ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/325647424

Varimax Rotation and Thereafter: Tutorial on PCA Using Linear Algebra, Visualization, and Python Programming for R and Q anal....

Article *in* Journal of Research Methodology · May 2018

DOI: 10.21487/jrm.2018.5.3.1.79

CITATION
1

READS
8,553

1 author:

Byung S. Lee
Elon University

11 PUBLICATIONS **127** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project

Q Methodology software tools View project

Project

Presidential Candidates' Twitter Use and the Linkage Pattern of Twitter users in the 2012 Presidential Election View project

All content following this page was uploaded by Byung S. Lee on 17 June 2018.

The user has requested enhancement of the downloaded file.

Citation: Lee, B. (2018). Varimax rotation and thereafter: Tutorial on PCA using linear algebra, visualization, and python programing for R and Q analysis. Journal of Research Methodology, 3(1), 79-130.

[DOI] http://dx.doi.org/10.21487/jrm.2018.5.3.1.79

【연구경향】

Varimax Rotation and Thereafter: Tutorial on PCA Using Linear Algebra, Visualization, and Python Programming for R and Q analysis

Byung Lee*

ABSTRACT

Principal component analysis (PCA) has been heavily used for both academic and practical purposes. This tutorial would help individuals who want to better utilize PCA as well as R scholars interested in Q analysis. Many statistical commercial packages can handle PCA, but one may reach a deeper understanding by running PCA code for oneself. Cranking out results using PCA features on one of the commercial statistical packages is easy, just importing data and clicking a few buttons. For a deep understanding of PCA, however, an individual needs to see what happens when running code. Many scholars may have a basic knowledge of a popular programming language like Python. But PCA code in Python is not neatly compiled in one place. A user must gather Python code that is scattered around the Internet, tweak it for compatibility, and fill any remaining gaps. Also, commercial statistical packages, which have been geared toward R analysis, cannot be used for rotation for theoretical or exploratory analysis in Q studies beyond varimax. This tutorial offered all Python code needed for PCA while comparing its results with Statistical Package for the Social Science (SPSS) output. This tutorial also covered the theoretical or exploratory rotation of factor axes, which is a must for Q analysis.

Keywords : PCA, Principal Component Analysis, linear algebra, graphs, Python code, varimax rotation, and R and Q analysis

* Associate Professor, Elon University(byunglee@elon.edu)

I. Introduction to PCA

A) What is PCA?

"A principal component analysis can be considered as a rotation of the axes of the original variable coordinate system to new orthogonal axes, called principal axes, such that the new axes coincide with directions of maximum variation of the original observations" (Campbell & Atchley, 1981, pp. 268).

Items, whether human beings or objects, can be observed by measuring their characteristics or attributes, so-called variables. These variables constitute data dimensions. When graphically displayed, they represent coordinate axes.

If one chooses to have the same number of principal components in PCA as original variables, for example, 10, one can account for all variations that the original ones can. PCA finds principal components in descending order of variations explained. The front components account for more variations than the later ones. The 1st principal component accounts for the maximum amount of variations possible in data, and the 2nd principal component extracts the maximum possible variations in data after excluding what was explained by the 1st component. Extractions can be done until all the variations are accounted for by the last principal component.

Original variables vary together in some pattern. PCA takes advantage of this covariation and accounts for more variations than the original ones in the case of the front-end principal components. If one keeps all principal components, there is no benefit. The benefit arises when it is used for data reduction by discarding the backend principal components, which can be interpreted as noise or small enough to be ignored without causing much loss to the original data. Also, the principal components of PCA are statistically independent of each other.

To understand the central factors that influence political behavior, for

example, a study (Markaki, Chadjipandelis, & Tomaras, 2014) using PCA extracted eight factors out of the original 64, explaining 34% of the total variation. Through PCA, the original dataset was reduced to a new dataset, which is one-eighth of the original amount.

PCA is "a data-reduction technique that transforms a larger number of correlated variables into a much smaller set of uncorrelated variables called principal components" (Kabacoff 2015, p. 319).

B) The purpose of this manuscript

To understand PCA and use it correctly, it requires more than punching numbers into software incorporating the PCA feature and cranking out results. It requires a linear algebra background to understand its theoretical foundation. Abstract concepts in linear algebra can be more easily followed by graphical geometrical displays. The operation of PCA beyond three dimensions can be better understood by actually running computer code on data.

Not much literature exists to simultaneously cover these three areas--linear algebra, visualization, and computer code--at a level that social scientists without much mathematical background are able to understand. Many books on factor analysis have described PCA as part of factor analysis. Since it involves linear algebra, some factor analysis books even include an introduction to matrix operations (Horst 1965; Cureton & D'Agostino, 1983; Gorsuch 1983). Linear algebra books always include a chapter on eigenvalues and eigenvectors, but only recently these books began to include PCA around at their end (Strang 2016; Lay 2005). Popular computer packages like SPSS, SAS, and Stata have PCA features, but how these programs operate under the hood is hidden. Many social scientists are familiar with Python, a general-purpose scripting language, but Python codes for PCA are often scattered around the Internet without much annotation. Python code has been

developed mainly for R analysis (traditional factor analysis in which variables are correlated), not for Q analysis, in which research participants are correlated, and exploratory or theoretical rotations of factor axes beyond varimax rotation are adopted. The commercial statistical programs are not different. The author found that literature rarely cover all three approaches together[1]: numerical analysis based on linear algebra, geometrical displays, and Python programming code, with which researchers can dirt their hands with data.

II. Graphical Geometric Explanation

This article will start with a visual description of PCA, the easiest among the three approaches. For a graphical illustration of PCA process, the author created a sample dataset measuring two variables with 30 observations in this example: people's favor for Chocolate and their favor for Candy, as shown below. This dataset is posted at "http://bestelon.com/pca/pca.csv".

	A	B	C
1	name	x_value	y_value
2	A	2.7	4.6
3	B	15	15
4	C	23.9	9.5
5	D	14.3	9.3
27	Z	20.4	16.6
28	A1	32.6	17.4
29	B1	28.6	12.7
30	C1	18.1	12.3
31	D1	46.1	7.9

1) When factor analysis began to be developed, some book authors introduced Fortran code along with linear algebra and geometrical illustrations.

A) Import the original dataset

```
import pandas as pd
filepath = "http://bestelon.com/pca/pca.csv"
df = pd.read_csv(filepath)
```

B) Draw a graph based on the dataset

```
import numpy as np
import matplotlib.pyplot as plt
import random
labels = df["name"]

random.seed( 5 )
randColor = np.random.random(30)

# to set the figure size explicitly
plt.figure(figsize=(6, 6))

plt.scatter( df.iloc[:, 1], df.iloc[:, 2], marker='o',
c=randColor , s = 100,  cmap=plt.get_cmap('Spectral') )

#c for color or sequence of color; s for size
# marker for marker style
#spectral for the black-purple-blue-green-yellow-red-
white spectrum
# refer to https://matplotlib.org/api/_as_gen/matplotlib.
pyplot.scatter.html

for label, x, y in zip(df.iloc[:, 0], df.iloc[:, 1],
df.iloc[:, 2]):
    plt.annotate(
        label,
        xy=(x, y), xytext=(5, 5),
        textcoords='offset points', ha='right', va='bottom')

plt.xlim(0,60)
plt.ylim(0, 60)
plt.xlabel('Favor for Chocolate')
plt.ylabel('Favor for Candy')
plt.title("Figure 1. People's favor for Chocolate vs.
Candy",fontsize = 16)
plt.show()
```

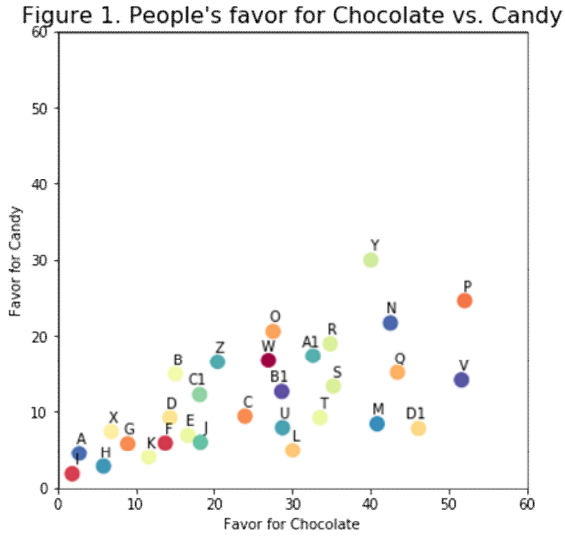
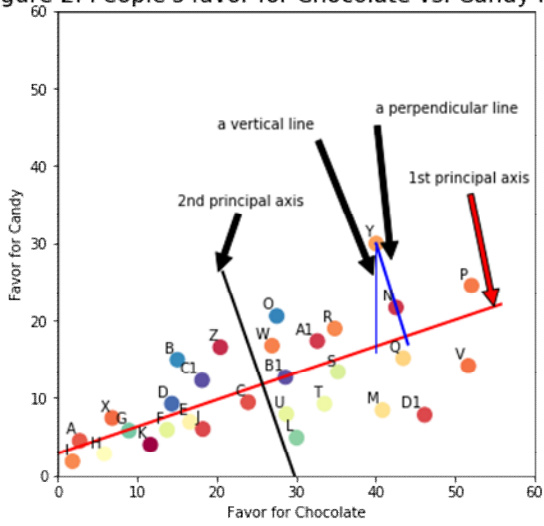


Figure 1 shows 30 people in a coordinate system who favor chocolate on the x-axis and who favor candy on the y-axis. Here one can see a positive correlation between people's favor for chocolate and favor for candy. Rather than measuring the people's favor twice to understand their taste, a researcher can measure favor for sweets at once with a straight line that goes through the middle of all dots.

The straight line should be placed so that the distance between each dot and the line is the shortest, more specifically when the sum of the squared distance for each dot is the smallest. The distance from the line here indicates the perpendicular distance, not the vertical distance (refer to Figure 2).

