**#################################################**

**Unsupervised learning**

**Class 09 – Image compression with PCA**

**Retriving the information in MDS**

Katarzyna Kopczewska, Associate professor

[kkopczewska@wne.uw.edu.pl](mailto:kkopczewska@wne.uw.edu.pl)

**#################################################**

**# Reading datasets and packages**

setwd("D:/My all/&Wykłady/Wykłady - WNE Unsupervised Learning/01. Clustering/dane ceny")

library(jpeg)

library(factoextra)

library(magick)

library(gridExtra)

library(ggplot2)

**# importing and plotting colour photo – our faculty**

photo<-readJPEG("wne-building.jpg")

plot(1, type="n") # plotting the rasterImage – colour photo

rasterImage(photo, 0.6, 0.6, 1.4, 1.4)

**# in colour photo one gets three matrices – pixel by pixel**

**# each for one component of RGB colour**

**# easy trick to convert to grey scale is to sum up RGB shades**

**# and divide by max value (to scale up to max. 1)**

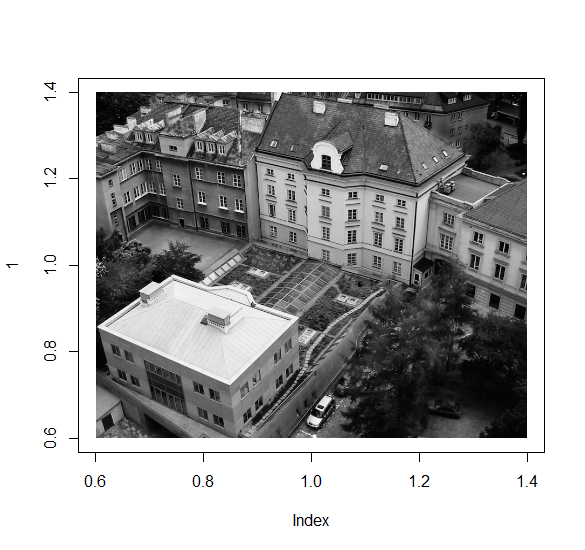
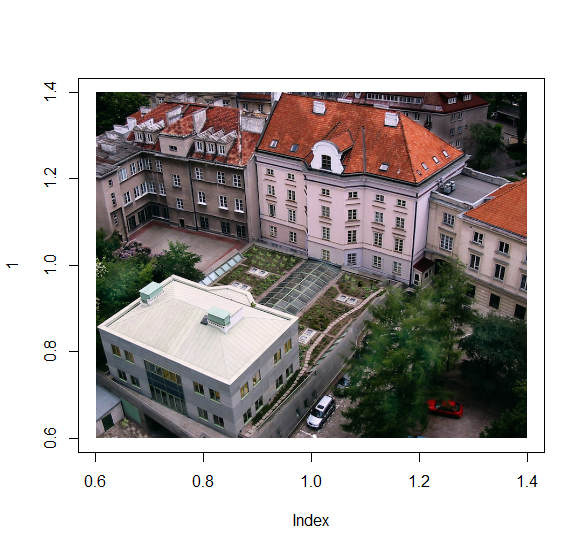
photo.sum<-photo[,,1]+photo[,,2]+photo[,,3] # summing up RGB shades

photo.bw<-photo.sum/max(photo.sum) # dividing by max

plot(1, type="n") # plotting the rasterImage – black & white photo

rasterImage(photo.bw, 0.6, 0.6, 1.4, 1.4)

writeJPEG(photo.bw, "photo\_bw.jpg")



