Performing Principal Components Analysis

Principal Components Analysis (**PCA**) is a statistical procedure to perform a reduction of dimension of a number of variables to a smaller subset that is linearly uncorrelated. Its practical application in population genetics is assisting the visualization of relationships of individuals that is being studied.

While most of the recipes in this chapter make use of Python as a "glue language" (Python calls external applications that actually do most of the work) with PCA, we have an option, that is, we can either use an external application (for example, EIGENSOFT smartpca) or use scikit-learn and perform everything on Python. We will perform both.

Getting ready

You will need to run the first recipe in order to use the `hapmap10_auto_noofs_ld_12` PLINK file (with alleles recoded as 1 and 2). PCA requires LD-pruned markers; we will not risk using the offspring here because it will probably bias the result. We will use the recoded PLINK file with alleles as 1 and 2 because this makes processing easier with smartpca and scikit-learn.

As with the second recipe, if you are not using Docker, you will also be using some of the code that I have produced. You can find this code at https://github.com/tiagoantao/pygenomics. You can install it with

```
pip install pygenomics
```

For this recipe, you will need to download EIGENSOFT (http://www.hsph.harvard.edu/alkes-price/software/), which includes the smartpca application that we will use.

There is a notebook in the `03_PopGen/PCA.ipynb` recipe, but you will still need to run the first recipe.

How to do it...

Take a look at the following steps:

1. Let's load the metadata as follows:

```
f = open('relationships_w_pops_121708.txt')
ind_pop = {}
f.readline()  # header
for l in f:
    toks = l.rstrip().split('\t')
    fam_id = toks[0]
    ind_id = toks[1]
    pop = toks[-1]
    ind_pop['/'.join([fam_id, ind_id])] = pop
f.close()
ind_pop['2469/NA20281'] = ind_pop['2805/NA20281']
```

- In this case, we will add an entry that is consistent with what is available on the PINK file.

2. Let's convert the PLINK file to the EIGENSOFT format:

```
from genomics.popgen.plink.convert import to_eigen
to_eigen('hapmap10_auto_noofs_ld_12', 'hapmap10_auto_noofs_ld_12')
```

- This uses a function that I have written to convert from PLINK to the EIGENSOFT format. This is mostly text manipulation, not precisely the most exciting code.

3. Now, we will run smartpca and parse its results as follows:

```
from genomics.popgen.pca import smart
ctrl = smart.SmartPCAController('hapmap10_auto_noofs_ld_12')
ctrl.run()
wei, wei_perc, ind_comp =\ smart.parse_evec('hapmap10_auto_noofs_ld_12.evec', 'hapmap10_aut
```

- Again, this will use a couple of functions from pygenomics to control smartpca and then to parse the output. The code is typical for this kind of operations, and while you are invited to inspect it, be aware that it's quite straightforward.
- The parse function will return PCA weights (which we will not use, but you should inspect), normalized weights, and then principal components (usually up to PC 10) per individual.

4. Then, we plot PC1 and PC2, as shown in the following code:

```
from genomics.popgen.pca import plot
plot.render_pca(ind_comp, 1, 2, cluster=ind_pop)
```

- This will produce the following figure. We will supply the plotting function and the population information retrieved from the metadata, which allows you to plot each population with a different color.
- The results are very similar to published results; we will find four groups. Most Asian populations are located on top, the African populations are located on the right-hand side, and the European populations are located at the bottom. Two more admixed populations (GIH and MEX) are located in the middle:

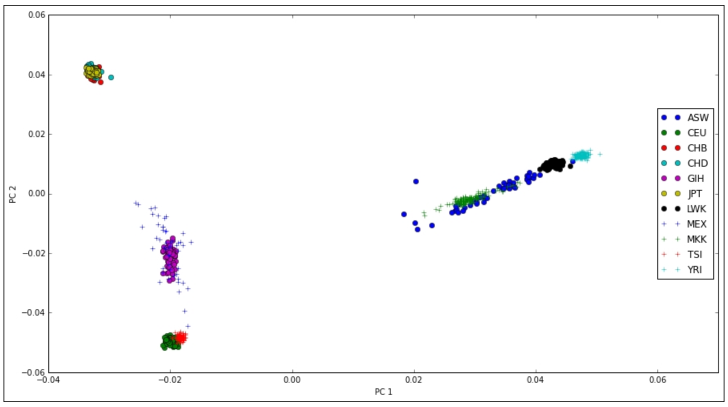

Figure 5: PC1 and PC2 of the HapMap data as produced by smartpca

5. Now, let's turn to a PCA plot produced by Python libraries only. To be able to run scikit-learn PCA on our data, let's get the individual order on the `PED` file and the number of SNPs first as follows:

```
f = open('hapmap10_auto_noofs_ld_12.ped')
ninds = 0
ind_order = []
for l in f:
    ninds += 1
    toks = l[:100].replace(' ', '\t').split('\t')
    fam_id = toks[0]
    ind_id = toks[1]
    ind_order.append('%s/%s' % (fam_id, ind_id))
nsnps = (len(line.replace(' ', '\t').split('\t')) - 6) // 2
f.close()
```