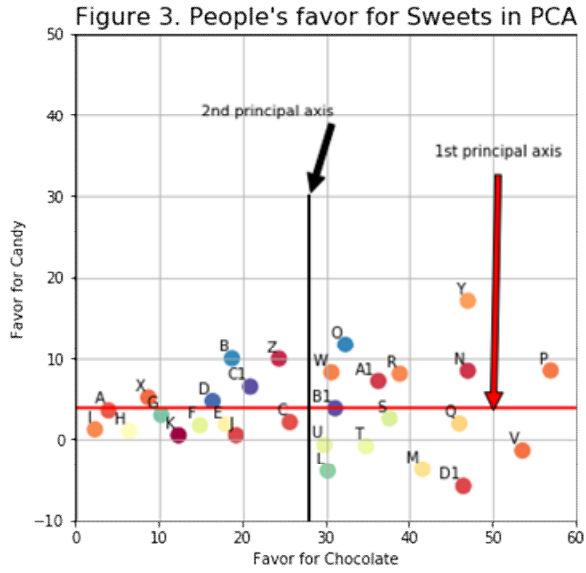
Then the line becomes a new axis, called the 1st principal component axis, and another line that is perpendicular to the 1st component axis, the shortest horizontally off all dots when measured from the 2nd component axis itself, becomes the 2nd principal component.

Figure 2. People's favor for Chocolate vs. Candy in PCA



C) Perspective change

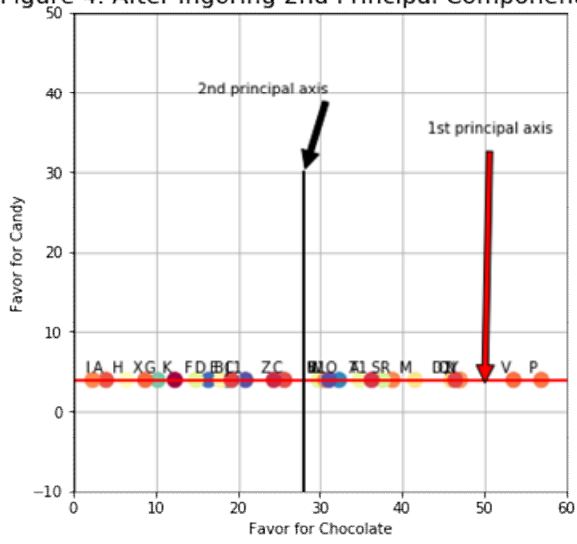
To better understand what the axis change means, let's look at the configuration of dots from a new perspective, meaning under new coordinates. To do that, one can imagine oneself hovering over the graph. Let's rotate oneself by about 15 degrees counterclockwise to align the 1st principal axis horizontally and the 2nd principal axis line vertically. They will be a new x-axis and y-axis respectively. Or you can just rotate the graph by 15 degrees clockwise so that the 1st and 2nd principal component axes will become x- and y-axes (refer to Figure 3). PCA allows one to see the data from a new frame of reference.



D) Discard back-end principal components

If one discards the 2nd principal component for data reduction, it means ignoring the data that is accounted for by this principal component. This disregard results in the effect that each data dot in Figure 3 moves to a new position as shown in Figure 4.

Figure 4. After Ingoring 2nd Principal Component Axis



III. Numerical Analysis: Comparison Between Eigendecomposition and SVD

The second approach is based on numerical analysis using linear algebra. Among many methods to find eigenvalues and eigenvectors, this article covered eigendecomposition and SVD.

A) Formula for calculating variance/covariance or correlation

In PCA, one needs to find eigenvalues and eigenvectors. One can use eigendecomposition, which uses the formula of $Ax = \lambda x$. Here A is an n x n matrix, x is a vector, and λ is a scalar. When a vector, x, is premultiplied by A, it can be a scalar multiple of x. If the x is not a zero vector, then the x becomes an eigenvector and the λ an eigenvalue.

Eigenvectors match the direction of principal component axes. Eigenvalues

and eigenvectors are paired. If an eigenvector pairs with a higher eigenvalue, it matches with a more front-end principal component. For example, if an eigenvector pairs with the highest eigenvalue, it matches the 1st principal component axis.

Since eigendecomposition requires a symmetric matrix, either the covariance or correlation of the dataset is used instead of the original dataset in rectangular format. If variables are measured using the same or similar measures, one can use a variance/covariance matrix, as shown in Equation 1. Covariance and variance can be calculated with equations 2, 3 and 4. To best estimate the variance or covariance of the population data, (n-1) is used with a sample dataset instead of n, the number of observations.

$$\begin{pmatrix} Var(x,x) & Cov(x,y) \\ Cov(y,x) & Var(y,y) \end{pmatrix} \qquad \text{Equation 1}$$

$$Cov(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(xi - \bar{x})(yi - \bar{y}) \qquad \text{Equation 2}$$

$$Var(x,x) = \frac{1}{n-1}\sum_{i=1}^{n}(xi - \bar{x})(xi - \bar{x}) \qquad \textit{Equation 3}$$

$$Var(y,y) = \frac{1}{n-1}\sum_{i=1}^{n}(yi - \bar{y})(yi - \bar{y}) \qquad \textit{Equation 4}$$

If variables are calculated with different measures, one is recommended to use correlations since the measurement unit may influence PCA results. For example, the use of inches or feet to measure heights will make a difference. The formula for calculating the correlation between x and y variables is shown in Equation 5. The correlation between x and x or between y and y can be calculated in the same way by substituting x for y or y for x.

$$Corr(x,y) = \frac{Cov(x,y)}{\sqrt{Var(x)}\sqrt{Var(y)}} \quad Equation\ 5$$

B) Center or standardize data

The original dataset was converted below into dfO (the original 30 x 3 dataset matrix), dfM (a centered dataset by subtracting the mean value from each observation), and dfZ (a standardized dataset). dfO2, dfM2, and dfZ2 represent the numerical portion of dfO, dfM, and dfZ, these datasets' last two columns each.

```
1. df0 = df
2. df02 = df.iloc[:, 1:]
3. dfM2 = df02 - df02.mean()
4. dfZ2 = (df02 - df02.mean()) / df02.std()
5. dfM = dfM2.copy()
6. dfM.insert(loc = 0, column = "name", value = df0.iloc[:, 0])
7. dfZ = dfZ2.copy()
8. dfZ.insert(loc = 0, column = "name", value = df0.iloc[:, 0])
```

C) Eigendecomposition with variances/covariances

Variance and covariance can be calculated using either the original dataset or centered dataset.

```
1. df02Cov = df02.cov()
2. print(df02Cov)
3. dfM2Cov = dfM2.cov()
4. print(dfM2Cov)
```

The covariance was plugged into the np.linalg.eig function to produce eigenvalues and eigenvectors. The resulting eigenvalues and their corresponding eigenvectors are sorted in descending order below:

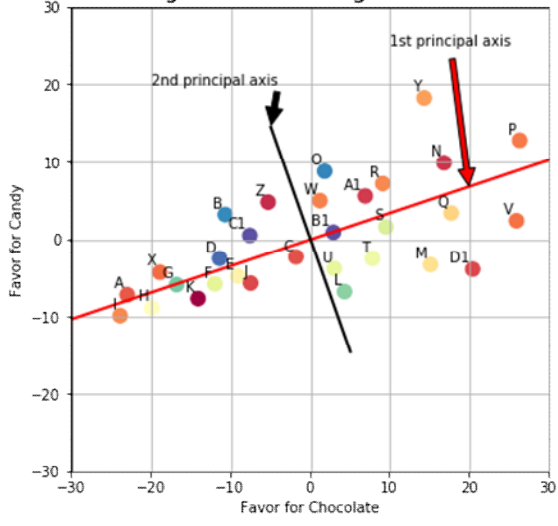
```
1. dfM2Cov = dfM2.cov()
2. eValM2Cov, eVecM2Cov = np.linalg.eig(dfM2Cov)
3.
4. #sort results based on eigenvalues
5. idxM2Cov = eValM2Cov.argsort()[::-1]
6. eValM2Cov, eVecM2Cov = eValM2Cov[idxM2Cov], eVecM2Cov[:,
   idxM2Cov]
7. print(eValM2Cov.round(3))
8. print(eVecM2Cov.round(3))
```

The first line in the following code indicates two eigenvalues, while lines 2 and 3 indicate two eigenvectors:

```
1. [ 231.35    25.482]
2. [[ 0.946 -0.325]
3.  [ 0.325  0.946]]
```

If centered values are plotted and lines are drawn based on the eigenvectors above, one can see exactly the same graph, as seen in figures 1 and 2, except for a movement of the center point by the mean of x and y variables (refer to Figure 5).

Figure 5. Centering the Data



D) Eigendecomposition with correlations

Correlations among variables can be calculated using any of the following four different ways: the covariance of the standardized dataset, or the correlation of the original dataset, the centered dataset, or the standardized dataset.

```
dfZ2Cov = dfZ2.cov()
df02Corr = df02.corr()
dfM2Corr = dfM2.corr()
dfZ2Corr = dfZ2.corr()
```

A matrix of correlations was plugged into the np.linalg.eig function to calculate eigendecomposition:

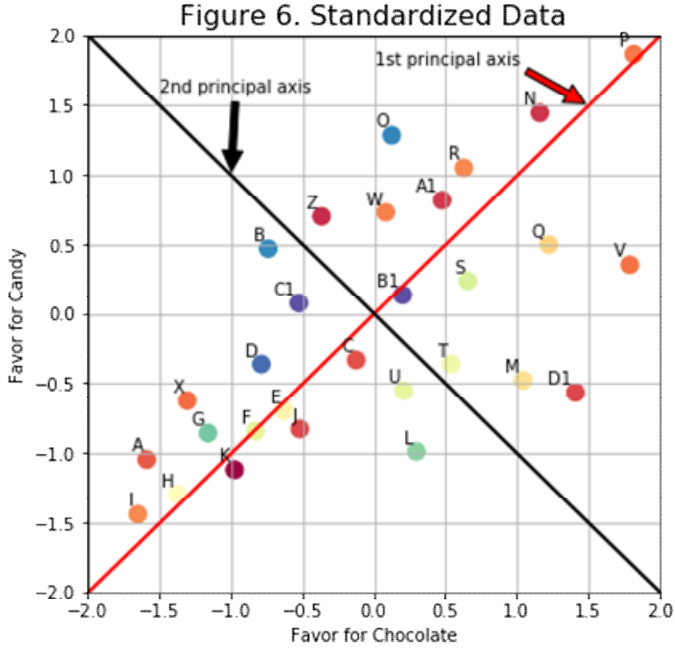
```
covZ = dfZ.corr()
eValZ1, eVecZ1 = np.linalg.eig(covZ)

#sort results based on eigenvalues
idxZ = eValZ1.argsort()[::-1]
eValZ1, eVecZ1 = eValZ1[idxZ], eVecZ1[:, idxZ]
print(eValZ1.round(3))
print(eVecZ1.round(3))
```

The result below differs from what the eigendecomposition produced with variances/covariances earlier:

```
[ 1.636  0.364]
[[ 0.707 -0.707]
 [ 0.707  0.707]]
```

The data standardization changes the variation of x and y values. Now x and y variables have the same amount of variance, 1. In Figure 6, one can

see the data dots are more widely scattered vertically than they were. It strengthens the influence of variables that were weak relative to stronger ones. Now the 1st principal component is only about 4.5 times as strong as the 2nd principal component. Before the standardization, it was 9.1 times.



