing the appropriate keyword arguments to the fit method.

**fit_var**(*order: int | None = None*, ***kwargs*)

Fits a Vector Autoregression (VAR) model to the input data.

**Parameters**

- **order** (*int, optional*) – The number of order to include in the VAR model.
- ****kwargs** – Additional keyword arguments for the VAR model.

**Raises**
**ValueError** – If the maximum allowed lag value is exceeded.

**Returns**
The fitted VAR model.

**Return type**
VARResultsWrapper

### Notes

The VAR model is fit using the statsmodels implementation. The default solver is 'bfgs' and the default optimization method is 'css'. The default maximum number of iterations is 50. These values can be changed by passing the appropriate keyword arguments to the fit method.

**property model_type: Literal['ar', 'arima', 'sarima', 'var', 'arch']**

The type of model to fit.

**property verbose: int**

The verbosity level controlling suppression.

Verbosity levels: - 2: No suppression (default) - 1: Suppress stdout only - 0: Suppress both stdout and stderr

**Returns**
The verbosity level.

**Return type**
int

**property y: ndarray | None**

Optional array of exogenous variables.

CHAPTER

# NINE

# TIME SERIES SIMULATOR

class tsbootstrap.time_series_simulator.**TimeSeriesSimulator**(*fitted_model*, *X_fitted: ndarray*, *rng=None*)

Class to simulate various types of time series models.

**n_samples**

Number of samples in the fitted time series model.

**Type**
int

**n_features**

Number of features in the fitted time series model.

**Type**
int

**burnin**

Number of burn-in samples to discard for certain models.

**Type**
int

**_validate_ar_simulation_params**(*params*)

Validate the parameters necessary for the simulation.

**_simulate_ar_residuals**(*lags*, *coefs*, *init*, *max_lag*)

Simulates an Autoregressive (AR) process with given lags, coefficients, initial values, and random errors.

**simulate_ar_process**(*resids_lags*, *resids_coefs*, *resids*)

Simulate AR process from the fitted model.

**_simulate_non_ar_residuals**()

Simulate residuals according to the model type.

**simulate_non_ar_process**()

Simulate a time series from the fitted model.

**generate_samples_sieve**(*model_type*, *resids_lags*, *resids_coefs*, *resids*)

Generate a bootstrap sample using the sieve bootstrap.

**property X_fitted: ndarray**

Get the array of fitted values.

**property fitted_model**

Get the fitted model.

**generate_samples_sieve**(*model_type: Literal['ar', 'arima', 'sarima', 'var', 'arch']*, *resids_lags: Integral | List[Integral] | None = None*, *resids_coefs: ndarray | None = None*, *resids: ndarray | None = None*) → ndarray

Generate a bootstrap sample using the sieve bootstrap.

**Parameters**

- **model_type** (*ModelTypes*) – The model type used for the simulation.
- **resids_lags** (*Optional[Union[Integral, List[Integral]]], optional*) – The lags to be used in the AR process. Can be non-consecutive.
- **resids_coefs** (*Optional[np.ndarray], optional*) – The coefficients corresponding to each lag. Of shape (1, len(lags)).
- **resids** (*Optional[np.ndarray], optional*) – The initial values for the simulation. Should be at least as long as the maximum lag.

**Returns**
The bootstrap sample.

**Return type**
np.ndarray

**Raises**
**ValueError** – If *resids_lags*, *resids_coefs*, or *resids* are not provided.

**property rng**

Get the random number generator instance.

**simulate_ar_process**(*resids_lags: Integral | List[Integral]*, *resids_coefs: ndarray*, *resids: ndarray*) → ndarray

Simulate AR process from the fitted model.

**Parameters**

- **resids_lags** (*Union[Integral, List[Integral]]*) – The lags to be used in the AR process. Can be non-consecutive, but when called from *generate_samples_sieve*, it will be sorted.
- **resids_coefs** (*np.ndarray*) – The coefficients corresponding to each lag. Of shape (1, len(lags)). Sorted by *generate_samples_sieve* corresponding to the sorted *lags*.
- **resids** (*np.ndarray*) – The initial values for the simulation. Should be at least as long as the maximum lag.

**Returns**
The simulated AR process as a 1D NumPy array.

**Return type**
np.ndarray

**Raises**

- **ValueError** – If *resids_lags*, *resids_coefs*, or *resids* are not provided. If *resids_coefs* is not a 1D NumPy array. If *resids_coefs* is not the same length as *resids_lags*. If *resids* is not the same length as *X_fitted*.
- **TypeError** – If *fitted_model* is not an instance of *AutoRegResultsWrapper*. If *resids_lags* is not an integer or a list of integers.

