## Contents:

1 Description .................................................. 3
2 Requirements ............................................. 5
3 Installation .................................................. 7
   3.1 factor_analyzer package ............................. 7
4 Indices and tables .......................................... 25
   Python Module Index ..................................... 27
   Index ....................................................... 29
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.
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 an 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)

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.
Requirements

- Python 3.4 or higher
- numpy
- pandas
- scipy
- scikit-learn
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
```

### 3.1 factor_analyzer package

#### 3.1.1 factor_analyzer.analyze Module

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

- **author** Jeremy Biggs (jbiggs@ets.org)
- **author** Nitin Madnani (nmadnani@ets.org)
- **date** 10/25/2017
- **organization** ETS

```python
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', rotation_kwargs=None)
```

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

A FactorAnalyzer class, which -
(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.*
  Methods 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 “LBFGS-B” optimization. Defaults to (0.005, 1).*
- **impute** *({'drop', 'mean', 'median'}, optional) – If missing values are present in the data, either use list-wise deletion (‘drop’) or impute the column median (‘median’) or column mean (‘mean’).*
- **use_corr_matrix** *(bool, optional) – Set to true if the data is the correlation matrix. Defaults to False.*
- **optional** *(rotation_kwargs,) – Additional key word arguments are passed to the rotation method.*

loadings

The factor loadings matrix. Default to None, if fit() has not been called.
**Type** numpy array

**corr**
The original correlation matrix. Default to None, if \texttt{fit()} has not been called.

**rotation_matrix**
The rotation matrix, if a rotation has been performed.

**structure**
The structure loading matrix. This only exists if the rotation is promax.

**psi**
The factor correlations matrix. This only exists if the rotation is oblique.

**Notes**
This code was partly derived from the excellent R package \texttt{psych}.

**References**

**Examples**

```python
>>> 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])
```

\texttt{fit(X, y=None)}

Fit the factor analysis model using either minres, ml, or principal solutions. By default, use SMC as starting guesses.

### 3.1. \texttt{factor_analyzer} package

9
Parameters

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

Examples

```python
>>> 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],
       [ 0.43874135, 0.61452341, -0.07255667],
       [ 0.45318006, 0.71926681, -0.07546472],
       [ 0.36688794, 0.44377343, 0.01737067],
       [ 0.74141382, -0.15008235, 0.29977512],
       [ 0.7416750 , -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 array

Examples

```python
>>> 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 array*) – The original eigen values
- **common_factor_eigen_values** (*numpy array*) – The common factor eigen values
Examples

```python
>>> 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 the factor variance information, including variance, proportional variance and cumulative variance for each factor

Returns

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

Examples

```python
>>> 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([3.51018854, 1.28371018, 0.73739507]),
   array([3.51018854, 1.28371018, 0.73739507]))
```

generate_uniquenesses()

Calculate the uniquenesses, given the factor loading matrix.

Returns

- uniquenesses (numpy array) – The uniquenesses from the factor loading matrix.

Return type

numpy array

Examples

```python
```
```ruby
>>> 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_kwarg={},
        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])
```

transform(X)
Get the factor scores for 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 array, shape (n_samples, n_components)`

**Examples**

```ruby
>>> 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_kwarg={},
        use_smc=True)
>>> fa.transform(df_features)
array([[-1.05141425,  0.57687826,  0.16587883],
       [-1.59940101,  0.89632125,  0.03824552],
       [-1.21768164, -1.16319406,  0.57135189],
       ...,
       [ 0.13601554,  0.03601086,  0.28813877],
       [ 1.86904519, -0.35323949, -0.68170573],
       [ 0.86133386,  0.18280695, -0.79170903]])
```

factor_analyzer.factor_analyzer.calculate_bartlett_sphericity(x)
Test the hypothesis that the correlation matrix is equal to the identity matrix.identity

**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:

\[-1 \times (n - 1 - ((2p + 5)/6)) \times ln(det(R))\]

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

**Parameters**

- **x** (*array-like*) – The array from 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 array) – The KMO score per item.
- kmo_total (float) – The KMO score overall.