E) Singular value decomposition (SVD) with a symmetric data matrix

Among the many ways to decompose a matrix, SVD is preferred by researchers because it always decomposes a matrix, while eigendecomponsion or other methods may fail to do it. Eigendecomposition has to use a symmetric matrix while SVD can use any rectangular matrix including a square form.

SVD produces eigenvalues and eigenvectors with a dataset in a symmetric

form, which is created by premultiplying a dataset with a transpose of itself like M.T.dot(M). Here M indicates a data matrix, T represents a transpose of the matrix, and "dot" means an inner product of the two matrices. If one divides this product of the two matrices by n-1 (n representing the number of observation), it will be the same as a variance/covariance matrix shown in Equation 1 earlier.

```
1. n = 30
2. import numpy as np
3. covMat2 = (1/ (n-1)) *dfM2.T.dot(dfM2)
4. U, S, V = np.linalg.svd(covMat2)
5. print(S)
6. print(V)
```

This code above produced the same result as the eigendecomposition method in C) Eigendecomposition with variances/covariances above, except for the sign of eigenvectors. The first line below represents eigenvalues and the rest, eigenvectors.

```
[ 231.35017587   25.48200804]
[[-0.94582661 -0.32467218]
 [-0.32467218  0.94582661]]
```

The line 3 of the code above used a division of a matrix by (n-1). The following code used the same code without the division:

```
1. import numpy as np
2. covMat2 = dfM2.T.dot(dfM2)
3. U, S, V = np.linalg.svd(covMat2)
4. print(S)
5. print(V)
```

The result shows the same eigenvectors, so the directions of eigenvectors remain the same. But the eigenvalues are (n-1) times as big as the earlier ones:

```
[ 6709.15510011   738.97823322]
[[-0.94582661 -0.32467218]
 [-0.32467218  0.94582661]]
```

F) Singular value decomposition (SVD) with a rectangular data matrix

SVD can decompose the original centered data matrix without relying on the correlation or covariance of datasets:

```
import numpy as np
U, S, V = np.linalg.svd(dfM2)
print(S)
print(V)
```

The result of the code above is:

```
[ 81.9094323   27.18415408]
[[ 0.94582661  0.32467218]
 [ 0.32467218 -0.94582661]]
```

The vectors in the second and third line are the same except for the sign difference. The numbers in the first line calculated with SVD using the non-symmetric matrix are called singular values. If each is multiplied by itself, however, it becomes the same as the eigenvalue of the symmetric metrix in E) Singular value decomposition (SVD) with a symmetric data matrix above.

```
81.9094323 * 81.9094323 = 6709.1511
27.18415408 * 27.18415408 = 738.978233
```

PCA can use any of these two methods, eigendecomposition or SVD, although the latter is more stable than the former since it always succeeds in producing eigenvalues and eigenvectors.

IV. Analysis of a Real Dataset

A) Lipset dataset

In real life, the data are more complex than the 2 dimensions the current author used as an example. If there are more than 3 variables, it is impossible to display the data graphically, so one has to deal with PCA by the numerical or computer coding approach.

The following data, so-called the Lipset data[2)], has been often used as an example by Q community when it illustrates how to use PCA and factor analysis for Q analysis. Generally, Q analysis is understood in R matrix users as correlating people instead of variables in PCA or factor analysis. In fact, the Q researchers who subscribe to William Stephenson's ideas use Q analysis only with a dataset collected in a specific way. They try to understand people's subjective schema on any events, people, issues, etc. To do so, the Q researchers usually select questions based on stratified sampling and have research participants consider other question items while rank ordering each

2) The questions items for the Lipset data were made by William Stephenson, and the Lipset data themselves were collected by Stephen Brown. This dataset as a sample is also included in the most popular Q analysis package, PQMethod. The author uses this after getting permission from Brown. For python analysis, the dataset has been adapted and posted at http://bestelon.com/pca/lipset.csv.

question item on a card in a Gestalt way in the two-step sorting process (Brown 1980; Stephenson 1953).

	A	B	C	D	E	F	G	H	I	J
1	item	v1	v2	v3	v4	v5	v6	v7	v8	v9
2	p01	-1	-1	2	3	-4	1	2	-2	3
3	p02	0	0	-2	1	-1	-3	0	2	1
4	p03	-2	-1	-2	-3	3	0	-2	0	0
5	p04	0	-3	4	-1	-1	3	1	-3	1
28	p27	4	2	1	-1	3	0	0	-4	-1
29	p28	2	3	-1	2	0	4	4	3	-4
30	p29	2	-1	0	0	1	-1	-3	0	0
31	p30	-3	-4	-1	2	-2	2	1	-2	-1
32	p31	-2	-2	-1	1	1	1	-1	-2	3
33	p32	-2	-3	4	2	-2	2	3	-2	-1
34	p33	4	4	1	0	-1	-3	-2	2	4

B) Import dataset and calculate correlations among variables

SPSS is the statistical package that quantitative social scientists are familiar with. As a reference point, SPSS results are compared with the results from Python code. First, let's import the Lipset data with 33 items and 9 variables:

```
1. import pandas as pd
2. filepath = "http://bestelon.com/pca/lipset.csv"
3. df = pd.read_csv(filepath)
4. print(df)
```

Correlations among variables were calculated with the following:

```
1. # select only numbers
2. numCols = df.iloc[:, 1:10]
3. numCorr = numCols.corr()
4. print(numCorr)
```

The calculation produced the 9 x 9 correlation matrix below:

```
          v1       v2       v3       v4       v5       v6       v7       v8       v9
v1   1.00000  0.53750  0.20625  0.22500  0.10000 -0.22500 -0.31875  0.23750  0.05000
v2   0.53750  1.00000 -0.07500  0.08750  0.17500 -0.02500 -0.16250  0.38125  0.06875
v3   0.20625 -0.07500  1.00000  0.40000 -0.54375  0.08750  0.05000 -0.09375  0.11250
v4   0.22500  0.08750  0.40000  1.00000 -0.56250  0.27500  0.16875  0.05625  0.02500
v5   0.10000  0.17500 -0.54375 -0.56250  1.00000 -0.06250 -0.13125  0.01875 -0.03125
v6  -0.22500 -0.02500  0.08750  0.27500 -0.06250  1.00000  0.61875 -0.36875 -0.21250
v7  -0.31875 -0.16250  0.05000  0.16875 -0.13125  0.61875  1.00000 -0.28750 -0.03125
v8   0.23750  0.38125 -0.09375  0.05625  0.01875 -0.36875 -0.28750  1.00000 -0.21250
v9   0.05000  0.06875  0.11250  0.02500 -0.03125 -0.21250 -0.03125 -0.21250  1.00000
```

SPSS produced the exact same correlation matrix, except for the numbers being rounded to 3 decimal places:

Correlation Matrix

		v1	v2	v3	v4	v5	v6	v7	v8	v9
Correlation	v1	1.000	.538	.206	.225	.100	-.225	-.319	.238	.050
	v2	.538	1.000	-.075	.088	.175	-.025	-.163	.381	.069
	v3	.206	-.075	1.000	.400	-.544	.088	.050	-.094	.113
	v4	.225	.088	.400	1.000	-.563	.275	.169	.056	.025
	v5	.100	.175	-.544	-.563	1.000	-.063	-.131	.019	-.031
	v6	-.225	-.025	.088	.275	-.063	1.000	.619	-.369	-.213
	v7	-.319	-.163	.050	.169	-.131	.619	1.000	-.288	-.031
	v8	.238	.381	-.094	.056	.019	-.369	-.288	1.000	-.213
	v9	.050	.069	.113	.025	-.031	-.213	-.031	-.213	1.000

C) Eigendecomposition using np.linalg.eig

The Python code below shows how to use the Python command of np.linalg.eig( ) for eigendecomposition:

```
import numpy as np
eVal_corr, eVec_corr = np.linalg.eig(numCorr)
idx_corr = eVal_corr.argsort()[::-1]
eVal_corr = eVal_corr[idx_corr]
eVec_corr = eVec_corr[:, idx_corr]
print(eVal_corr.round(3))
print(eVec_corr.round(3))
```

The code above produced nine eigenvalues and eigenvectors:

```
[ 2.384  2.015  1.341  1.127  0.732  0.512  0.401  0.32   0.168]
[[ 0.312  0.427 -0.197 -0.281 -0.424  0.08  -0.435 -0.266  0.396]
 [ 0.324  0.276 -0.468 -0.317  0.197 -0.157  0.539 -0.21  -0.317]
 [-0.251  0.47   0.229 -0.018 -0.378 -0.607  0.045  0.309 -0.233]
 [-0.27   0.511 -0.148  0.041  0.179  0.599 -0.214  0.258 -0.374]
 [ 0.34  -0.422 -0.25  -0.29  -0.219 -0.033 -0.32   0.521 -0.37 ]
 [-0.447 -0.076 -0.511 -0.153 -0.115  0.033  0.281  0.392  0.513]
 [-0.46  -0.128 -0.333 -0.155  0.304 -0.379 -0.494 -0.362 -0.158]
 [ 0.366  0.225 -0.202  0.459  0.481 -0.309 -0.213  0.345  0.27 ]
 [ 0.017  0.09   0.439 -0.69   0.471 -0.023 -0.043  0.218  0.225]]
```