# PCA for pictures is to run individual PCA on each of R, G & B shades.

# This generates eigenvectors of shades -

# first eigen vector explains the majority of shade, next vectors less.

# Trick is to integrate new shades into picture.

# This new value comes from multiplying “x” and “rotation” components of PCA.

# each color scale (R,G,B) gets own matrix and own PCA

r<-photo[,,1] # individual matrix of R colour component

g<-photo[,,2]

b<-photo[,,3]

r.pca<-prcomp(r, center=FALSE, scale.=FALSE) # PCA for R colour component

g.pca<-prcomp(g, center=FALSE, scale.=FALSE)

b.pca<-prcomp(b, center=FALSE, scale.=FALSE)

rgb.pca<-list(r.pca, g.pca, b.pca) # merging all PCA into one object

# let’s see the importance of PC

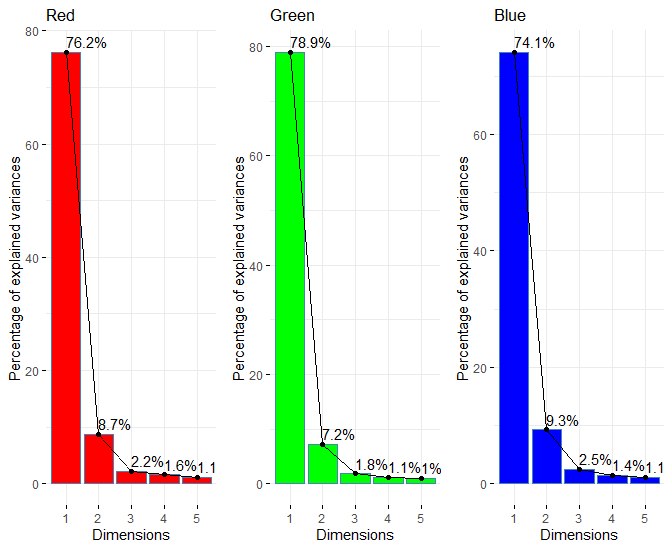
library(gridExtra)

f1<-fviz\_eig(r.pca, main="Red", barfill="red", ncp=5, addlabels=TRUE)

f2<-fviz\_eig(g.pca, main="Green", barfill="green", ncp=5, addlabels=TRUE)

f3<-fviz\_eig(b.pca, main="Blue", barfill="blue", ncp=5, addlabels=TRUE)

grid.arrange(f1, f2, f3, ncol=3)



# the loop for different photos

# number of pixels in photo defines the number of principal components (PC)

# In code below we make 9 photos, with min. 3 & max=n PC

# we multiply x \* rotation and apply it on existing pixels grid

# finally we save all photos into working directory and as objects

vec<-seq.int(3, round(nrow(photo)), length.out=9)

for(i in vec){

photo.pca<-sapply(rgb.pca, function(j) {

new.RGB<-j$x[,1:i] %\*% t(j$rotation[,1:i])}, simplify="array")

assign(paste("photo\_", round(i,0), sep=""), photo.pca) # saving as object

writeJPEG(photo.pca, paste("photo\_", round(i,0), "\_princ\_comp.jpg", sep=""))

}

# easy plotting of new photo

plot(image\_read(photo\_3))

round(vec,0)

[1] 3 84 164 244 325 406 486 566 647 # number of PC

# collective plotting 9 mages – mechanism of retrieving names of saved files

par(mfrow=c(3,3))

par(mar=c(1,1,1,1))

plot(image\_read(get(paste("photo\_", round(vec[1],0), sep=""))))

plot(image\_read(get(paste("photo\_", round(vec[2],0), sep=""))))

plot(image\_read(get(paste("photo\_", round(vec[3],0), sep=""))))

plot(image\_read(get(paste("photo\_", round(vec[4],0), sep=""))))

plot(image\_read(get(paste("photo\_", round(vec[5],0), sep=""))))

plot(image\_read(get(paste("photo\_", round(vec[6],0), sep=""))))

plot(image\_read(get(paste("photo\_", round(vec[7],0), sep=""))))

plot(image\_read(get(paste("photo\_", round(vec[8],0), sep=""))))

plot(image\_read(get(paste("photo\_", round(vec[9],0), sep=""))))



**# another option of doing the same**

library(abind)

pp=10 # how many principal components should be included

photo.pca2<-abind(r.pca$x[,1:pp] %\*% t(r.pca$rotation[,1:pp]),

g.pca$x[,1:pp] %\*% t(g.pca$rotation[,1:pp]),

b.pca$x[,1:pp] %\*% t(b.pca$rotation[,1:pp]),

along=3)

plot(image\_read(photo.pca2))

