factor_analyzer Documentation

Release 0.4.0

Jeremy Biggs

May 19, 2023

Contents

This is a Python module to perform exploratory and factor analysis (EFA), with several optional rotations. It also includes a class to perform confirmatory factor analysis (CFA), with certain pre-defined constraints. In exploratory factor analysis, factor extraction can be performed using a variety of estimation techniques. The *factor_analyzer* package allows users to perform EFA using either (1) a minimum residual (MINRES) solution, (2) a maximum likelihood (ML) solution, or (3) a principal factor solution. However, CFA can only be performed using an ML solution.

Both the EFA and CFA classes within this package are fully compatible with `scikit-learn`. Portions of this code are ported from the excellent R library `psych`, and the `sem` package provided inspiration for the CFA class.

Important: Please make sure to read the *important notes* section if you encounter any unexpected results.

CHAPTER 1

Documentation

1.1 Introduction

Exploratory factor analysis (EFA) is a statistical technique used to identify latent relationships among sets of observed variables in a dataset. In particular, EFA seeks to model a large set of observed variables as linear combinations of some smaller set of unobserved, latent factors. The matrix of weights, or factor loadings, generated from an EFA model describes the underlying relationships between each variable and the latent factors.

Confirmatory factor analysis (CFA), a closely associated technique, is used to test a priori hypothesis about latent relationships among sets of observed variables. In CFA, the researcher specifies the expected pattern of factor loadings (and possibly other constraints), and fits a model according to this specification.

Typically, a number of factors (K) in an EFA or CFA model is selected such that it is substantially smaller than the number of variables. The factor analysis model can be estimated using a variety of standard estimation methods, including but not limited MINRES or ML.

Factor loadings are similar to standardized regression coefficients, and variables with higher loadings on a particular factor can be interpreted as explaining a larger proportion of the variation in that factor. In the case of EFA, factor loading matrices are usually rotated after the factor analysis model is estimated in order to produce a simpler, more interpretable structure to identify which variables are loading on a particular factor.

Two common types of rotations are:

1. The **varimax** rotation, which rotates the factor loading matrix so as to maximize the sum of the variance of squared loadings, while preserving the orthogonality of the loading matrix.
2. The **promax** rotation, a method for oblique rotation, which builds upon the varimax rotation, but ultimately allows factors to become correlated.

This package includes a `factor_analyzer` module with a stand-alone `FactorAnalyzer` class. The class includes `fit()` and `transform()` methods that enable users to perform factor analysis and score new data using the fitted factor model. Users can also perform optional rotations on a factor loading matrix using the `Rotator` class.

The following rotation options are available in both `FactorAnalyzer` and `Rotator`:

(a) varimax (orthogonal rotation)

(b) promax (oblique rotation)

(c) oblimin (oblique rotation)

(d) oblimax (orthogonal rotation)

(e) quartimin (oblique rotation)

(f) quartimax (orthogonal rotation)

(g) equamax (orthogonal rotation)

(h) geomin_obl (oblique rotation)

(i) geomin_ort (orthogonal rotation)

In addition, the package includes a *confirmatory_factor_analyzer* module with a stand-alone *ConfirmatoryFactorAnalyzer* class. The class includes *fit()* and *transform()* that enable users to perform confirmatory factor analysis and score new data using the fitted model. Performing CFA requires users to specify in advance a model specification with the expected factor loading relationships. This can be done using the *ModelSpecificationParser* class.

1.1.1 Requirements

- Python 3.8 or higher
- `numpy`
- `pandas`
- `scipy`
- `scikit-learn`
- `pre-commit`

1.1.2 Installation

You can install this package via `pip` with:

```
$ pip install factor_analyzer
```

Alternatively, you can install via `conda` with:

```
$ conda install -c ets factor_analyzer
```

1.2 Important notes

1. It is possible that *factor_analyzer* may return the loading for a factor that has all negative entries whereas SPSS/R may return the same loading with all positive entries. This is not a bug. This can happen if the eigenvalue decomposition returns an eigenvector with all negative entries, which is not unusual since if v is an eigenvector, then so is $\alpha * v$, where α is any scalar ($\neq 0$). Additionally, signs on factor loadings are also kind of meaningless because all they do is flip the (already arbitrary) interpretation of the latent factor. For more details, please refer to this Github issue.
2. When using equamax rotation, you must compute the correct value of κ yourself and pass it using the *rotation_kwargs* argument. This is different from SPSS which computes the value of κ internally. For more details, please refer to this Github issue.

1.3 API documentation

1.3.1 factor_analyzer.factor_analyzer Module

Factor analysis using MINRES or ML, with optional rotation using Varimax or Promax.

author Jeremy Biggs (jeremy.m.biggs@gmail.com)

author Nitin Madnani (nmadnani@ets.org)

organization Educational Testing Service

date 2022-09-05

class factor_analyzer.factor_analyzer.**FactorAnalyzer**(*n_factors=3*, *rotation='promax'*, *method='minres'*, *use_smc=True*, *is_corr_matrix=False*, *bounds=(0.005*, *1)*, *impute='median'*, *svd_method='randomized'*, *rotation_kwargs=None*)

Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

The main exploratory factor analysis class.

