

HDS 9. Principal component analysis (PCA)

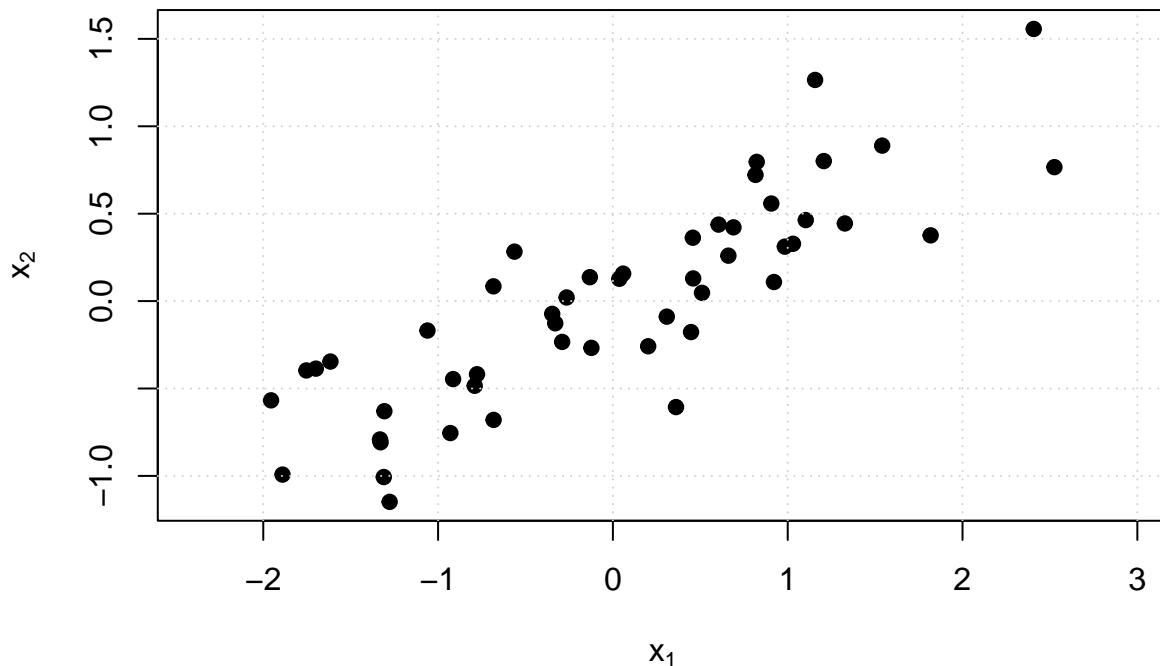
Matti Pirinen, University of Helsinki

12.1.2024

Now we move from variable selection to dimension reduction. The goal is to represent high dimensional data in a low dimensional subspace in a useful way, 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 the dimension. Intuitively, given a target dimension, we want to find such a mapping from the original data space to the target space that maintains as much of the original information as possible. PCA, first defined in 1901 by Karl Pearson, is one of the most widely used method for this purpose, and one with a clear statistical motivation and linear algebraic implementation. Thus, we will start from it.

An 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)) #A lways mean-center for PCA
plot(X, asp = 1, pch = 19, xlab = expression(x[1]), ylab = expression(x[2]))
grid()
```



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

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

These variables are quite correlated, so they are telling partly a redundant story about the study samples. Do we really need both dimensions to represent these data? Or could only one dimension be enough for some purposes? How could we project these points to one dimension in a useful way that would preserve the main structure of the points.

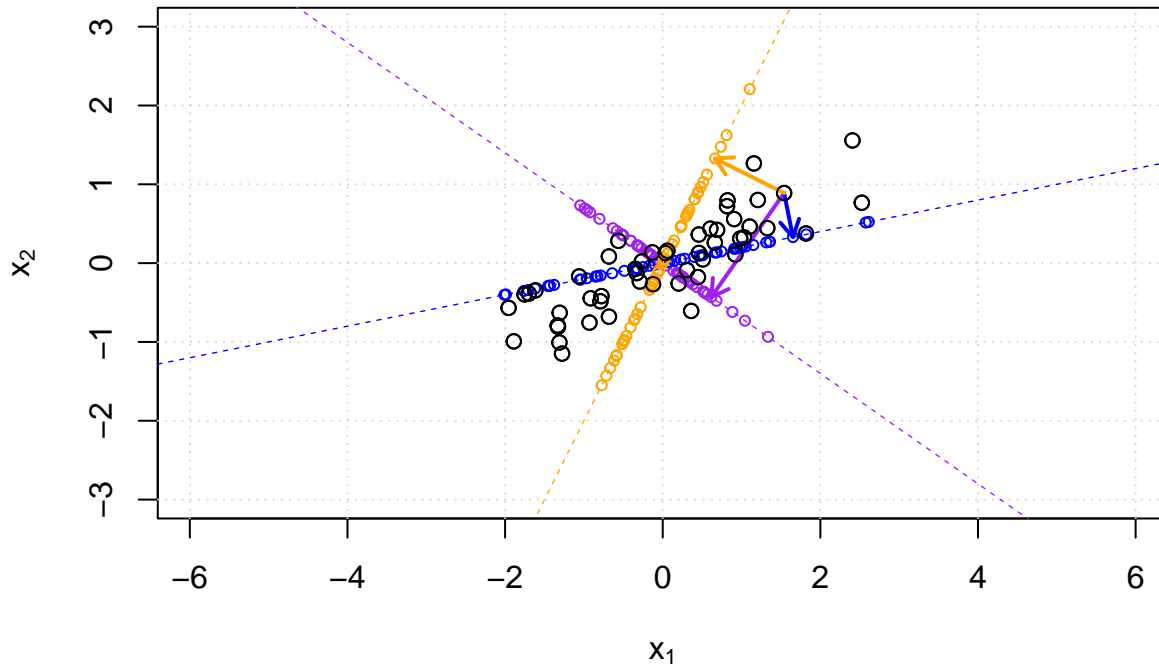
Let's draw some lines through the origin and project the data on those lines. (Since the data are mean centered, the origin is at the center of the data set.) Then we will have a set of one dimensional representations of the data. The unit vector of a line that passes through the origin with slope b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

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

And the coordinate of the point along the line is

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

