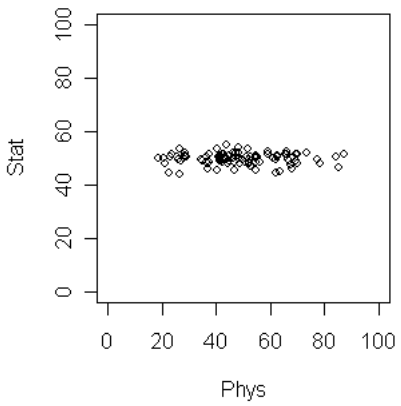
Miscellaneous statistical analyses with R

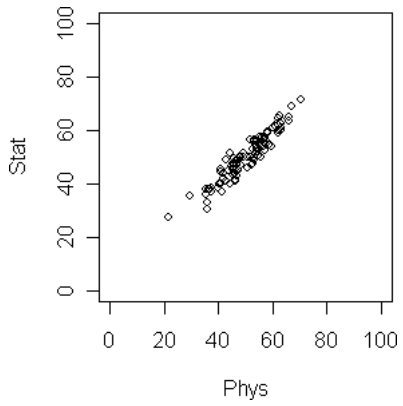
Contents

1 Principal Component Analysis

We start with a simple example to motivate this statistical tool. Consider 100 students with Physics and Statistics grades shown in the diagram below. The data set is in marks.dat.



If we want to compare among the students which grade should be a better discriminating factor? Physics or Statistics? Surely Physics, since the variation is larger there. This is a common situation in data analysis where the direction along which the data *varies the most* is of special importance.
Now suppose that the plot looks like the following. What is the best way to compare the students now?



Here the direction of maximum variation is like a slanted straight line. This means we should take linear combination of the two grades to get the best result. In this simple data set the direction of maximum variation is more or less clear. But for many data sets (especially high dimensional ones) such visual inspection is not adequate or even possible! So we need an objective method to find such a direction. **Principal Component Analysis (PCA)** is one way to do this.
The relevant data set looks like this:

```
 Phys     Stat
45        49
66        64
...       ...
```

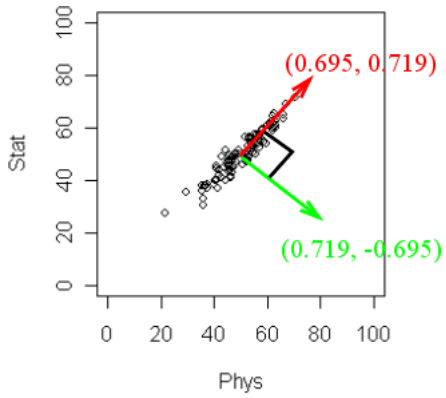
We load this into a `data.frame` called `dat`.

```
pc = princomp(~Stat+Phys,data=dat) #Don't run this
pc$loading                         #Don't run this
```

Notice the somewhat non-intuitive syntax of the `princomp` function. The first argument is a so-called formula object in R (we had encountered this beast in the regression tutorial). In `princomp` the first argument must start with a $\sim$ followed by a list of the variables (separated by plus signs).
Here is part of the output.

```
Loadings:
     Comp.1 Comp.2
Stat  0.695 -0.719
Phys  0.719  0.695
```

The output may not be readily obvious. The next diagram will help.



R has returned two **principal components**. (Two because we have two variables). These are a unit vector at right angles to each other. You may think of PCA as choosing a new coordinate system for the data, the principal components being the unit vectors along the axes. The first principal component gives the direction of the maximum spread of the data. The second gives the direction of maximum spread perpendicular to the first direction. These two directions are packed inside the matrix `pc$loadings`. Each column gives a direction. The direction of maximum spread (the first principal component) is in the first column, the next principal component in the second and so on. Notice, however, that the second principal direction as computed by R is actually negative of that shown in the diagram. This does not matter, as here direction means just a slope, not the orientation.
There is yet more information. Type

```
pc #Don't run this
```

to learn the amount of spread of the data along the chosen directions. Part of the output is:

```
Standard deviations:
   Comp.1    Comp.2
12.398827  1.983152
```