This class:

(1) Fits a factor analysis model using minres, maximum likelihood, or principal factor extraction and returns the loading matrix

(2) Optionally performs a rotation, with method including:

(a) varimax (orthogonal rotation)

(b) promax (oblique rotation)

(c) oblimin (oblique rotation)

(d) oblimax (orthogonal rotation)

(e) quartimin (oblique rotation)

(f) quartimax (orthogonal rotation)

(g) equamax (orthogonal rotation)

Parameters

- **n_factors** (*int, optional*) – The number of factors to select. Defaults to 3.
- **rotation** (*str, optional*) – The type of rotation to perform after fitting the factor analysis model. If set to None, no rotation will be performed, nor will any associated Kaiser normalization.

  Possible values include:

  (a) varimax (orthogonal rotation)

  (b) promax (oblique rotation)

  (c) oblimin (oblique rotation)

  (d) oblimax (orthogonal rotation)

(e) quartimin (oblique rotation)

(f) quartimax (orthogonal rotation)

(g) equamax (orthogonal rotation)

Defaults to 'promax'.

- **method** (*{'minres', 'ml', 'principal'}, optional*) – The fitting method to use, either MINRES or Maximum Likelihood. Defaults to 'minres'.
- **use_smc** (*bool, optional*) – Whether to use squared multiple correlation as starting guesses for factor analysis. Defaults to `True`.
- **bounds** (*tuple, optional*) – The lower and upper bounds on the variables for "L-BFGS-B" optimization. Defaults to (0.005, 1).
- **impute** (*{'drop', 'mean', 'median'}, optional*) – How to handle missing values, if any, in the data: (a) use list-wise deletion ('drop'), or (b) impute the column median ('median'), or impute the column mean ('mean'). Defaults to 'median'
- **is_corr_matrix** (*bool, optional*) – Set to `True if the ``data` is the correlation matrix. Defaults to *False*.
- **svd_method** (*{'lapack', 'randomized'}*) – The SVD method to use when `method` is 'principal'. If 'lapack', use standard SVD from `scipy.linalg`. If 'randomized', use faster `randomized_svd` function from scikit-learn. The latter should only be used if the number of columns is greater than or equal to the number of rows in in the dataset. Defaults to 'randomized'
- **optional** (*rotation_kwargs,*) – Dictionary containing keyword arguments for the rotation method.

loadings_
The factor loadings matrix. `None`, if `fit()` ` has not been called.

Type `numpy.ndarray`

corr_
The original correlation matrix. `None`, if `fit()` has not been called.

Type `numpy.ndarray`

rotation_matrix_
The rotation matrix, if a rotation has been performed. `None` otherwise.

Type `numpy.ndarray`

structure_
The structure loading matrix. This only exists if `rotation` is 'promax' and is `None` otherwise.

Type `numpy.ndarray` or None

phi_
The factor correlations matrix. This only exists if `rotation` is 'oblique' and is `None` otherwise.

Type `numpy.ndarray` or None

Notes

This code was partly derived from the excellent R package *psych*.

References

[1] https://github.com/cran/psych/blob/master/R/fa.R

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer
>>> df_features = pd.read_csv('tests/data/test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
FactorAnalyzer(bounds=(0.005, 1), impute='median', is_corr_matrix=False,
        method='minres', n_factors=3, rotation=None, rotation_kwargs={},
        use_smc=True)
>>> fa.loadings_
array([[-0.12991218,  0.16398154,  0.73823498],
       [ 0.03899558,  0.04658425,  0.01150343],
       [ 0.34874135,  0.61452341, -0.07255667],
       [ 0.45318006,  0.71926681, -0.07546472],
       [ 0.36688794,  0.44377343, -0.01737067],
       [ 0.74141382, -0.15008235,  0.29977512],
       [ 0.741675  , -0.16123009, -0.20744495],
       [ 0.82910167, -0.20519428,  0.04930817],
       [ 0.76041819, -0.23768727, -0.1206858 ],
       [ 0.81533404, -0.12494695,  0.17639683]])
>>> fa.get_communalities()
array([0.588758  , 0.00382308, 0.50452402, 0.72841183, 0.33184336,
       0.66208428, 0.61911036, 0.73194557, 0.64929612, 0.71149718])
```

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

Fit factor analysis model using either MINRES, ML, or principal factor analysis.

By default, use SMC as starting guesses.

Parameters

- **X** (*array-like*) – The data to analyze.
- **y** (*ignored*) –

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer
>>> df_features = pd.read_csv('tests/data/test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
FactorAnalyzer(bounds=(0.005, 1), impute='median', is_corr_matrix=False,
        method='minres', n_factors=3, rotation=None, rotation_kwargs={},
        use_smc=True)
>>> fa.loadings_
array([[-0.12991218,  0.16398154,  0.73823498],
       [ 0.03899558,  0.04658425,  0.01150343],
       [ 0.34874135,  0.61452341, -0.07255667],
       [ 0.45318006,  0.71926681, -0.07546472],
       [ 0.36688794,  0.44377343, -0.01737067],
```

(continues on next page)

(continued from previous page)

```
       [ 0.74141382, -0.15008235,  0.29977512],
       [ 0.741675  , -0.16123009, -0.20744495],
       [ 0.82910167, -0.20519428,  0.04930817],
       [ 0.76041819, -0.23768727, -0.1206858 ],
       [ 0.81533404, -0.12494695,  0.17639683]])
```

get_communalities()

Calculate the communalities, given the factor loading matrix.

Returns **communalities** – The communalities from the factor loading matrix.

