1 Lab Sheet 3: Principal Components Analysis

For this lab sheet, we will use the results from the 2016 Olympic Games Women's Heptathlon athletics event. The results are in the data set `heptathlon2016.csv`, and are taken from https://en.wikipedia.org/wiki/Athletics_at_the_2016_Summer_Olympics. The process involved was to right-click on the web page, select "View Page Source" (with Firefox—other browsers have equivalents), copy the relevant part of the HTML code, and clean it manually.

The heptathlon consists of 7 disciplines:

- `hurdles` : the 100m hurdles (*run*)
- `highjump` : the high jump (*jump*)
- `shotput` : the shot put (*throw*)
- `run200m` : the 200m (*run*)
- `longjump` : the long jump (*jump*)
- `javelin` : the javelin (*throw*)
- `run800m` : the 800m (*run*)

Let's make some comments to start.

- Events marked *run* are running events; the data is given as the time taken in seconds. The aim is to go as quickly as possible, which means that better performances will be faster, so correspond to **smaller** numerical data.
- Events marked *jump* are jumping events; the data is given as a distance in metres. The aim is to jump as high/far as possible, so that better performances correspond to **larger** numerical data.
- Events marked *throw* are throwing events; the data is given as a distance in metres. The aim is to throw as far as possible, so that better performances correspond to **larger** numerical data.

27 athletes completed all 7 events (there were also 4 who didn't complete the event, but they aren't included in this data set).

As with any data set in CSV form, first have a quick look at it by opening it in Excel.

2 Lab Sheet 3a: Principal Components Analysis in Python

As explained in the lecture notes, Principal Components Analysis (PCA) is the first way to try to reduce the dimensionality of a data set. It finds a *linear* subspace in which the data approximately lies.

Let's load up the data set, using the athlete's name as an index column, and open it in `pandas` as a data frame `hept`. That is, type

```
hept=pd.read_csv("heptathlon2016.csv",index_col="Name")
```

Now compute some summary statistics for the data.

Let's store the events for future reference:

```
events=hept.T.index
```

Plot all the data with the `seaborn` command `pairplot`. Can you get anything useful from this? I think one can discern some outliers for particular events; there are two good performances on the high jump, for example, and outliers for the 200m, long jump and 800m. In this case, we could have got this just from an examination of the data set, since it's so small.

In order to use PCA in Python, we will use one more (big) module: `scikit-learn`, often abbreviated as `sklearn`. This module contains pretty well all of the machine learning functions we will need in this course. Rather than load the whole module in for each method we use, it is normal to just load up the part we need.

Remark. Since all that is needed for PCA is little more than eigenvalue calculations for a variance matrix, there are ways to do this using the usual modules we are importing, but the `scikit-learn` method is easy, and we'll use it for other methods in the module, so we should get used to it with an easy method!

For PCA, we need to add:

```
from sklearn.decomposition import PCA
```

Then running PCA is remarkably simple—we just type

```
pca=PCA()
pca.fit(hept)
```

The first command here tells us that `pca` means that we run PCA. At this point, `pca` consists of an abstract method. After the second line, we have fit the abstract method to the concrete data in `hept`, and `pca` now contains all the results. (We will see this construction a lot!)

There are some additional options; try replacing `PCA()` with `PCA(n_components=5)`, say, to insist that 5 components are returned, or `PCA(n_components=0.9)` to insist that enough components are returned to account for 90% of the total variance (the behaviour is different whether the input is an integer or a value between 0 and 1).

It's worth remarking that the `scikit-learn` manual pages on the internet are pretty good. So a search for "`python scikit-learn PCA`" will probably bring up the manual page quickly, and will reveal further information about the function.

We now want to see what PCA tells us. Type

```
pca.explained_variance_ratio_
```

to find information about proportion of variance represented by the principal components. (The eigenvalues of the variance matrix are in `pca_explained_variance_`, incidentally, but the proportions are more likely to be useful.) The command

```
np.cumsum(pca.explained_variance_ratio_)
```

gives the cumulative totals of these. You should be able to see that already the first two components account for 98.5% of the total variation. This would suggest that the data is mostly spread out in two dimensions. This is something of a surprise, and usually indicates that you've done it wrong (and that's true here!).

Typing `pca.components_` reveals that the 800m run dominates the first component, and the javelin dominates the second. Is it really true that the heptathlon completely depends only on these two events? Of course not. What is happening is that the numerical data is much more spread out in these two events simply because the numbers in these columns are much greater.