Here the spread along the first direction is about 12.40. while that along the second is much smaller, just 1.98. These numbers are often not of main importance, it is their relative magnitude that matters.
To see all the stuff that is neatly tucked inside `pc` we can type

```
names(pc) #Don't run this
```

We shall not go into all these here. But one thing deserves mention: `scores`. These are the projections of the data points along the principal components. We have already mentioned that the principal components may be viewed as a new reference frame. Then the scores are the coordinates of the points w.r.t. this frame.

```
pc$scores #Don't run this
```

Here is how scores of the first few cases look:

```
     Comp.1       Comp.2
  -3.4863835 -2.916341631
  21.7134877  0.376570202
   3.4934991  2.823881335
      ...          ...
```

Returning to our original question, we may say that the `Comp.1` scores give the most discriminatory linear combination of the two sets of marks.

1.1 PCA of Webbink globular clusters

Most statisticians consider PCA a tool for reducing dimension of data without losing too much information. So PCA is commonly used with high dimensional data sets.
First let us load such a data set.

```
# Overview of Webbink globular cluster properties
GC = read.table(
"http://astrostatistics.psu.edu/MSMA/datasets/Webbink_GC_tab.txt"
,header=T,fill=T)
GC=na.omit(GC)
dim(GC)
names(GC)
```

The first 4 columns hold the location information. There's not much sense in running PCA on these. Let's extract the dynamic variables. Then we remove some extremely high values. Finally we standardize them using the `scale` function.

```
GCdyn = GC[,-c(1:4)]
GCdyn = GCdyn[-c(which.max(GCdyn[,4]), which.max(GCdyn[,11])),]
GCdyn1 = scale(GCdyn)
```

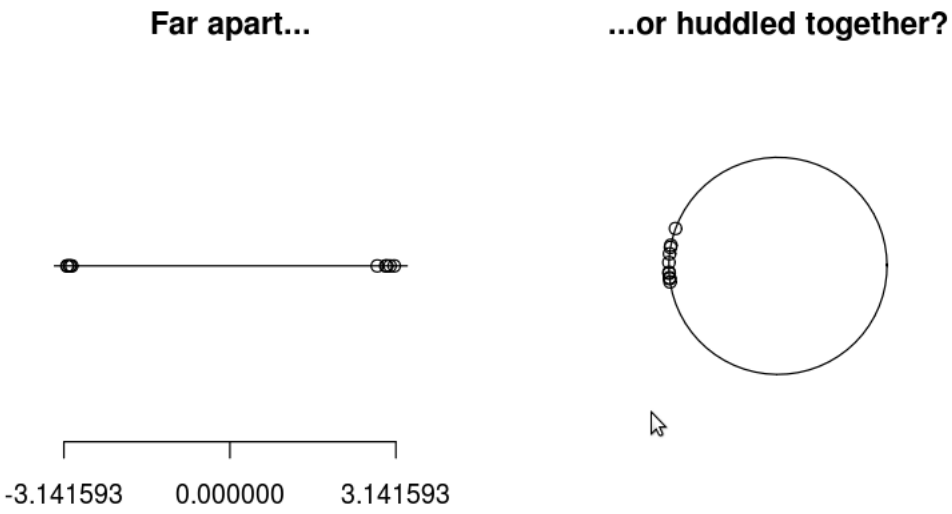
```
PCdyn = princomp(GCdyn1)
plot(PCdyn, main='')
summary(PCdyn)
loadings(PCdyn)
# Add principal component values into data frame
PCdat = data.frame(names=row.names(GCdyn1), GCdyn1, PCdyn$scores[,1:4])
```

2 Directional data

In astronomy we have to often work with angles like RA and declination. While apparently these variables take numerical values just like other variables, there is an important distinction. Consider, for example, the following angles (in radians):

2.79, -3.07, 2.95, -3.02, -3.03, -2.99, 3.11, 2.97, 3.03, -3.08.

We have plotted these points in two ways, first as ordinary numbers, and then as angles. While the first plots seems to suggest that the points are divided into two groups far from each other, the second plot correctly reveals them as a set of closely situated angles.