Return type numpy.ndarray

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer
>>> df_features = pd.read_csv('tests/data/test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
FactorAnalyzer(bounds=(0.005, 1), impute='median', is_corr_matrix=False,
        method='minres', n_factors=3, rotation=None, rotation_kwargs={},
        use_smc=True)
>>> fa.get_communalities()
array([0.588758  , 0.00382308, 0.50452402, 0.72841183, 0.33184336,
       0.66208428, 0.61911036, 0.73194557, 0.64929612, 0.71149718])
```

get_eigenvalues()

Calculate the eigenvalues, given the factor correlation matrix.

Returns

- **original_eigen_values** (numpy.ndarray) – The original eigenvalues
- **common_factor_eigen_values** (numpy.ndarray) – The common factor eigenvalues

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer
>>> df_features = pd.read_csv('tests/data/test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
FactorAnalyzer(bounds=(0.005, 1), impute='median', is_corr_matrix=False,
        method='minres', n_factors=3, rotation=None, rotation_kwargs={},
        use_smc=True)
>>> fa.get_eigenvalues()
(array([ 3.51018854,  1.28371018,  0.73739507,  0.1334704 ,  0.03445558,
        0.0102918 , -0.00740013, -0.03694786, -0.05959139, -0.07428112]),
 array([ 3.51018905,  1.2837105 ,  0.73739508,  0.13347082,  0.03445601,
        0.01029184, -0.0074    , -0.03694834, -0.05959057, -0.07428059]))
```

get_factor_variance()

Calculate factor variance information.

The factor variance information including the variance, proportional variance, and cumulative variance for each factor.

Returns

- **variance** (numpy.ndarray) – The factor variances.
- **proportional_variance** (numpy.ndarray) – The proportional factor variances.
- **cumulative_variances** (numpy.ndarray) – The cumulative factor variances.

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer
>>> df_features = pd.read_csv('tests/data/test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
FactorAnalyzer(bounds=(0.005, 1), impute='median', is_corr_matrix=False,
        method='minres', n_factors=3, rotation=None, rotation_kwargs={},
        use_smc=True)
>>> # 1. Sum of squared loadings (variance)
... # 2. Proportional variance
... # 3. Cumulative variance
>>> fa.get_factor_variance()
(array([3.51018854, 1.28371018, 0.73739507]),
 array([0.35101885, 0.12837102, 0.07373951]),
 array([0.35101885, 0.47938987, 0.55312938]))
```

get_uniquenesses()

Calculate the uniquenesses, given the factor loading matrix.

Returns **uniquenesses** – The uniquenesses from the factor loading matrix.

Return type numpy.ndarray

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer
>>> df_features = pd.read_csv('tests/data/test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
FactorAnalyzer(bounds=(0.005, 1), impute='median', is_corr_matrix=False,
        method='minres', n_factors=3, rotation=None, rotation_kwargs={},
        use_smc=True)
>>> fa.get_uniquenesses()
array([0.411242  , 0.99617692, 0.49547598, 0.27158817, 0.66815664,
       0.33791572, 0.38088964, 0.26805443, 0.35070388, 0.28850282])
```

sufficiency(*num_observations: int*) → Tuple[float, int, float]

Perform the sufficiency test.

The test calculates statistics under the null hypothesis that the selected number of factors is sufficient.

Parameters **num_observations** (*int*) – The number of observations in the input data that this factor analyzer was fit using.

Returns

- **statistic** (*float*) – The test statistic
- **degrees** (*int*) – The degrees of freedom
- **pvalue** (*float*) – The p-value of the test

References

[1] Lawley, D. N. and Maxwell, A. E. (1971). Factor Analysis as a Statistical Method. Second edition. Butterworths. P. 36.

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer
>>> df_features = pd.read_csv('tests/data/test01.csv')
>>> fa = FactorAnalyzer(n_factors=3, rotation=None, method="ml")
>>> fa.fit(df_features)
>>> fa.sufficiency(df_features.shape[0])
(1475.8755629859675, 663, 8.804286459822274e-64)
```

transform(*X*)

Get factor scores for a new data set.

Parameters **X** (array-like, shape (n_samples, n_features)) – The data to score using the fitted factor model.

Returns **X_new** – The latent variables of X.

Return type numpy.ndarray, shape (n_samples, n_components)

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer
>>> df_features = pd.read_csv('tests/data/test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
FactorAnalyzer(bounds=(0.005, 1), impute='median', is_corr_matrix=False,
        method='minres', n_factors=3, rotation=None, rotation_kwargs={},
        use_smc=True)
>>> fa.transform(df_features)
array([[-1.05141425,  0.57687826,  0.1658788 ],
       [-1.59940101,  0.89632125,  0.03824552],
       [-1.21768164, -1.16319406,  0.57135189],
       ...,
       [ 0.13601554,  0.03601086,  0.28813877],
       [ 1.86904519, -0.3532394 , -0.68170573],
       [ 0.86133386,  0.18280695, -0.79170903]])
```

factor_analyzer.factor_analyzer.**calculate_bartlett_sphericity**(*x*)

Compute the Bartlett sphericity test.

H0: The matrix of population correlations is equal to I. H1: The matrix of population correlations is not equal to I.

The formula for Bartlett's Sphericity test is:

$$-1 * (n - 1 - ((2p + 5)/6)) * ln(det(R))$$

Where R det(R) is the determinant of the correlation matrix, and p is the number of variables.

Parameters **x** (*array-like*) – The array for which to calculate sphericity.

Returns

- **statistic** (*float*) – The chi-square value.
- **p_value** (*float*) – The associated p-value for the test.

factor_analyzer.factor_analyzer.**calculate_kmo**(*x*)

Calculate the Kaiser-Meyer-Olkin criterion for items and overall.

This statistic represents the degree to which each observed variable is predicted, without error, by the other variables in the dataset. In general, a KMO < 0.6 is considered inadequate.