`simulate_non_ar_process()` → ndarray

Simulate a time series from the fitted model.

**Returns**
**np.ndarray**

**Return type**
The simulated time series.

CHAPTER

# TEN

# TSFIT

class tsbootstrap.tsfit.**TSFit**(*order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral]*, *model_type: Literal['ar', 'arima', 'sarima', 'var', 'arch']*, ***kwargs*)

Performs fitting for various time series models including 'ar', 'arima', 'sarima', 'var', and 'arch'.

**rescale_factors**

Rescaling factors for the input data and exogenous variables.

**Type**
dict

**model**

The fitted model.

**Type**
Union[AutoRegResultsWrapper, ARIMAResultsWrapper, SARIMAXResultsWrapper, VARResultsWrapper, ARCHModelResult]

**fit**(*X*, *y=None*)

Fit the chosen model to the data.

**get_coefs**()

Return the coefficients of the fitted model.

**get_intercepts**()

Return the intercepts of the fitted model.

**get_residuals**()

Return the residuals of the fitted model.

**get_fitted_X**()

Return the fitted values of the model.

**get_order**()

Return the order of the fitted model.

**predict**(*X*, *n_steps=1*)

Predict future values using the fitted model.

**score**(*X*, *y_true*)

Compute the R-squared score for the fitted model.

**Raises**
**ValueError** – If the model type or the model order is invalid.

### Notes

The following table shows the valid model types and their corresponding orders.

| Model | Valid orders | Invalid orders |
|---|---|---|
| 'ar' | int | list, tuple |
| 'arima' | tuple of length 3 | int, list, tuple |
| 'sarima'\| tuple of length 4 | | int, list, tuple |
| 'var' | int | list, tuple |
| 'arch' | int | list, tuple |

### Examples

```
>>> from tsbootstrap import TSFit
>>> import numpy as np
>>> X = np.random.normal(size=(100, 1))
>>> fit_obj = TSFit(order=2, model_type='ar')
>>> fit_obj.fit(X)
TSFit(order=2, model_type='ar')
>>> fit_obj.get_coefs()
array([[ 0.003, -0.002]])
>>> fit_obj.get_intercepts()
array([0.001])
>>> fit_obj.get_residuals()
array([[ 0.001],
          [-0.002],
            [-0.002],
                [-0.002],
                    [-0.002], ...
>>> fit_obj.get_fitted_X()
array([[ 0.001],
            [-0.002],
                [-0.002],
                    [-0.002],
                        [-0.002], ...
>>> fit_obj.get_order()
2
>>> fit_obj.predict(X, n_steps=5)
array([[ 0.001],
            [-0.002],
                [-0.002],
                    [-0.002],
                        [-0.002], ...
>>> fit_obj.score(X, X)
0.999
```

**fit**(*X: ndarray*, *y=None*) → TSFit

Fit the chosen model to the data.

**Parameters**

- **X** (*np.ndarray*) – Input data of shape (n_timepoints, n_features).

- **y** (*np.ndarray, optional*) – Exogenous variables, by default None.

**Returns**
The fitted TSFit object.

**Return type**
TSFit

**Raises**

- **ValueError** – If the model type or the model order is invalid.
- **RuntimeError** – If the maximum number of iterations is reached before the variance is within the desired range.

**get_coefs**() → ndarray

Return the coefficients of the fitted model.

**Returns**
The coefficients of the fitted model.

**Return type**
np.ndarray

**Raises**
**NotFittedError** – If the model is not fitted.

### Notes

The shape of the coefficients depends on the model type.

| Model | Coefficient shape |
|---|---|
| 'ar' | (1, order) |
| 'arima' | (1, order) |
| 'sarima'\| | (1, order) |
| 'var' | (n_features, n_features, order) |
| 'arch' | (1, order) |

**get_fitted_X**() → ndarray

Return the fitted values of the model.

**Returns**
The fitted values of the model.

**Return type**
np.ndarray

**Raises**
**NotFittedError** – If the model is not fitted.

### Notes

The shape of the fitted values depends on the model type.

| Model | Fitted values shape |
|---|---|
| 'ar' | (n, 1) |
| 'arima' | (n, 1) |
| 'sarima' | (n, 1) |
| 'var' | (n, k) |
| 'arch' | (n, 1) |

`get_intercepts()` → ndarray

Return the intercepts of the fitted model.