Standard statistical tools lose their intuitive interpretation for directional data. For example, the standard deviation will be quite high for the above angles, though they are actually quite tightly clustered. Similarly, if all the angles are rotated by a constant amount, the standard deviation will change, though the rotated points are just as tightly clustered as the original points.
Thus the fact that the angles π and $-\pi$ are actually the same needs to be taken care of. This leads to the need of directional statistics. We shall learn about this using the Hipparcos data set.
The first step is to load the data.

```
hip = read.table(
"http://astrostatistics.psu.edu/MSMA/datasets/HIP.dat"
,header=T,fill=T)
dim(hip)
names(hip)
hip = na.omit(hip)
```

Let's take a brief look at the data.

```
summary(hip)
hist(hip[,8], breaks=50)
```

We do not want cases with large `e_plx` values. Also we shall centre the proper motion variables `pmRA` and `pmDE` by subtracting their respective medians.

```
hip =  hip[hip[,'e_Plx']<5,]
pmRA = with(hip, pmRA - median(pmRA))
pmDE = with(hip, pmDE - median(pmDE))
```

Let's make a simple plot.

```
plot(pmRA, pmDE, pch=20, cex=0.6,
     xlab='Proper motion R.A. (mas/yr)',
     ylab='Proper motion Dec. (mas/yr)',main='')
abline(h=0,v=0, lty=2, lwd=2)
```

Next we consider the proper motion values as angles.

```
nstar = length(pmRA)
rad2deg = 360 / (2*pi)

thetaRad = atan2(pmDE, pmRA)
thetaDeg = thetaRad * rad2deg
```

We want to visualize the angles. First a crude attempt (ignoring the directional aspect of the data).

```
hist(thetaDeg, breaks=30, lwd=2,
     xlab='Position angle theta (degrees)',main='')
```

Next, we want to take the directional aspect into account. For this we need a new package called **CircStats**.

```
install.packages('CircStats')
library(CircStats)
circ.summary(thetaDeg)
circ.plot(thetaDeg, cex=0.3, pch=20, stack=T, bins=50, shrink=2)
```

The `circ.plot` functions plots the points on a circle. The `stack=T` options prevents the merging of points close to each other. If there are multiple close points, they are "stacked" to form a line jutting radially outward from the circle. If we want a smoothed version then we can use a different package called **circular**.

```
install.packages('circular')
library(circular)
theta = circular(thetaRad)
theta_kde = density(theta, bw=100, type='K') #1/100 is the
                                             #smoothing param
plot(theta_kde, main='', xlab='', ylab='', lwd=2)
```

Let's look at some summary statistics.

```
circ.summary(theta)
circ.disp(theta)[4]
```

A standard question that we ask ourselves is: Are the angles uniform in all directions? A number of tests are available for this.

2.1 Raleigh test

This test is based on the fact that if the angles are equally scattered in all directions, then the resultant should be close to zero. Here we are treating an angle θ as a unit vector $(\cos\theta, \sin\theta)$, and adding them to find the resultant. Of course, the resultant could be close to zero even otherwise, e.g., if half of the angles face east while the rest face west. So Rayleigh test basically tests uniformity as opposed to "too many angles in one direction".

```
r.test(theta)
```

2.2 Watson's test

This is another test based on a similar idea: if the resultant is too large, then most possibly the directions are not uniform. Watson's test provides an approximate threshold for the length of the resultant to be considered "too large".

```
watson(thetaRad)
```

2.3 Kuiper's test

This is a more sophisticated test that is based on the Kolmogorov-Smirnov idea.

```
kuiper(thetaRad)
```

2.4 Rao's spacing test

This test is based on the fact that if the angles are uniformly scattered in all directions, then the arc lengths between any two of the angles should have a particular type of distribution. To see this let us simulate 100 angles randomly from $[0, 2\pi)$.

```
angles = runif(100,0,2*pi)
circ.plot(angles * rad2deg, pch=20, stack=T, bins=50, shrink=2)
```

Now look at the arc lengths.

```
sorted.angles = sort(angles)
arcLen = diff(sorted.angles)
arcLen = c(arcLen, 2*pi-range(angles))
hist(arcLen,breaks=50)
```

Rao's spacing test compares the observed arc lengths with this distribution.

```
rao.spacing(thetaRad)
```