Parameters **x** (*array-like*) – The array from which to calculate KMOs.

Returns

- **kmo_per_variable** (numpy.ndarray) – The KMO score per item.
- **kmo_total** (*float*) – The overall KMO score.

1.3.2 factor_analyzer.confirmatory_factor_analyzer Module

Confirmatory factor analysis using machine learning methods.

author Jeremy Biggs (jeremy.m.biggs@gmail.com)

author Nitin Madnani (nmadnani@ets.org)

organization Educational Testing Service

date 2022-09-05

class factor_analyzer.confirmatory_factor_analyzer.**ConfirmatoryFactorAnalyzer**(*specification=None*, *n_obs=None*, *is_cov_matrix=False*, *bounds=None*, *max_iter=200*, *tol=None*, *impute='median'*, *disp=True*)

Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Fit a confirmatory factor analysis model using maximum likelihood.

Parameters

- **specification** (*ModelSpecification* or None, optional) – A model specification. This must be a *ModelSpecification* object or None. If None, a *ModelSpecification* object will be generated assuming that n_factors == n_variables, and that all variables load on all factors. Note that this could mean the factor model is not identified, and the optimization could fail. Defaults to *None*.

- **n_obs** (*int or None, optional*) – The number of observations in the original data set. If this is not passed and `is_cov_matrix` is `True`, then an error will be raised. Defaults to `None`.
- **is_cov_matrix** (*bool, optional*) – Whether the input `X` is a covariance matrix. If `False`, assume it is the full data set. Defaults to `False`.
- **bounds** (*list of tuples or None, optional*) – A list of minimum and maximum boundaries for each element of the input array. This must equal `x0`, which is the input array from your parsed and combined model specification.

  The length is: ((n_factors * n_variables) + n_variables + n_factors + (((n_factors * n_factors) - n_factors) // 2)

  If *None*, nothing will be bounded. Defaults to `None`.
- **max_iter** (*int, optional*) – The maximum number of iterations for the optimization routine. Defaults to 200.
- **tol** (*float or None, optional*) – The tolerance for convergence. Defaults to `None`.
- **disp** (*bool, optional*) – Whether to print the scipy optimization `fmin` message to standard output. Defaults to `True`.

Raises `ValueError` – If *is_cov_matrix* is *True*, and *n_obs* is not provided.

model
The model specification object.

Type *ModelSpecification*

loadings_
The factor loadings matrix. `None`, if `fit()` ` has not been called.

Type `numpy.ndarray`

error_vars_
The error variance matrix

Type `numpy.ndarray`

factor_varcovs_
The factor covariance matrix.

Type `numpy.ndarray`

log_likelihood_
The log likelihood from the optimization routine.

Type float

aic_
The Akaike information criterion.

Type float

bic_
The Bayesian information criterion.

Type float

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import (ConfirmatoryFactorAnalyzer,
...                              ModelSpecificationParser)
>>> X = pd.read_csv('tests/data/test11.csv')
>>> model_dict = {"F1": ["V1", "V2", "V3", "V4"],
...               "F2": ["V5", "V6", "V7", "V8"]}
>>> model_spec = ModelSpecificationParser.parse_model_specification_from_dict(X,␣
↪model_dict)
>>> cfa = ConfirmatoryFactorAnalyzer(model_spec, disp=False)
>>> cfa.fit(X.values)
>>> cfa.loadings_
array([[0.99131285, 0.        ],
       [0.46074919, 0.        ],
       [0.3502267 , 0.        ],
       [0.58331488, 0.        ],
       [0.        , 0.98621042],
       [0.        , 0.73389239],
       [0.        , 0.37602988],
       [0.        , 0.50049507]])
>>> cfa.factor_varcovs_
array([[1.        , 0.17385704],
       [0.17385704, 1.        ]])
>>> cfa.get_standard_errors()
(array([[0.06779949, 0.        ],
       [0.04369956, 0.        ],
       [0.04153113, 0.        ],
       [0.04766645, 0.        ],
       [0.        , 0.06025341],
       [0.        , 0.04913149],
       [0.        , 0.0406604 ],
       [0.        , 0.04351208]]),
 array([0.11929873, 0.05043616, 0.04645803, 0.05803088,
        0.10176889, 0.06607524, 0.04742321, 0.05373646]))
>>> cfa.transform(X.values)
array([[-0.46852166, -1.08708035],
       [ 2.59025301,  1.20227783],
       [-0.47215977,  2.65697245],
       ...,
       [-1.5930886 , -0.91804114],
       [ 0.19430887,  0.88174818],
       [-0.27863554, -0.7695101 ]])
```

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

Perform confirmatory factor analysis.

Parameters

- **X** (*array-like*) – The data to use for confirmatory factor analysis. If this is just a covariance matrix, make sure `is_cov_matrix` was set to `True`.
- **y** (*ignored*) –

Raises

- `ValueError` – If the specification is not None or a *ModelSpecification* object.
- `AssertionError` – If `is_cov_matrix` was `True` and the matrix is not square.
- `AssertionError` – If `len(bounds)` != `len(x0)`

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import (ConfirmatoryFactorAnalyzer,
...                              ModelSpecificationParser)
>>> X = pd.read_csv('tests/data/test11.csv')
>>> model_dict = {"F1": ["V1", "V2", "V3", "V4"],
...               "F2": ["V5", "V6", "V7", "V8"]}
>>> model_spec = ModelSpecificationParser.parse_model_specification_from_
↪dict(X, model_dict)
>>> cfa = ConfirmatoryFactorAnalyzer(model_spec, disp=False)
>>> cfa.fit(X.values)
>>> cfa.loadings_
array([[0.99131285, 0.        ],
       [0.46074919, 0.        ],
       [0.3502267 , 0.        ],
       [0.58331488, 0.        ],
       [0.        , 0.98621042],
       [0.        , 0.73389239],
       [0.        , 0.37602988],
       [0.        , 0.50049507]])
```

get_model_implied_cov()

Get the model-implied covariance matrix (sigma) for an estimated model.