D) Eigendecomposition using np.linalg.svd for SVD

SVD can produce eigenvalues and eigenvectors:

```
U, S, V = np.linalg.svd(numCorr)
print(S.round(3))
print(V.T.round(3))
[ 2.384  2.015  1.341  1.127  0.732  0.512  0.401  0.32   0.168]
[[-0.312  0.427  0.197 -0.281  0.424 -0.08   0.435 -0.266 -0.396]
 [-0.324  0.276  0.468 -0.317 -0.197  0.157 -0.539 -0.21   0.317]
 [ 0.251  0.47  -0.229 -0.018  0.378  0.607 -0.045  0.309  0.233]
 [ 0.27   0.511  0.148  0.041 -0.179 -0.599  0.214  0.258  0.374]
 [-0.34  -0.422  0.25  -0.29   0.219  0.033  0.32   0.521  0.37 ]
 [ 0.447 -0.076  0.511 -0.153  0.115 -0.033 -0.281  0.392 -0.513]
 [ 0.46  -0.128  0.333 -0.155 -0.304  0.379  0.494 -0.362  0.158]
 [-0.366  0.225  0.202  0.459 -0.481  0.309  0.213  0.345 -0.27 ]
 [-0.017  0.09  -0.439 -0.69  -0.471  0.023  0.043  0.218 -0.225]]
```

The two methods, eigendecomposition and SVD, show the same results except for the reversed signs of some values. This occurs because eigenvectors are not unique. Multiplying by any constant gives another valid eigenvector. This is clear given the formula describing an eigenvector: $A \cdot v = \lambda \cdot v$. If all values measuring a concept, for example, love, switch their number signs, they will mean all values for a concept of hate. To get consistent results, the author calculated the sum of eigenvectors for each column and flip the sign by multiplying each by -1 if their sum is less than 0.

```
def flip_vector_sign(eVec):
    for i in range( eVec.shape[1]): ## changed from 1 to 0 inside shape
        if (eVec[:, i].sum() < 0):
            eVec[:, i]  = -1 * eVec[:, i]

    return eVec
```

The function above was called below:

```
# To print each row on one line
np.set_printoptions(linewidth = 100)
Vt = flip_vector_sign(V.T)
Vt.round(3)
```

A new 9 x 9 matrix was produced:

```
array([[-0.312,  0.427,  0.197,  0.281, -0.424, -0.08 ,  0.435, -0.266, -0.396],
       [-0.324,  0.276,  0.468,  0.317,  0.197,  0.157, -0.539, -0.21 ,  0.317],
       [ 0.251,  0.47 , -0.229,  0.018, -0.378,  0.607, -0.045,  0.309,  0.233],
       [ 0.27 ,  0.511,  0.148, -0.041,  0.179, -0.599,  0.214,  0.258,  0.374],
       [-0.34 , -0.422,  0.25 ,  0.29 , -0.219,  0.033,  0.32 ,  0.521,  0.37 ],
       [ 0.447, -0.076,  0.511,  0.153, -0.115, -0.033, -0.281,  0.392, -0.513],
       [ 0.46 , -0.128,  0.333,  0.155,  0.304,  0.379,  0.494, -0.362,  0.158],
       [-0.366,  0.225,  0.202, -0.459,  0.481,  0.309,  0.213,  0.345, -0.27 ],
       [-0.017,  0.09 , -0.439,  0.69 ,  0.471,  0.023,  0.043,  0.218, -0.225]])
```

E) Select principal components

Many ways of deciding on the number of factors have been suggested (Brown 1980, pp. 222-224; Field 2005, pp. 632-634; Watts & Stenner, 2012, pp. 105-110). They include checking the inflection point of the scree plot as the cut-off point or counting principal components whose eigenvalues are larger than a specific number like 1 or 0.7. If these two methods are used, the following code would be beneficial since it organizes eigenvalues for individual principal components and those in cumulative form:

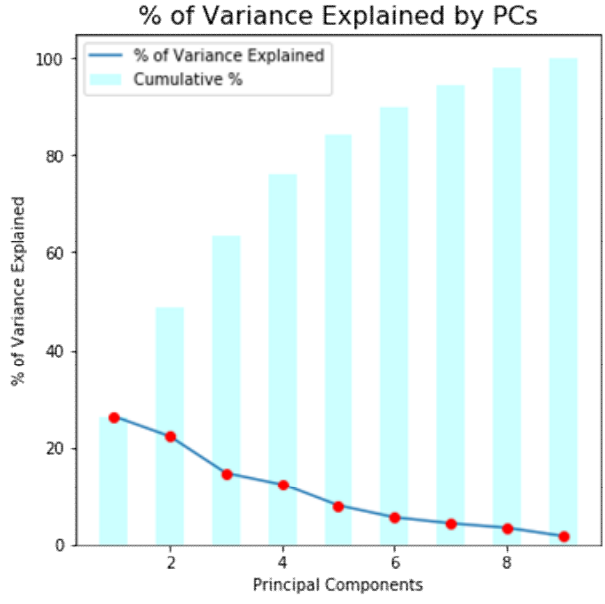
```
exp = eVal_corr *100 / np.sum(eVal_corr)
accSum = np.cumsum(exp)
pcNum = list(range(1, 10) )
data = np.array([pcNum, eVal_corr, exp, accSum])
eigenValues = pd.DataFrame(data.T,   columns = ['PC #', 'E
igenvalue', "% of Variance Exp", "Cumulative %"])
#Keep original for later use
eigenNumbers = eigenValues.copy()
format_mapping={'PC #': '{:,.0f}', 'Eigenvalue': '{:,.3f}'
, '% of Variance Exp': '{:.3f}%', 'Cumulative %': '{:.3f}%'}
for key, value in format_mapping.items():
    eigenValues[key] = eigenValues[key].apply(value.format)
eigenValues
```

	PC #	Eigenvalue	% of Variance Exp	Cumulative %
0	1	2.384	26.486%	26.486%
1	2	2.015	22.393%	48.879%
2	3	1.341	14.898%	63.777%
3	4	1.127	12.526%	76.303%
4	5	0.732	8.136%	84.439%
5	6	0.512	5.685%	90.125%
6	7	0.401	4.459%	94.584%
7	8	0.320	3.551%	98.135%
8	9	0.168	1.865%	100.000%

The following code was written to plot eigenvalues:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(6, 6))
eachExp = eigenNumbers.iloc[:, 2]
plt.bar(pcNum, accSum, width = 0.5,  color = 'cyan', alpha
= 0.2, label = "Cumulative %")
plt.plot(pcNum, eachExp, label = "% of Variance Explained")
plt.plot(pcNum, eachExp, 'ro', label = '_nolegend_')
plt.xlabel("Principal Components")
plt.ylabel("% of Variance Explained")
plt.title("% of Variance Explained by PCs", fontsize = 16)
plt.legend(loc = 'upper left')
plt.show()
```

A scree plot and a bar chart for the cumulated percentage of variance were drawn in the same graph.

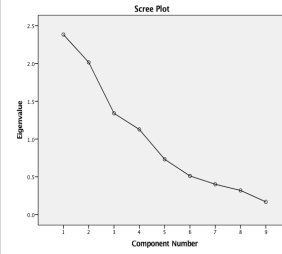


SPSS produced similar outcomes:

Total Variance Explained

Component	Initial Eigenvalues Total	% of Variance	Cumulative %	Extraction Sums of Squared Loadings Total	% of Variance	Cumulative %
1	2.384	26.486	26.486	2.384	26.486	26.486
2	2.015	22.393	48.879	2.015	22.393	48.879
3	1.341	14.898	63.777	1.341	14.898	63.777
4	1.127	12.526	76.303			
5	.732	8.136	84.439			
6	.512	5.685	90.125			
7	.401	4.459	94.584			
8	.320	3.551	98.135			
9	.168	1.865	100.000			

Extraction Method: Principal Component Analysis.



F) Loadings

Let's assume that three principal components were selected for analysis based on theory or an inflection point. The next analysis shows how to calculate factor loadings. Among the principal components (PCs), only the first three were selected:

```
1. eVec_corr3 = Vt[:, :3]
2. eVal_corr3 = eVal_corr[:3]
3. loadings3 = eVec_corr3 * np.sqrt(eVal_corr3)
4. print(loadings3.round(3))
```

Those three were transposed and titled with the following code:

```
1. loadings3T = loadings3.round(3).T
2. loadingsDF = pd.DataFrame(loadings3T, index = ["PC1", "PC2
   ", "PC3"],
3.     columns = ["v1", "v2", "v3", "v4", "v5", "v6", "v7", "
   v8", "v9" ])
4. print(loadingsDF)
```

The loadings matrix in output shows the relationship between old variables with new principal components by calculating the coordinate of the old variables along the PC (principal component) axes:

```
1.          v1     v2     v3     v4     v5     v6     v7     v8     v9
2.  PC1 -0.481 -0.501  0.388  0.416 -0.525  0.691  0.711 -0.565 -0.027
3.  PC2  0.607  0.392  0.667  0.725 -0.600 -0.107 -0.182  0.320  0.128
4.  PC3  0.228  0.542 -0.265  0.171  0.289  0.591  0.386  0.234 -0.509
```

The loadings can be interpreted as the Pearson correlation between a principal component and a variable. One can see v6 and v7 are high with PC1, meaning that these two variables contribute heavily to PC1. On the other hand, v9 is low, little contributing to the formation of PC1.

The loadings also can be interpreted as regression coefficients, as shown below (Field, 2005, p. 622).

For example, the three principal components are composed of the nine variables:

PC1 = -0.481 * v1 -0.501 * v2 + . . . -0.027 * v9,
PC2 = 0.607 * v1 +0.392 * v2 + . . . 0.128 * v9,
PC3 = 0.228 * v1 + 0.542 * v2 + . . . -0.509 * v9 Equations 6

The nine loading numbers are used as regression coefficients for each of the three equations above.

SPSS created Component Matrix, the same as the loadings matrix above.

Component Matrix[a]

	Component		
	1	2	3
v1	-.481	.607	.228
v2	-.501	.392	.542
v3	.388	.667	-.265
v4	.416	.725	.171
v5	-.525	-.600	.289
v6	.691	-.107	.591
v7	.711	-.182	.386
v8	-.565	.320	.234
v9	-.027	.128	-.509

Extraction Method: Principal Component Analysis.

a. 3 components extracted.

G) PC score coefficients and PC scores

Principal component scores for items can be calculated using Equations 6, which is known as a weighted average (Field, 2005, p. 625). However, more sophisticated ways were invented for calculating principal component scores. Among the most popular is the regression method. In this method, PC loadings are premultiplied by the inverse of the original matrix to calculate PC score coefficients (refer to Equation 7), which is to "take account of the initial correlations between variables...in doing so, differences in units of measurement and variable variances are stabilized." (Field, 2005, p. 626)

PC score coefficients = the inverse of the original matrix * PC loadings

Equation 7