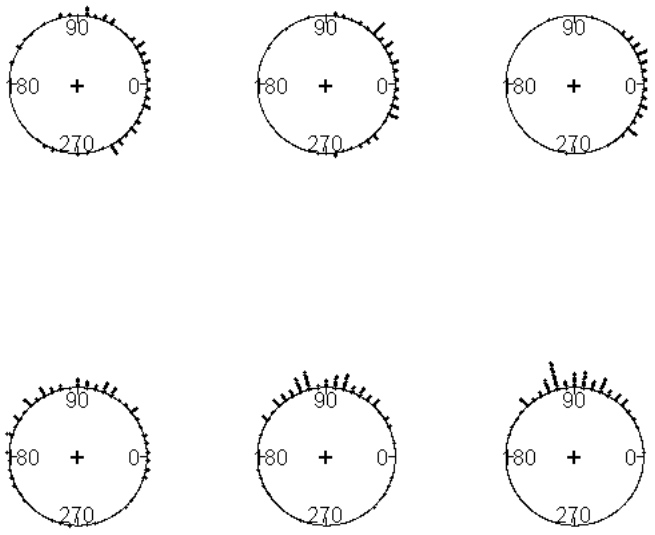
2.5 von Mises distribution

So far we are talking about only the uniform distribution of angles. This is a special case of the **von Mises distribution**, which also allows the angles to crowd more towards a certain *mean direction*. This distribution takes two parameters: the mean direction and κ, which measures the concentration of the angles around the mean direction.
To get a feel for the von Mises distribution, let us simulate some data from this distribution.

```
par(mfrow=c(2,3))
angles = rvm(100, 0, 1)
```

```
circ.plot(angles, pch=20, stack=T, bins=50)
angles = rvm(100, 0, 2)
circ.plot(angles, pch=20, stack=T, bins=50)
angles = rvm(100, 0, 4)
circ.plot(angles, pch=20, stack=T, bins=50)
angles = rvm(100, pi/2, 1)
circ.plot(angles, pch=20, stack=T, bins=50)
angles = rvm(100, pi/2, 2)
circ.plot(angles, pch=20, stack=T, bins=50)
angles = rvm(100, pi/2, 4)
circ.plot(angles, pch=20, stack=T, bins=50)
par(mfrow=c(1,1))
```

This produces a picture like this:



Exercise 1: If `rvm` is the function to generate random numbers from the von Mises distribution, what should the name of the R function computing the von Mises density? Use this function to make plot of the density for $\mu = 0, \kappa = 1$. Does the shape look familiar? ■

Given a data set we can estimate the parameters of the best fitting von Mises distribution.

```
est.kappa(thetaRad, bias=T)
vm.ml(theta,bias=T)
```

Can we test if a given data set is indeed from a von Mises distribution? Yes, we can use Watson's test for this purpose.

```
watson(theta,dist='vm')
```

3 Spatial statistics

3.1 Using package `spdep`

Looking for spatial patterns among the stars has long tradition starting with the formation of constellations. A typical approach is to join two nearby stars with an imaginary straight line. We shall use the `spdep` package to implement a modern version of the same idea.

```
install.packages('spdep')
library(spdep)
```

Let us create a mock firmament and populate it with stars.

```
constel = function() {
  plot(1:10,ty='n')
  stars = locator()
  starCoords = cbind(stars$x,stars$y)
  knn = knearneigh(starCoords)
  nb = knn2nb(knn)
  plot.nb(nb,starCoords,add=T)
}
```

This function opens a blank plot window (with x and y both ranging from 1 to 10) and let's you click on points to create stars. Then it runs the function `knearneigh` to find the k nearest neighbors for each point. By default $k = 1$. Our intention is to join each star and its nearest neighbor with a straight line. Unfortunately, the value returned by `knearneigh` is not directly suited for

plotting. So we have to filter it through the utility function knn2nb to produce the same result in a somewhat different structure suitable for plotting. Then we plot the lines.

```
constel()
```

Click on the empty plot window (try to have some close clusters). The function is crude and will not show the clicked points until you right-click and select "Stop". Then it will show the "constellations".
Now that we have finished playing with our toy sky, it's time to turn our gaze to the real one.

```
shap=read.table(
'http://astrostatistics.psu.edu/MSMA/datasets/Shapley_galaxy.dat'
,header=T,fill=T)
shap = na.omit(shap)
mat = as.matrix(shap[,1:2])
nbr = knearneigh(mat)
tmp1 = knn2nb(nbr)
plot.nb(tmp1,mat)
```