Returns **model_implied_cov** – The model-implied covariance matrix.

Return type numpy.ndarray

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import (ConfirmatoryFactorAnalyzer,
...                              ModelSpecificationParser)
>>> X = pd.read_csv('tests/data/test11.csv')
>>> model_dict = {"F1": ["V1", "V2", "V3", "V4"],
...               "F2": ["V5", "V6", "V7", "V8"]}
>>> model_spec = ModelSpecificationParser.parse_model_specification_from_
↪dict(X, model_dict)
>>> cfa = ConfirmatoryFactorAnalyzer(model_spec, disp=False)
>>> cfa.fit(X.values)
>>> cfa.get_model_implied_cov()
array([[2.07938612, 0.45674659, 0.34718423, 0.57824753, 0.16997013,
        0.12648394, 0.06480751, 0.08625868],
       [0.45674659, 1.16703337, 0.16136667, 0.26876186, 0.07899988,
        0.05878807, 0.03012168, 0.0400919 ],
       [0.34718423, 0.16136667, 1.07364855, 0.20429245, 0.06004974,
        0.04468625, 0.02289622, 0.03047483],
       [0.57824753, 0.26876186, 0.20429245, 1.28809317, 0.10001495,
        0.07442652, 0.03813447, 0.05075691],
       [0.16997013, 0.07899988, 0.06004974, 0.10001495, 2.0364391 ,
        0.72377232, 0.37084458, 0.49359346],
       [0.12648394, 0.05878807, 0.04468625, 0.07442652, 0.72377232,
        1.48080077, 0.27596546, 0.36730952],
       [0.06480751, 0.03012168, 0.02289622, 0.03813447, 0.37084458,
        0.27596546, 1.11761918, 0.1882011 ],
```

(continues on next page)

(continued from previous page)

```
       [0.08625868, 0.0400919 , 0.03047483, 0.05075691, 0.49359346,
        0.36730952, 0.1882011 , 1.28888233]])
```

get_standard_errors()

Get standard errors from the implied covariance matrix and implied means.

Returns

- **loadings_se** (numpy.ndarray) – The standard errors for the factor loadings.
- **error_vars_se** (numpy.ndarray) – The standard errors for the error variances.

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import (ConfirmatoryFactorAnalyzer,
...                              ModelSpecificationParser)
>>> X = pd.read_csv('tests/data/test11.csv')
>>> model_dict = {"F1": ["V1", "V2", "V3", "V4"],
...               "F2": ["V5", "V6", "V7", "V8"]}
>>> model_spec = ModelSpecificationParser.parse_model_specification_from_
↪dict(X, model_dict)
>>> cfa = ConfirmatoryFactorAnalyzer(model_spec, disp=False)
>>> cfa.fit(X.values)
>>> cfa.get_standard_errors()
(array([[0.06779949, 0.        ],
       [0.04369956, 0.        ],
       [0.04153113, 0.        ],
       [0.04766645, 0.        ],
       [0.        , 0.06025341],
       [0.        , 0.04913149],
       [0.        , 0.0406604 ],
       [0.        , 0.04351208]]),
 array([0.11929873, 0.05043616, 0.04645803, 0.05803088,
        0.10176889, 0.06607524, 0.04742321, 0.05373646]))
```

transform(*X*)

Get the factor scores for a new data set.

Parameters **X** (array-like, shape (n_samples, n_features)) – The data to score using the fitted factor model.

Returns **scores** – The latent variables of X.

Return type numpy array, shape (n_samples, n_components)

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import (ConfirmatoryFactorAnalyzer,
...                              ModelSpecificationParser)
>>> X = pd.read_csv('tests/data/test11.csv')
>>> model_dict = {"F1": ["V1", "V2", "V3", "V4"],
...               "F2": ["V5", "V6", "V7", "V8"]}
>>> model_spec = ModelSpecificationParser.parse_model_specification_from_
↪dict(X, model_dict)
```

(continues on next page)

(continued from previous page)