```
1. import numpy as np
2. pcScoreCoef = np.linalg.inv(numCorr).dot(loadings3)
3. print( pcScoreCoef.round(3))
```

The following are PC score coefficients:

```
1. [[-0.202  0.301  0.17 ]
2.  [-0.21   0.194  0.404]
3.  [ 0.163  0.331 -0.198]
4.  [ 0.175  0.36   0.127]
5.  [-0.22  -0.298  0.216]
6.  [ 0.29  -0.053  0.441]
7.  [ 0.298 -0.09   0.288]
8.  [-0.237  0.159  0.174]
9.  [-0.011  0.063 -0.379]]
```

These are exactly the same as what SPSS produced as Component Score Coefficient Matrix.

Component Score Coefficient Matrix

	Component 1	2	3
v1	-.202	.301	.170
v2	-.210	.194	.404
v3	.163	.331	-.198
v4	.175	.360	.127
v5	-.220	-.298	.216
v6	.290	-.053	.441
v7	.298	-.090	.288
v8	-.237	.159	.174
v9	-.011	.063	-.379

Extraction Method: Principal Component Analysis.
Component Scores.

PC scores were calculated by multiplying the data matrix by the PC score coefficients, as shown below:

```
zScore = (numCols - numCols.mean() ) / numCols.std()
pcScore = zScore.dot(pcScoreCoef)
pcScore.round(5)
```

The Python code produced the result (below left), and SPSS produced the same (below right):

	0	1	2
0	1.55125	0.92806	-0.85854
1	-0.57445	0.23950	-0.46792
2	-0.66767	-1.45335	-0.29508
3	1.42814	0.00750	-0.73250
4	0.57579	-1.00634	1.29582

	FAC1_1	FAC2_1	FAC3_1
8	1.55125	.92806	-.85854
9	-.57445	.23950	-.46792
0	-.66767	-1.45335	-.29508
9	1.42814	.00750	-.73250
7	.57579	-1.00634	1.29582

V. Varimax Rotation

A) The purpose of varimax rotation and algorithm

PCA often needs rotation for easier interpretation. The current author used the most popular method, called Varimax rotation. Varimax orthogonal rotation tries to maximize variance of the squared loadings in each factor so that each factor has only a few variables with large loadings and many other variables with low loadings. This process helps to obtain Thurstone's simple structure (ttnphns, 2015). Kaiser suggested normalization before rotating factors (Kaiser, 1958) and offered suggestions for computer programming (Kaiser, 1959).

Varimax can proceed by rotating two principal components successively or all principal components simultaneously (Horst, 1965). Multiple versions of Python code for varimax rotation can be found on the Internet (Biggs, 2017; "Factor_rotation," n.d.). Biggs's varimax function used Kaiser's normalization and a simultaneous rotation. Another method of successive rotation for comparing each pair of principal components is also available online (Zaiontz, n.d.).

For this article, the varimax function in Biggs' factor analysis class was saved as a separate function, varimaxFunction, and imported for data analysis, as shown in the code below. Since the function requires a Pandas' DataFrame input, line 1 below is used to convert the data format from a numpy array to a Pandas DataFrame.

```
1. loadings3 = pd.DataFrame(loadings3)
2. from varimaxFunction import varimax
3. rotatedLoading, rotationMtx = varimax(loadings3)
4. print(rotationMtx.round(3))
5. print(rotatedLoading.round(3))
```

The function created a varimax transformation matrix (lines 1-3 below) and a rotated loading matrix (lines 4-13).

```
1. [[ 0.637  0.529  0.56 ]
2.  [-0.557  0.818 -0.14 ]
3.  [-0.533 -0.223  0.816]]
4.        0      1      2
5. 0 -0.766  0.191 -0.168
6. 1 -0.826 -0.066  0.107
7. 2  0.016  0.811 -0.093
8. 3 -0.230  0.776  0.271
9. 4 -0.154 -0.833  0.026
10. 5  0.184  0.146  0.885
11. 6  0.349  0.141  0.739
12. 7 -0.663 -0.089 -0.171
13. 8  0.183  0.204 -0.448
```

If the original loadings matrix of loadings3 is multiplied by the varimax transformation matrix, rotationMtx, as shown below, it will produce the same rotated loading matrix shown above.

```
1. check = loadings3.dot(rotationMtx)
2. print(check.round(3))
```

SPSS created the following two, which are the same as what the Python code calculated except for some structural differences explained below.

Component Transformation Matrix

Component	1	2	3
1	.530	-.637	.560
2	.818	.557	-.140
3	-.223	.533	.816

Extraction Method: Principal Component Analysis.
Rotation Method: Varimax with Kaiser Normalization.

Rotated Component Matrix[a]

	Component 1	2	3
v1	.191	.766	-.168
v2	-.066	.826	.107
v3	.811	-.016	-.093
v4	.776	.230	.271
v5	-.833	.154	.026
v6	.146	-.184	.885
v7	.141	-.349	.739
v8	-.089	.663	-.171
v9	.204	-.183	-.448

Extraction Method: Principal Component Analysis.
Rotation Method: Varimax with Kaiser Normalization.

a. Rotation converged in 5 iterations.

B) A comparison of the SPSS and Python results

SPSS and Python produced the results, which showed differences in the signs and column order. The Python calculations need to be adjusted in two ways. First, the number sign was flipped with the earlier flip_vector_sign function, as shown below. The varimax loadings matrix can be directly modified without first modifying its corresponding transformation matrix before recalculating the varimax loadings matrix.

```
# flip_vector_sign requires a numpy matrix
rotLoadNPMatrix = rotatedLoading.as_matrix()
rotatedLoadingFlip = flip_vector_sign(rotLoadNPMatrix)
print(rotatedLoadingFlip.round(3))
```

Second, the principal component columns should be rearranged based on the amount of variance that the principal components account for. After varimax rotation, it is possible that the original 1st principal component may not explain the largest amount of variance. This is what happened with the data above. Therefore, the term factors is preferred over principal components after a varimax rotation is performed on principal component axes.

To check the amount of variance each principal component accounts for, the following matrixDescReorder function summed loadings for each column and rearranged all columns in descending order.

```
def matrixDescReorder(matrix):
### squared each number -- the portion of explanation
    rotatedPctExplained = matrix**2
### sum columnwise
    rotatedPctExplainedSum = np.sum(rotatedPctExplained,
axis = 0)

### sort them in descending order
```

```
### final loading matrix is rotatedLoadingFlipOrdered
    indexN = rotatedPctExplainedSum.argsort()[::-1]
    rotatedReordered = matrix[:, indexN]
    return rotatedReordered
```

The flipped rotated loading matrix was used as an argument for the matrixDescReorder function:

```
rotatedLoadingFlipReordered = matrixDescReorder(rotatedLoadingFlip)
print(rotatedLoadingFlipOrdered.round(3))
```

The result below produced the same as what SPSS did earlier. Columns were reordered and number signs were adjusted.

```
[[ 0.191  0.766 -0.168]
 [-0.066  0.826  0.107]
 [ 0.811 -0.016 -0.093]
 [ 0.776  0.23   0.271]
 [-0.833  0.154  0.026]
 [ 0.146 -0.184  0.885]
 [ 0.141 -0.349  0.739]
 [-0.089  0.663 -0.171]
 [ 0.204 -0.183 -0.448]]
```

C) Factor score coefficients

Factor score coefficient matrix can be calculated as the PC score coefficient matrix was calculated in Equation 7.

```
cscmRotated = np.linalg.inv(zScore.corr()).dot(rotatedLoadingFlipReordered)
print(cscmRotated.round(3))
```

The result (on the left below) is the same as that produced by SPSS (on the right).

```
[[ 0.102  0.387 -0.016]
 [-0.042  0.457  0.185]
 [ 0.401 -0.024 -0.117]
 [ 0.358  0.157  0.151]
 [-0.408  0.089  0.094]
 [ 0.012  0.021  0.53 ]
 [ 0.02  -0.087  0.415]
 [-0.034  0.332 -0.013]
 [ 0.131 -0.16  -0.325]]
```

Component Score Coefficient Matrix

	Component 1	2	3
v1	.102	.387	-.016
v2	-.042	.457	.185
v3	.401	-.024	-.117
v4	.358	.157	.151
v5	-.408	.089	.094
v6	.012	.021	.530
v7	.020	-.087	.415
v8	-.034	.332	-.013
v9	.131	-.160	-.325

Extraction Method: Principal Component Analysis.
Rotation Method: Varimax with Kaiser Normalization.
Component Scores.

Factor scores were calculated with the following code:

```
pcScoreRotated = zScore.dot(cscmRotated)
print(pcScoreRotated.round(5))
```

The Python code produced a little different result (on the left below) from what SPSS did (on the right). But the difference, which might occur due to the computer's rounding errors, was negligible. Only the first five rows are shown instead of the entire 33 rows to save space.