We need to adjust our data so that all the variables are comparable. What went wrong? We used the *variance* matrix, when we should have used the *correlation* matrix. You will almost always want to use the correlation matrix. So we need to normalise our data so that the variables are scaled to have the same variance. This uses another part of the `scikit-learn` library, which contains functions for preparing the data frame before applying the learning procedure:

```
from sklearn.preprocessing import StandardScaler
```

and then type

```
scaler=StandardScaler()
scaler.fit(hept)
hept_sc=scaler.transform(hept)
```

to transform the data. Now repeat the above steps, using this version of the data set. You won't be able to make the pairs plot (or at least, more work will be needed!), since `StandardScaler` transforms the `pandas` data frome into a Python `array` (but PCA will still work with this format).

Let's look now at Principal Component 1 (PC1). This accounts for 35.68% of the variance in the original data set (where can you see this?). Find the loadings for PC1.

You should see that the signs of the loadings are positive for the running events and throwing events (slightly more for the running events), and negative for the jumping events.

[*Note: it may be that your eigenvectors come out to be the negative of mine (remember that eigenvectors are only defined up to sign), and then you should change all the directions in what follows!*]

How can we get a higher score on PC1? We would expect that we would need larger numbers for the running and throwing events, and smaller numbers for the jumping events. Recalling that large numbers on the running events correspond to *weak* performance, etc., you can see that the highest scores on PC1 are likely to come from weak performances in running and jumping, but strong performances in throwing. So we would expect high scores on PC1 to come from overall weaker performers who might have some compensation from throwing events. Conversely, low scores on PC1 would be expected to come from stronger performers with a weakness in throwing.

In any plot with PC1 along the axis, you should then expect to locate strong performers with a weakness in throwing towards the negative direction.

We can see all the results rotated onto the PCs with `p=pca.fit_transform(hept_sc)`. Try this, and look at what we get. (Since we will use it several times, it makes sense to store it as a variable for future use.)

The two British athletes are the two with the lowest scores on PC1. What can you deduce from this? Look at their results in the original dataset and confirm this.

Let's turn to PC2. Now there are positive scores for jumping and throwing events, and very little contribution from running events. As for PC1, try to understand how performers can get high scores on PC2. We would need strong performances in both jumping and throwing. So low scores will come from weak throwers and jumpers, and conversely, strong throwers and jumpers will score highly on PC2.

The overall winner is by far the highest scorer on PC2. Check this on the data.

PC3 is clearly a mostly contrast between 800m and hurdles, so reflects primarily the running events, and high scores on PC3 will come from good hurdlers who are comparatively weak at the 800m, and vice versa.

There is an outlier on PC3—find them, and account for this from the data.

These three PCs already account for about 75% of the total variance in the data set. We could ignore the other PCs if we felt that most of the information we want to retain is already reflected in the first PCs. This reduces our dimensionality from 7 to 3! Actually, we will probably want more—75% is unlikely to be enough for most purposes (although this depends on the context), and in any case, this data set is small enough that it's easy to work with all the PCs. I'd definitely take PC4 too (can you try to interpret this component?)

A useful construction is the *scree plot* of cumulative variance against the number of PCs:

```
cumexp=np.concatenate([[0],pca.explained_variance_ratio_])
plt.plot(np.cumsum(cumexp))
plt.xlabel("number of components")
plt.ylabel("cumulative variance")
```

(Irritatingly, we have to add the initial 0 to the vector of explained variances so that the plot starts at $(0, 0)$. Try it without...)

It's worth asking how many PCs we should keep for further analysis. Obviously we win by taking fewer components, since it's easier to understand the data—on the other hand, there's a trade-off, since we throw away more of the information in the data set when we do this.

It makes little sense to take one PC and to ignore subsequent ones if the next one is of similar importance to one you take. It is therefore more sensible to stop at points where there is a sudden jump down in the variance, and you should think about why this corresponds to an "elbow" or kink in the scree plot.

In the heptathlon data set, there aren't really any elbows, since the importances are 35.7%, 24.1%, 14.7%, 9.6%, 6.6%, 5.0% and 4.4%. The jump after the 5th and 6th PCs are rather small, so it is sensible to stop after the 4th PC (if 84.1% of the total variation is enough), or to take all of them otherwise, but be aware that the later components are probably mostly random noise.

Actually, we probably wanted to plot the PCs on a scatterplot. As we've seen, the command `p=pca.fit_transform(hept_sc)` rotated the data onto the axes given in the PCs, and so we can see the first two PCs of the data with

```
plt.scatter(x=p[:,0],y=p[:,1])
plt.xlabel("PC1")
plt.ylabel("PC2")
```

(As an exercise, you might like to define a function which takes in two integers i and j, and plots the ith and jth principal components on a scatterplot, labelling the axes with something like "PC1 (35.7%)" etc.)

