HDS 8. Principal component analysis (PCA)

Matti Pirinen, University of Helsinki

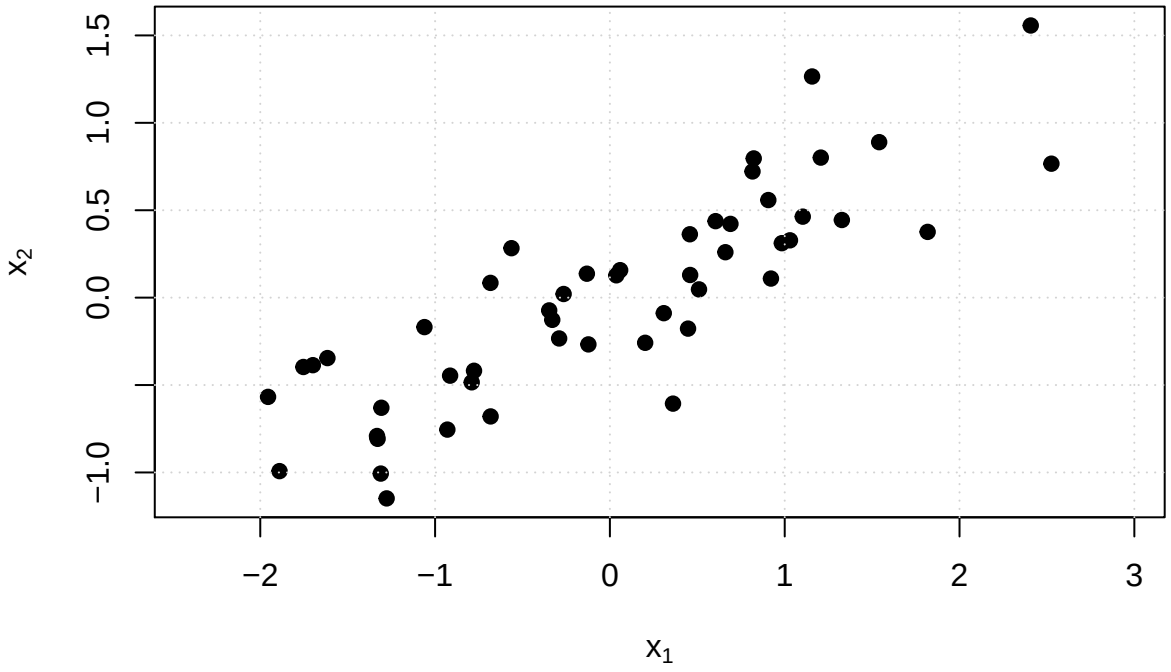
28.11.2019

Now we move from variable selection to dimension reduction. The goal is to represent high dimensional data in a useful way in a low dimensional subspace, for example, to visualize the data in human interpretable ways or to pre-process the data for downstream analyses that benefit from reduced data size. The key question is what is a *useful* way to reduce dimension. Intuitively, given a target dimension, we want to find a mapping from the original data to the target space by keeping as much of the original information as possible. PCA (first defined in 1901 by K. Pearson) is one of the most widely used method for this purpose, and one with a clear statistical motivation and linear algebraic implementation, so we will start from it.

Example in 2 dimensions

Let's consider a set of n=50 points on 2D space (p=2):

```
set.seed(18)
n = 50
x.1 = rnorm(n)
x.2 = 0.4*x.1 + rnorm(n, 0, 0.4)
X = cbind(x.1 - mean(x.1), x.2 - mean(x.2)) #Always mean-center for PCA
plot(X, asp = 1, pch = 19, xlab = expression(x[1]), ylab = expression(x[2]))
grid()
```




```
var(X) #what is variance in x and y direction? and what is the covariance?
```

```
##           [,1]      [,2]
## [1,] 1.2526074 0.5590945
## [2,] 0.5590945 0.3478777
```

The variables are quite correlated, so they are telling partly a redundant story. Do we really need 2 dimensions to represent these data? Or could 1 dimension be enough? How could we put these points to one dimension in a useful way that would preserve main properties of their relationships?

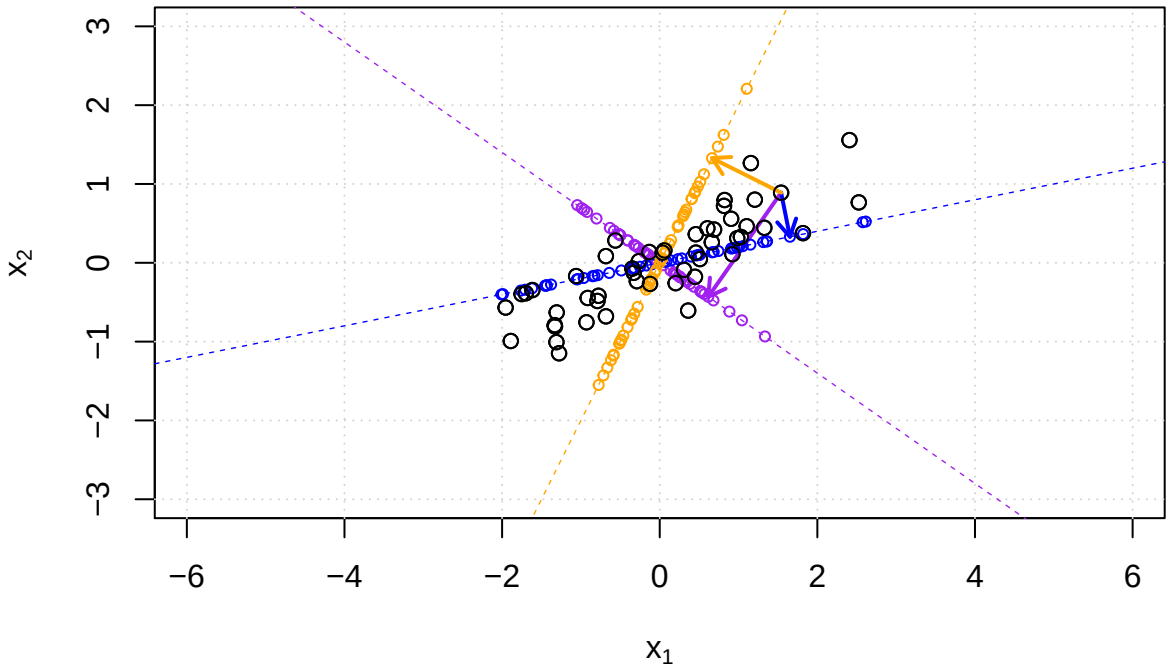
Let's draw some lines through the origin and project the data on those lines. (Since data are mean centered, the origin is at the center of the data set.) Then we will have 1-dimensional representations of the data. The unit vector of a line that passes through the origin with slope b is $\boldsymbol{u} = (1, b)^T/\sqrt{1+b^2}$. The projection of a point $\boldsymbol{x} = (x_1, x_2)^T$ on that line is

$$(\boldsymbol{u}^T\boldsymbol{x})\boldsymbol{u} = \left(\frac{(1,b)\cdot(x_1,x_2)}{\sqrt{1+b^2}}\right)\frac{(1,b)^T}{\sqrt{1+b^2}} = \left(\frac{x_1+bx_2}{1+b^2}\right)(1,b)^T$$