**# let’s check how the size of photo decreases under this PS compression**

# file.info() works for files not objects – so one should access files in WD

# paste() with option sep="" is equivalent to paste0() with no options

library(Metrics)

sizes<-matrix(0, nrow=9, ncol=4)

colnames(sizes)<-c("Number of PC", "Photo size", "Compression ratio", "MSE-Mean Squared Error")

sizes[,1]<-round(vec,0)

for(i in 1:9) {

path<-paste("photo\_", round(vec[i],0), "\_princ\_comp.jpg", sep="")

sizes[i,2]<-file.info(path)$size

photo\_mse<-readJPEG(path)

sizes[i,4]<-mse(photo, photo\_mse) # from Metrics::

}

sizes[,3]<-round(as.numeric(sizes[,2])/as.numeric(sizes[9,2]),3)

sizes

# for nicer display of table

library(knitr)

kable(sizes)

| Number of PC| Photo size| Compression ratio| MSE-Mean Squared Error|

|------------:|----------:|-----------------:|----------------------:|

| 3| 40979| 0.372| 0.0283008|

| 84| 101382| 0.921| 0.0042965|

| 164| 110175| 1.001| 0.0019264|

| 244| 111276| 1.011| 0.0010919|

| 325| 110020| 1.000| 0.0007884|

| 406| 109742| 0.997| 0.0006916|

| 486| 109960| 0.999| 0.0006557|

| 566| 110001| 0.999| 0.0006434|

| 647| 110064| 1.000| 0.0006388|

**# how to make own palette from image**

library(imgpalr)

photo<-readJPEG("wne-building.jpg")

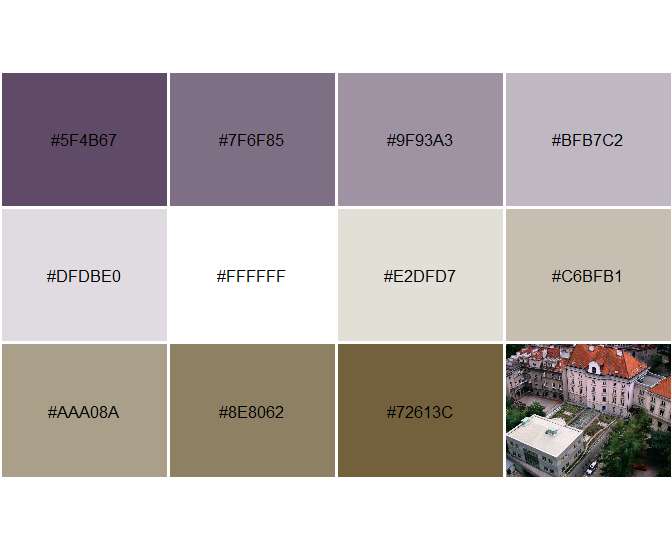
col1<-image\_pal("wne-building.jpg", n=8, type="div", saturation=c(0.75, 1), brightness=c(0.75, 1), plot=TRUE)

col2<-image\_pal("wne-building.jpg", n=11, type="seq", k=2, saturation=c(0.5, 1), brightness=c(0.25, 1), seq\_by="hsv", plot=TRUE)

col3<-image\_pal("wne-building.jpg", n=11, type="div", k=2, saturation=c(0.5, 1), brightness=c(0.25, 1), plot=TRUE)

col4<-image\_pal("wne-building.jpg", n=11, k=30, type="div", saturation=c(0.75, 1), brightness=c(0.75, 1), bw=c(0.1, 0.9), plot=TRUE)



**# website materials for image compression**

**# As extra material see papers of my students from previous years:**

<https://www.r-bloggers.com/2019/01/image-compression-with-pca-in-r/>

<https://rpubs.com/sgroszkiewicz/pca>

**# other nice resources**

<https://github.com/wee-analyze/machine-learning-kmeans-image-compression> - k-means compression

<https://towardsdatascience.com/image-compression-using-k-means-clustering-aa0c91bb0eeb> - k-means compression

<https://rpubs.com/JanpuHou/469414> - eigenfaces (PCA for face recognition)

<https://rpubs.com/dherrero12/543854> - eigenfaces (PCA for face recognition)