**Returns**
The intercepts of the fitted model.

**Return type**
np.ndarray

**Raises**
`NotFittedError` – If the model is not fitted.

### Notes

The shape of the intercepts depends on the model type.

| Model | Intercept shape |
|---|---|
| 'ar' | (1, trend_terms) |
| 'arima' | (1, trend_terms) |
| 'sarima' | (1, trend_terms) |
| 'var' | (n_features, trend_terms) |
| 'arch' | (0,) |

`get_order()` → Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral]

Return the order of the fitted model.

**Returns**
The order of the fitted model.

**Return type**
OrderTypesWithoutNone

**Raises**
`NotFittedError` – If the model is not fitted.

### Notes

The shape of the order depends on the model type.

| Model | Order shape |
|---|---|
| 'ar' | int |
| 'arima' | tuple of length 3 |
| 'sarima' | tuple of length 4 |
| 'var' | int |
| 'arch' | int |

**get_params**(*deep=True*)

Get parameters for this estimator.

**Parameters**
**deep** (*bool, optional*) – When set to True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns**
Parameter names mapped to their values.

**Return type**
dict

**get_residuals**() → ndarray

Return the residuals of the fitted model.

**Returns**
The residuals of the fitted model.

**Return type**
np.ndarray

**Raises**
**NotFittedError** – If the model is not fitted.

### Notes

The shape of the residuals depends on the model type.

| Model | Residual shape |
|---|---|
| 'ar' | (n, 1) |
| 'arima' | (n, 1) |
| 'sarima' | (n, 1) |
| 'var' | (n, k) |
| 'arch' | (n, 1) |

**property model_type: str**

The type of the model.

**property order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral]**

The order of the model.

**predict**(*X: ndarray*, *y=None*, *n_steps: int = 1*) → ndarray

Predict time series values using the fitted model.

**Parameters**

- **X** (*np.ndarray*) – Input data of shape (n_timepoints, n_features).
- **y** (*np.ndarray, optional*) – Exogenous variables, by default None.
- **n_steps** (*int, optional*) – Number of steps to forecast, by default 1.

**Returns**
Predicted values.

**Return type**
np.ndarray

**Raises**
**RuntimeError** – If the model is not fitted.

**score**(*X: ndarray*, *y_true: ndarray*) → float

Compute the R-squared score for the fitted model.

**Parameters**

- **X** (*np.ndarray*) – The input data.
- **y_true** (*np.ndarray*) – The true values.

**Returns**
The R-squared score.

**Return type**
float

**Raises**

- **NotFittedError** – If the model is not fitted.
- **ValueError** – If the number of lags is greater than the length of the input data.

**set_params**(***params*)

Set the parameters of this estimator.

**Parameters**
****params** – Estimator parameters.

**set_predict_request**(**, n_steps: bool | None | str = '$UNCHANGED$'*) → TSFit

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

---

**Note:** This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

---

**Parameters**

**n_steps** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `n_steps` parameter in `predict`.

**Returns**

**self** – The updated object.

**Return type**

object

**set_score_request**(*, *y_true: bool | None | str = '$UNCHANGED$'*) → TSFit

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

---

**Note:** This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

---

**Parameters**

**y_true** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `y_true` parameter in `score`.

**Returns**

**self** – The updated object.

**Return type**

object

class tsbootstrap.tsfit.**TSFitBestLag**(*model_type: str*, *max_lag: int = 10*, *order: Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral] | None = None*, *save_models=False*, ***kwargs*)

A class used to fit time series data and find the best lag for forecasting.

**rank_lagger**

An instance of the RankLags class.

**Type**
RankLags

**ts_fit**

An instance of the TSFit class.

**Type**
TSFit

**model**

The fitted time series model.

**Type**
Union[AutoRegResultsWrapper, ARIMAResultsWrapper, SARIMAXResultsWrapper, VARResultsWrapper, ARCHModelResult]

**rescale_factors**

The rescaling factors used for the input data and exogenous variables.

**Type**
Dict[str, Union[float, List[float] | None]]

**fit**(*X*, *y=None*)

Fit the time series model to the data.

**get_coefs**()

Return the coefficients of the fitted model.

**get_intercepts**()

Return the intercepts of the fitted model.

**get_residuals**()

Return the residuals of the fitted model.

**get_fitted_X**()

Return the fitted values of the model.

**get_order**()

Return the order of the fitted model.

**get_model**()

Return the fitted time series model.

**predict**(*X*, *n_steps=1*)

Predict future values using the fitted model.