And the coordinate of the point along the 1-dimensional line is

$$(\boldsymbol{u}^T\boldsymbol{x}) = \left(\frac{x_1+bx_2}{\sqrt{1+b^2}}\right).$$

```
plot(X, pch = 1, xlim = c(-3,3), ylim = c(-3,3), asp = 1,
     xlab = expression(x[1]), ylab = expression(x[2]))
grid()
b = c(-0.7, 0.2, 2) #example lines
cols = c("purple", "blue", "orange")
vars = rep(NA, length(b)) #variances of the projected point on each line
for(ii in 1:length(b)){
  abline(0, b[ii], col = cols[ii], lty = 2, lwd = 0.7)
  coord.on.line = (X[,1] + X[,2]*b[ii]) / sqrt(1 + b[ii]^2) #coordinate along each line
  z = coord.on.line/sqrt(1 + b[ii]^2) #temp variable
  points(z, z*b[ii], col = cols[ii], pch = 1, cex = 0.7) #points on the line
  #Show projection for point jj=10 using arrows from point to lines
  jj = 10
  arrows(X[jj,1], X[jj,2], z[jj], z[jj]*b[ii], code = 2,
         length = 0.1, lwd = 2, col = cols[ii])
  vars[ii] = var(coord.on.line)
  }
points(X, pch = 1) #redraw the original points on top
```



```r
cbind(b, vars) #see what is the variance of the projected points on each line
```

```
##         b      vars
## [1,] -0.7 0.4297552
## [2,]  0.2 1.4328465
## [3,]  2.0 0.9760993
```

Because the projections are orthogonal, we can think that the projections preserve (some) of the original relationships between the points. It seems that line 2 (with slope 0.2) separates the points most from each other as on that line the projected points have the largest empirical variance. In that sense it preserves more information about the original data than the other two lines. Think about the two extremes that would be the useless case where each original point is projected on the same point on a line, whence we lose all the information about the differences between the original data units, and the case of perfect information where the original data was already on a line and therefore we do not lose anything by simply recording the coordinates on that line. In reality we always work with data in between these extremes, and in PCA the criteria of finding the subspace is to choose the line on which the projections preserve the highest possible variance.

What would be the line on which the points had the largest variance? Let's find the unit vector $\boldsymbol{u} = (u_1, u_2)^T$ ($u_1^2 + u_2^2 = 1$) that maximizes the empirical variance of the projections of X on that line. As the columns of $\boldsymbol{X}$ are mean-centered then also the mean of $\boldsymbol{Xu}$ is 0, and variance is proportional to the sum of squares of $\boldsymbol{Xu}$:

$$\widehat{\mathrm{Var}}\,(\boldsymbol{Xu}) \propto \boldsymbol{u}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{u}.$$

Matrix theory (see Rayleigh quotient) says that this quantity is maximized (among all unit vectors $\boldsymbol{u}$) when $\boldsymbol{u}$ is the eigenvector of $\boldsymbol{X}^T\boldsymbol{X}$ corresponding to the largest eigenvalue. Let's see what it gives.

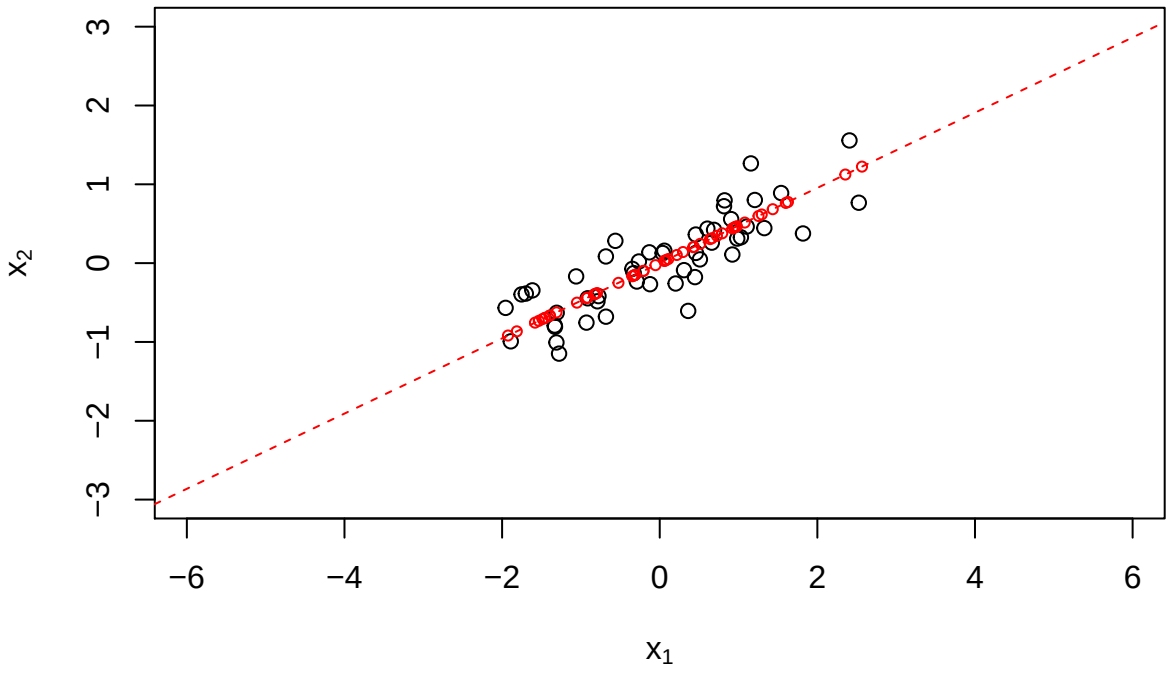
```r
eig = eigen(t(X) %*% X)
eig$values
```

```
## [1] 74.451730  3.972042
```

```r
eig$vectors #are in columns, in decreasing order of eigenvalues
```

```
##            [,1]       [,2]
## [1,] -0.9024967  0.4306969
## [2,] -0.4306969 -0.9024967
```

```r
u = eig$vectors[,1] #corresponds to the largest eigenvalue.
plot(X, pch = 1, xlim = c(-3,3), ylim = c(-3,3), asp = 1,
     xlab = expression(x[1]), ylab = expression(x[2]) ) #plot the data again
abline(0, u[2]/u[1], lty = 2, col = "red") #add max variance line whose unit vector is u
#do projections on the max variance line
coord.on.line = (X %*% u) #coordinates of points in X projected on line u
points( cbind(u[1]*coord.on.line, u[2]*coord.on.line), col = "red", cex = 0.7)
```




```
var(coord.on.line)
```

```
##           [,1]
## [1,] 1.519423
```

Looks plausible. Visually the projection on this lines preserves a lot of the information of the configuration of the points in the 2D plane.

For an interactive demo on the same idea see the top answer here: https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues

Since eigenvectors (of a symmetric matrix like $\boldsymbol{X}^T\boldsymbol{X}$) are orthogonal, they give a new coordinate system on which the data has the maximum variance on the 1st axis and the 2nd axis is the orthogonal direction (in which the data varies less compared to the 1st axis). These are called the principal components (PCs) of the data $\boldsymbol{X}$. The idea is that since PC 1 has been chosen to maximise the variability between the points, it is a good candidate for the 1-dimensional representation of the original data. We may reduce the dimension of the data by only storing the points' coordinates on the 1st PC and ignoring the 2nd PC. We will lose some information, but, as measured by variance, as little as possible given that we will reduce the dimension.

See this for some interactive play with 2D and 3D PCA: http://setosa.io/ev/principal-component-analysis/

PCA more generally

More generally, when $p > 2$, there exist $\min\{p, n\}$ PCs. The kth PC is defined by unit vector $\boldsymbol{u}_k = (u_{k1}, \ldots, u_{kp})^T$, ($||\boldsymbol{u}_k||_2 = 1$), and the **score** (previously called "coordinate") of sample i on PC k is $\text{pc}_k(i) = \boldsymbol{x}_i^T\boldsymbol{u}_k$. The coefficient u_{kj} by which the variable j contributes to PC k is called the **loading** of variable j on PC k. PCs have following properties

- PC 1 is the 1-dimensional subspace of $\mathbb{R}^p$ on which the projection of the rows of data $\boldsymbol{X}$ have the maximum empirical variance.
- PC $k > 1$ is the 1-dimensional subspace of $\mathbb{R}^p$ that is orthogonal to all PCs 1,...,$k-1$ and on which the projection of the rows of data $\boldsymbol{X}$ have the maximum empirical variance among all the subspaces orthogonal to the previous PCs.