```
>>> cfa = ConfirmatoryFactorAnalyzer(model_spec, disp=False)
>>> cfa.fit(X.values)
>>> cfa.transform(X.values)
array([[-0.46852166, -1.08708035],
       [ 2.59025301,  1.20227783],
       [-0.47215977,  2.65697245],
       ...,
       [-1.5930886 , -0.91804114],
       [ 0.19430887,  0.88174818],
   [-0.27863554, -0.7695101 ]])
```

References

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6157408/

class factor_analyzer.confirmatory_factor_analyzer.**ModelSpecification**(*loadings*, *n_factors*, *n_variables*, *factor_names=None*, *variable_names=None*)

Bases: object

Encapsulate the model specification for CFA.

This class contains a number of specification properties that are used in the CFA procedure.

Parameters

- **loadings** (*array-like*) – The factor loadings specification.
- **n_factors** (*int*) – The number of factors.
- **n_variables** (*int*) – The number of variables.
- **factor_names** (*list of str or None*) – A list of factor names, if available. Defaults to None.
- **variable_names** (*list of str or None*) – A list of variable names, if available. Defaults to None.

copy()
Return a copy of the model specification.

error_vars
Get the error variance specification.

error_vars_free
Get the indices of "free" error variance parameters.

factor_covs
Get the factor covariance specification.

factor_covs_free
Get the indices of "free" factor covariance parameters.

factor_names
Get list of factor names, if available.

get_model_specification_as_dict()
Get the model specification as a dictionary.

Returns **model_specification** – The model specification keys and values, as a dictionary.

Return type dict

loadings
Get the factor loadings specification.

loadings_free
Get the indices of "free" factor loading parameters.

n_factors
Get the number of factors.

n_lower_diag
Get the lower diagonal of the factor covariance matrix.

n_variables
Get the number of variables.

variable_names
Get list of variable names, if available.

class factor_analyzer.confirmatory_factor_analyzer.**ModelSpecificationParser**
Bases: object

Generate the model specification for CFA.

This class includes two static methods to generate a *ModelSpecification* object from either a dictionary or a numpy array.

static parse_model_specification_from_array(*X*, *specification=None*)
Generate the model specification from a numpy array.

The columns should correspond to the factors, and the rows should correspond to the variables. If this method is used to create the *ModelSpecification* object, then *no* factor names and variable names will be added as properties to that object.

Parameters

- **X** (*array-like*) – The data set that will be used for CFA.
- **specification** (*array-like or None*) – An array with the loading details. If None, the matrix will be created assuming all variables load on all factors. Defaults to None.

Returns A model specification object.

Return type *ModelSpecification*

Raises ValueError – If specification is not in the expected format.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from factor_analyzer import (ConfirmatoryFactorAnalyzer,
...                              ModelSpecificationParser)
>>> X = pd.read_csv('tests/data/test11.csv')
>>> model_array = np.array([[1, 1, 1, 1, 0, 0, 0, 0], [0, 0, 0, 0, 1, 1, 1,␣
↪1]])
```

(continues on next page)

(continued from previous page)

```
>>> model_spec = ModelSpecificationParser.parse_model_specification_from_
↪array(X,
...
↪ model_array)
```

static parse_model_specification_from_dict(*X*, *specification=None*)

Generate the model specification from a dictionary.

The keys in the dictionary should be the factor names, and the values should be the feature names. If this method is used to create the *ModelSpecification* object, then factor names and variable names will be added as properties to that object.

Parameters

- **X** (*array-like*) – The data set that will be used for CFA.
- **specification** (*dict or None*) – A dictionary with the loading details. If `None`, the matrix will be created assuming all variables load on all factors. Defaults to `None`.

Returns A model specification object.

Return type *ModelSpecification*

Raises `ValueError` – If `specification` is not in the expected format.

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import (ConfirmatoryFactorAnalyzer,
...                              ModelSpecificationParser)
>>> X = pd.read_csv('tests/data/test11.csv')
>>> model_dict = {"F1": ["V1", "V2", "V3", "V4"],
...               "F2": ["V5", "V6", "V7", "V8"]}
>>> model_spec = ModelSpecificationParser.parse_model_specification_from_
↪dict(X, model_dict)
```

1.3.3 factor_analyzer.rotator Module

Class to perform various rotations of factor loading matrices.

author Jeremy Biggs (jeremy.m.biggs@gmail.com)

author Nitin Madnani (nmadnani@ets.org)

organization Educational Testing Service

date 2022-09-05

class factor_analyzer.rotator.**Rotator**(*method='varimax'*, *normalize=True*, *power=4*, *kappa=0*, *gamma=0*, *delta=0.01*, *max_iter=500*, *tol=1e-05*)

Bases: `sklearn.base.BaseEstimator`

Perform rotations on an unrotated factor loading matrix.

The Rotator class takes an (unrotated) factor loading matrix and performs one of several rotations.

Parameters

- **method** (*str, optional*) –

The factor rotation method. Options include:

(a) varimax (orthogonal rotation)

(b) promax (oblique rotation)

(c) oblimin (oblique rotation)

(d) oblimax (orthogonal rotation)

(e) quartimin (oblique rotation)

(f) quartimax (orthogonal rotation)

(g) equamax (orthogonal rotation)

(h) geomin_obl (oblique rotation)

(i) geomin_ort (orthogonal rotation)

Defaults to 'varimax'.

- **normalize** (*bool or None, optional*) – Whether to perform Kaiser normalization and de-normalization prior to and following rotation. Used for 'varimax' and 'promax' rotations. If `None`, default for 'promax' is `False`, and default for 'varimax' is `True`. Defaults to `None`.
- **power** (*int, optional*) – The exponent to which to raise the promax loadings (minus 1). Numbers should generally range from 2 to 4. Defaults to 4.
- **kappa** (*float, optional*) – The kappa value for the 'equamax' objective. Ignored if the method is not 'equamax'. Defaults to 0.
- **gamma** (*int, optional*) – The gamma level for the 'oblimin' objective. Ignored if the method is not 'oblimin'. Defaults to 0.
- **delta** (*float, optional*) – The delta level for 'geomin' objectives. Ignored if the method is not 'geomin_*'. Defaults to 0.01.
- **max_iter** (*int, optional*) – The maximum number of iterations. Used for 'varimax' and 'oblique' rotations. Defaults to 1000.
- **tol** (*float, optional*) – The convergence threshold. Used for 'varimax' and 'oblique' rotations. Defaults to 1e-5.

loadings_
The loadings matrix.

Type `numpy.ndarray`, shape (`n_features`, `n_factors`)

rotation_
The rotation matrix.

Type `numpy.ndarray`, shape (`n_factors`, `n_factors`)

phi_
The factor correlations matrix. This only exists if `method` is 'oblique'.

Type `numpy.ndarray` or None

Notes

Most of the rotations in this class are ported from R's `GPARotation` package.

References

[1] https://cran.r-project.org/web/packages/GPArotation/index.html

Examples

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer, Rotator
>>> df_features = pd.read_csv('test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
>>> rotator = Rotator()
>>> rotator.fit_transform(fa.loadings_)
array([[-0.07693215,  0.04499572,  0.76211208],
       [ 0.01842035,  0.05757874,  0.01297908],
       [ 0.06067925,  0.70692662, -0.03311798],
       [ 0.11314343,  0.84525117, -0.03407129],
       [ 0.15307233,  0.5553474 , -0.00121802],
       [ 0.77450832,  0.1474666 ,  0.20118338],
       [ 0.7063001 ,  0.17229555, -0.30093981],
       [ 0.83990851,  0.15058874, -0.06182469],
       [ 0.76620579,  0.1045194 , -0.22649615],
       [ 0.81372945,  0.20915845,  0.07479506]])
```

