principal-components

September 30, 2014

1 Principal Components Analysis

1.0.1 The BioEnv data set

To demonstrate PCA we'll use a dataset called 'bioenv.txt' (see class wiki), obtained from a book called "Biplots in Practice" (M. Greenacre, 2010). Here is Greenacre's description of the dataset:

> The context is in marine biology and the data consist of two sets of variables observed at the same locations on the sea-bed: the first is a set of biological variables, the counts of five groups of species, and the second is a set of four environmental variables. The data set, called "bioenv", is shown in Exhibit 2.1. The species groups are abbreviated as "a" to "e". The environmental variables are "pollution", a composite index of pollution combining measurements of heavy metal concentrations and hydrocarbons; depth, the depth in metres of the sea-bed where the sample was taken; "temperature", the temperature of the water at the sampling point; and "sediment", a classification of the substrate of the sample into one of three sediment categories.

The first column of the data has no header, and corresponds to the site labels, so we'll make that the index for our Pandas `DataFrame`.

```
In [33]: import numpy as np
         import numpy.linalg as la
         import pandas as pd

         from matplotlib import pyplot as plt

         %matplotlib inline
```

```
In [9]: url = "https://raw.githubusercontent.com/pmagwene/Bio723/master/datasets/bioenv.txt"
        bioenv = pd.read_table(url, index_col=0)
```

```
In [10]: bioenv.columns
```

```
Out[10]: Index([u'a ', u'b ', u'c ', u'd', u'e ', u'Pollution', u'Depth', u'Temperature', u'Sediment'],
```

Notice that there are some extra trailing spaces on some of the columns labels. We can strip those spaces off as so:

```
In [29]: bioenv.columns = [name.strip() for name in bioenv.columns]
```

The line above uses a list comprehension to iterate through all the column names, and uses the string method `strip()` to strip off the extra spaces. See `string.strip` for more info on the strip function.

```
In [30]: bioenv.head()
```

```
Out[30]:      a   b   c   d  e  Pollution  Depth  Temperature Sediment
         s1   0   2   9  14  2        4.8     72          3.5        S
         s2  26   4  13  11  0        2.8     75          2.5        C
         s3   0  10   9   8  0        5.4     59          2.7        C
         s4   0   0  15   3  0        8.2     64          2.9        S
         s5  13   5   3  10  7        3.9     61          3.1        C
```