**Extensions for Dimensions reductions**

1. **PCA for mixed data – operates on quantitative and qualitative data - use PCAmixdata:: package** <https://cran.r-project.org/web/packages/PCAmixdata/vignettes/PCAmixdata.html#pcarot>, <https://arxiv.org/pdf/1411.4911.pdf>
2. **Correspondence Analysis – for categorical variables – use FactoMineR:: package**

<https://cran.r-project.org/web/packages/FactoMineR/index.html>

**##############################################################################**

**# How well data can be retrieved by MDS?**

**# let’s generate random data for a ring (in 4 quarters separately)**

x1<-runif(1000, 0,1)

y1<-runif(1000, 0,1)

x2<-runif(1000, 0,1)

y2<-runif(1000, -1,0)

x3<-runif(1000, -1,0)

y3<-runif(1000, -1,0)

x3<-runif(1000, -1,0)

x4<-runif(1000, -1,0)

y4<-runif(1000, 0,1)

r1<-(x1^2+y1^2)^0.5

r2<-(x2^2+y2^2)^0.5

r3<-(x3^2+y3^2)^0.5

r4<-(x4^2+y4^2)^0.5

p1<-which(r1>0.95 & r1<1.05)

p2<-which(r2>0.95 & r2<1.05)

p3<-which(r3>0.95 & r3<1.05)

p4<-which(r4>0.95 & r4<1.05)

v1<-cbind(x1[p1], y1[p1])

v2<-cbind(x2[p2], y2[p2])

v3<-cbind(x3[p3], y3[p3])

v4<-cbind(x4[p4], y4[p4])

**# let’s plot the ring**

plot(v1[,1], v1[,2], ylim=c(-1,1), xlim=c(-1,1))

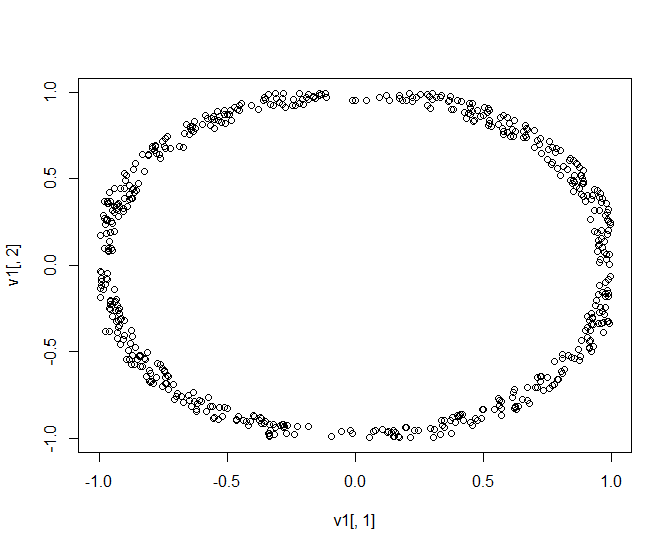
points(v2[,1], v2[,2])

points(v3[,1], v3[,2])

points(v4[,1], v4[,2])

**d<-rbind(v1,v2,v3,v4)**

plot(d[,1], d[,2], ylim=c(-1,1), xlim=c(-1,1))



**# MDS on the data**

dist.reg<-dist(d) # as a main input we need distance between units

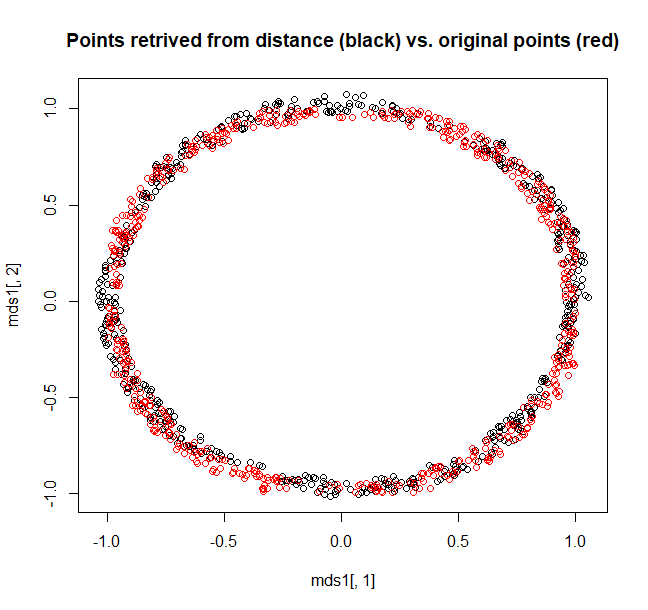
mds1<-cmdscale(dist.reg, k=2)

