CompressedFisher

Release 0.1

William Coulton

May 17, 2023

CONTENTS:

Compressed Fisher is a Python library containing tools for testing if Fisher forecasts using simulated components are converged. The library contains tools to compute standard Fisher estimates, estimate the level of bias due to the finite number of simulations, and compute the compressed Fisher information, as described Coulton & Wandelt (2022) and Coulton et al (2022b).

Note: Please report any requests via a github issue

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

CHAPTER

ONE

USAGE

1.1 Installation

To use CompressedFisher, first install it using pip:

```
$ pip install CompressedFisher
```

It requires the *numpy* and *scipy* packages.

1.2 Basic Usage

Typical usage of the code requires two ensembles of simulations: one set of simulations is given at the fiducial parameters (θ) and is used to estimate the covariance matrix. The second is a set of simulated derivatives; these can either be in the form of realizations of the derivatives themselves or simulations evaluate at a set of point in the neighborhood of the fiducial point that the code can use to estimate the derivatives (e.g. simulations at parameter points $\theta + \delta\theta_i$ and $\theta - \delta\theta_i$ where $\delta\theta_i$ is a small step in parameter, i .)

Here we sketch a potential workflow, with detailed examples available in the notebooks described in the *examples* section.

1. Choose the appropriate class based on the distribution of the data. Currently supported cases are:

- gaussianFisher
- poissonFisher

1. Provide the code with the two sets of simulations (one for the covariance, rates etc and the second set for the derivatives)
2. Choose a division of the simulations between the compression and Fisher estimation steps (1/2 in each typical works well)
3. Call the *compute_fisher_forecast*, *compute_compressed_fisher_forecast* and *compute_combined_fisher_forecast* to compute standard, compressed and combined Fisher forecasts.
4. There are a range of methods to assess whether these forecasts are converged (including *est_fisher_forecast_bias* and *run_fisher_deriv_stablity_test*). It is important to perform tests like these (and more) to ensure that your forecast constraints are converged. If they are not your parameter inferences will likely overestimate your ability to constrain that parameter.

1.3 Examples

In the *examples* folder of the repository are three examples showing how to used the code. Each considers a different test case: a Gaussian likelihood with a parameter independent mean, a Gaussian likelihood with a parameter dependent mean, and a Poisson case.

These examples show how to use the main functionality of the code. Further details on each method can be found in api

CHAPTER

TWO

API

class CompressedFisher.fisher.**baseFisher**

The base class for Fisher forecasting

This class contains common routines for all the Fisher forecasts including: compressed and combined Fisher forecasts, convergence tests and more.

compute_combined_fisher_forecast(*params_names*)

Computes the combined fisher parameter constraints The inverse of Eq. 18 in Coulton and Wandelt

Args:
params_names ([list]): list of the parameters that you want to included in the Fisher forecast.

Returns:
[np.array([n_params,n_params])]: An matrix or size ([n_params,n_params]) with the forecast Fisher covariance matrix.

compute_combined_fisher_forecast_wShuffle(*params_names*, *compress_fraction*, *nShuffles=10*, *verbose=False*)

Computes the combined fisher parameter constraints and this version uses repeated estimates (with the number given by nShuffles) where for each repeat the sims are shuffled and redivded between the compression and fisher estimation.

This reduces the variance of the estimator, but requires more simulations for it to be unbiased.

Args:
params_names ([list]): list of the parameters that you want to included in the Fisher forecast. compress_fraction (float): the fraction (between 0 and 1) of the simulations to use for the compression nShuffles (int): the number of times to iterate. Each iteration redivides the simulations and reestimates the fisher constraints

Returns:
[np.array([n_params,n_params])]: An matrix or size ([n_params,n_params]) with the forecast Fisher covariance matrix.

compute_compressed_fisher_forecast(*params_names*)

Compute the compressed fisher forecast parameter covariances The inverse of Eq. 10 in Coulton and Wandelt

Args:
params_names ([list]): A list of which parameters to include in the forecast.

Returns:
[matrix]: The compressed forecast parameter covariances

`compute_compressed_fisher_forecast_wShuffle`(*params_names*, *compress_fraction*, *nShuffles=10*, *verbose=False*)

Computes the compressed fisher parameter constraints and this version uses repeated estimates (with the number given by nShuffles) where for each repeat the sims are shuffled and redivded between the compression and fisher estimation.

This reduces the variance of the estimator, but requires more simulations for it to be valid.