You've already seen the scores of the two British athletes on PC1, and that of the overall winner on PC2. Identify the points on the scatterplot corresponding to these athletes.

Plot PC2 against PC3, and identify outliers. Check them with the data.

We might want to add further points to the plot. For example, perhaps we want to compare the results of Jessica Ennis from 2012, when she won the gold, to her silver medal performance from 2016 by adding that result to the plot. Here are her results from 2012:

```
Ennis-Hill,12.54,1.86,14.28,22.83,6.48,47.49,128.65
```

We can start with this data as a list, but PCA expects a matrix (rather, a 2-dimensional array) and not a list, so we can `reshape` it into the right size. Then we have to subject it to the same scaling process that the original data went through, using `scaler`, and finally transform it using the PCA coordinates:

```
eh=[12.54,1.86,14.28,22.83,6.48,47.49,128.65]
eh=np.reshape(eh,(1,-1))
eh1=scaler.transform(eh)
eh1=pca.transform(eh1)
```

Now we can add this point to the plot above:

```
plt.scatter(p[:,0], y=p[:,1])
plt.plot(eh1[:,0],eh1[:,1],"r*")
plt.xlabel("PC1")
plt.ylabel("PC2")
```

(Actually, `scikit-learn` provides something called "pipelines" for where we want to do the same operations on new data as for an original data set; we could set up a pipeline for this sequence of operations, and apply it as a single operation. We won't need these in this module, though.)

Remark. The `plot` command produces square plots, and chooses scales along the axes so that the data fills the space in the plot. One should realise that PC1, by definition, is the direction where the data is most spread out, so that the scale for PC1 will be more condensed than that for PC2. To get a realistic picture of the data, you may prefer to plot the data on axes with equal scaling. `matplotlib` can handle this with

```
plt.axis("equal")
```

We've said that the PCs are linear combinations of the original variables. It is possible to represent the original variables on a plot of the data on the first two principal components (a *biplot*). The variables are represented as arrows, with lengths proportional to the standard deviations, and angles between a pair of arrows proportional to the covariances. The orientation of the arrows on the plot relates to the correlations between the variables and the principal components and so can be an aid to interpretation. Although R can do this automatically, this doesn't seem to be in Python, but there are recipes online. Try

```
plt.xlim(-1,1)
plt.ylim(-1,1)
plt.xlabel("PC1")
plt.ylabel("PC2")
x=p[:,0]
y=p[:,1]
```

```
coeff=np.transpose(pca.components_[0:2,:])
n=hept.shape[1]
scalex=1.0/(x.max()-x.min())
scaley=1.0/(y.max()-y.min())
plt.scatter(x*scalex,y*scaley)
for i in range(n):
    plt.arrow(0,0,coeff[i,0],coeff[i,1],color="r",alpha=0.5)
    plt.text(coeff[i,0]*1.15,coeff[i,1]*1.15,events[i],color="g")
```

This gives the directions of each of the original variables on PC1 and PC2. Try to identify from the biplot what might be particularly good events for the medallists, and confirm this from the data set.

3 Lab Sheet 3b: Principal Components Analysis in R

As explained in the lecture notes, Principal Components Analysis (PCA) is the first way to try to reduce the dimensionality of a data set. It finds a *linear* subspace in which the data approximately lies.

Let's load up the data set, and open it in R as a data frame `hept`. That is, type

```
hept<-read.csv(file="heptathlon2016.csv",header=TRUE)
```

Now compute some summary statistics for the data.

Plot all the data with `pairs(hept[,2:8])`. Can you get anything useful from this? I think one can discern some outliers for particular events; there are two good performances on the high jump, for example, and outliers for the 200m, long jump and 800m. We could probably have got this from the data set, since it's a small set.

We now want to see what PCA tells us. Run

```
hept.pca<-princomp(hept[,2:8])
```

(Why did we use `hept[,2:8]` rather than just `hept`?) This runs PCA, and stores the results in `hept.pca`. Type `summary(hept.pca)` to see what it says. You will notice that it lists the relative importances of each component, and that already the first two components account for 98.5% of the total variation. This would suggest that the data is mostly spread out in two dimensions. This is something of a surprise, and usually indicates that you've done it wrong (and that's true!).