```
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) # slopes of the 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 the point to the 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) # re-draw the original points on top
```



```
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 the line 2, with the 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 structure of the original data than the other two lines. Think about the two extreme examples of possible projections on a line. First, the useless case where each original point is projected on the same point on a line, loses all the information about the differences between the original data units. Second, if the original data were already on a line, then we do not lose anything by simply recording the coordinates on that line instead of the original multidimensional coordinates. In reality, our data lies somewhere in between these extremes, and in the PCA the defining criterion of the target subspace is that in the subspace the projections of the data points preserve the highest possible variance.

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

$$\widehat{\text{Var}}(\mathbf{X}\mathbf{u}) \propto \mathbf{u}^T \mathbf{X}^T \mathbf{X} \mathbf{u}.$$

We can use the method of Lagrange multipliers to maximize this quantity among all unit vectors \mathbf{u} , that is, the vectors for which $\mathbf{u}^T \mathbf{u} - 1 = 0$. This method requires us to maximize the extended function $f(\mathbf{u}, \alpha) = \mathbf{u}^T \mathbf{X}^T \mathbf{X} \mathbf{u} + \alpha(\mathbf{u}^T \mathbf{u} - 1)$ with respect to \mathbf{u} and α . The derivative with respect to \mathbf{u} is

$$\frac{\partial f(\mathbf{u}, \alpha)}{\partial \mathbf{u}} = 2\mathbf{X}^T \mathbf{X} \mathbf{u} + 2\alpha \mathbf{u},$$

and equals to 0 when $\mathbf{X}^T \mathbf{X} \mathbf{u} = -\alpha \mathbf{u}$, that is, when \mathbf{u} is an eigenvector of the matrix $\mathbf{X}^T \mathbf{X}$ and $-\alpha$ is the corresponding eigenvalue. Among all eigenvectors, we thus choose the one that maximizes $\mathbf{u}^T \mathbf{X}^T \mathbf{X} \mathbf{u} = -\alpha \mathbf{u}^T \mathbf{u} = -\alpha$, which shows that \mathbf{u} should correspond to the largest eigenvalue of $\mathbf{X}^T \mathbf{X}$.

Let's apply this result in practice.

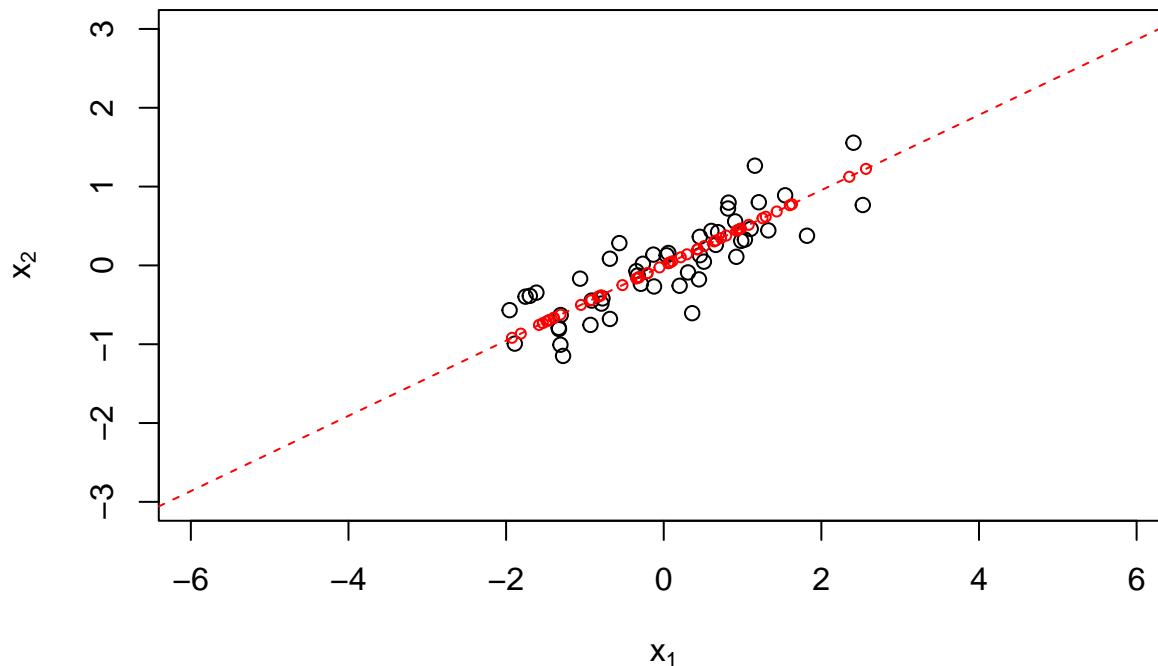
```
eig = eigen(t(X) %*% X)
eig$values
```