**score**(*X*, *y_true*)

Compute the R-squared score for the fitted model.

**fit**(*X: ndarray*, *y=None*)

Fit the time series model to the data.

**Parameters**

- **X** (*np.ndarray*) – The input data.
- **y** (*np.ndarray, optional, default=None*) – Exogenous variables to include in the model.

**Returns**
The fitted model.

**Return type**
self

**get_coefs**() → ndarray

Return the coefficients of the fitted model.

**Returns**
The coefficients of the fitted model.

**Return type**
np.ndarray

**get_fitted_X**() → ndarray

Return the fitted values of the model.

**Returns**
The fitted values of the model.

**Return type**
np.ndarray

**get_model**()

Return the fitted time series model.

**Returns**
The fitted time series model.

**Return type**
Union[AutoRegResultsWrapper, ARIMAResultsWrapper, SARIMAXResultsWrapper, VARResultsWrapper, ARCHModelResult]

**Raises**
**ValueError** – If models were not saved during initialization.

**get_order**() → Integral | List[Integral] | tuple[Integral, Integral, Integral] | tuple[Integral, Integral, Integral, Integral]

Return the order of the fitted model.

**Returns**
The order of the fitted model.

**Return type**
int, List[int], Tuple[int, int, int], Tuple[int, int, int, int]

**get_residuals**() → ndarray

Return the residuals of the fitted model.

**Returns**
The residuals of the fitted model.

**Return type**
np.ndarray

**predict**(*X: ndarray*, *n_steps: int = 1*)

Predict future values using the fitted model.

**Parameters**

- **X** (*np.ndarray*) – The input data.
- **n_steps** (*int, optional, default=1*) – The number of steps to predict.

**Returns**
The predicted values.

**Return type**
np.ndarray

**score**(*X: ndarray*, *y_true: ndarray*)

Compute the R-squared score for the fitted model.

**Parameters**

- **X** (*np.ndarray*) – The input data.
- **y_true** (*np.ndarray*) – The true values of the target variable.

**Returns**
The R-squared score.

**Return type**
float

**set_predict_request**(*, *n_steps: bool | None | str = '$UNCHANGED$'*) → TSFitBestLag

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

---

**Note:** This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

---

**Parameters**
**n_steps** (*str, True, False, or None, default=sklearn.utils.*

*metadata_routing.UNCHANGED*) – Metadata routing for `n_steps` parameter in `predict`.

**Returns**
**self** – The updated object.

**Return type**
object

**set_score_request**(*, *y_true: bool | None | str = '$UNCHANGED$'*) → TSFitBestLag

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

---

**Note:** This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

---

**Parameters**
**y_true** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `y_true` parameter in `score`.

**Returns**
**self** – The updated object.

**Return type**
object

CHAPTER

# ELEVEN

# ODDS AND ENDS

tsbootstrap.utils.odds_and_ends.**assert_arrays_compare**(*a: ndarray*, *b: ndarray*, *rtol=1e-05*, *atol=1e-08*, *check_same=True*) → bool

Assert that two arrays are almost equal.

This function compares two arrays for equality, allowing for NaNs and Infs in the arrays. The arrays are considered equal if the following conditions are satisfied: 1. The locations of NaNs and Infs in both arrays are the same. 2. The signs of the infinite values in both arrays are the same. 3. The finite values are almost equal.

**Parameters**

- **a** (*np.ndarray*) – The arrays to be compared.
- **b** (*np.ndarray*) – The arrays to be compared.
- **rtol** (*float, optional*) – The relative tolerance parameter for the np.allclose function. Default is 1e-5.
- **atol** (*float, optional*) – The absolute tolerance parameter for the np.allclose function. Default is 1e-8.
- **check_same** (*bool, optional*) – If True, raise an AssertionError if the arrays are not almost equal. If False, return True if the arrays are not almost equal and False otherwise. Default is True.

**Returns**

If check_same is False, returns True if the arrays are not almost equal and False otherwise. If check_same is True, returns True if the arrays are almost equal and False otherwise.

**Return type**

bool

**Raises**

- **AssertionError** – If check_same is True and the arrays are not almost equal.
- **ValueError** – If check_same is True and the arrays have NaNs or Infs in different locations. If check_same is True and the arrays have Infs with different signs.

tsbootstrap.utils.odds_and_ends.**check_generator**(*seed_or_rng: Generator | Integral | None*, *seed_allowed: bool = True*) → Generator

Turn seed into a np.random.Generator instance.