```
          0        1        2
0   1.77244 -0.92830  0.03845
1  -0.00379  0.24997 -0.73743
2  -1.47725 -0.54182 -0.41141
3   0.92569 -1.29567  0.20132
4  -0.80778 -0.23698  1.52144
```

FAC1_2	FAC2_2	FAC3_2
1.77245	-.92827	.03843
-.00380	.24998	-.73743
-1.47725	-.54184	-.41140
.92571	-1.29566	.20130
-.80776	-.23700	1.52144

VI. Theoretical or Exploratory Rotation

In Q analysis, factor axes are often rotated for theoretical reasons (Brown, 1980, pp. 224-239). Factor axes may be rotated toward variables with same characteristics (in the case of Q analysis, people are treated as variables). Of course, factor axes also can be rotated for serendipity that may lead to unexpected insights. This type of rotations is unique to Q analysis that Q community has adopted.

In the case of R community, it just reverses observations and variables of any dataset to obtain a Q matrix, but does not rotate factor axes after varimax rotation. Thus, SPSS and other statistical packages, which are not focused on the Q community, have not incorporated the feature for theoretical or exploratory rotations.

For rotation of axes, a user needs to see an arrangement of variables with factor axes. Either two or three axes can be displayed.

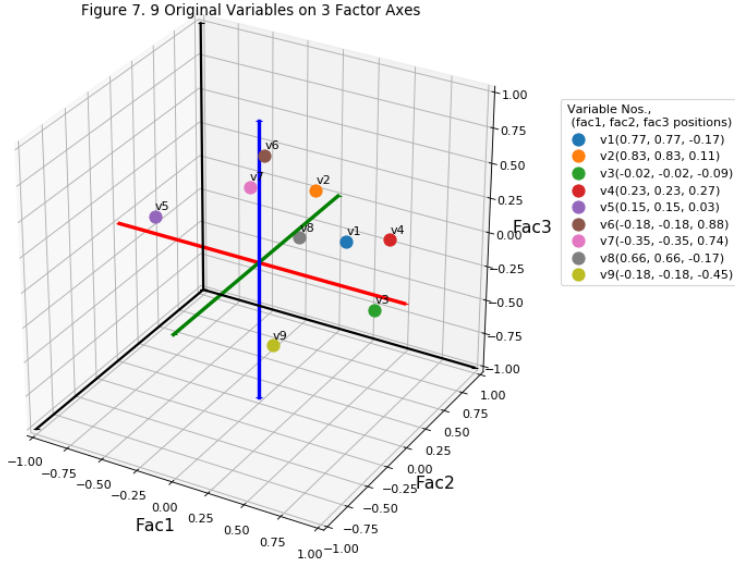
A) 3-dimension graph

A 3-dimensional graph can be drawn using facsGraph3 in Appendix 1. This function is applied to an argument of the varimax factor loadings matrix, loadingsRot, which is the Pandas' DataFrame version of rotatedLoadingFlipReordered.

```
1. # pcsGraph3 requires DataFrame
2. loadingRot = pd.DataFrame(rotatedLoadingFlipReordered.copy())
3. facsGraph3("Figure 7", loadingRot)
```

The following graph displays the coordinate of variables along the three factor axes, and the legend shows the specific coordinates of each variable (refer to Figure 7).

Figure 7. 9 Original Variables on 3 Factor Axes



B) Rotate two axes at a time while viewing a 3-dimensional graph

Although three axes can be changed theoretically, it will be daunting to human brains. Thus, the author wrote code, transMatrix3 in Appendix II, after incorporating the following three formulae ("Rotation Matrix," n.d.). Rotation theta degrees were multiplied by -1 since the computer code actually rotates variable dots on the graph, while people tend to think about the rotation of the axes. For example, rotating data dots by 5 degrees counterclockwise means rotating the axes by -5 degrees counterclockwise.

$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$	$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$	$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

The function above takes the number of degrees for a clockwise rotation and two axes as its arguments. One may consider rotating the x- and z-axis along the y-axis by some degrees, like -8 degrees, counterclockwise after viewing Figure 7. Then the following code will do the job:

```
import numpy as np
rotMatrix1 = transMatrix3(-8, 1, 3)
rotMatrix1.round(3)
```

In linear algebra, a rotation by some angles can be translated into a rotating matrix like the following:

```
array([[ 0.99 ,  0.   ,  0.139],
       [ 0.   ,  1.   ,  0.   ],
       [-0.139,  0.   ,  0.99 ]])
```

The rotMatrix1 above is used as a factor transformation matrix in the following code to rotate the varimax factor loadings matrix:

```
loadingRot1 = loadingRot.dot(rotMatrix1)
print(loadingRot1.round(3))
```

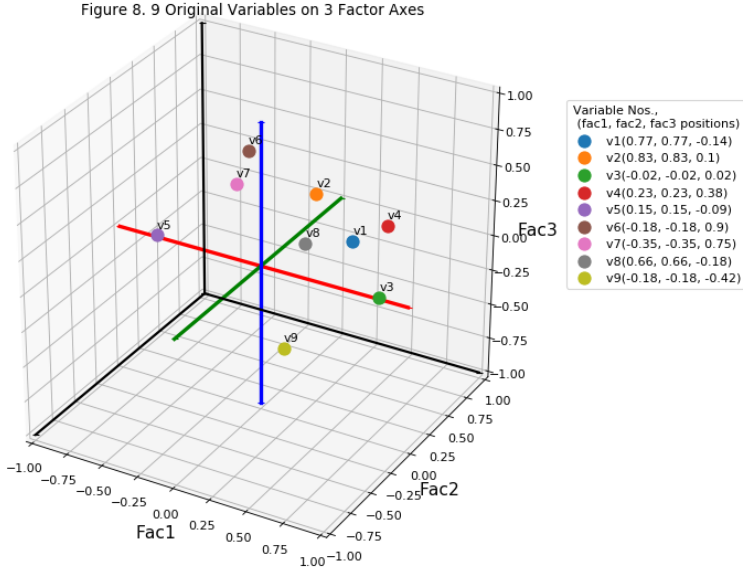
C) A new factor loadings matrix after a rotation

The newly emerged factor loadings changed the first and third columns while keeping constant the second column that represents the y-axis.

```
         0      1      2
0  0.213  0.766 -0.140
1 -0.080  0.826  0.097
2  0.816 -0.016  0.021
3  0.731  0.230  0.376
4 -0.829  0.154 -0.090
5  0.021 -0.184  0.897
```

```
8.   6   0.037 -0.349   0.751
9.   7  -0.064  0.663  -0.182
10.  8   0.264 -0.183  -0.415
```

When the new factor loadings matrix was plugged into the facsGraph3 function, a new 3-dimensional graph was created (refer to Figure 8). Notice the rotation of only the x- and z-axis in Figure 8 below.



D) Calculate a factor score coefficient matrix and factor scores

To create a factor score coefficient matrix, a loadings matrix is premultiplied by the inverse of the original correlation matrix as in Equation 7.

```
cscmRot1 = np.linalg.inv(zScore.corr()).dot(loadingRot1)
print(cscmRot1.round(3))
```

The following is the factor score coefficient matrix:

```
[[ 0.103  0.387 -0.002]
 [-0.069  0.457  0.177]
 [ 0.414 -0.024 -0.06 ]
 [ 0.335  0.156  0.2  ]
 [-0.416  0.089  0.037]
 [-0.062  0.023  0.526]
 [-0.039 -0.088  0.413]
 [-0.032  0.333 -0.018]
 [ 0.175 -0.159 -0.304]]
```

Then, factor scores can be calculated by multiplying the data matrix by this factor score coefficient matrix. The first five rows of the result are shown below:

```
         0      1      2
0    1.749 -0.929  0.284
1    0.099  0.248 -0.732
2   -1.405 -0.541 -0.613
3    0.889 -1.293  0.329
4   -1.014 -0.238  1.396
```

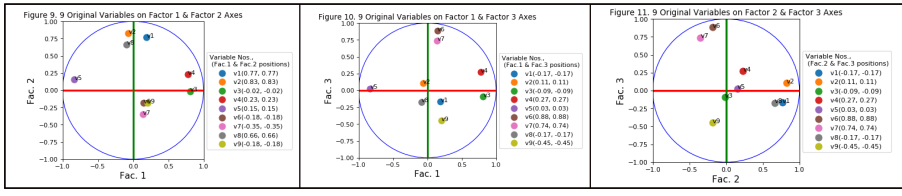
E) 2-dimensional graph and rotation

One may prefer to see loadings in 2-dimensional graphs instead to decide the first two axes to rotate. If one has three axes in total, like factor 1, factor 2 and factor 3, three graphs are possible: one with factors 1 and 2, and the second one with factors 1 and 3, and the last with factors 2 and 3. The four factors will lead to six 2-dimensional graphs.

The function facfsGraph2 in Appendix III was written to draw a 2-dimensional graph with function arguments of a graph title, a loading matrix, the number representing two axes (1 for factor 1, 2 for factor 2, etc.), and the width and height of a graph in centimeters. The three lines of code created

three figures 9, 10 and 11.

```
1. facsGraph2("Figure 9", loadingRot, 1, 2, 4, 4)
2. facsGraph2("Figure 10", loadingRot, 1, 3, 4, 4)
3. facsGraph2("Figure 11", loadingRot, 2, 3, 4, 4)
```



F) Rotate axes in a 2-dimensional graph

First, a rotation of axes by theta degrees can be converted into a rotation matrix with the following transMatrix2 function:

```
1. from math import sin, cos, radians
2. def trans2DMatrix(theta):
3.     theta = -theta
4.     a1 = cos(radians(theta))
5.     a2 = -sin(radians(theta))
6.     a3 = sin(radians(theta))
7.     a4 = cos(radians(theta))
8.     transMat = np.array([ [a1, a2], [a3, a4 ] ])
9.     return transMat
```

The two axes in Figure 11 may be rotated by 15 degrees counterclockwise. The trans2DMatrix is applied to the 15 degrees for a rotation transformation matrix:

```
1. trans2DMat1 = trans2DMatrix(15)
2. print(trans2DMat1.round(3))
```

The function call created the following matrix:

```
[[ 0.966 -0.259]
 [ 0.259  0.966]]
```

This 2D transformation matrix needs to be premultiplied by the varimax rotation matrix, specifically only two columns of the matrix that are related to the factors to be rotated. The following rotate2Factors function needs four arguments--the current loading matrix, the 2D transformation matrix, and the factor numbers representing the two factors.

```
def rotate2Factors(matrix, trans2DMat, fNum1, fNum2):
    #select 2 columns
    twoFacs = matrix.iloc[:, [fNum1 - 1, fNum2 - 1] ]
    twoFacRot = twoFacs.dot(trans2DMat)

    matrix.iloc[:, fNum1 - 1]= twoFacRot.iloc[:, 0]
    matrix.iloc[:, fNum2-1]= twoFacRot.iloc[:, 1]
    ## return the whole matrix
    return matrix
```

To prevent the same varimax rotation matrix from being used in mixed way in both 3d and 2d rotations, one needs to create a new copy from the varimax loadings matrix.