Each PC is a 1-dimensional subspace (line) in the p-dimensional space of all variables. Equivalently it is an image of a linear combination of all p variables. Suppose that we have a new sample $\boldsymbol{x}^*$ measured on the

same p variables on which we have constructed PCs earlier with data matrix $\boldsymbol{X}$. We can project this new sample on the existing PCs by the linear mapping $\text{pc}_k(\boldsymbol{x}^*) = \boldsymbol{x}^{*T}\boldsymbol{u}_k = \sum_{j=1}^{p} x_j^* u_{kj}$.

The dimension reduction to $k \leq p$ dimensions with PCA happens by simply ignoring the the PCs after the k largest components are maintained. For visualization of the data we use only 2 or 3 components.

It can be shown that the first k PCs minimize the total sum of squared distances between the original points and their projections on any k dimensional subspace. Linear regression minimizes error between a particular outcome variable y and its prediction made as a linear combination of a set of predictor variables x_j. PCA, instead, treats all variables equally (not separating any predictors from any outcome(s)) and minimizes the total squared distance between its predictions (that are the projections to the first k PCs) and the original points in p dimensional space.

Derivation with more detail

Above it was assumed known from the matrix theory that the maximum variance line problem is solved by the leading eigenvector of the data product matrix $\boldsymbol{X}^T\boldsymbol{X}$. We can derive the PCA solution also by relying on a slightly simpler result on the existence of an eigenvalue decomposition of a symmetric (p x p) matrix $\boldsymbol{S}$. This result says that $\boldsymbol{S}$ can be written as $\boldsymbol{S} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^T$, where (p x p) matrix $\boldsymbol{U}$ is orthonormal (that is, for any columns $i \neq j$ $\boldsymbol{u}_i^T\boldsymbol{u}_j = 0$ and $\boldsymbol{u}_i^T\boldsymbol{u}_i = 1$) and $\boldsymbol{D} = (d_j)_{j=1}^p$ is a diagonal matrix. Note that $\boldsymbol{U}\boldsymbol{U}^T = \boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}$.

Let's set as our goal to find a suitable orthogonal transformation of the p variables (defined by a $p \times p$ matrix $\boldsymbol{Q}$) that removes the covariances between our original variables. In other words, we look for an orthogonal $\boldsymbol{Q}$ that will transform our observed p-dimensional vector $\boldsymbol{x}$, with original covariance matrix $\boldsymbol{S}_x$ (with eigendecomposition $\boldsymbol{S}_x = \boldsymbol{U}_x\boldsymbol{D}_x\boldsymbol{U}_x^T$), into a new p-dim vector $\boldsymbol{y} = \boldsymbol{Q}\boldsymbol{x}$ whose covariance would be diagonal. Since $\boldsymbol{Q}$ is orthogonal, it can be seen as preserving all the structure in the data and it is simply changing the coordinate system on which the variables are measured. Since

$$\text{Var}(\boldsymbol{Q}\boldsymbol{x}) = \boldsymbol{Q}\boldsymbol{S}_x\boldsymbol{Q}^T = \boldsymbol{Q}\boldsymbol{U}_x\boldsymbol{D}_x\boldsymbol{U}_x^T\boldsymbol{Q}^T,$$

we see that by a choice $\boldsymbol{Q} = \boldsymbol{U}_x^T$ we have that $\text{Var}(\boldsymbol{Q}\boldsymbol{x}) = \boldsymbol{D}_x$ is diagonal and hence the transformed variables have zero covariances. Thus, by transforming the variables by the eigenvectors of the empirical covariance matrix of $\boldsymbol{x}$, which is $\widehat{\boldsymbol{S}}_x = \frac{1}{n-1}\boldsymbol{X}^T\boldsymbol{X}$, we will rotate the data into new coordinates where different dimensions do not covary at all. This is the PCA transformation, and the new variable with the largest variance corresponds to the 1st PC etc. (Note that the constant $\frac{1}{n-1}$ does not affect the eigenvectors but does affect the eigenvalues multiplicatively. Thus, for deriving the PCA transformation, it does not matter whether we compute the eigenvectors from $\boldsymbol{X}^T\boldsymbol{X}$ or $\frac{1}{n-1}\boldsymbol{X}^T\boldsymbol{X}$.)

PCA in practice

Let's use PCA on our favorite data set `Boston`. Let's use both the R's standard function `prcomp()` and check that our eigenvalue decomposition method agrees. Note that the help of `prcomp` says that the calculation is done by a singular value decomposition of the (centered and possibly scaled) data matrix, not by using `eigen()` on the covariance matrix.

```
library(MASS)
str(Boston) #remind what's in there
```

```
## 'data.frame':    506 obs. of  14 variables:
##  $ crim   : num  0.00632 0.02731 0.02729 0.03237 0.06905 ...
##  $ zn     : num  18 0 0 0 0 0 12.5 12.5 12.5 12.5 ...
##  $ indus  : num  2.31 7.07 7.07 2.18 2.18 2.18 7.87 7.87 7.87 7.87 ...
##  $ chas   : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ nox    : num  0.538 0.469 0.469 0.458 0.458 0.458 0.524 0.524 0.524 0.524 ...
##  $ rm     : num  6.58 6.42 7.18 7 7.15 ...
##  $ age    : num  65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100 85.9 ...
```

```
##  $ dis    : num  4.09 4.97 4.97 6.06 6.06 ...
##  $ rad    : int  1 2 2 3 3 3 5 5 5 5 ...
##  $ tax    : num  296 242 242 222 222 222 311 311 311 311 ...
##  $ ptratio: num  15.3 17.8 17.8 18.7 18.7 18.7 15.2 15.2 15.2 15.2 ...
##  $ black  : num  397 397 393 395 397 ...
##  $ lstat  : num  4.98 9.14 4.03 2.94 5.33 ...
##  $ medv   : num  24 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 ...
```

```
#We will do PCA WITHOUT lstat included.
keep = c(1:12,14) #keep these columns for PCA
outcome = which(!(1:14 %in% keep)) #leave this out from PCA and think as an outcome variable
#Do PCA with eigendecomposition:
Boston.scaled = scale(as.matrix(Boston)) #always mean center for PCA. scaling not obligatory
eig = eigen(1/(nrow(Boston)-1) * t(Boston.scaled[,keep]) %*% Boston.scaled[,keep])
#Do PCA with prcomp:
prc = prcomp(Boston[,keep], center = T, scale = T)
str(prc)
```

```
## List of 5
##  $ sdev    : num [1:13] 2.44 1.264 1.147 0.931 0.895 ...
##  $ rotation: num [1:13, 1:13] 0.25555 -0.26151 0.35116 -0.00139 0.34458 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:13] "crim" "zn" "indus" "chas" ...
##   .. ..$ : chr [1:13] "PC1" "PC2" "PC3" "PC4" ...
##  $ center  : Named num [1:13] 3.6135 11.3636 11.1368 0.0692 0.5547 ...
##   ..- attr(*, "names")= chr [1:13] "crim" "zn" "indus" "chas" ...
##  $ scale   : Named num [1:13] 8.602 23.322 6.86 0.254 0.116 ...
##   ..- attr(*, "names")= chr [1:13] "crim" "zn" "indus" "chas" ...
##  $ x       : num [1:506, 1:13] -1.85 -1.3 -2.08 -2.52 -2.56 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:506] "1" "2" "3" "4" ...
##   .. ..$ : chr [1:13] "PC1" "PC2" "PC3" "PC4" ...
##  - attr(*, "class")= chr "prcomp"
```

```
#square roots of the eigenvalues = SD of PCs
cbind(prc$sdev[1:3],sqrt(eig$values[1:3]))
```

```
##          [,1]     [,2]
## [1,] 2.439967 2.439967
## [2,] 1.263604 1.263604
## [3,] 1.146852 1.146852
```

```
#matrix of PC loadings in columns = eigenvectors in columns
cbind(prc$rotation[1:3,1], eig$vectors[1:3,1])
```

```
##             [,1]       [,2]
## crim   0.2555546  0.2555546
## zn    -0.2615085 -0.2615085
## indus  0.3511626  0.3511626
```

```
#PC scores of sample 5 on PC 3 (note that sign is arbitrary!)
c(prc$x[5,3], t(eig$vectors[,3]) %*% Boston.scaled[5,keep])
```

```
## [1] 0.03657017 0.03657017
```