Well, this produces too much clutter. Recall that we had created a pickRect function earlier. We shall now use it to pick a less crowded portion of the data.

```
pickRect = function(coords) {
  plot(coords)
  cat("Pick a rectangle.\n")
  cat("First click on its bottom left corner.\n")
  bl = locator(1)
  cat("Now click on its top right corner.\n")
  tr = locator(1)
  rect(bl$x,bl$y,tr$x,tr$y)
  ind = coords[,1] > bl$x &
        coords[,1] < tr$x &
        coords[,2] > bl$y &
        coords[,2] < tr$y
  ind
}
submat = mat[pickRect(mat),]

subNbr = knearneigh(submat)
tmp1 = knn2nb(subNbr)
```

```
plot.nb(tmp1,submat)
```

3.1.1 Spatial autocorrelation

We shall next learn about spatial autocorrelation which measures if neighboring stars behave similarly or not (w.r.t. some variable of interest). For this we have to specify

1. the coordinates of each star
2. the values of the variable of interest for each star
3. how many neighbors we want to consider for each star

The output will be a number that measures the degree of association. There are a number of such measures. We shall discuss two: Moran's I and Geary's C.

$$I = (n \sum_i \sum_j w_{ij}(x_i - \bar{x})(x_j - \overline{x}))/(S_0 \sum_i (x_i - \bar{x})^2).$$

Here

- x_i is the value of the variable of interest for star i,
- w_{ij} measures the "degree of neighborhood" between stars i and j. In the simplest version we can take w_{ij} as 1 or 0 according as stars i and j are neighbors or not.
- S_0 is the sum of all w_{ij}'s.

```
nbr = knearneigh(mat)
tmp = knn2nb(nbr)
wt = nb2listw(tmp,style='B') #B for basic, ie, 0-1 weights
```

```
moran(mat[,1], wt, n=length(mat[,1]), S0=Szero(wt))
```

Well, this function requires redundant parameters. We can wrap it up as

```
my.moran = function(x,wt) moran(x,wt,n=length(x),S0=Szero(wt))
```

Then we can simply call

```
my.moran(mat[,1],wt)
```

Moran's I turns out to be pretty close to 1 in this case. This means that stars are rather close together in terms of RA. How do we quantify this "closeness to 1"? We shall show two methods.
The first method compares the observed value with the values obtained by randomization. No randomization is performed in reality, the expressions are all computed analytically *under the assumption of randomization.*

```
moran.test(mat[,1], wt)
```

The second approach actually performs the randomization:

```
moran.mc(mat[,1], wt, nsim=10000) #be patient!
```

Here is a related index called Geary's C.

```
geary(mat[,1], wt, n=length(mat[,1]), n1=length(mat[,1])-1,
   S0=Szero(wt))
```

As you can guess this may be conveniently abbreviated to

```
my.geary = function(x,wt) {
   n=length(x)
   geary(x,wt,n=n,n1=n-1,S0=Szero(wt))
}
```

Testing it is very similar to testing Moran's I.

```
geary.test(mat[,1], wt)
```

3.2 Using package `gstat`

A different way to discern spatial patterns is via a **variogram**. Roughly speaking it measures the variability of the values as a function of the distances. Imagine a piece of hilly terrain, where the height of point (x, y) is $z(x, y)$. Given two

random points P, Q a specific distance h apart, we want to know the average value of $|z(P) - z(Q)|^2$. If we plot its half against h, we get a (semi-)variogram. Note that this technique loses information about the direction, and hence is sometimes called an *isotropic* variogram.

```
install.packages('gstat')
library(gstat)
```

We continue to work with the same data set.

```
variog1 = variogram(Vel~1, locations=~R.A.+Dec., data=shap)
plot(variog1)
```