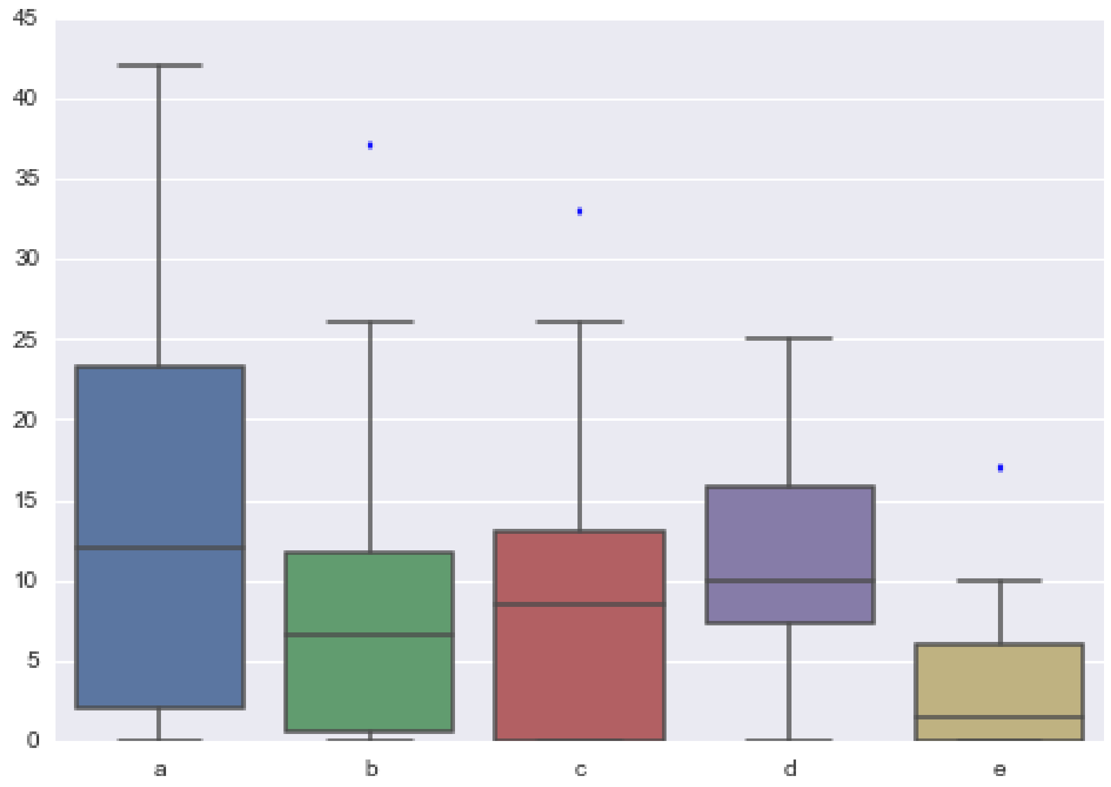
```
In [31]: bioenv.describe()
```

```
Out[31]:                a          b          c          d          e  Pollution  \
         count  30.000000  30.000000  30.000000  30.000000  30.000000  30.000000
         mean   13.466667   8.733333   8.400000  10.900000   2.966667   4.516667
         std    12.555349   9.134751   8.580652   6.666178   3.960872   2.141234
         min     0.000000   0.000000   0.000000   0.000000   0.000000   1.900000
         25%     2.000000   0.500000   0.000000   7.250000   0.000000   2.800000
         50%    12.000000   6.500000   8.500000  10.000000   1.500000   4.300000
         75%    23.250000  11.750000  13.000000  15.750000   6.000000   5.550000
         max    42.000000  37.000000  33.000000  25.000000  17.000000  10.000000

                     Depth  Temperature
         count   30.000000    30.000000
         mean    74.433333     3.056667
         std     15.615384     0.281233
         min     51.000000     2.500000
         25%     61.000000     2.900000
         50%     73.500000     3.000000
         75%     84.750000     3.300000
         max    100.000000     3.600000
```

The columns labeled “a” to “e” contain the counts of the five species at each site. We’ll work with this abundance data for now.

```
In [32]: abundance = bioenv[["a","b","c","d","e"]]
```

```
In [36]: # let's use the Seaborn library introduced last week.
         import seaborn as sns

         g = sns.boxplot(abundance)
```



From the boxplot it looks like the counts for species “e” are smaller on average, and less variable. The mean and variance functions confirm that.

```
In [56]: # here we use a "empty" groupby (all observations belong to the same group)
         # so we can use the convenient aggregate(agg) fxn to
         # produce a nice table

         # we also use the pandas set_option function to set the precision
         # with which our tables print in the IPython notebook
         pd.set_option('precision',3)

         abundance.groupby(lambda x: True).agg([np.mean, np.var])
```