If we have a new Boston area with measured values for these 13 variables, how would we project it to these 2D plots? Let's do projection for area 15 (even though it was already on PC analysis) and compare to the correct values.

```
cbind(as.numeric(predict(prc, newdata = Boston[15,keep])), prc$x[15,])
```

```
##             [,1]        [,2]
## PC1  -0.12206133 -0.12206133
## PC2  -0.67659968 -0.67659968
## PC3  -1.23467605 -1.23467605
## PC4   0.15573688  0.15573688
## PC5  -0.53868484 -0.53868484
## PC6   0.60565279  0.60565279
## PC7   0.27826995  0.27826995
## PC8   0.80364271  0.80364271
## PC9  -0.09900318 -0.09900318
## PC10 -0.11513169 -0.11513169
## PC11  0.11364220  0.11364220
## PC12 -0.35107680 -0.35107680
## PC13  0.10032419  0.10032419
```

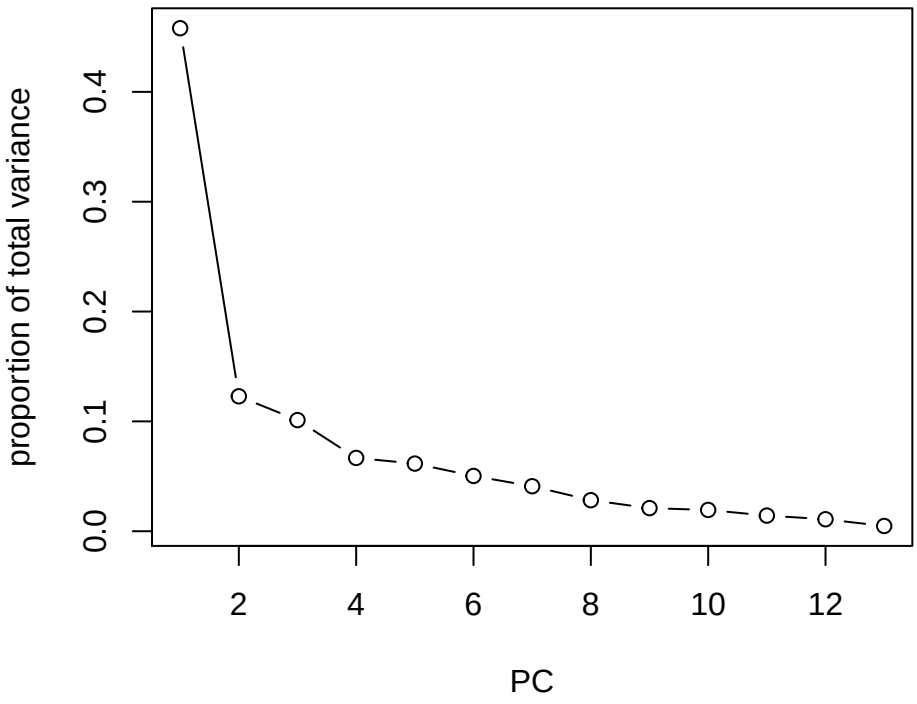
Let's see how large a proportion of total variance PCs cumulatively capture.

```
cumsum(prc$sdev^2)/sum(prc$sdev^2)
```

```
##  [1] 0.4579570 0.5807797 0.6819543 0.7486714 0.8102330 0.8605405 0.9015064
##  [8] 0.9297357 0.9507612 0.9701338 0.9843621 0.9952522 1.0000000
```

1st component alone explains 46% and first 7 components explain 90%. Often the variance explained by components is plotted, and the resulting **scree plot** can be used to evaluate (visually) how many components have clearly more weight than the rest.

```
par(mar = c(5,5,1,0.2))
plot(prc$sdev^2/sum(prc$sdev^2), t = "b", xlab = "PC", ylab = "proportion of total variance")
```



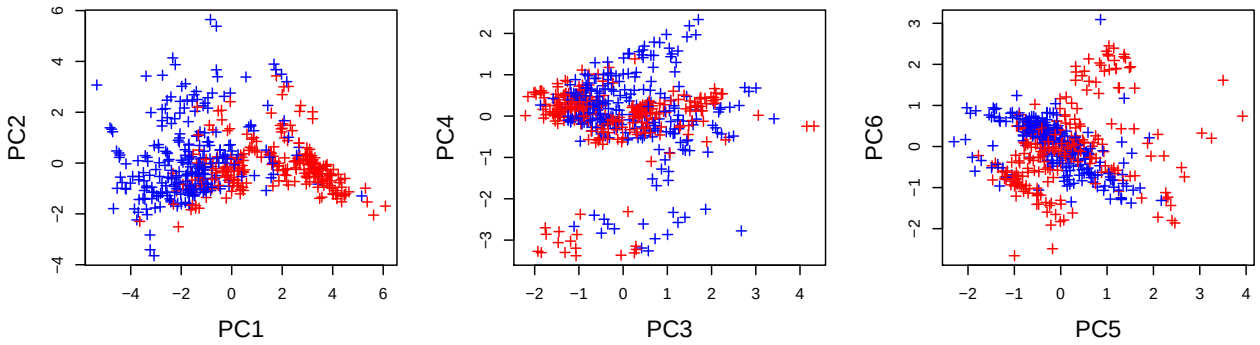
Let's see how the variables make up the leading PC:

```
prc$rotation[,1]
```

```
##         crim           zn        indus         chas          nox
##  0.255554649 -0.261508469  0.351162650 -0.001387083  0.344581670
##           rm          age          dis          rad          tax
## -0.197336951  0.311089512 -0.319149629  0.325450533  0.345858250
##      ptratio        black         medv
##  0.218842202 -0.207663732 -0.264809986
```

Let's visualize first six PCs and color each point according to `lstat` value (higher in red, lower in blue).

```
cols = rep("blue", nrow(Boston))
cols[Boston[,outcome] > median(Boston[,outcome])] = "red"
par(mfrow = c(1,3))
for(ii in seq(1,6,2)){
  plot(prc$x[,ii],prc$x[,ii+1], pch = 3, col = cols, cex.lab = 1.5,
       xlab = paste0("PC",ii), ylab = paste0("PC",ii+1))}
```



We see clear structure (at least in 1st and 3rd plots) where scores of PCs seem to associate with `lstat` value, which was not included in the PCA.

Principal components regression (PCR)

Let's do linear regression of `lstat ~ PC1` and see how it compares to how the individual variables explain `lstat`.

```
summary(lm(Boston[, outcome] ~ prc$x[,1]))
```

```
##
## Call:
## lm(formula = Boston[, outcome] ~ prc$x[, 1])
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -14.2733  -2.6826  -0.5661   2.0816  19.8784
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 12.65306    0.21343   59.28   <2e-16 ***
## prc$x[, 1]   2.16828    0.08756   24.76   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.801 on 504 degrees of freedom
```

```
## Multiple R-squared:  0.5489, Adjusted R-squared:  0.548
## F-statistic: 613.2 on 1 and 504 DF,  p-value: < 2.2e-16
```

```
#How do the individual variables explain lstat?
for(ii in keep){
  print(paste(names(Boston)[ii],":",
              summary(lm(Boston[, outcome] ~ Boston[,ii]))$adj.r.squared))}
```

```
## [1] "crim : 0.20601869232111"
## [1] "zn : 0.16891881326848"
## [1] "indus : 0.363313331937917"
## [1] "chas : 0.000930012813023517"
## [1] "nox : 0.347846506084054"
## [1] "rm : 0.375524008489809"
## [1] "age : 0.361547440694417"
## [1] "dis : 0.245510817911551"
## [1] "rad : 0.237294251952689"
## [1] "tax : 0.294531865698533"
## [1] "ptratio : 0.138202621403982"
## [1] "black : 0.132301404549657"
## [1] "medv : 0.543241825954707"
```

PC 1 explains similarly `lstat` as `medv` alone, and much more than any one variable alone.