```
# newly create a copy of the loading matrix after varimax
loadingRot2D = pd.DataFrame(rotatedLoadingFlipReordered.copy())
# the earlier loading matrix, loading transmission matrix,
two columns to be rotated
loadingRot2D1 = rotate2Factors(loadingRot2D, trans2DMat1,
2, 3)
print(loadingRot2D1.round(3))
```

A new loading matrix was calculated:

```
        0      1      2
0  0.191  0.696 -0.361
1 -0.066  0.826 -0.110
```

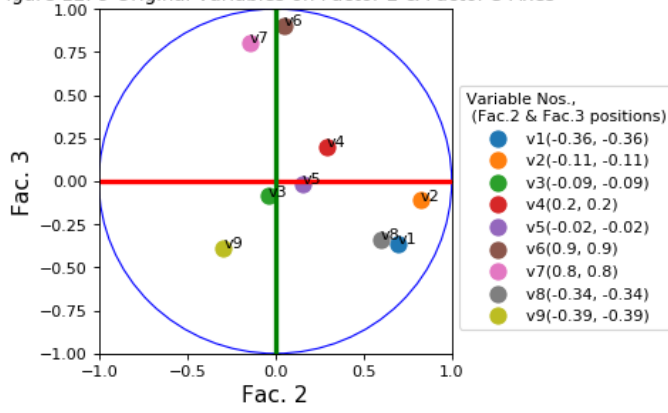
```
4.  2   0.811 -0.040 -0.086
5.  3   0.776  0.292  0.202
6.  4  -0.833  0.155 -0.015
7.  5   0.146  0.051  0.902
8.  6   0.141 -0.146  0.804
9.  7  -0.089  0.596 -0.337
10.  8   0.204 -0.293 -0.385
```

G) Check the graph for rotation

The new loadings matrix, loadingRot2D1, can be plugged into facsGraph2 for a new graph.

```
1. pcsGraph2("Figure 12", loadingRot, 1, 2, 4, 4)
```

Figure 12. 9 Original Variables on Factor 2 & Factor 3 Axes



One can notice the rotation based on the changes to the position of data dots. Of course, more rotations can be done as needed. Since rotation may change the number sign and the amount of variance each factor accounts for, one may need to flip the number sign and reorder the data columns if needed.

VII. A Summary of All Procedures

PCA goes through many different steps to produce the results a researcher wants. As the principal component analysis (PCA) process chart in Appendix IV shows, eigenvalues and eigenvectors have to be calculated for eigendecomposition in the beginning. This can be done by creating a correlation or covariance matrix from the data (step 1 in Appendix IV), which is further processed through either eigendecomposition (edc) or singular value decomposition (svd) (step 2). They also can be derived directly from the original data matrix through svd (step 3).

PC loadings, which is called Component Matrix in SPSS, can be calculated with the formula of eigenvectors $*\sqrt{eigenvalues}$ (step 4).

If rotation is not needed, the principal component (PC) score coefficient matrix can be calculated using one of many methods, like regression method, with the formula, the inner product of the inverse of the original correlation or covariance matrix and the loadings matrix (step 5). This is called Component Score Coefficient Matrix in SPSS. If the matrix of the original standardized or centered data is multiplied by PC score coefficient matrix, it will produce PC scores (step 6). In SPSS, a user has to select "Analyze" from the top menu, "Dimension Reduction" from the pulldown menu, "Scores" in the Factor Analysis dialog box, "Regression" in the Factor Analysis Factor Scores dialog box, and "Save as variables." Then principal factor scores will be appended to the end of the original dataset.

Varimax rotation will rearrange the amount of variance among principal components, so principal components lose their characteristics of extracting the maximum variance possible each time when an additional principal component is added. Thus, the term factors is preferred over principal components after varimax rotation. Rotation of principal components can be

performed on two principal components at a time or all principal components simultaneously. All the rotation angles are translated into the factor loading transformation matrix (step 7). This transformation matrix is called Component Transformation Matrix in SPSS. If the original PC loadings is multiplied by the transformation matrix, a new varimax factor loadings matrix is produced (step 8). SPSS calls it Rotated Component Matrix. One can calculate a new factor score coefficient matrix (step 9) and factors scores (step 10) using the same formula as in steps 5 and 6. SPSS appends new factor scores to the end of the dataset.

If one wants to continue to rotate factor axes after varimax rotation for further insights, rotation of factors axes needs to be expressed as a transformation matrix (step 11). This matrix can be used to produce a new factor loadings matrix (step 12), the same as in step 8. This process can be done until one is satisfied with the loadings of variables with factors. When one comes to a satisfactory loadings matrix, one can calculate its corresponding factor score coefficient matrix (step 13) and factor scores (step 14).

Q scholars who are interested in theoretical rotations may consult theory (step 15) and rotate factor axes as the theory suggests (step 16). Then, the factor score coefficient matrix and factor scores can be calculated in the same way as in step 13 and step 14 earlier.

VIII. Conclusion

Cranking out results using PCA analysis on one of the commercial statistical packages is easy, just importing data and clicking a few buttons. For a deep understanding of PCA, however, one needs to go inside the program and see

what happens inside the black box by running code. Many scholars may have a basic knowledge of a popular programming language like Python. But PCA code in Python is not compiled in one place. A user has to gather code scattered around the Internet, tweak it for compatibility and fill any gaps left.

Also, commercial statistical packages have been geared toward R analysis so they cannot be used for theoretical or exploratory analysis beyond varimax rotation, a unique Q approach. Q analysis emphasizes the importance of factor scores. R analysis, on the other hand, often ignores them.

This tutorial shows comparisons of Python and SPSS. Also, this tutorial shows all the steps needed for PCA along with Python code beyond varimax rotation. Therefore, this would help anyone who wants to run PCA at a deeper level and R scholars who want to learn Q analysis.

References

Biggs, J. 2017. *Factor_analyzer* [Web page]. Retrieved from github.com: https://github.com/EducationalTestingService/factor_analyzer/blob/master/factor_analyzer/factor_analyzer.py

Brown, S. R. 1980. *Political subjectivity: Applications of Q methodology in political science*. New Haven, CT: Yale University Press.

Campbell, N. A., and Atchley, W. R. 1981. The geometry of canonical variate analysis. *Systematic Zoology* 30(3): 268-280. Retrieved from http://www.jstor.org/stable/2413249

Cureton, E. E., and D'Agostino, R. B. 1983. *Factor analysis: An applied approach*. Hiilsdale, New Jersey: Lawrence Erlbaum Associates.

Markaki, E. N., Chadjipandelis, T., and Tomaras, P. 2014. *Procedia - Social and Behavioral Sciences* 147: 554-560.

Factor_rotation. n.d. [Web page]. Retrieved from github.com: https://github.com/mvds314/factor_rotation/blob/master/README.md

Field, A. 2005. *Discovering statistics using SPSS* : (And sex, drugs and rock 'n' roll) (2nd ed.). London: Sage.

Gorsuch, R. L. 1983. *Factor analysis*. Hillsdale, N.J: L. Erlbaum Associates.

Horst, P. 1965. *Factor analysis of data matrices*. New York: Holt, Rinehart and Winston Inc.

Kabacoff, R. A. 2015. *R in action : Data analysis and graphics with R*. Shelter Island, N.Y.: Manning Publications. Retrieved from Amazon.

Kaiser, H. 1959. Computer program for varimax rotation in factor analysis. *Education and Psychological Measurement* 19(3): 413-420.

Kaiser, H. F. 1958. The varimax criterion for analytic rotation in factor analysis. *Psychometrika* 23(3): 187-200.

Lay, D. C. 2005. *Linear algebra and its applications* (3rd ed.). Boston: Addison-Wesley.

Rotation Matrix. n.d. [Web page]. Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Rotation_matrix

Stephenson, W. 1953. *The study of behavior: Q-technique and its methodology*. Chicago, IL: University of Chicago Press.

Strang, G. 2016. *Introduction to linear algebra*. Wellesley, Massachusetts: Wellesley-Cambridge Press.

ttnphns. 2015. *Factor rotation methods (varimax, oblimin, etc.) - what do the names mean and what do the methods do?* [Web page]. Retrieved from stats.stackexchange.com: https://stats.stackexchange.com/questions/185216/factor-rotation-methods-varimax-oblimin-etc-what-do-the-names-mean-and-wh

Watts, S., and Stenner, P. 2012. *Doing Q methodological research: Theory, method and interpretation*. Los Angeles: Sage.

Zaiontz, C. n.d. Rotation | *Real Statistics Using Excel*. Retrieved from www.real-statistics.com: http://www.real-statistics.com/multivariate-statistics/factor-analysis/rotation/

Appendix 1: facsGraph3

```
def facsGraph3(figNum, loadingMat):
  # import modules
  from mpl_toolkits.mplot3d import Axes3D
  #from matplotlib.collections import LineCollection
  import matplotlib.pyplot as plt
  import numpy as np

  #basic frame
  fig=plt.figure(figsize=(9, 9), dpi= 80, facecolor='w', edgecolor='k')

   # start to ceate a 3d graph
   #111 means 1 row, 1 column, and the first slot
   ax = fig.add_subplot(111, projection='3d')

   #displaying dots
   for i in range(9):
       fac1 = loadingMat.iloc[i:i+1, 0 ]
       fac2 = loadingMat.iloc[i:i+1, 1 ]
       fac3 = loadingMat.iloc[i:i+1, 2 ]
       #ax.scatter(fac1, fac2, fac3,   c = "r", marker = "o")
       scatter1 = ax.scatter(fac1, fac2, fac3, s = 100, cmap="Diverging")

   arrX = loadingMat.iloc[:, 0  ]
   arrY = loadingMat.iloc[:, 1 ]
   arrZ = loadingMat.iloc[:, 2 ]
   # create labels starting with v1
   labels = []
   for i in range(loadingMat.shape[0]):
       labels.append("v" + str(i + 1) )
   labels = np.array(labels)
   #give the labels to each point
   for x, y, z, label in zip(arrX, arrY, arrZ, labels):
       ax.text(x, y, z+ 0.05, label )

   # 3 principal component axes
   ax.plot([-1, 1], [0, 0], [0, 0], color='r',  lw=3, marker="_", label='_nolegend_')
   ax.plot( [0, 0],[-1, 1], [0, 0], color='g',  lw=3,   marker="_", label='_nolegend_')
   ax.plot( [0, 0], [0, 0], [-1, 1], color='b',  lw=3,  marker="_", label='_nolegend_')
```