Args:
params_names ([list]): list of the parameters that you want to included in the Fisher forecast. compress_fraction (float): the fraction (between 0 and 1) of the simulations to use for the compression nShuffles (int): the number of times to iterate. Each iteration redivides the simulations and reestimates the fisher constraints

Returns:
[np.array([n_params,n_params])]: An matrix or size ([n_params,n_params]) with the forecast Fisher covariance matrix.

`compute_fisher_forecast`(*params_names*)

Compute the standard fisher forecast parameter covariances Eq. 5 in Coulton and Wandelt

Args:
params_names ([list]): A list of which parameters to include in the forecast.

Returns:
[matrix]: The forecast parameter covariances

`est_compressed_fisher_forecast_bias`(*params_names*)

Estimate the bias to the compressed fisher forecast parameter variances. The ratio of this to the compressed fisher parameter *variances* gives a measure of whether there are enough simulations for the forecast to be converged. Computed with Eq. 11 and Eq. 42 in Coulton and Wandelt

Args:
params_names ([list]): Parameters to include in the fisher forecast

Returns:
[matrix]: The bias terms to each element in the fisher forecast.

`est_fisher_forecast_bias`(*params_names*)

Estimate the bias to the standard fisher forecast parameter variances. The ratio of this to the parameter *variances* gives a measure of whether there are enough simulations for the forecast to be converged Computed Eq. 6 and Eq. 42. in Coulton and Wandelt

Args:
params_names ([list]): Parameters to include in the fisher forecast

Returns:
[matrix]: The bias terms to each element in the fisher forecast.

`generate_covmat_sim_splits`(*compress_fraction=None*, *compress_number=None*)

A function to split the covariance matrix simulations between the compression and the fisher evaluation. Typically this is not needed as the 'noise' on the covariance matrix is typically subdominant. Two modes 1) To specify some fraction of the total number of simulations to be used for compression (compress_fraction) 2) To specfiy a specific number of simulations to be used for the compression (compress_number)

Args:
compress_fraction ([float]): For mode 1) The fraction of simulations to be used for the compression(default: *None*) compress_number ([int]): For mode 2) The number of simulations to be used for the compression (default: *None*)

`generate_deriv_sim_splits`(*compress_fraction=None*, *compress_number=None*, *ids_comp=None*, *ids_fish=None*)

Perform the split of the sims into the set for computing the compression and the set for computing the derivatives. There are three different ways. 1) To specify some fraction of the total number of simulations to be used for compression (compress_fraction) 2) To specfiy a specific number of simulations to be used for the compression (compress_number) 3) To specify specificly which sims should be used for which part (ids_comp and ids_fish)

Args:

compress_fraction ([float]): For mode 1) The fraction of simulations to be used for the compression(default: *None*) compress_number ([int]): For mode 2) The number of simulations to be used for the compression (default: *None*) ids_comp ([list]): For mode 3) IDs of simulations to be used for computing the compression (default: *None*) ids_fish ([list]): For mode 3) IDs of simulations to be used for computing the fisher information (default: *None*)

`initailize_spline_weights`(*dict_param_spline_weights=None*)

Set the weights for the splines for the numerical derivatives.

Args:

dict_param_spline_weights ([dictionary]): The per parameter weights for the derivatives. (default: *None*)

If not argument is supplied the default weights, given on line 5 are used.

`run_combined_fisher_deriv_stablity_test`(*params_names*, *compress_fraction*, *sample_fractions=None*, *verbose=False*, *max_repeats=None*)

This function provides a second means to test the convergence of the compbined Fisher forecasts. The combined fisher forecasts are performed using only a subset of the total available set of derivative simulations (as set by sample_fractions). The combined forecast should be roughly constant to changes of the number of simulations, when it is converged.

For small subsets the Fisher forecast is repeated using as different divisions, by default all the possibe divissions. Eg. 2 for a sample_fraction of 1/2.

Args:

params_names ([list of names]): A list of the parameters to consider in the Fisher forecast compress_fraction (float) : The fraction of the total sims used in the compression. sample_fractions (array of floats): An array of fractions (between 0 and 1) specifing the subsets of the total number of simulations to consider.

If no arguement is given, use a default set of (1/10,1/5,1/3,2/5,1/2,1) (default: *None*)

verbose (bool): Print the Fisher forecast values for each sample fraction as they are evaluated (default: *False*) max_repeats (int): Limit the number of subdivisions to consider. E.g. for a sample fraction of 0.1 there are 10 divisions of the data.