summary(mds1)

plot(mds1[,1], mds1[,2]) # retrieved data

points(d, col="red") # empirical data

title(main="Points retrived from distance (black) vs. original points (red)")



**# let’s check the impact of distance metric on quality of MDS**

plot(d[,1], d[,2], ylim=c(-2,2), xlim=c(-2,2))

dist.reg<-**dist**(d, method="euclidean")

mds1<-**cmdscale**(dist.reg, k=2)

points(mds1[,1], mds1[,2], col="red")

dist.reg<-**dist**(d, method="manhattan")

mds2<-**cmdscale**(dist.reg, k=2)

points(mds2[,1], mds2[,2], col="blue")

dist.reg<-**dist**(d, method="canberra")

mds3<-**cmdscale**(dist.reg, k=2)

points(mds3[,1], mds3[,2], col="green")

dist.reg<-**dist**(d, method="maximum")

mds4<-**cmdscale**(dist.reg, k=2)

points(mds4[,1], mds4[,2], col="yellow")

dist.reg<-**dist**(d, method="minkowski", p=0.7)

mds5<-**cmdscale**(dist.reg, k=2)

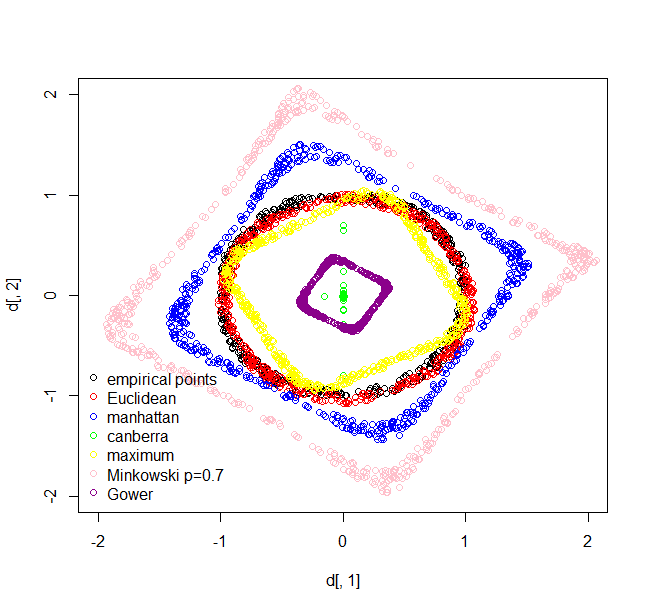
points(mds5[,1], mds5[,2], col="pink")

dist.gower<-gower.dist(d) # library(StatMatch)

mds6<-**cmdscale**(dist.gower, k=2)

points(mds6[,1], mds6[,2], col="darkmagenta")

legend("bottomleft", col=c("black", "red", "blue", "green", "yellow", "pink", "darkmagenta"), c("empirical points", "Euclidean", "manhattan", "canberra", "maximum", "Minkowski p=0.7", "Gower"), pch=21, bty="n")