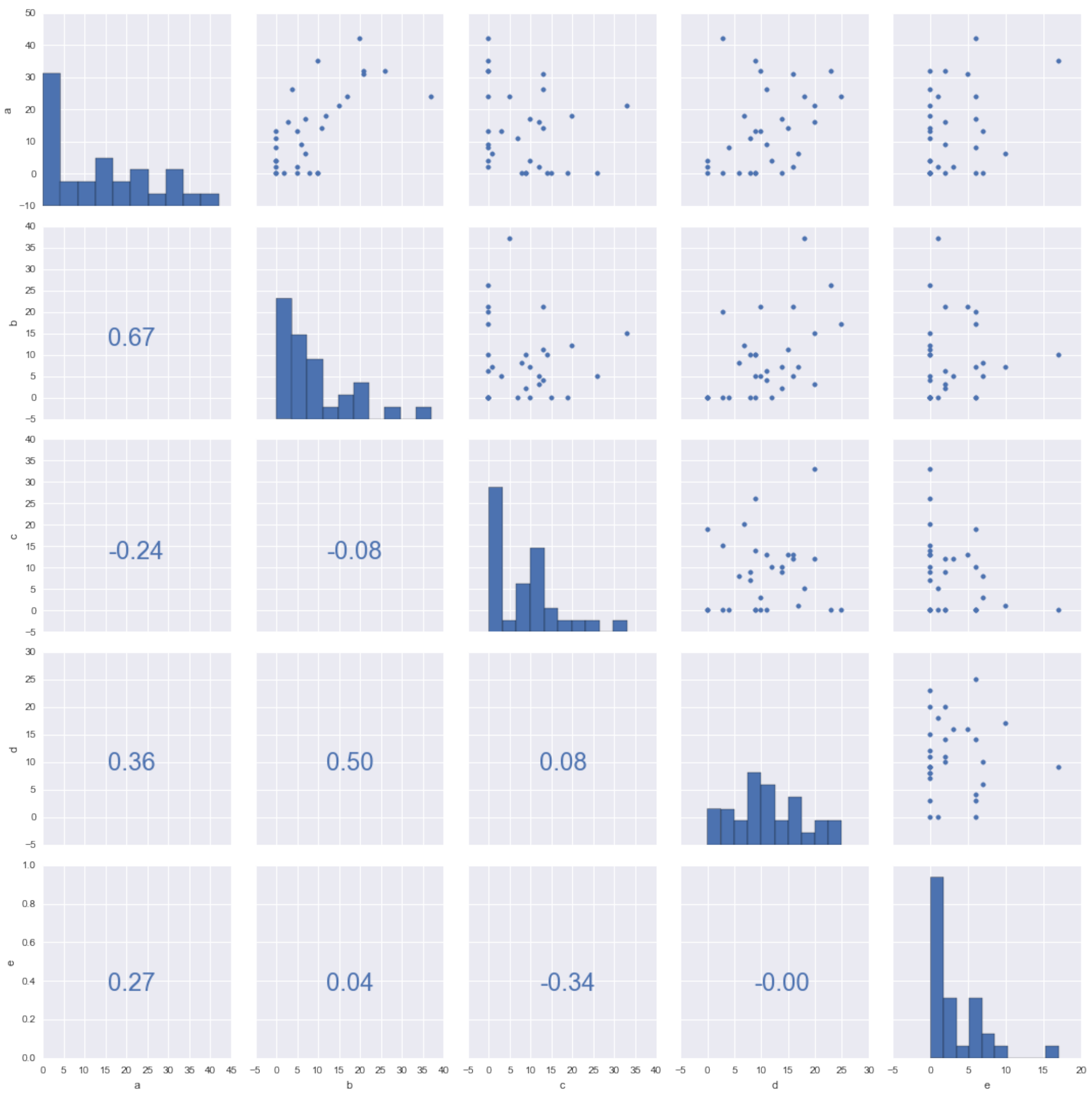
```
Out[56]:          a              b             c             d             e
              mean     var  mean    var  mean    var  mean    var  mean    var
         True 13.47 157.64  8.73  83.44   8.4  73.63  10.9  44.44  2.97  15.69
```