When many predictors in a regression model are highly correlated, one approach for a robust regression analysis is first to compute the leading PCs out of the correlated predictors and then do the regression on some of the leading PCs instead. This is called *principal components regression (PCR)*. For example, when regressing the outcome variable y on PC1 and PC2, the model is

$$\begin{aligned} y_i &= \mu + \mathrm{pc}_1(i)\beta_1 + \mathrm{pc}_2(i)\beta_2 + \varepsilon \\ &= \mu + \beta_1 \sum_{j=1}^{p} x_{ij}u_{1j} + \beta_2 \sum_{j=1}^{p} x_{ij}u_{2j} + \varepsilon \end{aligned}$$

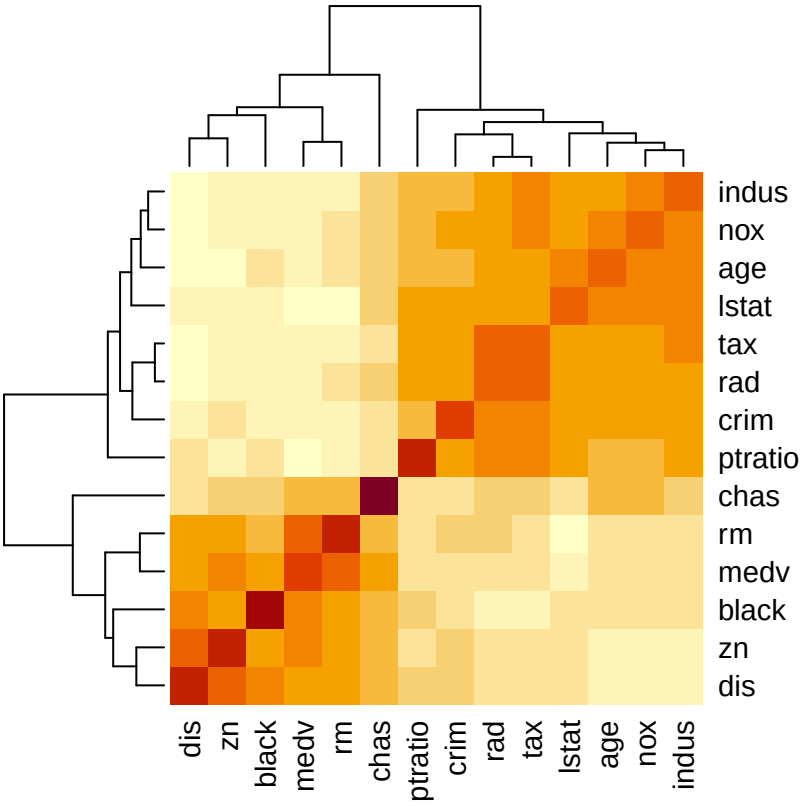
Thus, the model has only two coefficients, but still each of the p original predictors x_j is contributing to the model by $x_{ij}(\beta_1 u_{1j} + \beta_2 u_{2j})$. The idea is that by focusing on leading PCs, we can capture more of the main structure of the predictors than with equal number of individual predictors, but we will reduce the variance compared to the standard linear model with p coefficients that is prone to overfit when p is high.

The advantage of PCR is that the PCs are uncorrelated with each other, and still capture a large part of the joint information of the original correlated predictors. However, the interpretation of the regression coefficient is not anymore given in terms of the original predictors, which is a problem in cases where the original predictors had a clear meaning for the analyst, but PCs don't.

PCA on samples (rather than on variables)

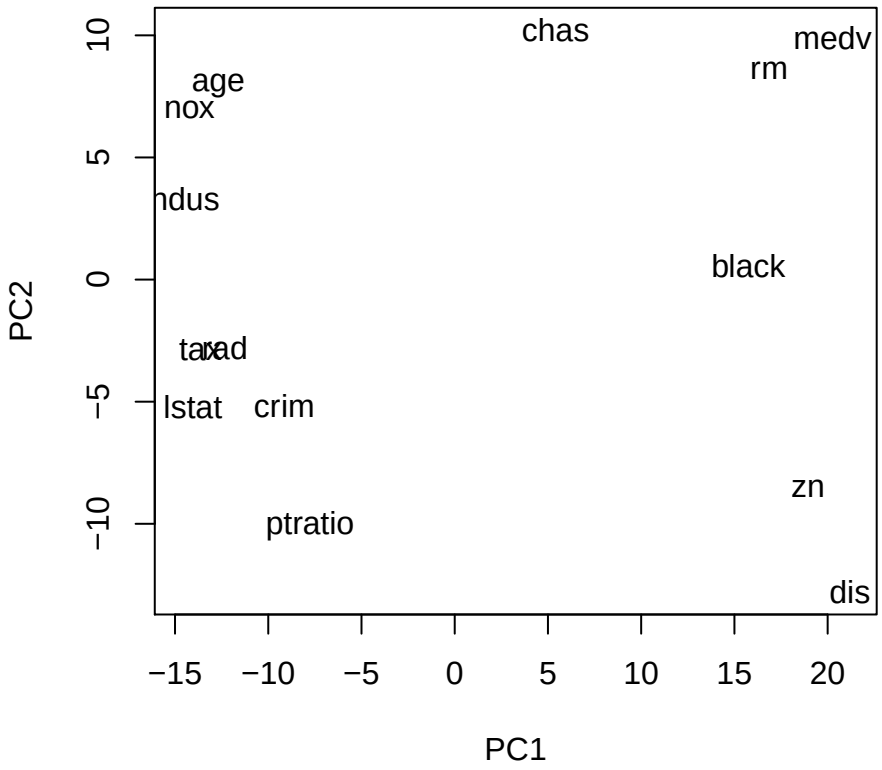
Let's then use PCA the other way around. We now consider the $p = 14$ variables as the "samples" that have been measured on $n = 506$ "variables" which are the Boston areas. We want to find the 2-dimensional description of the relationships between these p variables. Let's first show them with a heatmap.

```
heatmap(cor(Boston))
```



It seems that there are two main groups. Let's see what the leading PC says:

```
prc = prcomp(t(Boston.scaled))
plot(prc$x[,1], prc$x[,2], col = "white", xlab = "PC1", ylab = "PC2")
text(prc$x[,1], prc$x[,2], labels = names(Boston))
```



So the leading PC also divides the variables into the same two main groups as the correlation matrix.

In general, the correlation matrix of p variables has $p(p-1)/2$ unique elements, whereas their first k PCs have together $p \cdot k$ elements. Already for p in hundreds there is a considerable difference in favor of PCs (with small k, say $k \leq 10$) in the amount of data reduction, and this difference becomes crucial in applications with p in thousands.

See examples of PCA in slides HDS8_slides.pdf

Read ISL section 10.2 PCA

- What does PCA aim to do?
- What are *loadings* and what are *scores*?
- PCA solution minimizes something and maximizes something else. What are these quantities?
- Should we scale variables before PCA?

Tutorial on PCA by Jon Shlens

https://arxiv.org/pdf/1404.1100.pdf

This is an excellently written tutorial, but unfortunately it uses the rows and columns of data matrix the other way as we are using so the data matrix is given as $p \times n$ matrix, and p (the number of variables) is called m.