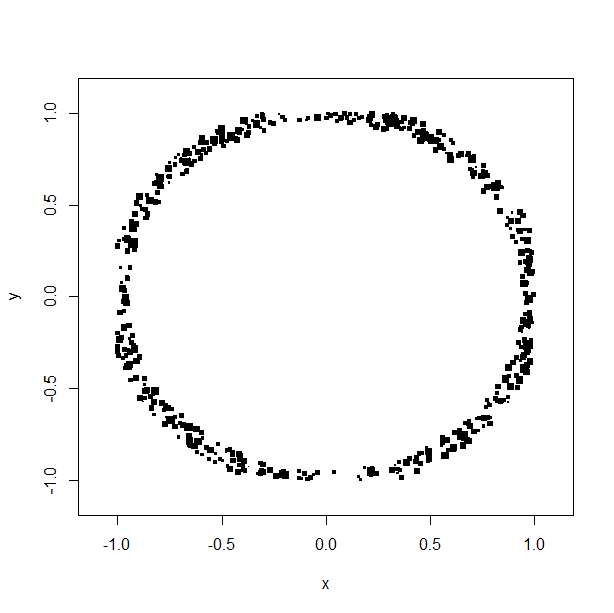
**# let’s generate *z* value for points (*z* as value of dot)**

z<-as.data.frame(rnorm(dim(d)[1], 10,3))

d<-cbind(d,z)

colnames(d)<-c("x","y","z")

plot(d[,1:2], cex=d$z/2, pch=".", ylim=c(-1.1, 1.1), xlim=c(-1.1, 1.1))



**# MDS on the xyz data**

dist.reg<-dist(d) # distance but for three variables (x,y,z)

mds1<-cmdscale(dist.reg, k=2)

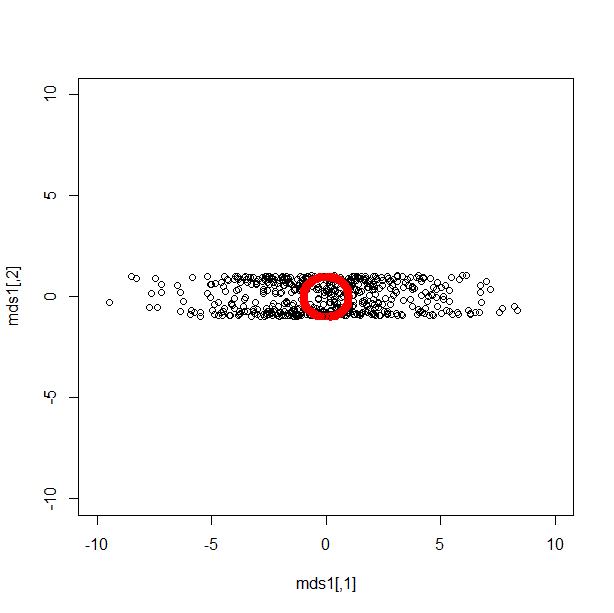
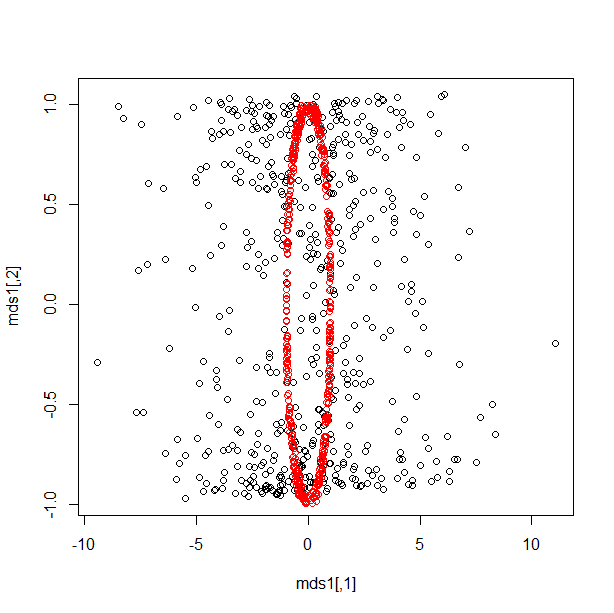
summary(mds1)

plot(mds1[,1], mds1[,2])

points(d[,1:2], col="red") # empirical data

plot(mds1[,1], mds1[,2], ylim=c(-10, 10), xlim=c(-10, 10)) # retrieved data

points(d[,1:2], col="red") # empirical data



**#############################################################################**

**### TASK ###**

Try to generate (x,y) numbers which build a triangle (any). Add vector of features (z). Check if pattern only can be well retrived with MDS. What it changes when z added?

**##############################################################################**