Leaving this as None will use all 10, however setting it to say 3 would mean only 3 of these are considered. (default: *None*)

Returns:

3 arrays : The number of simulations at each sample_fraction, the mean fisher information at each sample_fraction and the error on the mean fisher information (computed using the subsets)

`run_combined_wShuffles_fisher_deriv_stablity_test`(*params_names*, *compress_fraction*, *nShuffles=10*, *sample_fractions=None*, *verbose=True*, *max_repeats=None*)

This function provides a second means to test the convergence of the compbined Fisher forecasts using the reshuffling. There is an iterative step, where for each of nShuffles repeats the sims are shuffled and redivded between the compression and fisher estimation. The combined fisher forecasts are performed using only a subset of the total available set of derivative simulations (as set by sample_fractions). The combined forecast should be roughly constant to changes of the number of simulations, when it is converged.

For small subsets the Fisher forecast is repeated using as different divisions, by default all the possibe divissions. Eg. 2 for a sample_fraction of 1/2.

Args:
params_names ([list of names]): A list of the parameters to consider in the Fisher forecast compress_fraction (float) : The fraction of the total sims used in the compression. nShuffles (int): The number of redivisions of the simulations between compression and Fisher estimation to perform. sample_fractions (array of floats): An array of fractions (between 0 and 1) specifing the subsets of the total number of simulations to consider.

If no arguement is given, use a default set of (1/10,1/5,1/3,2/5,1/2,1) (default: *None*)

verbose (bool): Print the Fisher forecast values for each sample fraction as they are evaluated (default: *False*) max_repeats (int): Limit the number of subdivisions to consider. E.g. for a sample fraction of 0.1 there are 10 divisions of the data.

Leaving this as None will use all 10, however setting it to say 3 would mean only 3 of these are considered. (default: *None*)

Returns:
3 arrays : The number of simulations at each sample_fraction, the mean fisher information at each sample_fraction and the error on the mean fisher information (computed using the subsets)

`run_compressed_fisher_deriv_stablity_test`(*params_names*, *compress_fraction*, *sample_fractions=None*, *verbose=False*, *max_repeats=None*)

This function provides a second means to test the convergence of the compressed Fisher forecasts. The compressed fisher forecasts are performed using only a subset of the total available set of derivative simulations (as set by sample_fractions). If the size of the forecasts increases rapidly with the number of simulations, it is a sign that the Fisher forecast is not converged. If the size of the compressed decreases wit the number of simulations, it is actually a sing the compressed forecast is reliable

For small subsets the Fisher forecast is repeated using as different divisions, by default all the possibe divissions. Eg. 2 for a sample_fraction of 1/2.

Args:
params_names ([list of names]): A list of the parameters to consider in the Fisher forecast compress_fraction (float) : The fraction of the total sims used in the compression. sample_fractions (array of floats): An array of fractions (between 0 and 1) specifing the subsets of the total number of simulations to consider.

If no arguement is given, use a default set of (1/10,1/5,1/3,2/5,1/2,1) (default: *None*)

verbose (bool): Print the Fisher forecast values for each sample fraction as they are evaluated (default: *False*) max_repeats (int): Limit the number of subdivisions to consider. E.g. for a sample fraction of 0.1 there are 10 divisions of the data.

Leaving this as None will use all 10, however setting it to say 3 would mean only 3 of these are considered. (default: *None*)

Returns:
3 arrays : The number of simulations at each sample_fraction, the mean fisher information at each sample_fraction and the error on the mean fisher information (computed using the subsets)

`run_fisher_deriv_stablity_test`(*params_names*, *sample_fractions=None*, *verbose=False*, *max_repeats=None*)

This function provides a second means to test the convergence of the standard Fisher forecasts. Fisher forecasts are performed using only a subset of the total available set of derivative simulations (as set by sample_fractions). If the size of the forecasts increases rapidly with the number of simulations, it is a sign that the Fisher forecast is not converged.

For small subsets the Fisher forecast is repeated using as different divisions, by default all the possibe divissions. Eg. 2 for a sample_fraction of 1/2.

Args:

params_names ([list of names]): A list of the parameters to consider in the Fisher forecast sample_fractions (array of floats): An array of fractions (between 0 and 1) specifing the subsets of the total number of simulations to consider.

If no arguement is given, use a default set of (1/10,1/5,1/3,2/5,1/2,1) (default: *None*)

verbose (bool): Print the Fisher forecast values for each sample fraction as they are evaluated (default: *False*) max_repeats (int): Limit the number of subdivisions to consider. E.g. for a sample fraction of 0.1 there are 10 divisions of the data.