If you do not like the way the first two parameters are specified you may use a wrapper function

```
my.variogram = function(x,y,z,data,plot.it=TRUE) {
   fml1 = as.formula(paste(z,'~1'))
   fml2 = as.formula(paste('~',x,'+',y))
   variog = variogram(fml1,locations=fml2,data=data)
   if(plot.it) plot(variog)
   invisible(variog)
}
```

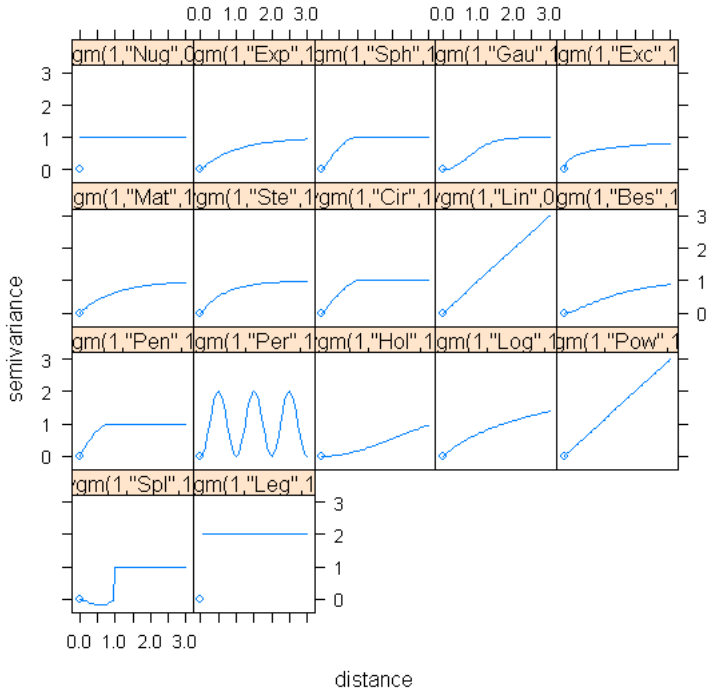
Then you can use

```
my.variogram('R.A.', 'Dec.', 'Vel', data=shap)
```

Next we want to model the variogram. For this we can use a combination of models. A gallery of available models may be seen using the command

```
show.vgms()
```

This produces the following gallery.



We typically choose some of these models, roughly specify the parameters by eyeballing, and then propose the models to the `fit.variogram` function, which polishes our guesses.

```
proposed.model1 = vgm(3e7,"Sph",1,1e7)
fitted.model1 = fit.variogram(variog1,proposed.model1)
plot(variog1,model = fitted.model1,
     col='black', pch=20 ,xlab='Distance (degree)',
     ylab="Semivariance (km/s*km/s)", lwd=2)
```

3.3 Using package `geoR`

First let us load the package.

```
install.packages('geoR')
library(geoR)
```

We are still working with the same data set. But we need to prepare it for the `geoR` package.

```
shap.geo = as.geodata(shap,coords.col=1:2, data.col=4)
```

This identifies the first two columns as the coordinates, and the fourth column as containing the data.
The `points` function gives a useful graphical summary. Unlike the default `points` function, it does not need an existing plot to add to.

```
points.geodata(shap.geo,cex.min=0.2, cex.max=1.0,
               pt.div='quart',
               col='gray')
```

Here is another variant.

```
plot.geodata(shap.geo, breaks=30)
```

This package also allows us to compute and plot variogram.

```
shap.vario = variog(shap.geo, uvec=seq(0, 10, by=0.5))
plot(shap.vario,lwd=2, cex.lab=1.3, cex.axis=1.3, lty=1)
```

However, it takes a different approach to modelling the data. Instead of fitting *ad hoc* structural functions (e.g., linear, spherical etc) it considers the data as the sum of a *trend*, a *Gaussian process*, and a *nugget error*. Then it computes the variogram of the fitted model.

```
shap.GRF1 = likfit(shap.geo, ini.cov.pars=c(4e7,0.2))#Be patient!
summary.likGRF(shap.GRF1)
lines.variomodel(shap.GRF1, lwd=2, lty=2)
```

Well, it is a pretty bad fit!

4 Hints for selected exercises

1. The function is called `dvm`. Use can try

```
f = function(x) dvm(x,mu=0,kappa=1)
curve(f,-pi,pi)
```

The curve is much like a Gaussian curve.