Singular value decomposition (SVD)

The SVD is the fundamental decomposition of a matrix $\boldsymbol{X}$ (of any size $n \times p$) into three components:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T, \text{ where}$$

- $\boldsymbol{U}$ is an $n \times n$ orthonormal matrix,
- $\boldsymbol{V}$ is a $p \times p$ orthonormal matrix,
- $\boldsymbol{\Sigma}$ is $n \times p$ rectangular diagonal matrix with elements $\sigma_m \geq 0$ on the diagonal, where $m = \min\{n, p\}$, and
- $\boldsymbol{X}\boldsymbol{v}_k = \sigma_k \boldsymbol{u}_k$, for all $k \leq m$, where $\boldsymbol{v}_k$ and $\boldsymbol{u}_k$ are the kth columns of $\boldsymbol{V}$ and $\boldsymbol{U}$ respectively.

We call the columns of $\boldsymbol{U}$ as the left-singular vectors of $\boldsymbol{X}$, the columns of $\boldsymbol{V}$ as the right-singular vectors of $\boldsymbol{X}$ and values $\sigma_k \geq 0$ as the singluar values of $\boldsymbol{X}$. We order the singular values in decreasing order $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_k \geq 0$ (and we define $\sigma_j = 0$ for all $j > m$).

SVD exposes the basis $\boldsymbol{V}$ for $\mathbb{R}^p$ and basis $\boldsymbol{U}$ for $\mathbb{R}^n$ that are particularly suitable to understand the structure of the original matrix $\boldsymbol{X}$.

Interpretation of SVD

Linear transformation.

Let's think about what matrix $\boldsymbol{X}$ does when it is used as a matrix of a linear mapping from $\mathbb{R}^p$ to $\mathbb{R}^n$. For example, consider how the parameter vector $\widehat{\boldsymbol{\beta}}$ turns into predicted values $\widehat{\boldsymbol{y}}$ through the linear mapping $\widehat{\boldsymbol{y}} = \boldsymbol{X}\widehat{\boldsymbol{\beta}}$ defined by the data matrix $\boldsymbol{X}$.

SVD says that the transformation $\widehat{\boldsymbol{\beta}} \mapsto \widehat{\boldsymbol{y}}$ between the standard bases of $\mathbb{R}^p$ and $\mathbb{R}^n$ happens in 3 basic steps.

- Rotate the standard coordinate system of $\mathbb{R}^p$ so that the (orthonormal) columns of $\boldsymbol{V}$ become the basis vectors, and express $\widehat{\boldsymbol{\beta}}$ in this orthonormal basis, whence the new coordinates of $\widehat{\boldsymbol{\beta}}$ are $\widehat{\boldsymbol{\beta}}^{(V)} = \boldsymbol{V}^T\widehat{\boldsymbol{\beta}}$.
- Scale the p elements of $\widehat{\boldsymbol{\beta}}^{(V)}$ by matrix $\boldsymbol{\Sigma}$, i.e., get the scaled coordinates

  $$\widehat{\boldsymbol{\beta}}^{(V,\Sigma)} = \boldsymbol{\Sigma}\widehat{\boldsymbol{\beta}}^{(V)} = \left(\sigma_1\widehat{\beta_1}^{(V)}, \ldots, \sigma_m\widehat{\beta}_m^{(V)}, 0, \ldots, 0\right)^T,$$

  (Note that if $m = p < n$ then the last $n - p$ elements of the vector are 0. Otherwise $m = n$ and there are no extras zeros at the end of the vector after element m.) Important is that this scaling happens by non-negative constants $\sigma_k \geq 0$ along each coordinate axis of the basis $\boldsymbol{V}$ in the input space $\mathbb{R}^p$ and ends up in the output space $\mathbb{R}^n$.
- Rotate the scaled coordinates $\widehat{\boldsymbol{\beta}}^{(V,\Sigma)}$ from the orthonormal basis $\boldsymbol{U}$ to the standard basis of $\mathbb{R}^n$ to get the final

  $$\widehat{\boldsymbol{y}} = \boldsymbol{U}\widehat{\boldsymbol{\beta}}^{(V,\Sigma)} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\widehat{\boldsymbol{\beta}} = \boldsymbol{X}\widehat{\boldsymbol{\beta}}.$$

  Important is that the output space coordinate $\sigma_k\widehat{\beta_k}^{(V)}$ along the basis vector $\boldsymbol{u}_k$ depends only on the input space coordinate along the basis vector $\boldsymbol{v}_k$ and not on any other direction of the input space.

Thus, SVD tells that the linear transformation defined by **any** matrix $\boldsymbol{X}$ is simply a (non-negative) scaling along certain characteristic directions, combined with two suitable rotations that reveal those characteristic directions with respect to the standard bases of input and output spaces of the matrix.

https://en.wikipedia.org/wiki/Singular-value_decomposition#/media/File:Singular-Value-Decomposition.svg

In the special case where $\boldsymbol{X} = \boldsymbol{X}^T$ is a symmetric square matrix, SVD coincides with the eigenvalue decomposition and $U = V$ is the basis formed by the eigenvectors of $\boldsymbol{X}$ and the singular values are the eigenvalues of $\boldsymbol{X}$.

Low rank approximation to $\boldsymbol{X}$.

Let's get back to our main topic of dimension reduction. Since matrix $\boldsymbol{\Sigma}$ is diagonal in SVD $\boldsymbol{X} = \boldsymbol{U\Sigma V}^T$, we can use the SVD to decompose $\boldsymbol{X}$ into a sum of simple $n \times p$ matrices, each of which has rank 1

$$\boldsymbol{X} = \sum_{k=1}^{m} \sigma_k \boldsymbol{u}_k \boldsymbol{v}_k^T.$$

This tells how the structure of $\boldsymbol{X}$ piles up from the individual pieces determined by the pairs of left and right singular vectors weighted by the singluar values. Each rank 1 matrix $\boldsymbol{u}_k \boldsymbol{v}_k^T$ can only capture one property for (all) rows (encoded by $\boldsymbol{v}_k$) and one property for (all) columns (encoded by $\boldsymbol{u}_k$), and the components are given in the decreasing order of contribution to the final matrix. It can be shown (Eckart-Young theorem) that by truncating this sum after K components leads to the rank K approximation to $\boldsymbol{X}$ that minimizes the elementwise squared error (Frobenius norm) from $\boldsymbol{X}$ among all rank K matrices. Thus, by keeping only the leading K components of SVD we reduce the amount of information from np elements (original size of $\boldsymbol{X}$) to $K(n + p + 1)$ elements.

A concrete way to visualize the rank K approximations is to consider an image as a rectangular matrix and then compare the different approximations to the original one. In particular, notice how the complexity of the image has a large effect on how many components are needed to represent it well. http://timbaumann.info/svd-image-compression-demo/

Connection between SVD and PCA

We determined earlier that PCs were the eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$. If we write this crossproduct matrix using the SVD of
$\boldsymbol{X} = \boldsymbol{U\Sigma V}^T$, we have that

$$\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{V\Sigma}^T\boldsymbol{U}^T\boldsymbol{U\Sigma V}^T = \boldsymbol{V\Sigma}^T\boldsymbol{\Sigma V}^T = \boldsymbol{VDV}^T,$$

where $n \times n$ diagonal matrix $\boldsymbol{D} = \boldsymbol{\Sigma}^T\boldsymbol{\Sigma}$. Thus, the eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$ can be read from columns of $\boldsymbol{V}$ and the corresponding eigenvalues are the squares of the singular values, i.e., $\lambda_k = \sigma_k^2$. Furthermore, the scores of the samples on the PCs are

$$\boldsymbol{XV} = \boldsymbol{U\Sigma},$$

and can be formed as linear combinations of the columns of $\boldsymbol{U}$. Hence, by doing the SVD for $\boldsymbol{X}$ we also get both the loadings and the scores of the PCs of the matrix $\boldsymbol{X}$. The function `prcomp` actually uses SVD for making the PCA.

Computing PCA

What is the quickest way to do PCA?