3.1.3 factor_analyzer.confirmatory_factor_analyzer Module

Confirmatory factor analysis using ML.

author Jeremy Biggs (jbiggs@ets.org)
date 2/05/2019
organization ETS

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

A ConfirmatoryFactorAnalyzer class, which fits a confirmatory factor analysis model using maximum likelihood.

Parameters
- specification (ModelSpecification object or None, optional) – A model specification. This must be a ModelSpecification object or None. If None, the ModelSpecification 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=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:

\[(n_{factors} \times n_{variables}) + n_{variables} + n_{factors} + (((n_{factors} \times n_{factors}) - n_{factors}) / 2)\]

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.

Type: numpy array

**error_vars**

The error variance matrix.

Type: numpy array

**factor_varcovs**

The factor covariance matrix.

Type: numpy array

**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**

```python
>>> 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.  ]],
(continues on next page)```
factor_analyzer Documentation, Release 0.3.1

(continued from previous page)

```python
>>> 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=True` and the matrix is not square.

- **AssertionError** – If len(bounds) != len(x0)

Examples

```python
>>> 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)
```

(continues on next page)
get_model_implied_cov()

Get the model-implied covariance matrix (sigma), if the model has been estimated.

Returns  
model_implied_cov – The model-implied covariance matrix.

Return type  
numpy array

Examples

get_standard_errors()

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

Returns

• loadings_se (numpy array) – The standard errors for the factor loadings.
• error_vars_se (numpy array) – The standard errors for the error variances.
Examples

```python
>>> 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 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

```python
>>> 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.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 ]])
```
class factor_analyzer.confirmatory_factor_analyzer.ModelSpecification (loadings, n_factors, n_variables, factor_names=None, variable_names=None)

Bases: object

A class to 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.
  
  Type numpy array

- **error_vars** *(array-like)* – The error variance specification
  
  Type numpy array

- **factor_covs** *(array-like)* – The factor covariance specification.
  
  Type numpy array

- **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.

**loadings**

The factor loadings specification.

**error_vars**

The error variance specification

**factor_covs**

The factor covariance specification.

**n_factors**

The number of factors.

**n_variables**

The number of variables.

**n_lower_diag**

The number of elements in the factor_covs array, which is equal to the lower diagonal of the factor covariance matrix.

**loadings_free**

The indexes of “free” factor loading parameters.
error_vars_free
The indexes of “free” error variance parameters.
Type numpy array

factor_covs_free
The indexes of “free” factor covariance parameters.
Type numpy array

factor_names
A list of factor names, if available.
Type list of str or None

variable_names
A list of variable names, if available.
Type list of str or None

copy()

error_vars
error_vars_free
factor_covs
factor_covs_free
factor_names

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
loadings_free
n_factors
n_lower_diag
n_variables
variable_names

class factor_analyzer.confirmatory_factor_analyzer.ModelSpecificationParser
Bases: object

A class to generate the model specification for CFA. This class includes two static methods to generate the ModelSpecification object from either a dictionary or a numpy array.

static parse_model_specification_from_array(X, specification=None)
Generate the model specification from an 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, 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**

```python
>>> 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]])
>>> 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*, 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**

```python
>>> 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)
```

---

**3.1.4 factor_analyzer.rotator Module**

Rotator class to perform various rotations of factor loading matrices.
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)
  
  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 power to which to raise the promax loadings (minus 1). Numbers should generally range from 2 to 4. Defaults to 4.

- **kappa** *(int, 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.

- **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 array, shape (n_features, n_factors)

**rotation_**

The rotation matrix

**Type** numpy array, shape (n_factors, n_factors)

**psi_**

The factor correlations matrix. This only exists if the rotation is oblique.

**Type** numpy array 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

```python
>>> 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)*

Computes the factor rotation.

**Parameters**

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

**Returns**

**Return type** self

**Example**

```python
>>> 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)*

Computes the factor rotation, and returns the new loading matrix.