Leaving this as None will use all 10, however setting it to say 3 would mean only 3 of these are considered. (default: *None*)

Returns:

3 arrays : The number of simulations at each sample_fraction, the mean fisher information at each sample_fraction and the error on the mean fisher information (computed using the subsets)

`class CompressedFisher.distributions.gaussian.`**`gaussianFisher`**(*param_names*, *n_sims_derivs*, *include_covmat_param_depedence=False*, *n_sims_covmat=None*, *deriv_finite_dif_accuracy=None*)

`compress_vector`(*vector*, *with_mean=True*)

Apply the compression to a data vector. The optimal compression requires the mean to be subtracted, however in the Fisher forecast this term drops out so the compression can be performed without this subtraction. This also minimizes the noise in the fisher estimate. Eq. 25 in Coulton and Wandelt

Args:

vector ((. . . , Dimension)): The data vector (can be multidimensional but the last dimension should be the dimension of the mean). with_mean (bool): Subtract the mean or not (True)

Returns:

[vector (. . . , n_parameters)]: The compressed data vector (the compression is performed on the last axis)

`property` **`covmat_comp`**

Access the covariance matrix used in the compression operation

Returns:

[DxD matrix]: The compression covariance matrix

`property` **`covmat_fisher`**

Access the covariance matrix used in the Fisher forecasting

Returns:

[DxD matrix]: The covariance matrix used in the fisher analysis.

`property` **`deriv_covmat_comp`**

Access the derivatives of the covariance matrix used in the compression step This is computed from the fraction of simulations assigned to the compression step

Returns:
[dict]: a dictionary containing the derivatives of the mean with respect to each parameter.

`property deriv_covmat_fisher`

Access the derivatives of the covariance matrix used in the fisher forecasting This is computed from the fraction of simulations assigned to the fisher analysis

Returns:
[dict]: a dictionary containing the derivatives of the mean with respect to each parameter.

`property deriv_mean_comp`

Access the derivatives of the mean used in the compression steps This is computed from the fraction of simulations assigned to the compression.

Returns:
[dict]: a dictionary containing the derivatives of the mean with respect to each parameter.

`property deriv_mean_fisher`

Access the derivatives of the mean used in the fisher forecasting. This is computed from the fraction of simulations assigned to the fisher analysis

Returns:
[dict]: a dictionary containing the derivatives of the mean with respect to each parameter.

`initailize_covmat`(*covmat_sims*, *store_covmat_sims=False*)

This function supplies the estimator with the simulations used to compute the covariance matrix. If store_covmat_sims is False then all the simulations are used in both the compression and the fisher estimation. The bias from this has generally found to be small and subdominant to that from the derivatives and for most cases this is the recommended approach. This can be relaxed by setting store_variance_sims to True and using the generate_covmat_sim_splits to split simulations.

Args:
covmat_sims (matrix [N_sims x Dimension]): The simulations used to compute the covariance matrix. store_covmat_sims (bool): Whether to store the covmat sims in the objects. (default: *False*)

`initailize_deriv_sims`(*dic_deriv_sims=None*, *dict_param_steps=None*, *deriv_mean_function=None*, *deriv_covmat_function=None*)

Pass the set of simulations used to estimate the derivatives. There are three modes for this: 1) Pass a dictionary containing the simulations for finite difference derivatives. Each element should be an array of shape (Number Sims, number of param step, Dimension) - where number of param steps corresponds to the points used to estimate a difference derivative. E.g. for a second order central difference index =0 should have sims at heta-delta heta and index =1 should have sims at heta+delta heta. If this mode is used dict_param_steps should be supplied and will be a dictionary containing the step size for each parameter. This is the preferred mode of operation for the code. 2) Pass a dictionary containing the simulations of the derivatives of the mean themselves. Each element should be an array of shape (Number Sims, Dimension). This cannot be used when considering a parameter dependence covariance matrix. 3) Pass a function, deriv_mean_function, which takes as arguments the parameter name and the sim id and returns the derivative of the mean for that simulation. The return should be an array of shape (dimension). If you are considering a model with a parameter dependent covariance matrix you also need to pass the deriv_covmat_function, with identical inputs as the above function. The return should be an array of shape (dimension x dimension).