```
39.
40.     # 3 axes on the border
41.     ax.plot([-1, 1], [1, 1], [-1, -1],  color='k',  lw=2, marker="_", label
    ='_nolegend_')
42.     ax.plot( [-1, -1], [-1, 1],[-1, -1],  color='k',  lw=2, marker="_", lab
    el='_nolegend_')
43.     ax.plot([-1, -1], [1, 1], [-1, 1],  color='k',  lw=2, marker="_", label
    ='_nolegend_')
44.
45.     arrX = arrY.round(2)
46.     arrY = arrY.round(2)
47.     arrZ = arrZ.round(2)
48.
49.     lgd = ax.legend(  [str(label)+ "(" + str(x) + ", " +  str(y) +  ", " +
     str(z) + ")" for x, y, z, label in zip(arrX, arrY, arrZ, labels) ], loc='u
    pper left', bbox_to_anchor=(1, 0.8))
50.     lgd.set_title("Variable Nos.,\n (fac1, fac2, fac3 positions)")
51.
52.     ax.set_xlabel('Fac1', fontsize = 14)
53.     ax.set_ylabel('Fac2', fontsize = 14)
54.     ax.set_zlabel('Fac3', fontsize = 14)
55.     ax.set_title(figNum + ". " + str(loadingMat.shape[0]) + " Original Vari
    ables on " +  str(loadingMat.shape[1]) +  " Factor Axes")
56.
57.     ax.set_xlim3d(-1, 1)
58.     ax.set_ylim3d(-1, 1)
59.     ax.set_zlim3d(-1, 1)
60.     plt.show()
61.     return None
```

Appendix II. transMatrix3

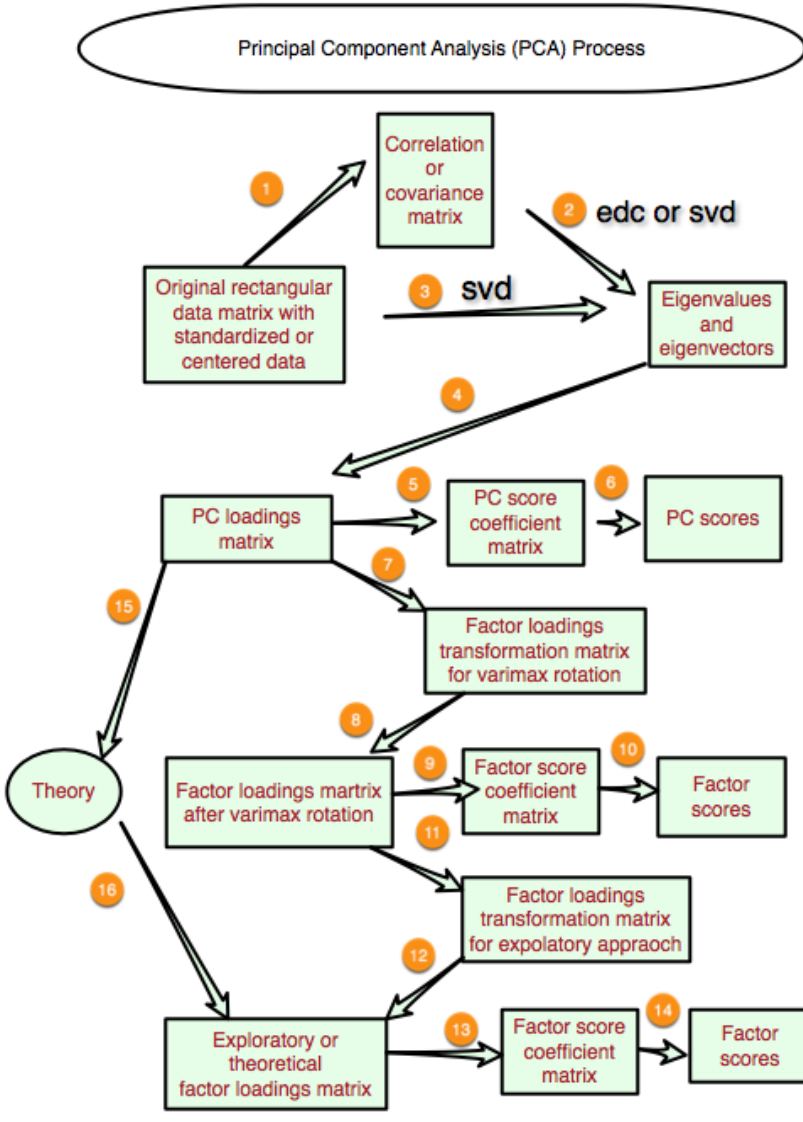
```
def transMatrix3(theta, axis1, axis2):
    import math
    theta = -theta
    if(axis1 == 2 and axis2 == 3):
        ##transMatrixX(theta):
        transMat  = np.array([ [1, 0, 0 ],
                        [0, cos(math.radians(theta)), -sin(math.radians(theta)) ],
                       [ 0, sin(math.radians(theta)),  cos(math.radians(theta))]
                      ])
    elif(axis1 == 1 and axis2 == 3):

         #transMatrixY(theta):
         transMat  = np.array([ [cos(math.radians(theta)), 0,  sin(math.radians(theta)) ],
                        [0, 1, 0 ],
                        [-sin(math.radians(theta)), 0, cos(math.radians(theta))]
                       ])


    elif(axis1 == 1 and axis2 == 2):
            #transMatrixZ(theta):
        transMat  = np.array([ [cos(math.radians(theta)),  -sin(math.radians(theta)), 0 ],
                        [sin(math.radians(theta)),  cos(math.radians(theta)), 0],
                        [0, 0, 1 ],
                       ])
    else:
        pass;

    return transMat
```

Appendix III. facsGraph2

```
import matplotlib.pyplot as plt
import numpy as np

def facsGraph2(figNum, loadingMat, f_input1 = 1, f_input2 = 2, figW =8, figH = 8):
    # import modules
    f1 = f_input1 -1
    f2 = f_input2 -1

    #basic frame
    fig=plt.figure(figsize=(figW, figH), dpi= 80, facecolor='w', edgecolor='k')

    # start to ceate a 3d graph
    #111 means 1 row, 1 column, and the first slot
    ax = fig.add_subplot(111 )

    #displaying dots
    for i in range(loadingMat.shape[0]):
        fac1 = loadingMat.iloc[i:i+1, f1 ]
        fac2 = loadingMat.iloc[i:i+1, f2 ]

        #ax.scatter(f1, f2, f3,   c = "r", marker = "o")
        scatter1 = ax.scatter(fac1, fac2, s = 100, cmap="Diverging")

    arrX = loadingMat.iloc[:, f1]
    arrY = loadingMat.iloc[:, f2]

    # create labels starting with v1
    labels = []
    for i in range(loadingMat.shape[0]):
        labels.append("v" + str(i + 1) )
    labels = np.array(labels)

    #give the labels to each point
    for x, y, label in zip(arrX, arrY, labels):
        ax.text(x, y, label)

    # 3 principal component axes
```

```
    ax.plot([-1, 1], [0, 0], color='r',  lw=3, marker="_", label='_nolegend_')
    ax.plot( [0, 0],[-1, 1], color='g',  lw=3, marker="_", label='_nolegend_')

    circle = plt.Circle((0, 0), 1, color='b', fill=False)
    ax.add_artist(circle)

    arrX = arrY.round(2)
    arrY = arrY.round(2)

    lgd = ax.legend(  [str(label)+ "(" + str(x) + ", " +  str(y) + ")" for x, y, label in zip(arrX, arrY,  labels) ], loc='upper left', bbox_to_anchor=(1, 0.8))
    lgd.set_title("Variable Nos.,\n (Fac." + str(f_input1) + " & Fac." + str(f_input2) + " positions)")

    # [ 'Lag ' + str(lag) for lag in arrX]
    #xyz labels
    ax.set_xlabel('Fac. ' + str(f_input1), fontsize = 14)
    ax.set_ylabel('Fac. ' + str(f_input2), fontsize = 14)
    ax.set_title(figNum + ". " + str(loadingMat.shape[0]) + " Original Variables on Factor " + str(f_input1) + " & Factor " + str(f_input2)  +  " Axes ")

    #xyz range
    ax.set_xlim(-1, 1)
    ax.set_ylim(-1, 1)

    #display

    plt.show()

    return None
```

Appendix IV. Principal Component Analysis(PCA) Process



Notes: edc in step 2 indicates eigendecomposition, and svd in step 3, singular value decomposition.

Varimax 회전 및 그 이후: R 및 Q 분석을 위한 선형 대수, 시각화 및 Python 프로그래밍을 사용한 PCA에 대한 사용지침서

이 병 수*

논문요약

주성분 분석(PCA)은 학문 및 실제 목적을 위해 많이 사용되었습니다. 이 사용지침서는 Q 분석에 관심있는 R 학자 뿐만 아니라 PCA를 더 잘 활용하고자 하는 개인에게 도움이 될 것입니다. 많은 상용 통계 패키지가 PCA를 처리할 수 있지만 PCA 코드를 직접 실행하면 PCA에 대한 더 깊게 이해할 수 있습니다. 상업용 통계 패키지에서 PCA 기능을 사용하여 데이터를 입력하고 몇 개의 버튼을 누르기만 하면 통계 결과가 나옵니다. 하지만 PCA에 대한 깊은 이해를 위해서는 개인이 코드를 실행할 때 어떤 일이 일어나는지 알아야합니다. 많은 학자들이 파이썬과 같은 인기있는 프로그래밍 언어에 대한 기본 지식을 보유하고 있을 수 있습니다. 그러나 파이썬 코드는 한 곳에서 깔끔하게 정리되어 모아져 있지 않습니다. 사용자는 인터넷에 흩어져 있는 파이썬 코드를 수집하고 호환성을 위해 조정한 다음 부족한 부분이 있다면 채워야 합니다. 또한 R 분석을 대상으로 한 상업 통계 패키지는 배리맥스 방법을 넘어선, 즉 Q 연구를 위한 이론적 또는 탐색적 분석을 위한 팩터 회전에 사용할 수 없습니다. 이 사용지침서에서는 PCA에 필요한 모든 Python 코드를 제공하면서 결과를 SPSS (Statistical Package for the Social Science)출력과 비교했습니다. 이 사용지침서는 Q 분석을 위한 필수 요소인 팩터 축의 이론적 또는 탐색적 팩터 회전도 다루었습니다.

주제어 : PCA, 주성분 분석, 선형 대수, 그래프, 파이썬 코드, varimax 회전, R 및 Q 분석

투고일: 2018.04.15. **심사일:** 2018.04.16. **게재확정일:** 2018.04.23.

* 일란대학교, 부교수

View publication stats