We can type `loadings(hept.pca)`, and find that the 800m run dominates the first component, and the javelin dominates the second. Is it really true that the heptathlon completely depends only on these two events? Of course not. What is happening is that the numerical data is much more spread out in these two events simply because the numbers in these columns are much greater.

We need to adjust our data so that all the variables are comparable. What went wrong? We used the *variance* matrix, when we should have used the *correlation* matrix. R's `princomp` command uses the variance matrix by default, but you will almost always want to use the correlation matrix. Instead, we should do

```
hept.pca<-princomp(hept[,2:8],cor=TRUE)
```

to ensure you are using the right matrix. Now repeat the above steps, using this version of `hept.pca`.

Let's look now at Principal Component 1 (PC1). This accounts for 35.68% of the variance in the original data set (where can you see this?). Find the loadings for PC1.

You should see that the signs of the loadings are positive for the running events and throwing events (slightly more for the running events), and negative for the jumping events.

[*Note: it may be that your eigenvectors come out to be the negative of mine (remember that eigenvectors are only defined up to sign), and then you should change all the directions in what follows!*]

How can we get a higher score on PC1? We would expect that we would need larger numbers for the running and throwing events, and smaller numbers for the jumping events. Recalling that large numbers on the running events correspond to *weak* performance, etc., you can see that the highest scores on PC1 are likely to come from weak performances in running and jumping, but strong performances in throwing. So we would expect high scores on PC1 to come from overall weaker performers who might have some compensation from throwing events. Conversely, low scores on PC1 would be expected to come from stronger performers with a weakness in throwing.

In any plot with PC1 along the axis, you should then expect to locate strong performers with a weakness in throwing towards the negative direction.

We can rotate all the results onto the PCs with `predict(hept.pca)`. Try this, and look at what we get.

The two British athletes are the two with the lowest scores on PC1. What can you deduce from this? Look at their results in the original dataset and confirm this.

Let's turn to PC2. Now there are negative scores for jumping and throwing events, and very little contribution from running events. As for PC1, try to understand how performers can get high scores on PC2. We would need weak performances in both jumping and throwing. So high scores will come from weak throwers and jumpers, and conversely, strong throwers and jumpers will score negatively on PC2.

The overall winner is by far the lowest scorer on PC2. Check this on the data.

PC3 is clearly a mostly contrast between 800m and hurdles, so reflects primarily the running events, and high scores on PC3 will come from good hurdlers who are comparatively weak at the 800m, and vice versa.

There is an outlier on PC3—find them, and account for this from the data.

These three PCs already account for about 75% of the total variance in the data set. We could ignore the other PCs if we felt that most of the information we want to retain is already reflected in the first PCs. This reduces our dimensionality from 7 to 3! Actually, we will probably want more—75% is unlikely to be enough for most purposes (although this depends on the context), and in any case, this data set is small enough that it's easy to work with all the PCs. I'd definitely take PC4 too (can you try to interpret this component?)

Let's try

```
plot(hept.pca)
```

This produces something slightly unexpected (perhaps). We get a sort of bar chart. An examination of the plot, and thinking about the data, might suggest that it contains exactly the relative importances of the PCs, which you can read off from `summary()`.

Here's a function to print the cumulative variance of the PCs (a *scree plot*):

```
screeplot<-function(mydata,cor=F,maxcomp=10) {
my.pc<-princomp(mydata, cor=cor)
k<-min(dim(mydata),maxcomp)
x<-c(0:k)
y<-my.pc$sdev[1:k]*my.pc$sdev[1:k]
y<-c(0,y)
z<-100*cumsum(y)/sum(my.pc$sdev*my.pc$sdev)

plot(x,z,type="l",xlab="number of dimensions",
    cex.main=1.5, lwd=3, col="red",ylim=c(0,100),
    ylab="cumulative percentage of total variance",
    main="Scree plot of variances",
    xaxt="n", yaxt="n")

axis(1,at=x,lwd=2)
axis(2,at=c(0,20,40,60,80,100),lwd=2)
abline(a=100,b=0,lwd=2,lty="dashed",col="orange")
text(x,z,labels=x,cex=0.8,adj=c(1.2,-.1),col="blue")
}
```

Calling `screeplot(hept[,2:8],cor=TRUE)` will give the cumulative plot of variance coming from PCA with the correlation matrix. As an exercise, those of you wishing to work with `ggplot2` should try to recreate this plot (my solution is in the R script file) and the other plots on this lab sheet; not all are immediately in the `ggplot2` package, and you may need to install packages (`GGally` for the pairs plot earlier, for example).