A correlation matrix suggests weak to moderate associations between the variables, but the scatterplot matrix suggests that many of the relationships have a strong non-linear element.

```
In [58]: abundance.corr()
```

```
Out[58]:      a     b     c     d     e
         a  1.00  0.67 -0.24  0.36  0.27
         b  0.67  1.00 -0.08  0.50  0.04
         c -0.24 -0.08  1.00  0.08 -0.34
         d  0.36  0.50  0.08  1.00 -0.00
         e  0.27  0.04 -0.34 -0.00  1.00
```

```
In [97]: # there's currently a bug in seaborn that causes
         # PairGrid to fail when the index is not a numeric sequence
         # the reset_index method allows us to work around this

         g = sns.PairGrid(abundance.reset_index(drop=True))

         # put histograms on the diagonal
         g.map_diag(plt.hist)

         # put scatters on the upper diagonals
         g.map_upper(plt.scatter)


         # put text giving correlations on lower diagonals
         # using our custom function

         def corr_text(x,y, **kwargs):
             meanx, meany = np.mean(x), np.mean(y)
             corrxy = np.corrcoef(x,y)[0,1]
             ax = plt.gca()  # gets current axis
             # figure out !1/3 of way along each axis limit
             # to print the text
             xlim = ax.get_xlim()
             midx = (xlim[0] + xlim[1]) * 0.35
             ylim = ax.get_ylim()
             midy = (ylim[0] + ylim[1]) * 0.35
             return ax.text(midx, midy, "{:0.2f}".format(corrxy), fontsize=24, **kwargs)

         g.map_lower(corr_text)
```



Linearity is not a requirement for PCA, as it's simply a rigid rotation of the original data. So we'll continue with our analysis.

1.1 PCA using StatsModels

The StatsModels package has two routines for carrying out PCA – `pca` and `pcasvd`. The `pca` function uses the `np.linalg.eig` to conduct principal components analysis, while `pcasvd` uses a technique called Singular Value Decomposition (SVD). The SVD approach is more efficient and potentially more accurate when the data is singular, but we'll wait until we discuss SVD before we explore the differences in detail.

It can be instructive to read source code from packages you use. The `pca` implementation is very simple and you can browse the source code for its implementation at this link.

```
In [314]: import statsmodels.sandbox.tools.tools_pca as pca

          # read the help for the pca fxn
          ?pca.pca
```

```
In [331]: ctrd = abundance - abundance.mean()
          orig, scores, evals, evecs = pca.pca(ctrd)
```

```
In [332]: # standard deviations associated with each PC
          # the standard deviations are the square roots
          # of the eigenvalues
          np.sqrt(evals)
```

```
Out[332]: array([ 14.86530583,   8.8149115 ,   6.21925038,   5.03477441,   3.48230841])
```

```
In [333]: # percent variance explained by each PC
          evals / np.sum(evals)
```

```
Out[333]: array([ 0.58953305,  0.2072986 ,  0.10318975,  0.06762706,  0.03235154])
```

```
In [334]: # cumulative percent variance explained
          np.cumsum(evals / np.sum(evals))
```

```
Out[334]: array([ 0.58953305,  0.79683166,  0.9000214 ,  0.96764846,  1.        ])
```

We see that approximately 59% of the variance in the data is captured by the first PC, and approximately 90% by the first three PCs.

Let's look at the PC loadings:

```
In [414]: loadings = np.dot(evecs, np.diag(np.sqrt(evals)))

          # create a dataframe to hold the loadings info
          loadings_df = pd.DataFrame(data=loadings,
                                     index=['a','b','c','d','e'],
                                     columns=['PC{}'.format(i+1) for i in range(len(evals))])

          print loadings_df
```

```
PC1   PC2   PC3   PC4   PC5
a  12.05  0.62  3.30  0.93  0.51
b   7.62 -2.45 -2.97 -3.19 -0.60
c  -2.41 -7.82  2.54 -0.06 -0.49
d   3.30 -2.79 -3.50  3.67  0.15
e   0.98  1.56  0.51  0.90 -3.35
```

The magnitude of the loadings is what you want to focus on. For example, species "a" and "b" contribute most to the first PC, while species "c" has the largest influence on PC2.