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

Compute the factor rotation.

Parameters

- **X** (*array-like*) – The factor loading matrix, shape (n_features, n_factors)
- **y** (*ignored*) –

Returns

Return type self

Example

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer, Rotator
>>> df_features = pd.read_csv('test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
>>> rotator = Rotator()
>>> rotator.fit(fa.loadings_)
```

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

Compute the factor rotation, and return the new loading matrix.

Parameters

- **X** (*array-like*) – The factor loading matrix, shape (n_features, n_factors)
- **y** (*Ignored*) –

Returns **loadings_** – The loadings matrix.

Return type numpy,ndarray, shape (n_features, n_factors)

Raises `ValueError` – If `method` is not in the list of acceptable methods.

Example

```
>>> import pandas as pd
>>> from factor_analyzer import FactorAnalyzer, Rotator
>>> df_features = pd.read_csv('test02.csv')
>>> fa = FactorAnalyzer(rotation=None)
>>> fa.fit(df_features)
>>> rotator = Rotator()
>>> rotator.fit_transform(fa.loadings_)
array([[-0.07693215,  0.04499572,  0.76211208],
       [ 0.01842035,  0.05757874,  0.01297908],
       [ 0.06067925,  0.70692662, -0.03311798],
       [ 0.11314343,  0.84525117, -0.03407129],
       [ 0.15307233,  0.5553474 , -0.00121802],
       [ 0.77450832,  0.1474666 ,  0.20118338],
       [ 0.7063001 ,  0.17229555, -0.30093981],
       [ 0.83990851,  0.15058874, -0.06182469],
       [ 0.76620579,  0.1045194 , -0.22649615],
       [ 0.81372945,  0.20915845,  0.07479506]])
```

1.3.4 `factor_analyzer.utils` Module

Utility functions, used primarily by the confirmatory factor analysis module.

author Jeremy Biggs (jeremy.m.biggs@gmail.com)

author Nitin Madnani (nmadnani@ets.org)

organization Educational Testing Service

date 2022-09-05

`factor_analyzer.utils.`**`apply_impute_nan`**(*x*, *how='mean'*)

Apply a function to impute `np.nan` values with the mean or the median.

Parameters

- **x** (*array-like*) – The 1-D array to impute.
- **how** (*str, optional*) – Whether to impute the 'mean' or 'median'. Defaults to 'mean'.

Returns **x** – The array, with the missing values imputed.

Return type `numpy.ndarray`

`factor_analyzer.utils.`**`commutation_matrix`**(*p*, *q*)

Calculate the commutation matrix.

This matrix transforms the vectorized form of the matrix into the vectorized form of its transpose.

Parameters

- **p** (*int*) – The number of rows.
- **q** (*int*) – The number of columns.

Returns **commutation_matrix** – The commutation matrix

Return type `numpy.ndarray`

References

https://en.wikipedia.org/wiki/Commutation_matrix

factor_analyzer.utils.**corr**(*x*)
Calculate the correlation matrix.

Parameters **x** (*array-like*) – A 1-D or 2-D array containing multiple variables and observations. Each column of x represents a variable, and each row a single observation of all those variables.

Returns **r** – The correlation matrix of the variables.

Return type numpy array

factor_analyzer.utils.**cov**(*x*, *ddof=0*)
Calculate the covariance matrix.

Parameters

- **x** (*array-like*) – A 1-D or 2-D array containing multiple variables and observations. Each column of x represents a variable, and each row a single observation of all those variables.
- **ddof** (*int, optional*) – Means Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements. Defaults to 0.

Returns **r** – The covariance matrix of the variables.

Return type numpy array

factor_analyzer.utils.**covariance_to_correlation**(*m*)
Compute cross-correlations from the given covariance matrix.

This is a port of R `cov2cor()` function.

Parameters **m** (*array-like*) – The covariance matrix.

Returns **retval** – The cross-correlation matrix.

Return type `numpy.ndarray`

Raises `ValueError` – If the input matrix is not square.

factor_analyzer.utils.**duplication_matrix**(*n=1*)
Calculate the duplication matrix.

A function to create the duplication matrix (Dn), which is the unique n2 × n(n+1)/2 matrix which, for any n × n symmetric matrix A, transforms vech(A) into vec(A), as in Dn vech(A) = vec(A).

Parameters **n** (*int, optional*) – The dimension of the n x n symmetric matrix. Defaults to 1.

Returns

- **duplication_matrix** (`numpy.ndarray`) – The duplication matrix.
- Raises‘
- ——
- *ValueError* – If `n` is not a positive integer greater than 1.

References

https://en.wikipedia.org/wiki/Duplication_and_elimination_matrices

factor_analyzer.utils.**duplication_matrix_pre_post**(*x*)
Transform given input symmetric matrix using pre-post duplication.

Parameters **x** (*array-like*) – The input matrix.

Returns **out** – The transformed matrix.

Return type numpy.ndarray

Raises AssertionError – If x is not symmetric.

factor_analyzer.utils.**fill_lower_diag**(*x*)
Fill the lower diagonal of a square matrix, given a 1-D input array.