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

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

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

```
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(u[1]*coord.on.line, u[2]*coord.on.line, col = "red", cex = 0.7)
```



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

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

The chosen line looks plausible. Visually, the projections on this line preserves a lot of the information of the original 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 the eigenvectors (of a symmetric matrix like $\mathbf{X}^T \mathbf{X}$) are orthogonal, they give a new coordinate system on which the data have 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 \mathbf{X} . The idea is that since the PC 1 has been chosen to maximize the variability between the points, it is a good candidate for the one dimensional representation of the original data. Now we may reduce the dimension of the data by storing only the points' coordinates on the PC 1 and ignoring the PC 2. We will lose some information, but, as measured by the variance, this loss is as small as it can be given that we are reducing the dimension.

You can study this via interactive play with 2D and 3D PCA: <http://setosa.io/ev/principal-component-analysis/>

PCA more generally More generally, when $p > 2$, the number of PCs is $\min\{p, n\}$. The PC k is defined by the unit vector $\mathbf{u}_k = (u_{k1}, \dots, u_{kp})^T$, ($\|\mathbf{u}_k\|_2 = 1$), and the **score** (previously called “coordinate”) of data point i on the PC k is $\text{pc}_k(i) = \mathbf{x}_i^T \mathbf{u}_k$. The coefficient u_{kj} , by which the variable j contributes to the PC k , is called the **loading** of the variable j on the PC k . PCs have the following properties

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

Each PC is a one dimensional subspace (a 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 data point \mathbf{x}^* measured on the same p variables on which we have constructed the PCs earlier with the data matrix \mathbf{X} . We can project this new point on the existing PCs by the linear mapping $\text{pc}_k(\mathbf{x}^*) = \mathbf{x}^{*T} \mathbf{u}_k = \sum_{j=1}^p x_j^* u_{kj}$.

The dimension reduction to $k \leq p$ dimensions with the PCA happens by simply ignoring the PCs after the k largest components. 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 the squared distances between the original points and their projections on any k dimensional subspace. Remember that linear regression minimizes the 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 outcomes), and minimizes the total squared distance between its predictions, that are the projections on the first k PCs, and the original points in the p dimensional space.

Another derivation of PCA Above we saw that the question of the maximum variance line is solved by the leading eigenvector of the product matrix of the data $\mathbf{X}^T \mathbf{X}$. We can derive the PCA solution also by relying on a result on the existence of an eigenvalue decomposition of a symmetric ($p \times p$) matrix \mathbf{S} (here taken to be the covariance matrix of the data vector \mathbf{x} when considered as a random variable). This result says that \mathbf{S} can be written as $\mathbf{S} = \mathbf{U} \mathbf{D} \mathbf{U}^T$, where ($p \times p$) matrix \mathbf{U} is orthonormal, (that is, for any columns $i \neq j$ $\mathbf{u}_i^T \mathbf{u}_j = 0$ and $\mathbf{u}_i^T \mathbf{u}_i = 1$) and where $\mathbf{D} = (d_j)_{j=1}^p$ is a diagonal matrix. Note that $\mathbf{U} \mathbf{U}^T = \mathbf{U}^T \mathbf{U} = \mathbf{I}$.

Let's set as our goal to find such an orthogonal $p \times p$ matrix \mathbf{Q} that when the data are transformed by the linear mapping \mathbf{Q} , the covariances between our original variables are removed. In other words, we look for an orthogonal \mathbf{Q} that will transform our observed p -dimensional vector \mathbf{x} , with original covariance matrix \mathbf{S}_x (with eigendecomposition $\mathbf{S}_x = \mathbf{U}_x \mathbf{D}_x \mathbf{U}_x^T$), into a new p -dimensional vector $\mathbf{y} = \mathbf{Q} \mathbf{x}$ whose covariance would be diagonal. Since we require \mathbf{Q} to be 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

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

we see that by a choice $\mathbf{Q} = \mathbf{U}_x^T$ we have that $\text{Var}(\mathbf{Q}\mathbf{x}) = \mathbf{D}_x$ is diagonal and hence the transformed variables have zero covariances. Thus, by transforming the variables by the eigenvectors of the empirical covariance matrix of \mathbf{x} , which is $\hat{\mathbf{S}}_x = \frac{1}{n-1}\mathbf{X}^T\mathbf{X}$, 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 PC 1 etc. Note that the constant $\frac{1}{n-1}$ does not affect the eigenvectors (although it does affect the eigenvalues multiplicatively). Thus, for deriving the PCA transformation, it does not matter whether we compute the eigenvectors from $\mathbf{X}^T\mathbf{X}$ or $\frac{1}{n-1}\mathbf{X}^T\mathbf{X}$.

PCA in practice

Let's use PCA on the Boston data set. Let's apply both the R's standard function `prcomp()` and also check that the above defined 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, and not by applying `eigen()` on the data 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 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 ...
## $ chas   : int   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 ...
## $ 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 ...
## $ 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 ...
## $ 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 = setdiff(1:14, keep) # leave this out from PCA and think it 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 = TRUE, scale = TRUE)
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 the sign is arbitrary!
c(prc$x[5,3], t(eig$vectors[,3]) %*% Boston.scaled[5,keep])
```

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

If we had a new Boston area with measured values for these 13 variables, how could we project it to these 2D plots? Let's do projection for the area 15, even though it was already on PC analysis, and compare to the observed 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
```

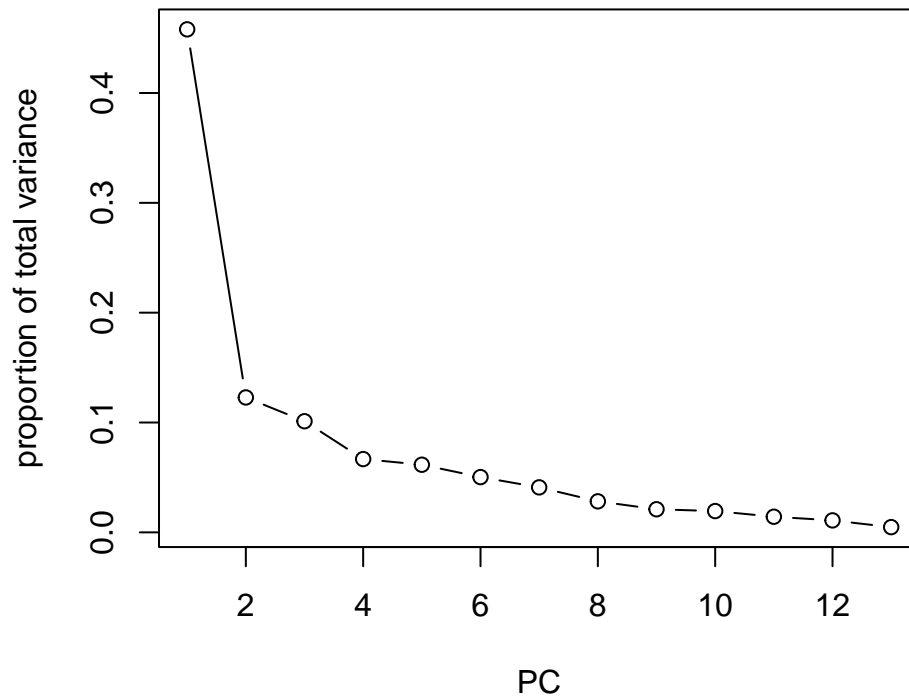
Let's see how large a proportion of the total variance the PCs cumulatively capture.

```
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
```

The 1st component alone explains 46% and the first 7 components explain 90%. Often the variance explained by the 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]
```

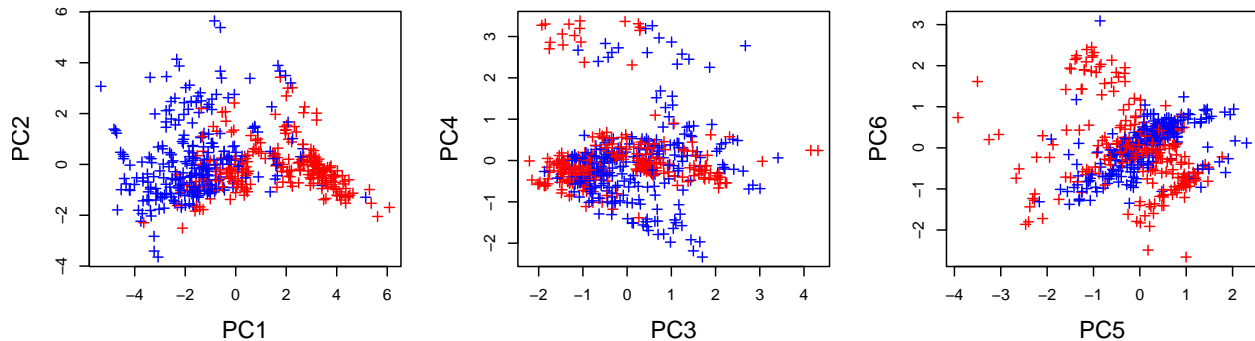
```
##      crim      zn      indus      chas      nox      rm  
## 0.255554649 -0.261508469 0.351162650 -0.001387083 0.344581670 -0.197336951  
##      age      dis      rad      tax      ptratio      black  
## 0.311089512 -0.319149629 0.325450533 0.345858250 0.218842202 -0.207663732  
##      medv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

Let's visualize the first six PCs and color each point according to the `lstat` value (higher values 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 the 1st and 3rd plots, where the scores of the PCs seem to associate with the `lstat` value, which was not included in the PCA.

Principal components regression (PCR) Let's do linear regression of `lstat ~ PC1` and compare it 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"
```

The PC 1 explains similarly `lstat` as `medv`, and much more than any other 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 to do the regression on some of the leading PCs. This is called *principal components regression (PCR)*. For example, when regressing the outcome variable y on the PC 1 and PC 2, the model is

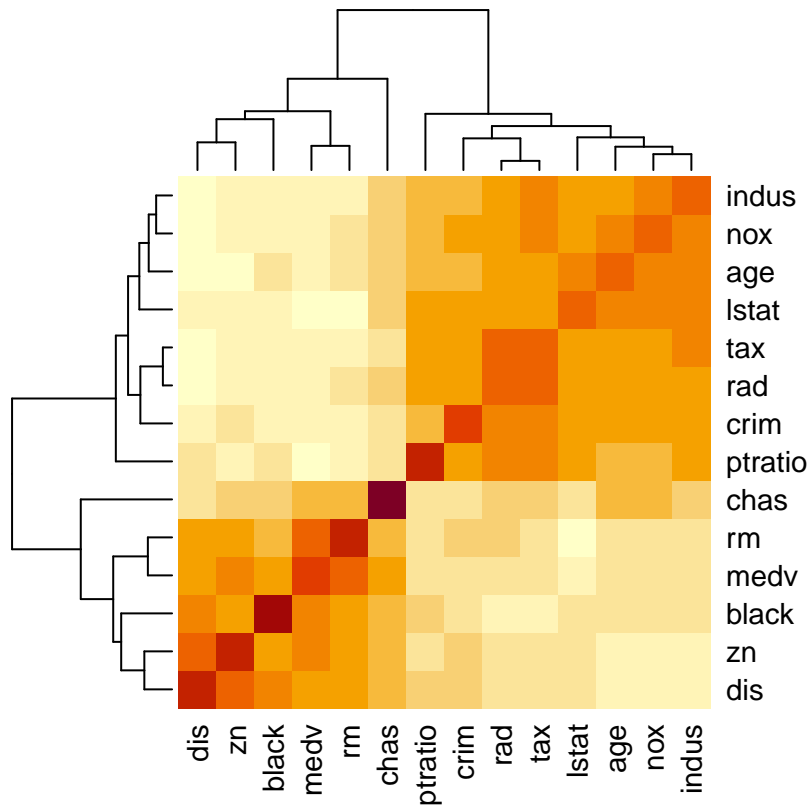
$$\begin{aligned} y_i &= \mu + \text{pc}_1(i)\beta_1 + \text{pc}_2(i)\beta_2 + \varepsilon \\ &= \mu + \beta_1 \sum_{j=1}^p x_{ij}u_{1j} + \beta_2 \sum_{j=1}^p x_{ij}u_{2j} + \varepsilon \end{aligned}$$

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 the leading PCs, we can capture more of the main structure of the predictors than with an 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.

An 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. A disadvantage is that 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, whereas the PCs didn't.

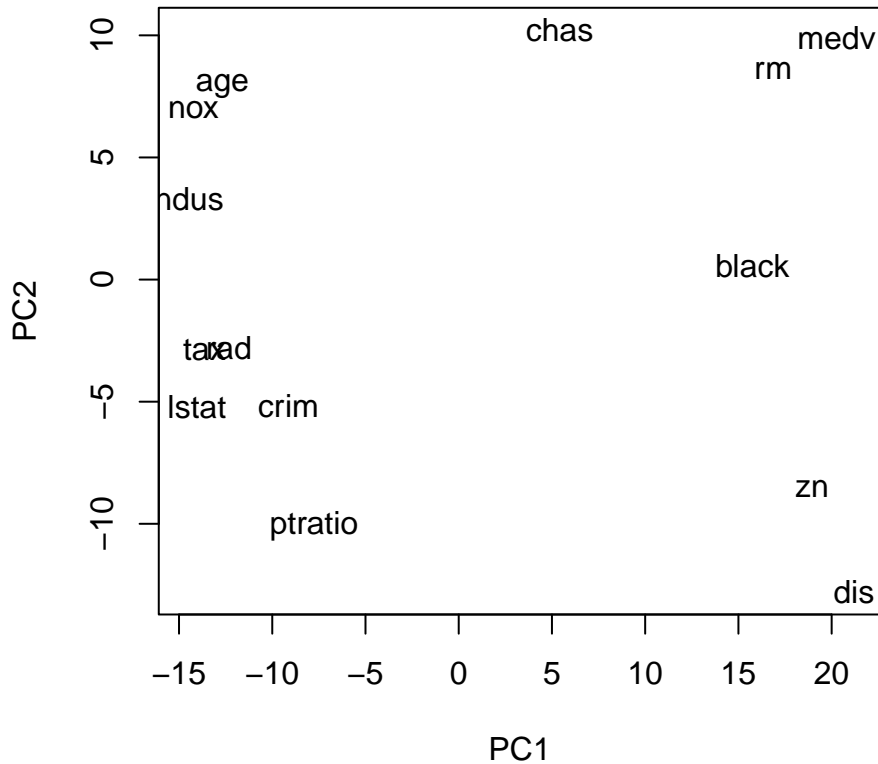
PCA on the samples rather than on the variables Let's then use the PCA the other way around. We now consider the $p = 14$ variables as the "samples" that have been measured on $n = 506$ "variables" which are the Boston areas. We want to find the 2-dimensional description of the relationships between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

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



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

```
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 the first k PCs have together $p \cdot k$ elements. Already when p is in hundreds, and we keep k small, say $k \leq 10$, there is a considerable difference in favor of the PCs in the efficiency of the data reduction, and this difference becomes crucial in applications where p is in thousands.

See examples of PCA in slides [HDS9_slides.pdf](#)

Read ISL section 12.2 PCA

- What does PCA aim to do?
- What are the *loadings* and what are the *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 the data matrix the other way as we are using so the data matrix is given as a $p \times n$ matrix, and p (the number of variables) is called m .

Singular value decomposition (SVD)

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

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

- \mathbf{U} is an $n \times n$ orthonormal matrix,
- \mathbf{V} is a $p \times p$ orthonormal matrix,
- $\mathbf{\Sigma}$ is $n \times p$ rectangular diagonal matrix with elements $\sigma_1, \dots, \sigma_m \geq 0$ on the diagonal, where $m = \min\{n, p\}$, and
- $\mathbf{X}\mathbf{v}_k = \sigma_k\mathbf{u}_k$, for all $k \leq m$, where \mathbf{v}_k and \mathbf{u}_k are the k th columns of \mathbf{V} and \mathbf{U} respectively.

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

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

Interpretation of SVD Linear transformation.

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

The SVD says that the transformation $\hat{\boldsymbol{\beta}} \mapsto \hat{\mathbf{y}}$ 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 \mathbf{V} become the basis vectors, and express $\hat{\boldsymbol{\beta}}$ in this orthonormal basis, when the new coordinates of $\hat{\boldsymbol{\beta}}$ are $\hat{\boldsymbol{\beta}}^{(V)} = \mathbf{V}^T\hat{\boldsymbol{\beta}}$.
- Scale the p elements of $\hat{\boldsymbol{\beta}}^{(V)}$ by matrix $\mathbf{\Sigma}$, i.e., get the scaled coordinates

$$\hat{\boldsymbol{\beta}}^{(V,\Sigma)} = \mathbf{\Sigma}\hat{\boldsymbol{\beta}}^{(V)} = \left(\sigma_1\hat{\beta}_1^{(V)}, \dots, \sigma_m\hat{\beta}_m^{(V)}, 0, \dots, 0 \right)^T,$$

(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 the element m .) It is important that this scaling happens by the non-negative constants $\sigma_k \geq 0$ along each coordinate axis of $\hat{\mathbf{y}}$ of the basis \mathbf{V} in the input space \mathbb{R}^p and ends up in the output space \mathbb{R}^n .

- Rotate the scaled coordinates $\hat{\boldsymbol{\beta}}^{(V,\Sigma)}$, that are given in the orthonormal basis \mathbf{U} , to the standard basis of \mathbb{R}^n to get the final

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

It is important that the output space coordinate $\sigma_k\hat{\beta}_k^{(V)}$ along the basis vector \mathbf{u}_k depends only on the input space coordinate along the basis vector \mathbf{v}_k and not on any other direction of the input space.

Thus, the SVD tells that the linear transformation defined by **any** matrix \mathbf{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 the 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 $\mathbf{X} = \mathbf{X}^T$ is a symmetric square matrix, the SVD coincides with the eigenvalue decomposition and $\mathbf{U} = \mathbf{V}$ is the basis formed by the eigenvectors of \mathbf{X} and the singular values are the eigenvalues of \mathbf{X} .

A low rank approximation to \mathbf{X} .

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

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

This tells how the structure of \mathbf{X} piles up from the individual pieces determined by the pairs of left and right singular vectors weighted by the singular values. Each rank 1 matrix $\mathbf{u}_k \mathbf{v}_k^T$ can only capture one quantitative pattern shared by all rows (encoded by \mathbf{v}_k) and one pattern shared by all columns (encoded by \mathbf{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 of rank 1 matrices after the first K components leads to the rank K approximation to \mathbf{X} that minimizes the elementwise squared error (*Frobenius norm*) from \mathbf{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 \mathbf{X}) to $K(n + p + 1)$ elements in the optimal way in terms of the MSE.

Example. Let's consider a 2×3 matrix X and see how it is formed as a sum of two rank 1 matrices determined from its SVD.

```
X = matrix(c(1, 0, -1,
            2, 1, 1), ncol = 3, byrow = T)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    0   -1
## [2,]    2    1    1
```

```
# make SVD
S = svd(X)
S
```

```
## $d
## [1] 2.497212 1.328131
##
## $u
##      [,1] [,2]
## [1,] 0.2297529 0.9732490
## [2,] 0.9732490 -0.2297529
##
## $v
##      [,1] [,2]
## [1,] 0.8714722 0.3868166
## [2,] 0.3897342 -0.1729896
## [3,] 0.2977305 -0.9057856
```

```
# Two rank 1 matrices r.k = u_k %*% v_k^T, for k = 1,2:
r.1 = S$u[,1] %*% t(S$v[,1])
r.2 = S$u[,2] %*% t(S$v[,2])
r.1
```

```
##      [,1] [,2] [,3]
## [1,] 0.2002233 0.08954258 0.06840444
## [2,] 0.8481594 0.37930844 0.28976586
```

```
r.2
```

```
##           [,1]      [,2]      [,3]
## [1,]  0.37646888 -0.16836200 -0.8815549
## [2,] -0.08887225  0.03974488  0.2081069
```

```
# when they are weighted by singular values, they will make up the original X:
S$d[1]*r.1 + S$d[2]*r.2
```

```
##           [,1] [,2] [,3]
## [1,]      1   0  -1
## [2,]      2   1   1
```

```
# and the best rank 1 approximation to X (measured by MSE) is given by
S$d[1]*r.1
```

```
##           [,1]      [,2]      [,3]
## [1,]  0.500000  0.2236068  0.1708204
## [2,]  2.118034  0.9472136  0.7236068
```

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

$$\mathbf{X}^T \mathbf{X} = \mathbf{V}\mathbf{\Sigma}^T \mathbf{U}^T \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \mathbf{V}\mathbf{\Sigma}^T \mathbf{\Sigma}\mathbf{V}^T = \mathbf{V}\mathbf{D}\mathbf{V}^T,$$

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

$$\mathbf{X}\mathbf{V} = \mathbf{U}\mathbf{\Sigma},$$

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

Computing the PCA

What is the quickest way to do the PCA?

If $p \leq n$, then the eigenvalue decomposition of a $p \times p$ matrix $\mathbf{X}^T \mathbf{X}$ 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 the computation of all PC scores takes an additional $\mathcal{O}(np^2)$ time. Computation of the crossproduct $\mathbf{X}^T \mathbf{X}$ 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 eigendecomposition of $n \times n$ matrix

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

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

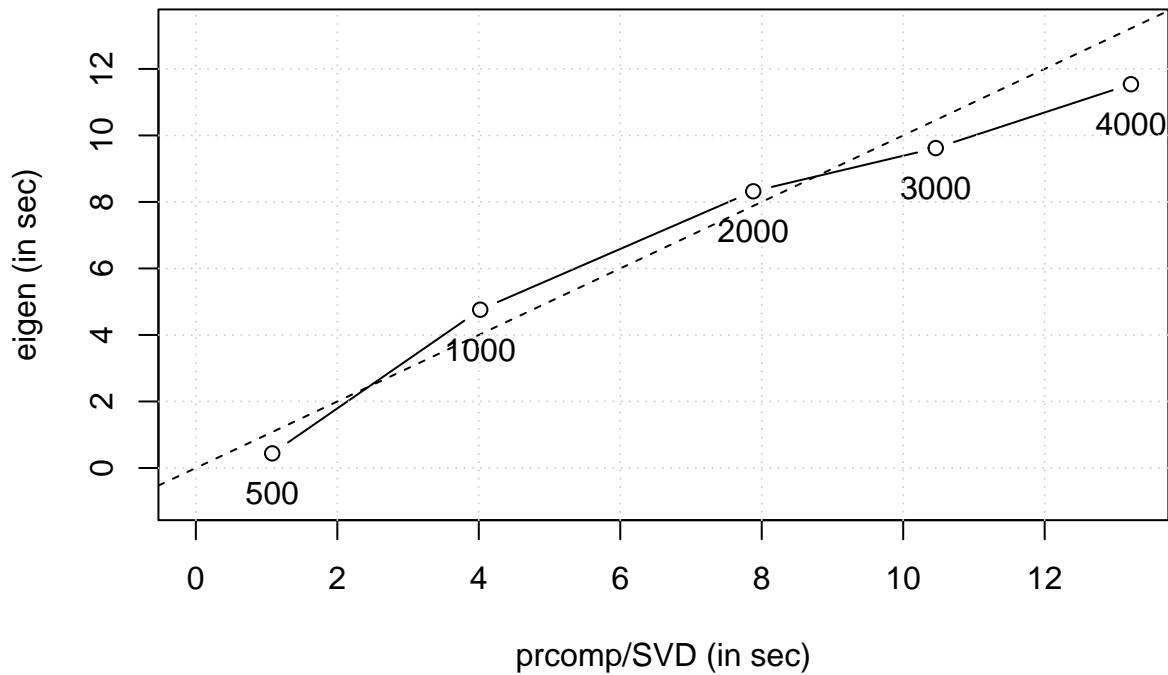
All in all, PCA through an eigendecomposition takes $\mathcal{O}(\min\{p^3 + np^2, n^3 + pn^2\})$ time.

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

```
set.seed(67)
file.name = "rprof.out" # time the 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 is using the SVD
  Rprof(file.name)
  prc = prcomp(X)
  Rprof(NULL)
  times[ii,1] = summaryRprof(file.name)$sampling.time

  Rprof(file.name)
  if(n > p){
    # use the pxp matrix: t(X) %*% X
    eig = eigen(1 / (n-1) * crossprod(X)) # has eig. values and loadings
    scores = X %*% eig$vectors # compute scores
  }else{
    # use the nxn matrix: X %*% t(X)
    eig = eigen(tcrossprod(X)) # has eig. values and scores
    loadings = crossprod(X, eig$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()
```


n=1000; p varies



Seems quite similar for smaller p but seems to favor the eigendecomposition for larger p . This could be because when $n = 1000$ 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 because the pre-processing time grows only linearly in p .

In practice, when both n and p are large, the 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 the 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 A to get a sequence z, Az, A^2z, \dots , which converges towards the leading eigenvector of A , if a suitable normalization is carried out at each step. Let's check that the power iteration works in practice.

```
set.seed(14)
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 eigenvector we want
eig = eigen(A)
eig$values[1:3] # check that the 1st is clearly larger than others
```

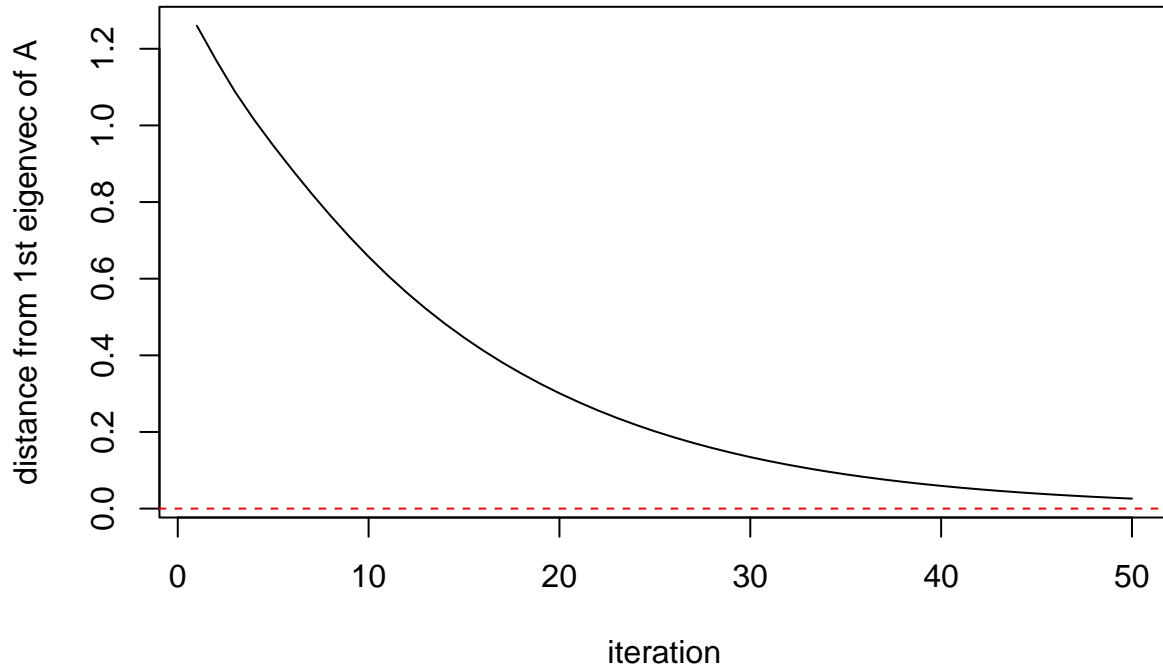
```
## [1] 1.541419 1.419703 1.268965
```

```
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 the 1st eig.vector and the 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 in use, such as Implicitly Restarted Arnoldi Method, are more sophisticated and robust than this simple power iteration. See the slides for an example analysis of *FastPCA* on the UK biobank data. Another method is *FlashPCA*.

A connection between the PCA and ridge regression

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

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

$$\hat{\mathbf{y}}_{\text{OLS}} = \mathbf{X}\hat{\beta}_{\text{OLS}} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T(\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^{-1}\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T\mathbf{y} = \mathbf{U}\mathbf{U}^T\mathbf{y}.$$

We have assumed in this SVD that

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

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

This shows, unsurprisingly, that the ordinary least squares predictions are obtained by an orthogonal projection of the outcome variable to the basis \mathbf{U} that is the column space of \mathbf{X} .

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

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

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

Thus, also for the ridge regression prediction, we project the outcome variable to the column space, but not orthogonally. Instead, we will shrink (if $\lambda > 0$) the coordinates along each \mathbf{u}_k direction by the coefficient $\frac{\sigma_k^2}{\sigma_k^2 + \lambda}$, which means that the directions with the smallest singular values will be shrunk the most. These are the directions in the column space, where the data have the smallest variance. We see that the ridge regression penalty induces relatively stronger shrinkage in the directions where the data vary the least. This sounds like a good approach if those directions are dominated by noise rather than by 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. And as the cost, we will also lose some of the original information.