**Parameters**

- **seed_or_rng** (*int, Generator, or None*) – If seed_or_rng is None, return the Generator singleton used by np.random. If seed_or_rng is an int, return a new Generator instance seeded with seed_or_rng. If seed_or_rng is already a Generator instance, return it. Otherwise raise ValueError.

- **seed_allowed** (*bool, optional*) – If True, seed_or_rng can be an int. If False, seed_or_rng cannot be an int. Default is True.

**Returns**
A numpy.random.Generator instance.

**Return type**
Generator

**Raises**
**ValueError** – If seed_or_rng is not None, an int, or a numpy.random.Generator instance. If seed_or_rng is an int and seed_allowed is False. If seed_or_rng is an int and it is not between 0 and 2**32 - 1.

tsbootstrap.utils.odds_and_ends.**generate_random_indices**(*num_samples: Integral*, *rng: Generator | Integral | None = None*) → ndarray

Generate random indices with replacement.

This function generates random indices from 0 to *num_samples-1* with replacement. The generated indices can be used for bootstrap sampling, etc.

**Parameters**

- **num_samples** (*Integral*) – The number of samples for which the indices are to be generated. This must be a positive integer.
- **rng** (*Integral, optional*) – The seed for the random number generator. If provided, this must be a non-negative integer. Default is None, which does not set the numpy's random seed and the results will be non-deterministic.

**Returns**
A numpy array of shape (*num_samples*,) containing randomly generated indices.

**Return type**
np.ndarray

**Raises**
**ValueError** – If *num_samples* is not a positive integer or if *random_seed* is provided and it is not a non-negative integer.

### Examples

```
>>> generate_random_indices(5, random_seed=0)
array([4, 0, 3, 3, 3])
>>> generate_random_indices(5)
array([2, 1, 4, 2, 0])  # random
```

tsbootstrap.utils.odds_and_ends.**suppress_output**(*verbose: int = 2*)

A context manager for controlling the suppression of stdout and stderr.

**Parameters**
**verbose** (*int, optional*) – Verbosity level controlling suppression. 2 - No suppression (default) 1 - Suppress stdout only 0 - Suppress both stdout and stderr

**Return type**
None

### Examples

**with suppress_output(verbose=1):**
print('This will not be printed to stdout')

tsbootstrap.utils.odds_and_ends.**time_series_split**(*X: ndarray*, *test_ratio: float*)

Splits a given time series into training and test sets.

**Parameters**

- **X** (*np.ndarray*) – The input time series.
- **test_ratio** (*float*) – The ratio of the test set size to the total size of the series.

**Returns**
A tuple containing the training set and the test set.

**Return type**
Tuple[np.ndarray, np.ndarray]

CHAPTER

# TWELVE

# TYPES

tsbootstrap.utils.types.**FittedModelTypes**() → tuple

Return a tuple of fitted model types for use in isinstance checks.

**Returns**
**tuple**

**Return type**
A tuple containing the result wrapper types for fitted models.

CHAPTER

# THIRTEEN

# VALIDATE

tsbootstrap.utils.validate.**add_newaxis_if_needed**(*X: ndarray*, *model_is_var: bool*) → ndarray

Add a new axis to the given array if it's needed.

tsbootstrap.utils.validate.**check_are_1d_integer_arrays**(*input_list*, *input_name: str*)

Check if all NumPy arrays in the input list are 1D and contain integer values.

tsbootstrap.utils.validate.**check_are_2d_arrays**(*input_list*, *input_name: str*)

Check if all NumPy arrays in the input list are 2D.

tsbootstrap.utils.validate.**check_are_finite**(*input_list*, *input_name: str*)

Check if all elements in the NumPy arrays in the input list are finite.

tsbootstrap.utils.validate.**check_are_nonnegative**(*input_array: ndarray*, *input_name: str*) → ndarray

Check if all elements in the input NumPy array are nonnegative.

tsbootstrap.utils.validate.**check_are_np_arrays**(*input_list*, *input_name: str*)

Check if all elements in the input list are NumPy arrays.

tsbootstrap.utils.validate.**check_are_real**(*input_array: ndarray*, *input_name: str*) → ndarray

Check if all elements in the input NumPy array are real.

tsbootstrap.utils.validate.**check_array_shape**(*X: ndarray*, *model_is_var: bool*, *allow_multi_column: bool*) → ndarray

Check if the given array meets the required shape constraints.