6. Then, we create an array required for the PCA function reading in the `PED` file:

```
import numpy as np
pca_array = np.empty((ninds, nsnps), dtype=int)

f = open('hapmap10_auto_noofs_ld_12.ped')
for ind, l in enumerate(f):
    snps = l.replace(' ', '\t').split('\t')[6:]
    for pos in range(len(snps) // 2):
        a1 = int(snps[2 * pos])
        a2 = int(snps[2 * pos])
        my_code = a1 + a2 - 2
        pca_array[ind, pos] = my_code
f.close()
```

- This code will be slow to execute.
- The most import part of the code is coding of alleles, and the ability for PCA to produce meaningful results rely on a good coding here. Remember that we will use a PLINK file that has a 1 and 2 allele coding. We will use the following strategy, that is 11s are converted to 0, 12 (and 21) are converted to 1 and 22 are converted to 2.

7. We can now call the scikit-learn PCA function, which requests `8` components:

```
from sklearn.decomposition import PCA
my_pca = PCA(n_components=8)
my_pca.fit(pca_array)
trans = my_pca.transform(pca_array)
```

8. Finally, let's print eight PCs as follows:

```
sc_ind_comp = {}
for i, ind_pca in enumerate(trans):
    sc_ind_comp[ind_order[i]] = ind_pca
plot.render_pca_eight(sc_ind_comp, cluster=ind_pop)
```

- We will use a different function to perform the plotting here; you will able to see up to component `8`.
- The result is qualitatively similar to the smartpca version (it will be a worrying situation if it had been otherwise). Note that this is a mirror image from the previous figure; swapping signals on PCA is not a major issue at all:

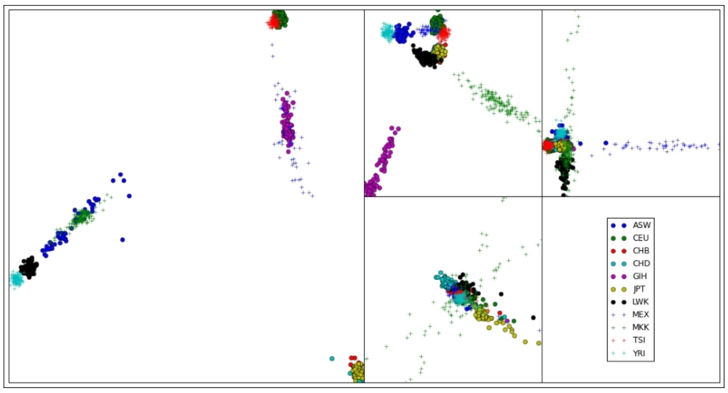

Figure 6: PC1 to PC8 of the HapMap data as produced by scikit-learn

There's more...

An interesting question here is which method should you use? smartpca or scikit-learn? The results are similar, so if you are performing your own analysis, you are free to choose. However, if you publish your results in a scientific journal, smartpca is probably a safer choice because it's based on the published piece of software in the field of genetics; reviewers will probably prefer this.

See also

- The paper that probably popularized the use of PCA in genetics was Novembre et al *Genes mirror geography within Europe* on Nature, where a PCA of Europeans map almost perfectly to the map of Europe at **http://www.nature.com/nature/journal/v456/n7218/abs/nature07331.html**. Note that there is nothing in PCA that assures it will map to geographical features (just check our PCA earlier).
- The smartpca is described in Patterson et al, *Population structure and eigenanalysis*, *PLoS Genetics* at **http://journals.plos.org/plosgenetics/article?id=10.1371/journal.pgen.0020190**.
- A discussion of the meaning of PCA can be found in McVean's paper on, *A Genealogical Interpretation of Principal Components Analysis*, *PLoS Genetics* at **http://journals.plos.org/plosgenetics/article?id=10.1371/journal.pgen.1000686**.
- As usual, the Wikipedia page is nicely written at **http://en.wikipedia.org/wiki/Principal_component_analysis**.