It's worth asking how many PCs we should keep for further analysis. Obviously we win by taking fewer components, since it's easier to understand the data—on the other hand, there's a trade-off, since we throw away more of the information in the data set when we do this.

It makes little sense to take one PC and to ignore subsequent ones if the next one is of similar importance to one you take. It is therefore more sensible to look at this plot, and see if there are components after which there is a sudden jump down in the variance (this corresponds to an "elbow" or kink in the scree plot). That isn't really true in the heptathlon data set, since the importances are 35.7%, 24.1%, 14.7%, 9.6%, 6.6%, 5.0% and 4.4%. The jump after the 5th and 6th PCs are rather small, so it is sensible to stop after the 4th PC (if 84.1% of the total variation is enough), or to take all of them otherwise, but be aware that the later components are probably mostly random noise.

Actually, we probably wanted to plot the PCs on a scatterplot. As we've seen, the command `predict(hept.pca)` rotates the data onto the axes given in the PCs, and so we can see the first two PCs of the data with

```
plot(predict(hept.pca)[,1:2])
```

You've already seen the scores of the two British athletes on PC1, and that of the overall winner on PC2. Identify the points on the scatterplot corresponding to these athletes.

You can check this by typing

```
identify(predict(hept.pca)[,1:2])
```

and clicking on the point on the plot that you want to identify (press Esc to quit this process). Plot PC2 against PC3, and identify outliers. Check them with the data.

We might want to add further points to the plot. For example, we want to compare the results of Jessica Ennis from 2012, when she won the gold, to her silver medal performance from 2016 by adding that result to the plot. Here are her results from 2012:

```
Ennis-Hill,12.54,1.86,14.28,22.83,6.48,47.49,128.65
```

It turns out to be mildly awkward to do this! Here's one way to do this:

```
eh2012<-c(12.54,1.86,14.28,22.83,6.48,47.49,128.65)
plot(predict(hept.pca)[,1:2])
eh2012.pca<-scale(t(eh2012),hept.pca$center,hept.pca$scale)%*%hept.pca$loadings
points(eh2012.pca[1],eh2012.pca[2],pch=16,col=red)
```

With more points, it's probably simplest to merge the data frames, and add all the points using `predict`. There's some R code for this in the R script file for this lab sheet.

Let's explain the mysterious code here – remember that you can see how commands in R work with `?scale`. There aren't that many details there, in this case. But we can see what happens by getting R to produce `hept.pca$center`, which stores the means of the variables, and `hept.pca$scale`, which stores their standard deviations.

`scale(x,hept.pca$center,hept.pca$scale)` just tells you to take the vector (or matrix) `x`, subtract the mean from each column, then divide each column by the standard deviation – this is equivalent

to converting `x` in the same way that we did with the whole data set by using the correlation matrix. (When we run princomp with the correlation matrix, it is equivalent to working with the data set in this scaled form. We want to look at the new data in the same way, so we need to scale the new data so that it looks like the same coordinates as the original data.)

After doing that, we need to take a linear combination of the variables in the same way as the original data, by multiplying by the loadings, to get the values of the vector `x` in each principal component, and then to add the first two principal components to the plot.

Remark. The `plot` command produces square plots, and chooses scales along the axes so that the data fills the space in the plot. One should realise that PC1, by definition, is the direction where the data is most spread out, so that the scale for PC1 will be more condensed than that for PC2. To get a realistic picture of the data, you may prefer to plot the data on axes with equal scaling. There is a command for this in the `MASS` library, which we will mention more in the next lab sheet. Type instead:

```
library(MASS)
eqscplot(predict(hept.pca)[,1:2])
```

and compare this plot with the one given by `plot`. (Note that RStudio scales the plots according to the window size anyway, but this works well in the plain R interface.)

We've said that the PCs are linear combinations of the original variables. It is possible to represent the original variables on a plot of the data on the first two principal components (a *biplot*). The variables are represented as arrows, with lengths proportional to the standard deviations, and angles between a pair of arrows proportional to the covariances. The orientation of the arrows on the plot relates to the correlations between the variables and the principal components and so can be an aid to interpretation. Try

```
biplot(hept.pca)
```

This automatically labels all the observations, and gives the directions of each of the original variables on PC1 and PC2. Try to identify from the biplot what might be particularly good events for the medallists, and confirm this from the data set. (If you are recreating all the plots in `ggplot2`, this is a bit harder, and you should search online for "`ggplot biplot`"; there are many solutions available, although biplots aren't naturally implemented in `ggplot2`; there is a `ggbiplot` package, for example.)