**Parameters**

- **X** (*np.ndarray*) – The input array to be checked.
- **model_is_var** (*bool*) – Flag indicating if the model is a VAR (Vector Autoregression) model.
- **allow_multi_column** (*bool*) – Flag indicating if multiple columns are allowed in the array.

**Returns**

The original array if it meets the constraints.

**Return type**

np.ndarray

**Raises**

**ValueError** – If the array does not meet the required shape constraints.

### Examples

```
>>> check_array_shape(np.array([[1, 2], [3, 4]]), True, True)
array([[1, 2], [3, 4]])
```

```
>>> check_array_shape(np.array([1, 2, 3]), False, False)
array([1, 2, 3])
```

tsbootstrap.utils.validate.**check_array_size**(*X: ndarray*) → ndarray

Check if the given array contains at least two elements.

tsbootstrap.utils.validate.**check_array_type**(*X: ndarray*) → ndarray

Check if the given array is a NumPy array of floats.

tsbootstrap.utils.validate.**check_have_at_least_one_element**(*input_list*, *input_name: str*)

Check if all NumPy arrays in the input list have at least one element.

tsbootstrap.utils.validate.**check_have_at_least_one_feature**(*input_list*, *input_name: str*)

Check if all NumPy arrays in the input list have at least one feature.

tsbootstrap.utils.validate.**check_have_at_least_one_index**(*input_list*, *input_name: str*)

Check if all NumPy arrays in the input list have at least one index.

tsbootstrap.utils.validate.**check_have_same_num_of_features**(*input_list*, *input_name: str*)

Check if all NumPy arrays in the input list have the same number of features.

tsbootstrap.utils.validate.**check_indices_within_range**(*input_list*, *input_length: Integral*, *input_name: str*)

Check if all indices in the NumPy arrays in the input list are within the range of the input length.

tsbootstrap.utils.validate.**check_is_1d_or_2d_single_column**(*input_array: ndarray*, *input_name: str*) → ndarray

Check if the input NumPy array is a 1D array or a 2D array with a single column.

tsbootstrap.utils.validate.**check_is_finite**(*input_array: ndarray*, *input_name: str*) → ndarray

Check if all elements in the input NumPy array are finite.

tsbootstrap.utils.validate.**check_is_list**(*input_list: list*, *input_name: str*) → list

Check if the input is a list.

tsbootstrap.utils.validate.**check_is_nonempty**(*input_list: list*, *input_name: str*) → list

Check if the input list is nonempty.

tsbootstrap.utils.validate.**check_is_not_all_zero**(*input_array: ndarray*, *input_name: str*) → ndarray

Check if the input NumPy array is not all zeros.

tsbootstrap.utils.validate.**check_is_np_array**(*input_array: ndarray*, *input_name: str*) → ndarray

Check if the input is a NumPy array.

tsbootstrap.utils.validate.**validate_X**(*X: ndarray*, *model_is_var: bool*, *allow_multi_column: bool | None = None*) → ndarray

Validate the input array X based on the given model type.

**Parameters**

- **X** (*np.ndarray*) – The input array to be validated. It must be a NumPy array of floats (i, u, or f type).

- **model_is_var** (*bool*) – A flag to determine whether the model is of VAR (Vector Autoregression) type. If True, the function will validate it as a VAR array. If False, the function will validate it as a non-VAR array.
- **allow_multi_column** (*bool, optional*) – A flag to determine whether the array is allowed to have more than one column. If not specified, it defaults to the value of *model_is_var*.

**Returns**
A validated array.

**Return type**
np.ndarray

**Raises**

- **TypeError** – If X is not a NumPy array or its data type is not float.
- **ValueError** – If X contains fewer than two elements, or does not meet the dimensionality requirements.

tsbootstrap.utils.validate.**validate_X_and_y**(*X: ndarray*, *y: ndarray | None*, *model_is_var: bool = False*, *model_is_arch: bool = False*)

Validate and reshape input data and exogenous variables.

This function uses `validate_X()` and `validate_exog()` to perform detailed validation.

**Parameters**

- **X** (*np.ndarray*) – The input array to be validated.
- **y** (*Optional[np.ndarray]*) – The exogenous variable array to be validated. Can be None.
- **model_is_var** (*bool, optional*) – A flag to determine if the model is of VAR type. Default is False.
- **model_is_arch** (*bool, optional*) – A flag to determine if the model is of ARCH type. Default is False.