You can think of the loadings, as defined above, as the components (i.e lengths of the projected vectors) of the original variables with respect to the PC basis vectors. Since vector length is proportional to the standard deviation of the variables they represent, you can think of the loadings as giving the standard deviation of the original variables with respect the PC axes. This implies that the loadings squared sum to the total variance in the original data, as illustrated below.

```
In [336]: np.sum(loadings**2,axis=1)
```

```
Out[336]: array([ 157.63678161,   83.44367816,   73.62758621,   44.43793103,
                   15.68850575])
```

```
In [337]: abundance.var()
```

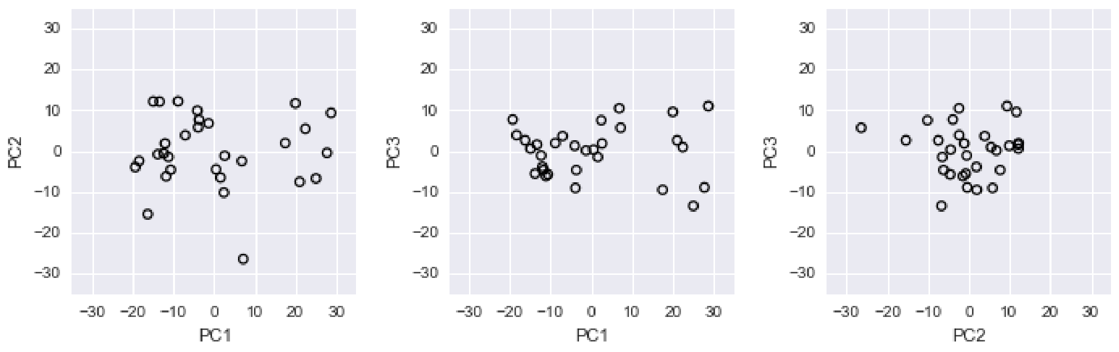
```
Out[337]: a    157.64
          b     83.44
          c     73.63
          d     44.44
          e     15.69
          dtype: float64
```

1.2 Drawing Figures to Represent PCA

1.2.1 PC Score Plots

The simplest PCA figure is to depict the PC scores, i.e. the projection of the observations into the space defined by the PC axes. Let's make a figure with three subplots, depicting PC1 vs PC2, PC1 vs PC3, and PC2 vs. PC3.

```
In [393]: fig, axes = plt.subplots(1,3)
          fig.set_size_inches(10,12)

          ax = axes[0]
          plt.sca(ax)  # set current axis
          ax.scatter(scores[:,0], scores[:,1], facecolors='none',
                     s=30, linewidths=1.25)
          plt.xlim(-35,35)
          plt.ylim(-35,35)
          plt.xlabel("PC1")
          plt.ylabel("PC2")
          ax.set_aspect('equal')


          ax = axes[1]
          plt.sca(ax)
          ax.scatter(scores[:,0], scores[:,2], facecolors='none',
                     s=30, linewidths=1.25)
          plt.xlim(-35,35)
          plt.ylim(-35,35)
          plt.xlabel("PC1")
          plt.ylabel("PC3")
          ax.set_aspect('equal')


          ax = axes[2]
          plt.sca(ax)
          ax.scatter(scores[:,1], scores[:,2], facecolors='none',
                     s=30, linewidths=1.25)
          plt.xlim(-35,35)
          plt.ylim(-35,35)
          plt.xlabel("PC2")
          plt.ylabel("PC3")
          ax.set_aspect('equal')

          # minimizes overlap
          plt.tight_layout(pad=2)
```