If $p \leq n$, then the eigenvalue decomposition of a $p \times p$ matrix $\boldsymbol{X}^T\boldsymbol{X}$ is a cubic procedure, that is, requires the number of basic operations (such as multiplications and additions) that grows as p^3 as a function of p. We denote this by saying that its (time) complexity is $\mathcal{O}(p^3)$, ("of the order of p^3"). The eigenvalue decomposition gives us the loadings and computation of all the PC scores takes additional $\mathcal{O}(np^2)$ time. Computation of the crossproduct $\boldsymbol{X}^T\boldsymbol{X}$ to start with takes yet another $\mathcal{O}(np^2)$ time. Thus the whole process is $\mathcal{O}(2np^2 + p^3)$.

If $p > n$, then we will have only n PCs and we can instead consider the eigen decompoisition of $n \times n$ matrix

$$\boldsymbol{XX}^T = \boldsymbol{U\Sigma V}^T\boldsymbol{V\Sigma}^T\boldsymbol{U}^T = \boldsymbol{U\Sigma}\Sigma^T\boldsymbol{U}^T,$$

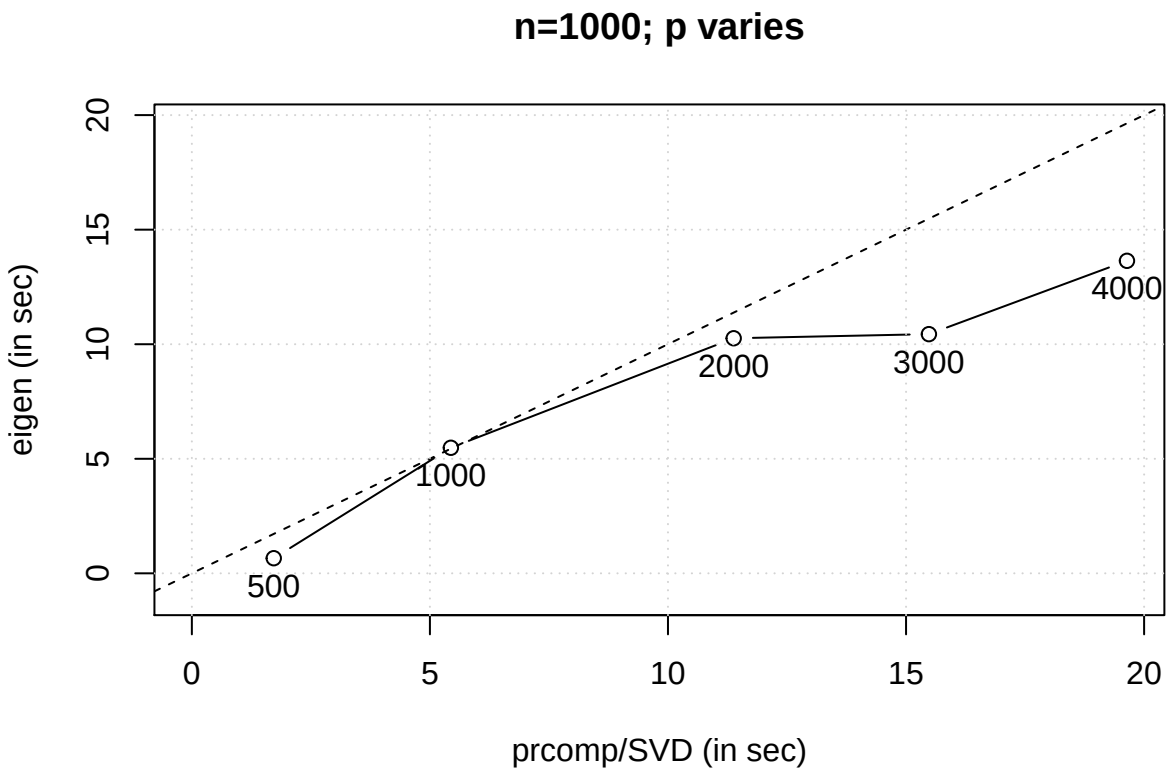
that gives us the scores on PCs (in columns of $\boldsymbol{U\Sigma}$) and the eigenvalues of each PC (the diagonal of $\Sigma\Sigma^T$). From there we can compute the loadings $\boldsymbol{V}$ from relationship $\boldsymbol{V\Sigma}^T = \boldsymbol{X}^T\boldsymbol{U}$ which is a $\mathcal{O}(pn^2)$ operation. Computing $\boldsymbol{XX}^T$ takes $\mathcal{O}(pn^2)$. Thus the whole process is $\mathcal{O}(2pn^2 + n^3)$.

All in all, PCA through an eigen decomposition takes $\mathcal{O}(\min\{(p^3 + np^2, n^3 + pn^2\})$ time.

The algorithms for SVD have similar theoretical complexity (depending on relationship between n and p). Let's see how they compare in practice.

```
set.seed(67)
file.name = "rprof.out" #time algorithms
n = 1000
ps = c(500, 1000, 2000, 3000, 4000)
times = matrix(NA, ncol = 2, nrow = length(ps))
for(ii in 1:length(ps)){
  p = ps[ii]
  X = scale(matrix(rnorm(n*p), ncol = p))
  #prcomp (Using SVD)
  Rprof(file.name)
  prc = prcomp(X)
  Rprof(NULL)
  times[ii,1] = summaryRprof(file.name)$sampling.time

  Rprof(file.name)
  if(n > p){
  #Use pxp matrix: t(X) %*% X
    eig = eigen(1/(n-1)*crossprod(X)) #has eig. values and loadings
    scores = X %*% eig$vectors #compute scores
  }else{
    #Use nxn matrix: X %*% t(X)
    eig = eigen(tcrossprod(X)) #has eig. values and scores
    loadings = crossprod(X, eig$vectors)%*%diag(1/sqrt(eig$values)) #compute loadings
  }
  Rprof(NULL)
  times[ii,2] = summaryRprof(file.name)$sampling.time
}
plot(times,xlab="prcomp/SVD (in sec)", ylab="eigen (in sec)", t="b",
     xlim=c(0,max(times)), ylim=c(-1,max(times)), main=paste0("n=",n,"; p varies"))
text(times[,1], times[,2]-1.2, labels = ps)
abline(0,1,lty=2)
grid()
```



Seems quite similar for smaller p but seems to deviate in favor of the eigendecomposition for larger p. This could be because as $n = 1000$ remains constant, the eigendecomposition for matrices with large $p\,(> n)$ has always the time complexity $\mathcal{O}(n^3)$, and hence the computing time does not grow that much with p as the pre-processing time grows only linearly in p.