**Returns**
A tuple containing the validated X array and optionally the validated exog array.

**Return type**
Tuple[np.ndarray, Optional[np.ndarray]]

**See also:**

**validate_X**
Function for validating the input array X.

**validate_exog**
Function for validating the exogenous variable array.

tsbootstrap.utils.validate.**validate_block_indices**(*block_indices: List[ndarray]*, *input_length: Integral*) → None

Validate the input block indices. Each block index must be a 1D NumPy array with at least one index and all indices must be within the range of X.

**Parameters**

- **block_indices** (*List[np.ndarray]*) – The input block indices.
- **input_length** (*Integral*) – The length of the input data.

**Raises**

- **TypeError** – If block_indices is not a list or if it contains non-NumPy arrays.
- **ValueError** – If block_indices is empty or if it contains NumPy arrays with non-integer values, or if it contains NumPy arrays with no indices, or if it contains NumPy arrays with indices outside the range of X.

tsbootstrap.utils.validate.**validate_blocks**(*blocks: List[ndarray]*) → None

Validate the input blocks. Each block must be a 2D NumPy array with at least one element.

**Parameters**
**blocks** (*List[np.ndarray]*) – The input blocks.

**Raises**

- **TypeError** – If blocks is not a list or if it contains non-NumPy arrays.
- **ValueError** – If blocks is empty or if it contains NumPy arrays with non-finite values, or if it contains NumPy arrays with no elements, or if it contains NumPy arrays with no features, or if it contains NumPy arrays with different number of features.

tsbootstrap.utils.validate.**validate_exog**(*exog: ndarray*) → ndarray

Validate the exogenous variable array *exog*, ensuring its dimensionality and dtype.

**Parameters**
**exog** (*np.ndarray*) – The exogenous variable array to be validated. Must be a NumPy array of floats.

**Returns**
A validated exogenous variable array.

**Return type**
np.ndarray

**Raises**

- **TypeError** – If *exog* is not a NumPy array or its data type is not float.
- **ValueError** – If *exog* contains fewer than two elements.

tsbootstrap.utils.validate.**validate_fitted_model**(*fitted_model*) → None

Validate the input fitted model. It must be an instance of a fitted model class.

**Parameters**
**fitted_model** (*FittedModelTypes*) – The input fitted model.

**Raises**
**TypeError** – If fitted_model is not an instance of a fitted model class.

tsbootstrap.utils.validate.**validate_integer_array**(*value: ndarray*, *min_value: Integral | None = None*, *max_value: Integral | None = None*) → None

Validate a 1D numpy array of integers against an optional minimum value.

tsbootstrap.utils.validate.**validate_integers**(**values*, *min_value: Integral | None = None*, *max_value: Integral | None = None*) → None

Validates that all input values are integers and optionally, above a minimum value.

Each value can be an integer, a list of integers, or a 1D numpy array of integers. If min_value is provided, all integers must be greater than or equal to min_value.

**Parameters**

- ***values** (*Union[Integral, List[Integral], np.ndarray]*) – One or more values to validate.
- **min_value** (*Integral, optional*) – If provided, all integers must be greater than or equal to min_value.
- **max_value** (*Integral, optional*) – If provided, all integers must be less than or equal to max_value.

**Raises**

**TypeError** – If a value is not an integer, list of integers, or 1D array of integers, or if any integer is less than min_value or greater than max_value.

tsbootstrap.utils.validate.**validate_list_of_integers**(*value*, *min_value: Integral | None = None*, *max_value: Integral | None = None*) → None

Validate a list of integer values against an optional minimum value.

tsbootstrap.utils.validate.**validate_literal_type**(*input_value: str*, *literal_type: Any*) → None

Validate the type of *input_value* against a Literal type or dictionary keys.

**Parameters**

- **input_value** (*str*) – The value to validate.
- **literal_type** (*type, or list*) – if type: Literal type or dictionary against which to validate the *input_value*. if list: list of valid values against which to validate the *input_value*.

**Raises**

- **TypeError** – If *input_value* is not a string.
- **ValueError** – If *input_value* is not among the valid types in *literal_type* or dictionary keys.

### Examples

```
>>> validate_literal_type("a", Literal["a", "b", "c"])
>>> validate_literal_type("x", {"x": 1, "y": 2})
>>> validate_literal_type("z", Literal["a", "b", "c"])
ValueError: Invalid input_value 'z'. Expected one of 'a', 'b', 'c'.
>>> validate_literal_type("z", {"x": 1, "y": 2})
ValueError: Invalid input_value 'z'. Expected one of 'x', 'y'.
```

tsbootstrap.utils.validate.**validate_order**(*order*) → None

Validates the type of the resids_order order.