**Parameters**

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

**Returns** `loadings_` – The loadings matrix (n_features, n_factors)

**Return type** numpy array, shape (n_features, n_factors)

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

**Example**

```python
>>> 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, 0.55534740, -0.00121802],
       [ 0.77450832, 0.14746660, 0.20118338],
       [ 0.70630010, 0.17229555, -0.30093981],
       [ 0.83990851, 0.15058874, -0.06182469],
       [ 0.76620579, 0.10451940, -0.22649615],
       [ 0.81372945, 0.20915845, 0.07479506]])
```
CHAPTER 4

Indices and tables

• genindex
• search
factor_analyzer.confirmatory_factor_analyzer,
13
factor_analyzer.factor_analyzer, 7
factor_analyzer.rotator, 20
## Python Module Index

<table>
<thead>
<tr>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>factor_analyzer</td>
</tr>
</tbody>
</table>

factor_analyzer Documentation, Release 0.3.1
Index

A
aic_ (factor_analyzer.confirmatory_factor_analyzer.ConfirmatoryFactorAnalyzer attribute), 14

B
bic_ (factor_analyzer.confirmatory_factor_analyzer.ConfirmatoryFactorAnalyzer attribute), 14

C
calculate_bartlett_sphericity() (in module factor_analyzer.factor_analyzer), 12
calculate_kmo() (in module factor_analyzer.factor_analyzer), 13
ConfirmatoryFactorAnalyzer (class in factor_analyzer.confirmatory_factor_analyzer), 13
copy() (factor_analyzer.confirmatory_factor_analyzer.ModelSpecification method), 19
corr (factor_analyzer.factor_analyzer.FactorAnalyzer attribute), 9

E
error_vars (factor_analyzer.confirmatory_factor_analyzer.ModelSpecification attribute), 18, 19
error_vars_ (factor_analyzer.confirmatory_factor_analyzer.ConfirmatoryFactorAnalyzer attribute), 14
error_vars_free (factor_analyzer.confirmatory_factor_analyzer.ModelSpecification attribute), 18, 19

F
factor_analyzer.confirmatory_factor_analyzer (module), 13
factor_analyzer.factor_analyzer (module), 7
factor_analyzer.rotator (module), 20
factor_covs (factor_analyzer.confirmatory_factor_analyzer.ModelSpecification attribute), 18, 19
factor_covs_free (factor_analyzer.confirmatory_factor_analyzer.ModelSpecification attribute), 19
factor_names (factor_analyzer.confirmatory_factor_analyzer.ModelSpecification attribute), 19
factor_varcovs_ (factor_analyzer.confirmatory_factor_analyzer.ConfirmatoryFactorAnalyzer attribute), 14
FactorAnalyzer (class in factor_analyzer.factor_analyzer), 7
fit() (factor_analyzer.confirmatory_factor_analyzer.ConfirmatoryFactorAnalyzer method), 15
fit() (factor_analyzer.factor_analyzer.FactorAnalyzer method), 9
fit() (factor_analyzer.rotator.Rotator method), 22
fit_transform() (factor_analyzer.rotator.Rotator method), 22

G
get_communalities() (factor_analyzer.factor_analyzer.FactorAnalyzer method), 10
get_eigenvalues() (factor_analyzer.factor_analyzer.FactorAnalyzer method), 11
get_factor_variance() (factor_analyzer.factor_analyzer.FactorAnalyzer method), 11
get_model_implied_cov() (factor_analyzer.confirmatory_factor_analyzer.ConfirmatoryFactorAnalyzer method), 16
get_model_specification_as_dict() (factor_analyzer.confirmatory_factor_analyzer.ModelSpecification method), 19
get_standard_errors() (factor_analyzer.confirmatory_factor_analyzer.ConfirmatoryFactorAnalyzer method), 19
get_uniquenesses() (factor_analyzer.factor_analyzer.FactorAnalyzer method), 11
method), 11 Rotator (class in factor_analyzer.rotator), 21