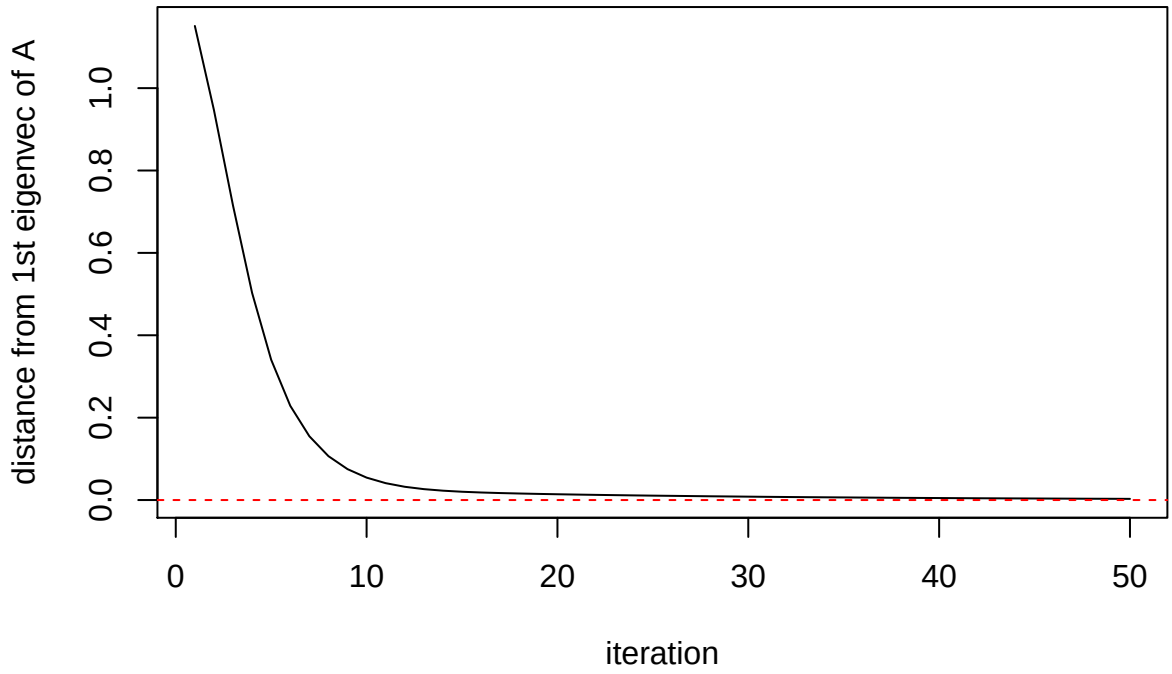
In practice, when both n and p are large, PCA is done by finding only a few of the leading components. (`prcomp()` can take in a parameter `rank` that specifies the number of components.) There are efficient methods that extract the leading components one-by-one without ever computing crossproduct matrices or the SVD. These methods are based on the idea of power iteration, that is, they iterate sequential multiplication of a random vector z by the target square matrix $\boldsymbol{A}$ to get a sequence $\boldsymbol{z}, \boldsymbol{Az}, \boldsymbol{A}^2\boldsymbol{z}, \ldots$, which converges towards the leading eigenvector of $\boldsymbol{A}$ when a suitable normalization is carried out at each step. Let's check that the power iteration works in practice.

```
set.seed(12)
n = 100
p = 10
X = scale(matrix(rnorm(n*p),ncol=p)) #random matrix
A = (t(X)%*%X)/(n-1) # A = X^T X, whose 1st eigenv vector we want
eig = eigen(A)
eig$values[1:3] #check that 1st is clearly larger than others
```

```
## [1] 1.622358 1.538839 1.189090
```

```
A.eig1 = eig$vector[,1] #this is our target eigenvector
z = rnorm(p) #a random vector in p-space
niter = 50
res = rep(NA, niter) #distance between 1st eig.vector and power iterated vector
for(ii in 1:niter){
  z = z/sqrt(sum(z^2)) #normalize
  res[ii] = sqrt(sum( (A.eig1-z)^2 )) #distance to target
  z = A %*% z
```

```
}
plot(res,t="l", xlab="iteration", ylab="distance from 1st eigenvec of A")
abline(h=0,lty=2, col="red")
```



The actual methods are more sophisticated and robust than this simple power iteration and form a very active research area. For example, the PCA algorithms that are currently applied to biobank scale data in human genomics (n and p are both $\sim 10^5$) have not been around for more than two years. See slides for an example analysis of *FastPCA*. Another method is *FlashPCA*.

Connection between PCA and ridge regression

Consider the regression problem of estimating $\boldsymbol{\beta} \in \mathbb{R}^p$ in $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$, where $\boldsymbol{y}$ is mean centered and size of $\boldsymbol{X}$ is $n \times p$.

Let's apply a SVD of $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ to the predicted outcome values from the OLS estimate

$$\widehat{\boldsymbol{y}}_{\text{OLS}} = \boldsymbol{X}\widehat{\boldsymbol{\beta}}_{\text{OLS}} = \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\left(\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\right)^{-1}\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{y} = \boldsymbol{U}\boldsymbol{U}^T\boldsymbol{y}.$$

We have assumed in this SVD that

- $\boldsymbol{U}$ is an $n \times r$ matrix and spans the column space of $\boldsymbol{X}$,
- $\boldsymbol{\Sigma}$ is an $r \times r$ diagonal matrix with positive diagonal elements $\sigma_1 \geq \ldots \geq \sigma_r > 0$
- $\boldsymbol{V}$ is an $r \times r$ matrix that spans the row space of $\boldsymbol{X}$,
- $r \leq n$ is the rank of $\boldsymbol{X}$.

Thus, if $r < n$, we have discarded those $n - r$ basis vectors of $\mathbb{R}^n$ from $\boldsymbol{U}$ that correspond to the singular value 0. Similarly, if $r < p$, we have discarded those $p - r$ basis vectors of $\mathbb{R}^p$ from $\boldsymbol{V}$ that correspond to the singular value 0. (Had we kept these unnecessary columns in $\boldsymbol{U}$ and $\boldsymbol{V}$ to make them square matrices, the above formula would not hold.)

This shows, unsurprisingly, that the OLS predictions are obtained by an orthogonal projection of the outcome variable to basis $\boldsymbol{U}$ that is the column space of $\boldsymbol{X}$.

Consider then the ridge regression (RR) estimates with the penalty parameter λ. The fitted values are

$$\begin{aligned}
\widehat{\boldsymbol{y}}_{\mathrm{RR}} &= \boldsymbol{X}\widehat{\boldsymbol{\beta}}_{\mathrm{RR}} = \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} \\
&= \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\left(\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{y} \\
&= \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\left(\boldsymbol{V}\left(\boldsymbol{\Sigma}^2 + \lambda\boldsymbol{I}\right)\boldsymbol{V}^T\right)^{-1}\boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^T\boldsymbol{y} \\
&= \boldsymbol{U}\boldsymbol{\Sigma}\left(\boldsymbol{\Sigma}^2 + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{\Sigma}\boldsymbol{U}^T\boldsymbol{y} \\
&= \boldsymbol{U}\left(\boldsymbol{\Sigma}^{-1}\left(\boldsymbol{\Sigma}^2 + \lambda\boldsymbol{I}\right)\boldsymbol{\Sigma}^{-1}\right)^{-1}\boldsymbol{U}^T\boldsymbol{y} \\
&= \boldsymbol{U}\left(\boldsymbol{I} + \lambda\boldsymbol{\Sigma}^{-2}\right)^{-1}\boldsymbol{U}^T\boldsymbol{y} \\
&= \boldsymbol{U}\boldsymbol{\Delta}\left(\frac{\sigma_k^2}{\sigma_k^2 + \lambda}\right)\boldsymbol{U}^T\boldsymbol{y},
\end{aligned}$$

where $\boldsymbol{\Delta}(d_k)$ is the diagonal matrix with sequence $(d_k)_{k=1}^r$ on the diagonal.

Thus, for ridge regression prediction, we also project the outcome variable to the column space but not orthogonally. Instead, we will shrink (if $\lambda > 0$) the coordinates along each $\boldsymbol{u}_k$ direction by the coefficient $\frac{\sigma_k^2}{\sigma_k^2+\lambda}$, which means that the directions with smallest singular values will be shrunk the most. These are the directions in column space, where the data has the smallest variance. We see that the ridge regression penalty induces relatively stronger shrinkage in the directions where the data vary less. This sounds like a good approach if those directions are dominated by noise rather than the true signal. The idea is similar to that of PCA: we remove or downweight the directions in which the data vary little and get both dimension reduction and more robust analyses. (But we will also lose some of the original information.)

Read from ISL 6.3 Dimension Reduction

- What is Principal components regression (PCR)?
- What are advantages and disadvantages of PCR compared to OLS?
- What is Partial Least Squares (PLS)?
- What is advantages and disadvantages of PLS compared to OLS?
- What is the difference between PCR and PLS?
- How do PCR and PLS compare to ridge regression and LASSO?

Read from ISL 6.4 Considerations in high dimensions (HDs)

- What are the challenges in high dimensional analyses?
- Which measures of goodness-of-fit one should / should not use in HDs?
- Which are suitable methods for regression in HDs?