Parameters **x** (*array-like*) – The flattened input matrix that will be used to fill the lower diagonal of the square matrix.

Returns **out** – The output square matrix, with the lower diagonal filled by x.

Return type numpy.ndarray

References

[1] https://stackoverflow.com/questions/51439271/ convert-1d-array-to-lower-triangular-matrix

factor_analyzer.utils.**get_first_idxs_from_values**(*x*, *eq=1*, *use_columns=True*)
Get the indexes for a given value.

Parameters

- **x** (*array-like*) – The input matrix.
- **eq** (*str or int, optional*) – The given value to find. Defaults to 1.
- **use_columns** (*bool, optional*) – Whether to get the first indexes using the columns. If False, then use the rows instead. Defaults to True.

Returns

- **row_idx** (*list*) – A list of row indexes.
- **col_idx** (*list*) – A list of column indexes.

factor_analyzer.utils.**get_free_parameter_idxs**(*x*, *eq=1*)
Get the free parameter indices from the flattened matrix.

Parameters

- **x** (*array-like*) – The input matrix.
- **eq** (*str or int, optional*) – The value that free parameters should be equal to. np.nan fields will be populated with this value. Defaults to 1.

Returns **idx** – The free parameter indexes.

Return type numpy.ndarray

factor_analyzer.utils.**get_symmetric_lower_idxs**(*n=1*, *diag=True*)
Get the indices for the lower triangle of a symmetric matrix.

Parameters

- **n** (*int, optional*) – The dimension of the n x n symmetric matrix. Defaults to 1.
- **diag** (*bool, optional*) – Whether to include the diagonal.

Returns **indices** – The indices for the lower triangle.

Return type numpy.ndarray

factor_analyzer.utils.**get_symmetric_upper_idxs**(*n=1*, *diag=True*)
Get the indices for the upper triangle of a symmetric matrix.

Parameters

- **n** (*int, optional*) – The dimension of the n x n symmetric matrix. Defaults to 1.
- **diag** (*bool, optional*) – Whether to include the diagonal.

Returns **indices** – The indices for the upper triangle.

Return type numpy.ndarray

factor_analyzer.utils.**impute_values**(*x*, *how='mean'*)
Impute np.nan values with the mean or median, or drop the containing rows.

Parameters

- **x** (*array-like*) – An array to impute.
- **how** (*str, optional*) – Whether to impute the 'mean' or 'median'. Defaults to 'mean'.

Returns **x** – The array, with the missing values imputed or with rows dropped.

Return type numpy.ndarray

factor_analyzer.utils.**inv_chol**(*x*, *logdet=False*)
Calculate matrix inverse using Cholesky decomposition.

Optionally, calculate the log determinant of the Cholesky.

Parameters

- **x** (*array-like*) – The matrix to invert.
- **logdet** (*bool, optional*) – Whether to calculate the log determinant, instead of the inverse. Defaults to False.

Returns

- **chol_inv** (*array-like*) – The inverted matrix.
- **chol_logdet** (*array-like or None*) – The log determinant, if logdet was True, otherwise, None.

factor_analyzer.utils.**merge_variance_covariance**(*variances*, *covariances=None*)
Merge variances and covariances into a single variance-covariance matrix.

Parameters

- **variances** (*array-like*) – The variances that will be used to fill the diagonal of the square matrix.
- **covariances** (*array-like or None, optional*) – The flattened input matrix that will be used to fill the lower and upper diagonal of the square matrix. If None, then only the variances will be used. Defaults to None.

Returns **variance_covariance** – The variance-covariance matrix.

Return type numpy.ndarray

factor_analyzer.utils.**partial_correlations**(*x*)
Compute partial correlations between variable pairs.

This is a python port of the `pcor()` function implemented in the `ppcor` R package, which computes partial correlations for each pair of variables in the given array, excluding all other variables.

Parameters **x** (*array-like*) – An array containing the feature values.

Returns **pcor** – An array containing the partial correlations of of each pair of variables in the given array, excluding all other variables.

Return type numpy.ndarray

factor_analyzer.utils.**smc**(*corr_mtx*, *sort=False*)

Calculate the squared multiple correlations.

This is equivalent to regressing each variable on all others and calculating the r-squared values.

Parameters

- **corr_mtx** (*array-like*) – The correlation matrix used to calculate SMC.
- **sort** (*bool, optional*) – Whether to sort the values for SMC before returning. Defaults to `False`.

Returns **smc** – The squared multiple correlations matrix.

Return type numpy.ndarray

factor_analyzer.utils.**unique_elements**(*seq*)

Get first unique instance of every list element, while maintaining order.

Parameters **seq** (*list-like*) – The list of elements.

Returns **seq** – The updated list of elements.

Return type list

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

f

Index

V