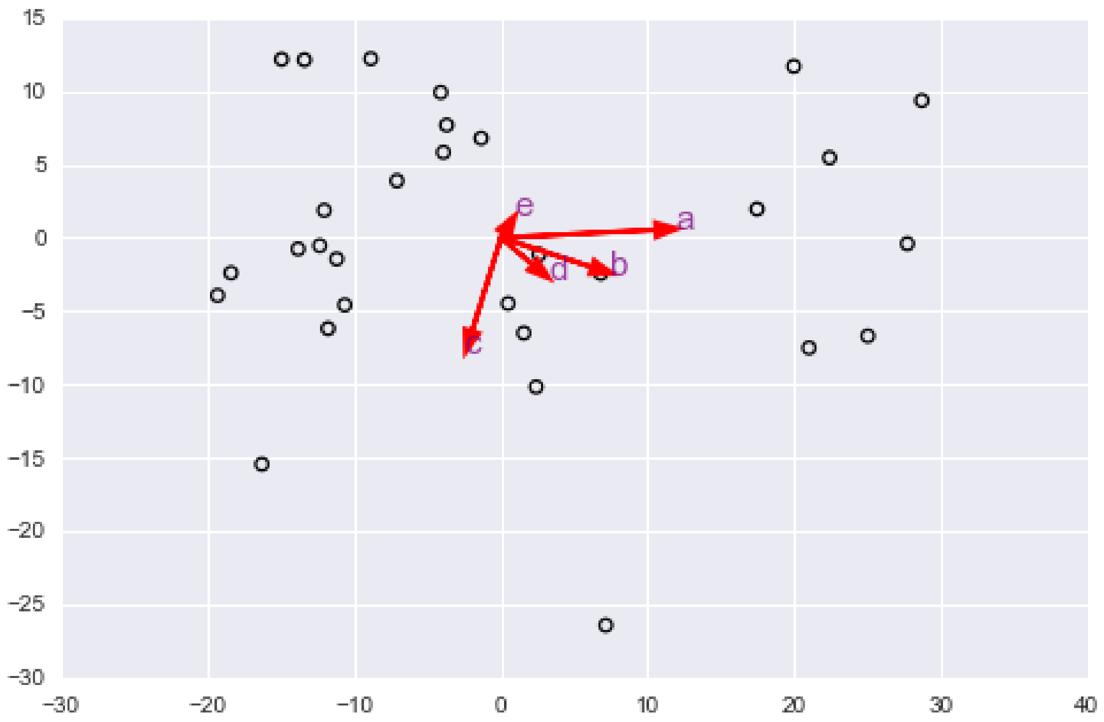


Note that you should always set the aspect ratio to be equal when plotting PC scores, so that the distances between points are accurate representations. Note too that I used the `xlim` and `ylim` functions to keep the axis limits the same in all plots; comparable scaling of axes is important when comparing PCA plots.

1.2.2 A first brush with biplots

As we discussed above, the loadings are projections of the original variables in the space of the PCs. This implies we can depict those loadings in the same PC basis that we use to depict the scores.

```
In [452]: fig, ax = plt.subplots()
          plt.scatter(scores[:,0], scores[:,1], facecolors='none',
                      s=30, linewidths=1.25)

          ax.set_aspect('equal')


          # a function to take care of drawing the loading vecdtors
          def draw_loadings(loadingsdf, idx, col1, col2):
              x = loadingsdf.ix[idx,col1]
              y = loadingsdf.ix[idx,col2]
              plt.arrow(0, 0, x, y,
                     color='red', linewidth=2, head_width=1,
                     length_includes_head=True)
              plt.text(x, y, idx, fontsize=15, color='purple', alpha=0.75)


          # draw loadings

          draw_loadings(loadings_df, "a", "PC1", "PC2")
          draw_loadings(loadings_df, "b", "PC1", "PC2")
          draw_loadings(loadings_df, "c", "PC1", "PC2")
          draw_loadings(loadings_df, "d", "PC1", "PC2")
          draw_loadings(loadings_df, "e", "PC1", "PC2")
```



The output of the code above is called a "biplot", as it simultaneously depicts both the observations and variables in the same space. From this biplot we can immediately see that variable "a" is highly correlated with PC1, but only weakly associated with PC2. Conversely, variable "c" is strongly correlated with PC2 but only weakly so with PC1. We can also approximate the correlations among the variables themselves – for example "b" and "d" are fairly strongly correlated, but weakly correlated with "c". Keep in mind that with respect to the relationships among the variables, this visualization is a 2D projection of a 5D space so the geometry is only approximate.

The biplot is a generally useful tool for multivariate analyses and there are a number of different ways to define biplots. We'll study biplots more formally in a few weeks after we've covered singular value decomposition.