**Parameters**

**order** (*Any*) – The order to validate.

**Raises**

- **TypeError** – If the order is not of the expected type (Integral, list, or tuple).
- **orderError** – If the order is an integral but is negative. If the order is a list/tuple and not all elements are positive integers.

tsbootstrap.utils.validate.**validate_rng**(*rng: Generator | Integral | None*, *allow_seed: bool = True*) → Generator

Validate the input random number generator.

This function validates if the input random number generator is an instance of the numpy.random.Generator class or an integer. If allow_seed is True, the input can also be an integer, which will be used to seed the default random number generator.

**Parameters**

- **rng** (*RngTypes*) – The input random number generator.
- **allow_seed** (*bool, optional*) – Whether to allow the input to be an integer. Default is True.

**Returns**
The validated random number generator.

**Return type**
Generator

**Raises**

- **TypeError** – If rng is not an instance of the numpy.random.Generator class or an integer.
- **ValueError** – If rng is an integer and it is negative or greater than or equal to 2**32.

tsbootstrap.utils.validate.**validate_single_integer**(*value: Integral*, *min_value: Integral | None = None*, *max_value: Integral | None = None*) → None

Validate a single integer value against an optional minimum value.

tsbootstrap.utils.validate.**validate_weights**(*weights: ndarray*) → None

Validate the input weights. Each weight must be a non-negative finite value.

**Parameters**
**weights** (*np.ndarray*) – The input weights.

**Raises**

- **TypeError** – If weights is not a NumPy array.
- **ValueError** – If weights contains any non-finite values, or if it contains any negative values, or if it contains any complex values, or if it contains all zeros, or if it is a 2D array with more than one column.

CHAPTER

# FOURTEEN

# RANKLAGS

**class** tsbootstrap.ranklags.**RankLags**(*X: ndarray*, *model_type: Literal['ar', 'arima', 'sarima', 'var', 'arch']*, *max_lag: Integral = 10*, *y=None*, *save_models: bool = False*)

A class that uses several metrics to rank lags for time series models.

**rank_lags_by_aic_bic**()

Rank lags based on Akaike information criterion (AIC) and Bayesian information criterion (BIC).

**rank_lags_by_pacf**()

Rank lags based on Partial Autocorrelation Function (PACF) values.

**estimate_conservative_lag**()

Estimate a conservative lag value by considering various metrics.

**get_model**(*order*)

Retrieve a previously fitted model given an order.

### Examples

```
>>> from tsbootstrap import RankLags
>>> import numpy as np
>>> X = np.random.normal(size=(100, 1))
>>> rank_obj = RankLags(X, model_type='ar')
>>> rank_obj.estimate_conservative_lag()
2
>>> rank_obj.rank_lags_by_aic_bic()
(array([2, 1]), array([2, 1]))
>>> rank_obj.rank_lags_by_pacf()
array([1, 2])
```

**property X: ndarray**

The input data.

**Returns**

The input data.

**Return type**

np.ndarray

**estimate_conservative_lag**() → int

Estimate a conservative lag value by considering various metrics.

**Returns**

A conservative lag value.

**Return type**
int

`get_model`(*order: int*)

Retrieve a previously fitted model given an order.

**Parameters**
**order** (*int*) – Order of the model to retrieve.

**Returns**
The fitted model.

**Return type**
Union[AutoRegResultsWrapper, ARIMAResultsWrapper, SARIMAXResultsWrapper, VARResultsWrapper, ARCHModelResult]

`property max_lag: Integral`

Maximum lag to consider.

**Returns**
Maximum lag to consider.

**Return type**
int

`property model_type: Literal['ar', 'arima', 'sarima', 'var', 'arch']`

The type of model to fit.

**Returns**
The type of model to fit.

**Return type**
str

`rank_lags_by_aic_bic`()

Rank lags based on Akaike information criterion (AIC) and Bayesian information criterion (BIC).

**Returns**
aic_ranked_lags: Lags ranked by AIC. bic_ranked_lags: Lags ranked by BIC.

**Return type**
Tuple[np.ndarray, np.ndarray]

`rank_lags_by_pacf`() → ndarray

Rank lags based on Partial Autocorrelation Function (PACF) values.

**Returns**
Lags ranked by PACF values.

**Return type**
np.ndarray

`property y: ndarray`

Exogenous variables to include in the model.

**Returns**
Exogenous variables to include in the model.

**Return type**
np.ndarray

CHAPTER

FIFTEEN

# INDICES AND TABLES

- genindex
- modindex
- search