Args:
dic_deriv_sims ([dictionary]): A dictionary containing the either simulations for computing finite differences for each parameter or a dictionary of the derivatives for each parameter. The key for the dictionary should be the parameter name. Needed for mode 1 or 2. (default: *None*) dict_param_steps ([dictionary]): A dictionary of the parmaeter step sizes. Only needed for mode 1) (default: *None*) deriv_rate_function ([function]): A function that returns the derivative of the mean for a single realizaiton. Arguments of the function are parameter name and sim index. Only needed for mode 3

(default: *None*) deriv_covmat_function ([function]): A function that returns the derivative of the covariance matrix for a single realizaiton. Arguments of the function are parameter name and sim index. Only needed for mode 3 (default: *None*)

`initailize_mean`(*mean_sims*)

This function supplies the estimator with the simulations used to compute the mean. There are no issues if these are the same sims as the covmat. sims.

Args:
mean (matrix [N_sims x Dimension]): The simulations used to compute the mean.

`class CompressedFisher.distributions.poisson.poissonFisher`(*param_names*, *n_sims_derivs*, *deriv_finite_dif_accuracy=None*, *n_sims_variance=None*)

`compress_vector`(*data*, *with_rate=True*)

Apply the compression to a data vector. The optimal compression requires the rate to be subtracted, however in the Fisher forecast this term drops out so the compression can be performed without this subtraction. This also minimizes the noise in the fisher estimate. The compression is computed using Eq. 41 in Coulton and Wandelt.

Args:
vector ((. . . , Dimension)): The data vector (can be multidimensional but the last dimension should be the dimension of the rate). with_rate (bool): Subtract the rate or not (True)

Returns:
[vector (. . . , n_parameters)]: The compressed data vector (the compression is performed on the last axis)

`property deriv_rate_comp`

Access the derivative of the rate used in the compression. These derivatives are computed with the simulations assigned to the compression operation

Returns:
[dictionary] : a dictionary of the derivatives for each parameter

`property deriv_rate_fisher`

Access the derivative of the rate used in the fisher forecast. These derivatives are computed with the simulations assigned to the Fisher estimation

Returns:
[dictionary] : a dictionary of the derivatives for each parameter

`initailize_deriv_sims`(*dic_deriv_sims=None*, *dict_param_steps=None*, *deriv_rate_function=None*)

Pass the set of simulations used to estimate the derivatives. There are three modes for this: 1) Pass a dictionary containing the simulations for finite difference derivatives. Each element should be an array of shape (Number Sims, number of param step, Dimension) - where number of param steps corresponds to the points used to estimate a difference derivative. E.g. for a second order central difference index =0 should have sims at heta-delta heta and index =1 should have sims at heta+delta heta. If this mode is used dict_param_steps should be supplied and will be a dictionary containing the step size for each parameter. This is the preferred mode of operation for the code. 2) Pass a dictionary containing the simulations of the derivatives themselves. Each element should be an array of shape (Number Sims, Dimension) 3) Pass a function, deriv_rate_function, which takes as arguments the parameter name and the sim id and returns the derivative for that simulation. The return should be an array of shape (dimension)

Args:
dic_deriv_sims ([dictionary]): A dictionary containing the either simulations for computing finite differences for each parameter or a dictionary of the derivatives for each parameter. The key for the

dictionary should be the parameter name. Needed for mode 1 or 2. (default: *None*) dict_param_steps ([dictionary]): A dictionary of the parmaeter step sizes. Only needed for mode 1) (default: *None*) deriv_rate_function ([function]): A function that returns the derivative for a single realizaiton. Arguments of the function are parameter name and sim index. Only needed for mode 3 (default: *None*)

`initailize_variance`(*variance_sims*, *store_variance_sims=False*)

Pass the set of simulations used be the code to estimate the Poisson rate and measurement variances. If store_variance_sims is False then all the simulations are used in both the compression and the fisher estimation. The bias from this has generally found to be small and subdominant to that from the derivatives and for most cases this is the recommended approach. This can be relaxed by setting store_variance_sims to True and using the generate_covmat_sim_splits to split simulations.

Args:

variance_sims (matrix [n_sims, d]): A set of n_sims simulations each with lenght, d store_variance_sims (bool): To divide these simulations between the fisher and compression set they must be stored. This is typically not necessary (default: *False*)

`property rate_comp`

Access the rate used in the compression

Returns:

[array]: The array of Poisson rates

`property variance_fisher`

Access the variance used in the Fisher forecast. This is computed from the simulations.

Returns:

[D array]: The array of variances

CHAPTER THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX