Matrix algebra and linear projections

Linear Regression

In the first part, we look at how multiple linear regression can be performed directly using linear algebra using a *single* line of code.

Lets start with something very simple: linear regression with one feature. Recall that the goal of linear regression is to find a function:

$$f(x) = \beta_0 + \beta_1 \cdot x_1$$

The notation can be simplified if we pretend that the intercept is simply a parameter for a feature x_0 which is always equal to 1.

$$f(x) = \beta_0 \cdot 1 + \beta_1 \cdot x_1 = \beta_0 \cdot x_0 + \beta_1 \cdot x_1$$

Linear regression at its simplest is when there is one feature and with two data points: x_1, x_2 with targets y_1, y_2. Finding the line that goes through these two values simply reduces to solving a system of linear equations:

$$y_1 = f(x_1) = \beta_0 \cdot x_{1,0} + \beta_1 \cdot x_{1,1}$$
$$y_2 = f(x_2) = \beta_0 \cdot x_{2,0} + \beta_1 \cdot x_{2,1}$$

We also can write this system of linear equations as matrix multiplication:

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} x_{1,0} & x_{1,1} \\ x_{2,0} & x_{2,1} \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}}_{\beta}$$

The matrix X here is called the *design matrix* and the general linear system is:

$$y = X\beta$$

If there are K features and N data points then the dimensions of X are $N \times (K + 1)$.

Computing the solution to our linear system with two data points is as easy as computing the inverse matrix X^{-1} to X and multiplying both sides by this inverse matrix:

$$X^{-1}y = X^{-1}X\beta = \beta$$

Consider now a concrete example with two data points. The value of the feature is: $x_{1,1} = 2$ and $x_{2,1} = 5$. The target is $y_1 = 7$ and $y_2 = 3$. The design matrix X is then (don't forget the intercept feature):

```
X <- rbind(c(1,2),
           c(1,5))
print(X)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    5
```

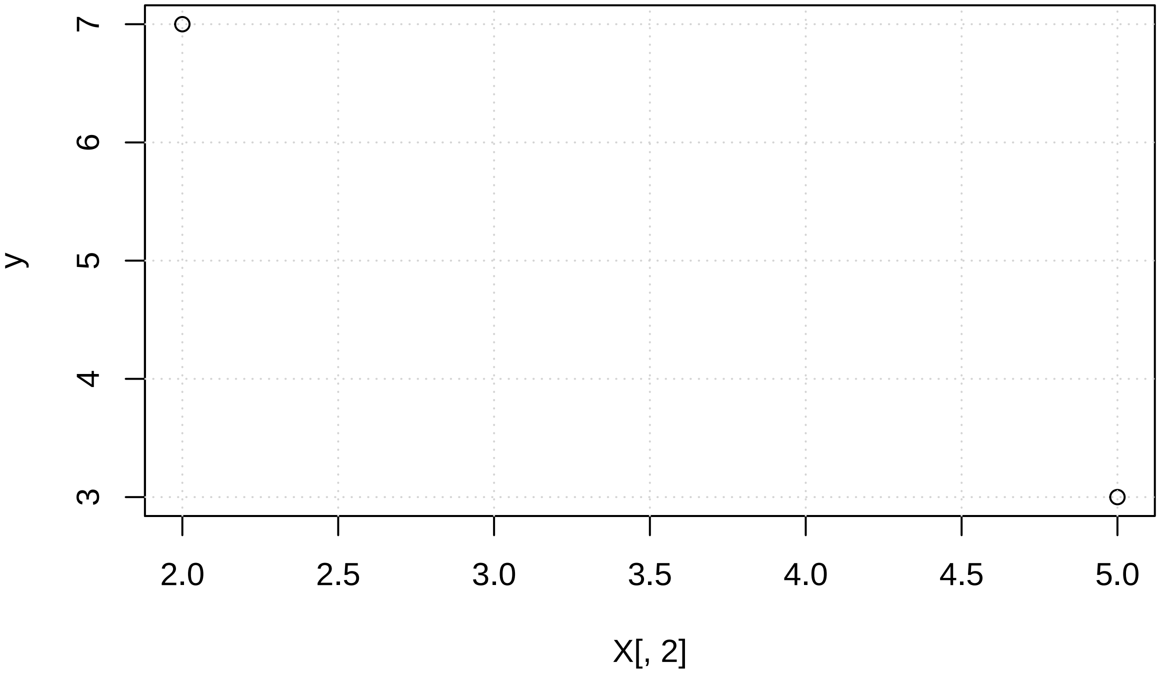
The target vector y is:

```
y <- c(7,3)
print(y)
```

```
## [1] 7 3
```

The two data points plotted look as follows:

```
plot(X[,2],y); grid()
```



We can now invert the matrix and get our solution to the linear regression problem!

```
Xinv <- solve(X)
print(Xinv)
```

```
##            [,1]       [,2]
## [1,]  1.6666667 -0.6666667
## [2,] -0.3333333  0.3333333
```

Lets double check that this is indeed a proper matrix inverse.

```
Xinv %*% X
```

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

```
X %*% Xinv
```

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

Yes the inverse works!

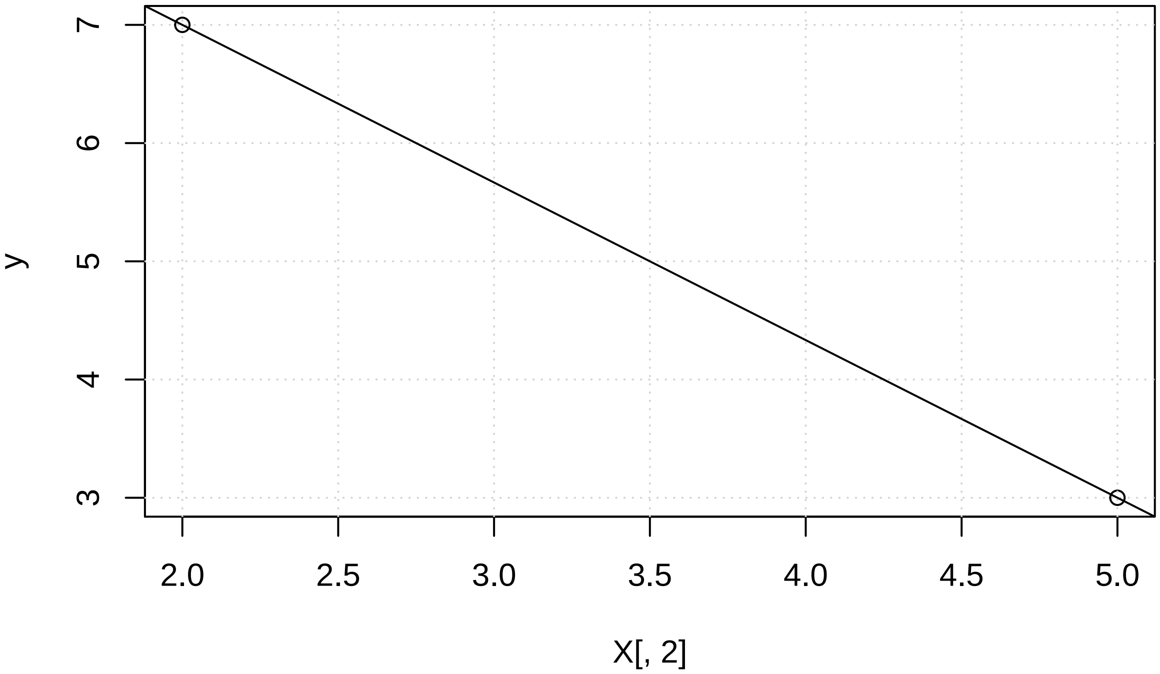
Computing the coefficients is now super easy:

```
beta <- Xinv %*% y
print(beta)
```

```
##           [,1]
## [1,]  9.666667
## [2,] -1.333333
```

Lets make sure that this line is in fact correct and goes through both of our data points.

```
plot(X[,2],y); grid()
abline(beta[1], beta[2])
```



Does the built-in linear regression give us the same result?

```
lm(y ~ X[,2])$coeff
```

```
## (Intercept)      X[, 2]
##    9.666667   -1.333333
```

```
c(beta)
```

```
## [1]  9.666667 -1.333333
```

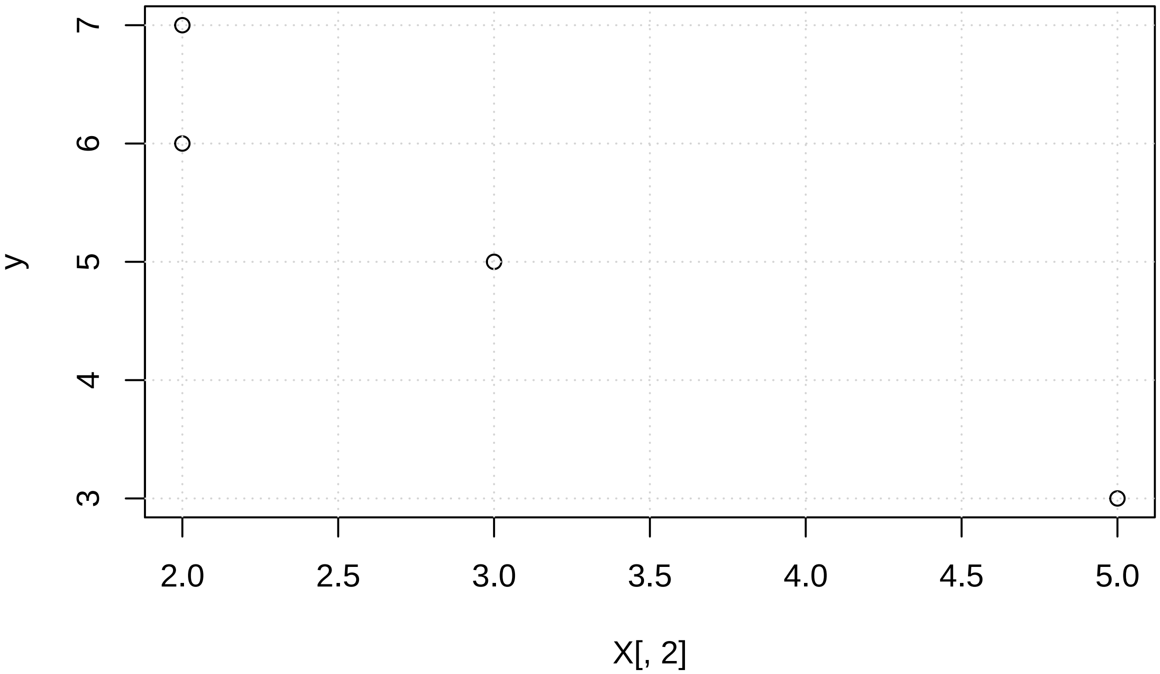
Yes! The results are the same.

Questions:

1. How can we compute the the parameters β when there is only a single data point?
2. How about when there are more data points than features?

OK, lets try 4 data points.

```
X <- rbind(c(1,2),
           c(1,5),
           c(1,3),
           c(1,2))
y <- c(7,3,5,6)
plot(X[,2],y); grid()
```



It does not look like one can find a line through these points. The system of linear equations will not have a solution. The error will happen when we try to invert the new design matrix.

```
# > solve(X)
# Fails with: Error in solve.default(X) : 'a' (4 x 2) must be square
```

The answer is to find the line that will minimize the RSS.

Minimizing RSS

Recall that RSS is the residual sum of squares:

$$\text{RSS} = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

It would be nice to be able to write it in a form of linear algebra. There is actually a tool for this called the L_2-norm or the *Euclidean distance*:

$$\|z\|_2^2 = \sum_{i=1}^{n} z_i^2 = z^T z$$

Using linear algebra, the RSS can be written much more compactly.

$$\text{RSS} = \|y - X\beta\|_2^2 = (y - X\beta)^T(y - X\beta) = y^T y - 2y^T X\beta + \beta^T X^T X\beta$$

Linear regression chooses β to minimize the RSS and thus we have to solve the following optimization problem.

$$\min_{\beta} \|y - X\beta\|_2^2$$

Luckily, this is a convex minimization problem. All we have to do is to look for a value of β in which the gradient is zero.

$$\begin{aligned}
\nabla_\beta \, \|y - X\beta\|_2^2 &= 0 \\
\nabla_\beta \left(y^T y - 2y^T X\beta + \beta^T X^T X\beta \right) &= 0 \\
\nabla_\beta \left(-2y^T X\beta + \beta^T X^T X\beta \right) &= 0 \\
-2X^T y + 2X^T X\beta &= 0 \\
X^T X\beta &= X^T y \\
\beta &= (X^T X)^{-1} X^T y
\end{aligned}$$

So what if X is not square? It is not a problem. If the dimensions of X are $N \times (K+1)$ then the dimensions of X are $(K+1) \times (K+1)$ which is always a square.

Question: Can any square matrix be inverted?

The implementation of linear regression is now really just a *single line*!

```r
beta <- solve(t(X) %*% X) %*% t(X) %*% y
print(beta)
```

```
##           [,1]
## [1,]  8.750000
## [2,] -1.166667
```

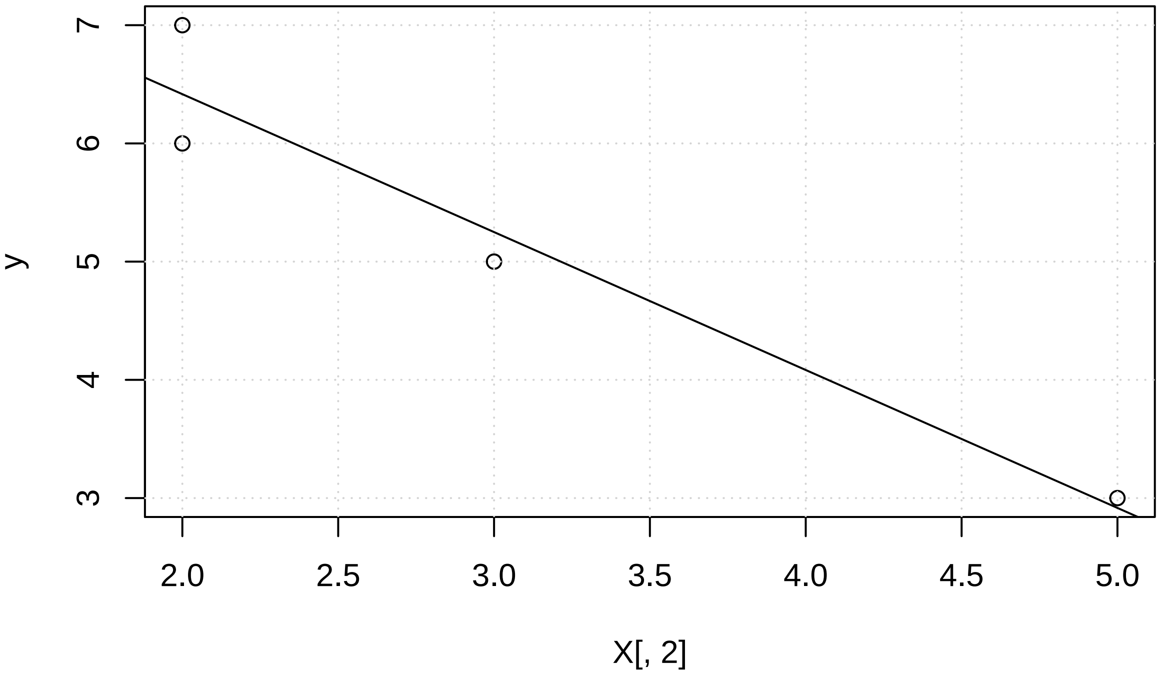
To make sure that everything is OK, we should compare our implementation with the built-in linear regression.

```r
beta_in <- lm(y ~ X[,2])$coeff
print(beta_in)
```

```
## (Intercept)      X[, 2]
##    8.750000   -1.166667
```

And finally, the plot.

```r
plot(X[,2],y); grid()
abline(beta[1], beta[2])
```



Computational Issues

The first rule of numerical linear algebra is: **never compute a matrix inverse**. Computing a matrix inverse is:

1. *Slow*: There are faster ways of solving systems of linear equations
2. *Unstable*: Linear algebra implementation if finite precision can lead to disastrously large errors for ill-conditioned matrices

Luckily, there are many other ways of computing

$$\beta = (X^T X)^{-1} X^T y$$

```r
solve(t(X) %*% X) %*% t(X) %*% y
```

```
##           [,1]
## [1,]  8.750000
## [2,] -1.166667
```

The most common alternatives are:

1. Gaussian elimination: Directly solve the system of linear equation. This is related to how a matrix inverse is often computed, but is faster by about a factor of K and much more numerically stable.

```r
solve(t(X) %*% X, t(X) %*% y)
```

```
##           [,1]
## [1,]  8.750000
## [2,] -1.166667
```

2. Cholesky decomposition (LDL): The idea is that for any *positive-definite* symmetric matrix $A = U^T U$ where U is an upper triangular matrix. Triangular matrices are very easy to invert and the procedure is computationally stable. Matrix $X^T X$ is symmetric and positive definite. Compute the Cholesky decomposition of $X^T X$.

The symmetric matrix is:

```
t(X) %*% X
```

```
##      [,1] [,2]
## [1,]    4   12
## [2,]   12   42
```

The Cholesky decomposition is:

```
U <- chol(t(X) %*% X)
U
```

```
##      [,1]    [,2]
## [1,]    2 6.00000
## [2,]    0 2.44949
```

```
t(U) %*% U
```

```
##      [,1] [,2]
## [1,]    4   12
## [2,]   12   42
```

The linear regression can now be expressed as:

```
chol2inv(U) %*% t(X) %*% y
```

```
##           [,1]
## [1,]  8.750000
## [2,] -1.166667
```

3. QR decomposition: Any matrix can be decomposed to $A = QR$ where Q is an *orthogonal matrix* and R is upper triangular. Orthogonal matrix satisfies $Q^Q = I$. The transpose of Q is also its inverse. QR is also stable and fast. We do not need to even compute X^TX but instead compute the QR decomposition of X.

When $X = QR$ then

$$X^TX = R^TQ^TQR = R^TR$$

```
qr.R(qr(X))
```

```
##      [,1]     [,2]
## [1,]   -2 -6.00000
## [2,]    0 -2.44949
```

Notice that R is the same as U from the Cholesky decomposition and is easier to compute.

```
R <- qr.R(qr(X))
chol2inv(R) %*% t(X) %*% y
```

```
##           [,1]
## [1,]  8.750000
## [2,] -1.166667
```

Column view of linear regression

Another way to view linear regression is a computing linear combination of the columns. Let X_i be the vector that represent the feature i for all samples. Then we are looking for a function that minimizes RSS for a linear combination of the feature vectors and the target.

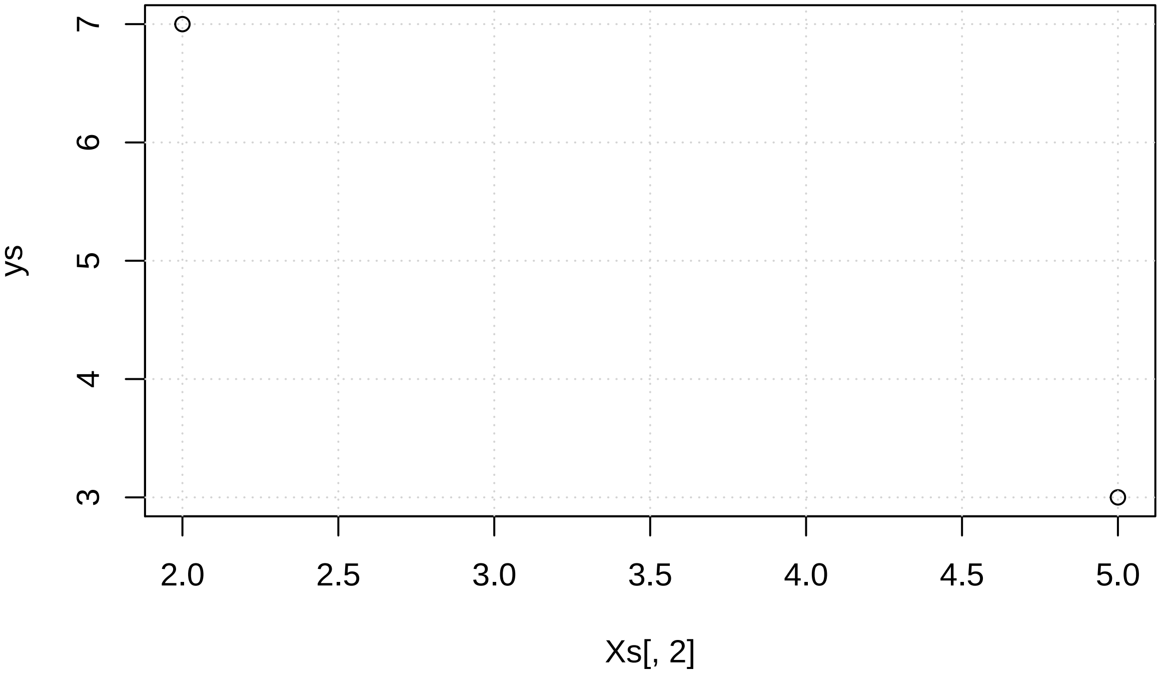
$$\min_{\beta} \|y - X_1\beta_1 - \ldots - X_K\beta_K\|_2^2$$

Before discussing some benefits, lets visualize the simple example from before.

```
Xs <- rbind(c(1,2),
            c(1,5))
ys <- c(7,3)
```

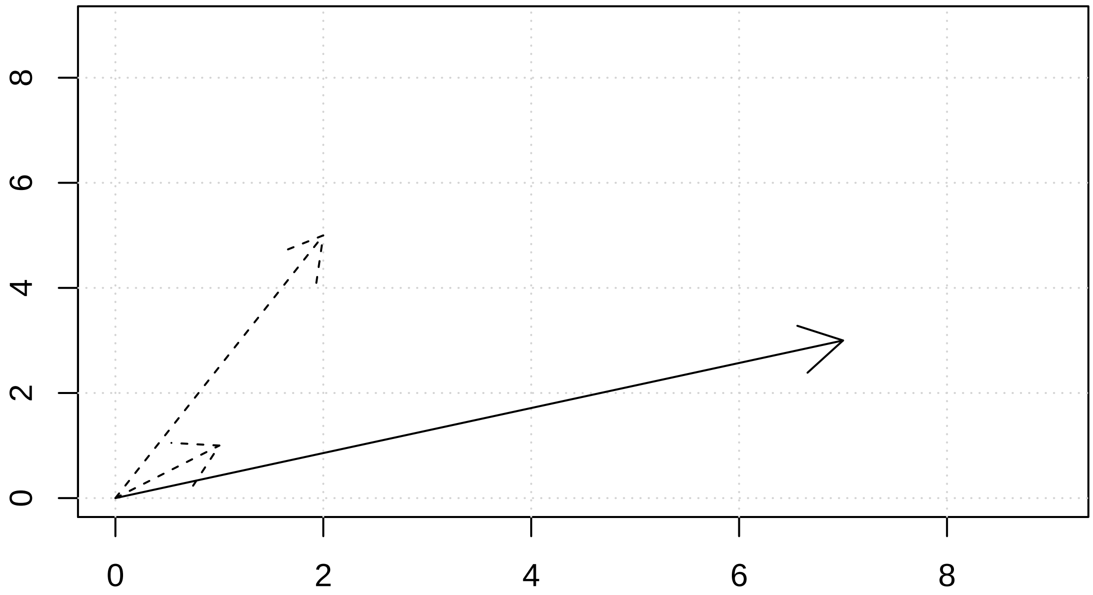
The standard row view looks at each row as a data point. This is the plot (ignoring the intercept feature). The goal is again to connect the two points using a line.

```
plot(Xs[,2],ys); grid()
```



The column view looks at each feature as a *vector*. Now, we include the intercept feature and get three vectors, including y. The goal is to linearly combine the dashed vectors to get the solid one.

```
plot(NULL, xlab="", ylab="", xlim=c(0,9), ylim=c(0,9)); grid();
arrows(0,0,Xs[1,1], Xs[2,1],lty=2)
arrows(0,0,Xs[1,2], Xs[2,2],lty=2)
arrows(0,0,ys[1],ys[2])
```



This view can help to see, for example, that adding a feature that is linearly dependent will not reduce the RSS. As an example, consider our previous design matrix X and add another feature.

```
Y <- cbind(X, c(3,6,5,8))
Y
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    1    5    6
## [3,]    1    3    5
## [4,]    1    2    8
```

Compute the coefficients of linear regression (make sure to remove the intercept):

```
beta <- lm(y ~ Y - 1)$coeff
beta
```

```
##         Y1         Y2         Y3
##  9.6590909 -1.1363636 -0.1818182
```

Lets verify that the numbers really do add up. First the matrix of the values is:

```
sapply(1:3, function(i) {beta[i] * Y[,i]})
```

```
##          [,1]      [,2]       [,3]
## [1,] 9.659091 -2.272727 -0.5454545
## [2,] 9.659091 -5.681818 -1.0909091
## [3,] 9.659091 -3.409091 -0.9090909
## [4,] 9.659091 -2.272727 -1.4545455
```

```
rowSums(sapply(1:3, function(i) {beta[i] * Y[,i]}))
```

```
## [1] 6.840909 2.886364 5.340909 5.931818
```

```
y
```

```
## [1] 7 3 5 6
```

Does RSS decrease when we add a feature that is a linear combination of the others?

```
Z <- cbind(Y, Y[,2] + Y[,3])
Z
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    5
## [2,]    1    5    6   11
## [3,]    1    3    5    8
## [4,]    1    2    8   10
```

Nope, it does not decrease the error at all.

```
summary(lm(y ~ Z - 1))$r.squared
```

```
## [1] 0.9986631
```

```
summary(lm(y ~ Y - 1))$r.squared
```

```
## [1] 0.9986631
```

PCA

PCA is all about Normal distributions and *covariance matrices*. First, lets look at some examples of points generated from a normal distribution with different covariance matrices in 2 dimensions. That means that there are two features. To keep things simple, we will just assume that the mean is 0.

```
mu <- c(0,0)
```

The simplest covariance matrix is just an identity matrix

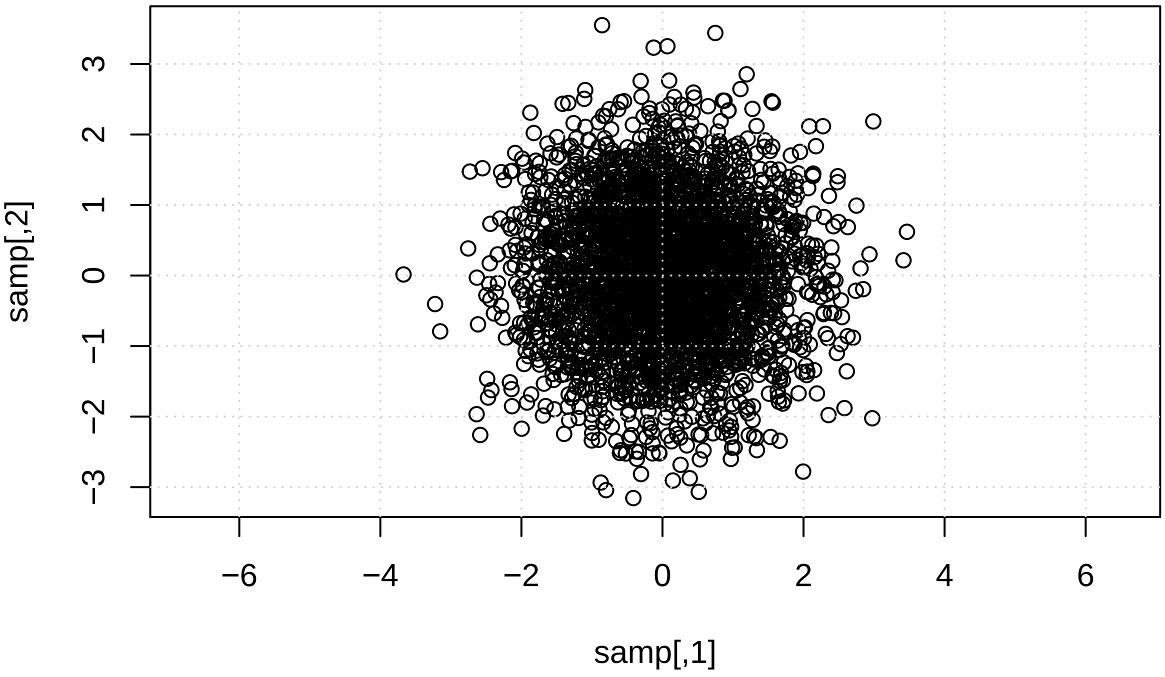
```
Sigma <- rbind(c(1,0),
               c(0,1))
```

Lets sample from this distribution. The result will look very much like the design matrix with rows corresponding to data points and columns corresponding to features.

```
library(MASS)
samp <- mvrnorm(10, mu = mu, Sigma = Sigma)
samp
```

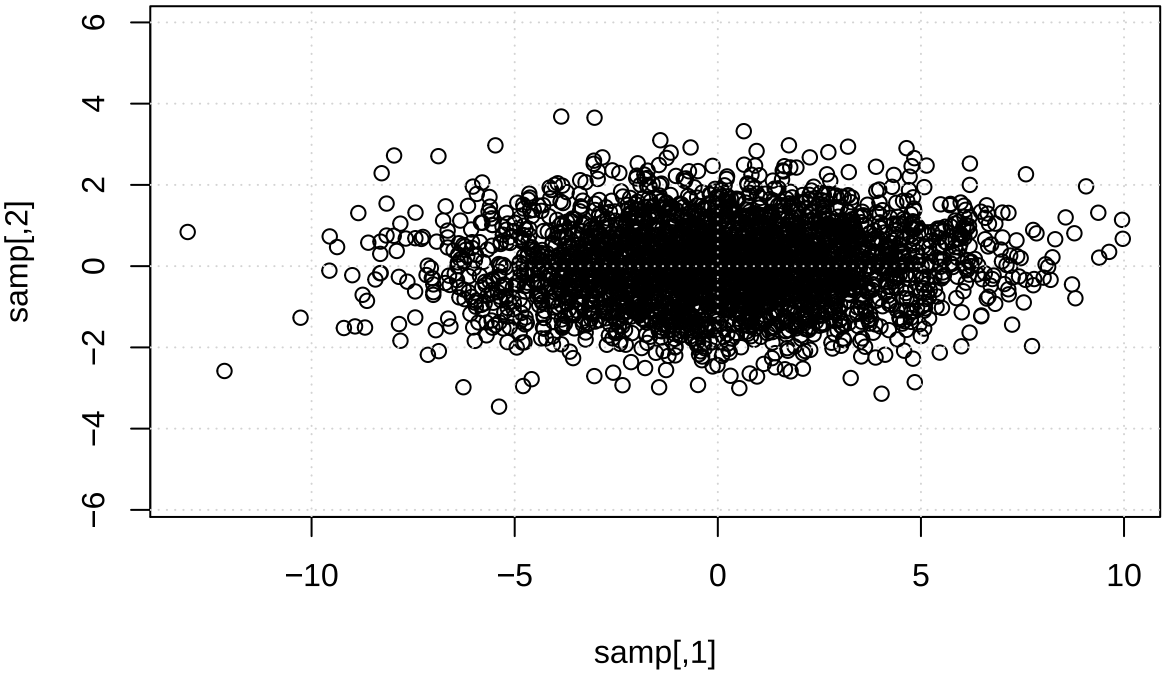
```
##              [,1]        [,2]
##  [1,]  0.54608931  3.58380951
##  [2,] -0.03560938  0.15350915
##  [3,]  1.55933848  0.07830426
##  [4,]  0.99593476  0.48335402
##  [5,] -1.50163437 -1.18847234
##  [6,] -0.79608857 -0.23343491
##  [7,] -0.19858881 -0.48535757
##  [8,]  0.01340952  1.93113804
##  [9,]  1.14539518  0.23447336
## [10,] -0.69755906 -0.26346883
```

```
samp <- mvrnorm(3000, mu = mu, Sigma = Sigma)
plot(samp, type="p", asp=1); grid()
```

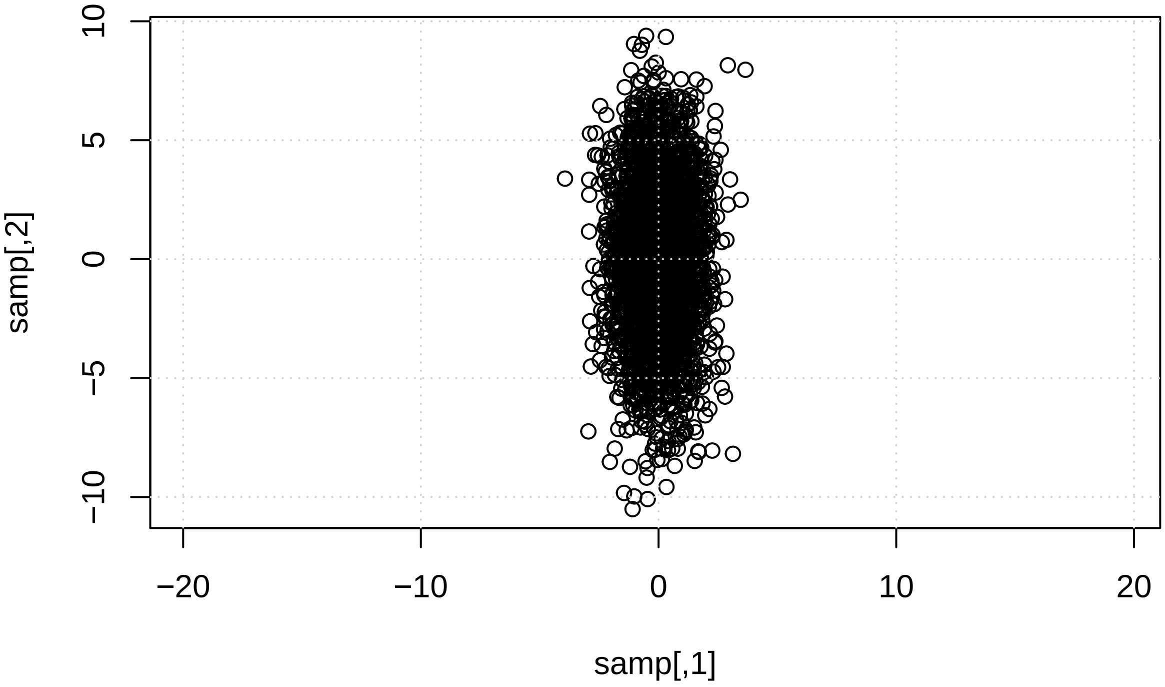


What happens when we choose a different matrix?

```
Sigma <- rbind(c(10,0),
               c(0,1))
samp <- mvrnorm(3000, mu = mu, Sigma = Sigma)
plot(samp, type="p", asp=1); grid()
```



```
Sigma <- rbind(c(1,0),
               c(0,10))
samp <- mvrnorm(3000, mu = mu, Sigma = Sigma)
plot(samp, type="p", asp=1); grid()
```



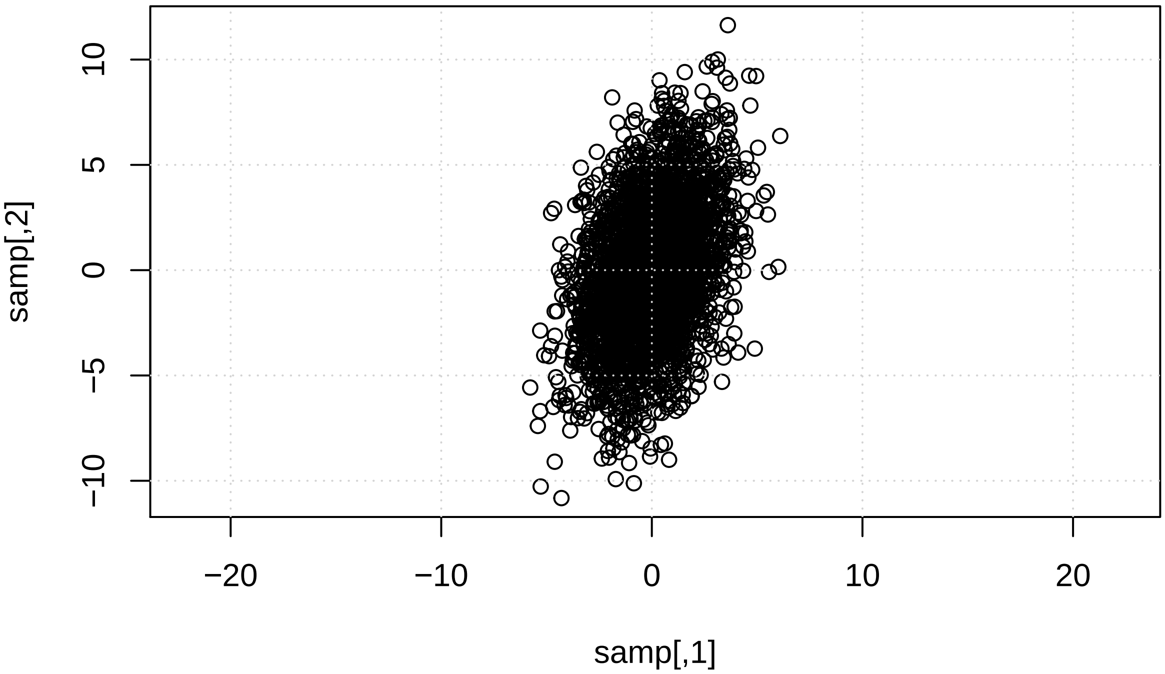
Computing PCA on this data is very simple – it is the axis with the highest variance and there are only to choose from.

```
prcomp(samp)
```

```
## Standard deviations:
## [1] 3.1489431 0.9944544
##
## Rotation:
##                 PC1         PC2
## [1,] -0.004902079 0.999987985
## [2,]  0.999987985 0.004902079
```

But what if the data is rotated?

```
Sigma <- rbind(c(3,2),
               c(2,10))
samp <- mvrnorm(3000, mu = mu, Sigma = Sigma)
plot(samp, type="p", asp=1); grid()
```



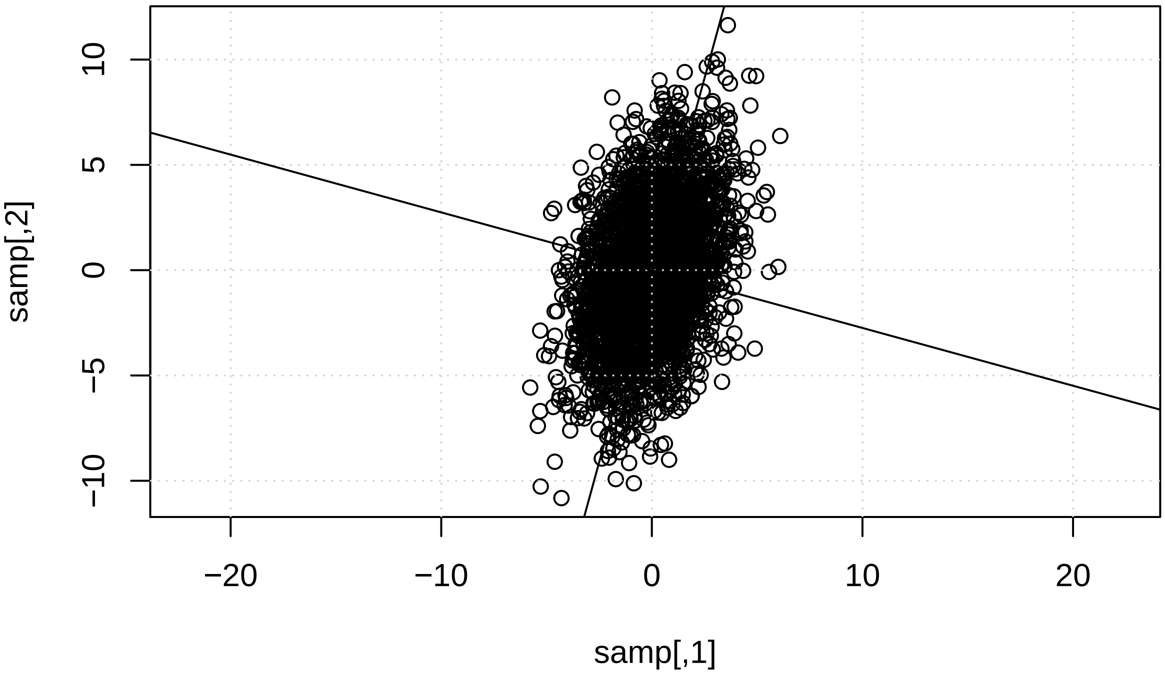
PCA can recover this rotation:

```
prcomp(samp)
```

```
## Standard deviations:
## [1] 3.260602 1.570133
##
## Rotation:
##              PC1        PC2
## [1,] -0.2645809  0.9643635
## [2,] -0.9643635 -0.2645809
```

Lets check this visually:

```
prc <- prcomp(samp)
plot(samp, type="p", asp=1); grid()
abline(0,prc$rotation[2,1]/prc$rotation[1,1])
abline(0,prc$rotation[2,2]/prc$rotation[1,2])
```



How is it done? In two easy steps.

1. Compute the *covariance matrix* from the data
2. Compute eigenvectors of the matrix. Looking for a linear transformation of the features that will give us a diagonal matrix.

Lets start with the second step. If we have our covariance matrix, we can compute the eigenvalues and eigen-vectors, which satisfy:

$$Ax = \lambda x$$

The eigenvectors can be computed as follows:

```
eigen(Sigma)
```

```
## $values
## [1] 10.531129  2.468871
##
## $vectors
##           [,1]       [,2]
## [1,] 0.2566679 -0.9664996
## [2,] 0.9664996  0.2566679
```

This of eigenvectors as dimensions in which the matrix behaves as diagonal. A very nice property of symmetric matrices (such as covariance matrices) is that their eigenvectors are *orthogonal*. So we can invert a matrix just by transposing it. Now we can diagonalize the matrix using the eigenvectors:

$$V^{-1}\Sigma V = D$$

where D is a diagonal matrix of *eigenvalues* and V is the matrix of *eigenvectors*. Lets see:

```
E = eigen(Sigma)
t(E$vectors) %*% Sigma %*% E$vectors
```

```
##          [,1]     [,2]
## [1,] 10.53113 0.000000
## [2,]  0.00000 2.468871
```

Question: How can we compute the covariance matrix from data?