Chapter 1

The Gaussian Linear Model

Statistics 849 is part of a two-semester sequence, 849 & 850, on *Theory and Applications of Regression and Analysis of Variance*. In these courses we study statistical models that relate a *response* to values of one or more *covariates*, which are variables that are observed in conjunction with the response.

The statistical inferences are based on a probability model that characterizes the distribution of the vector-valued *random variable*, $\mathcal{Y}$, as it depends on values of the covariates. We build the model based on observed values of the responses, represented by the vector $\boldsymbol{y}$, and corresponding values of the covariates.

All the models we will study are based on a *linear predictor* expression, $\boldsymbol{X\beta}$, where the $n \times p$ matrix $\boldsymbol{X}$ is the *model matrix* created from a model specification and the values of the covariates. Here n is the number of observations and p is the dimension of the *coefficient vector*, $\boldsymbol{\beta}$, The coefficients are *parameters* in the model. We form *estimates*, $\widehat{\boldsymbol{\beta}}$, of these parameters from the observed data.

We assume that $n \geq p$. That is, we have at least as many observations are we have coefficients in the model.

1.1 Gaussian Linear Model

A basic model for a response, $\mathcal{Y}$, that is measured on a continuous scale, is the *Gaussian Linear Model*

$$\mathcal{Y} \sim \mathcal{N}(\boldsymbol{X\beta}_T, \sigma^2 \boldsymbol{I}_n) \tag{1.1}$$

where $\boldsymbol{I}_n$ is the n-dimensional identity matrix, $\boldsymbol{\beta}_T$ is the "true", but unknown, value of the coefficient vector and $\mathcal{N}$ denotes the multivariate Gaussian (also called *normal*) distribution.

The probability density of $\mathcal{Y}$,

$$f_{\mathcal{Y}}(\boldsymbol{y}) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left(\frac{-\|\boldsymbol{y} - \boldsymbol{X\beta}_T\|^2}{2\sigma^2}\right), \tag{1.2}$$

is called a *spherical normal* density, because contours of constant density are concentric spheres centered at $\boldsymbol{X\beta}_T$.

The *likelihood*, $L(\boldsymbol{\beta}, \sigma|\boldsymbol{y})$, of the parameters, $\boldsymbol{\beta}$ and σ, given the observed responses, $\boldsymbol{y}$, and the model matrix, $\boldsymbol{X}$, is the same expression as the probability density, $f_{\mathcal{Y}}(\boldsymbol{y})$, but regarded as a function of the parameters given the data, as opposed to the density, which is a function of $\boldsymbol{y}$ for known values of the parameters.

$$L(\boldsymbol{\beta}, \sigma|\boldsymbol{y}) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left(\frac{-\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2}{2\sigma^2}\right). \tag{1.3}$$

The *maximum likelihood estimates* (*mle*s) of the parameters are, as the name suggests, the values of the parameters that maximize the likelihood

$$\left(\widehat{\boldsymbol{\beta}}', \widehat{\sigma}_L\right)' = \arg\max_{\boldsymbol{\beta},\sigma} L(\boldsymbol{\beta}, \sigma|\boldsymbol{y}) \tag{1.4}$$

As often happens it is much easier to maximize the expression for the *log-likelihood*

$$\ell(\boldsymbol{\beta}, \sigma|\boldsymbol{y}) = \log\left(L(\boldsymbol{\beta}, \sigma|\boldsymbol{y})\right) = -\frac{n}{2}\log(2\pi\sigma^2) - \frac{\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2}{2\sigma^2} \tag{1.5}$$

than to maximize the likelihood. Because the logarithm function is monotonic increasing, the values of the parameters that maximize the log-likelihood, written $\arg\max_{\beta,\sigma} \ell(\boldsymbol{\beta}, \sigma|\boldsymbol{y})$, are exactly the same as the values that maximize the likelihood, $\arg\max_{\beta,\sigma} L(\boldsymbol{\beta}, \sigma|\boldsymbol{y})$.

The expression can be simplified further by converting to the *deviance*, which is negative twice the log-likelihood,

$$d(\boldsymbol{\beta}, \sigma|\boldsymbol{y}) = -2\ell(\boldsymbol{\beta}, \sigma|\boldsymbol{y}) = n\log(2\pi\sigma^2) + \frac{\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2}{\sigma^2}. \tag{1.6}$$

Because of the negative sign, the mle's are the values that minimize the deviance. For any fixed value of σ^2, the deviance is minimized with respect to $\boldsymbol{\beta}$ when the *residual sum of squares*,

$$S(\boldsymbol{\beta}|\boldsymbol{y}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2,$$

is minimized. Thus the mle of the coefficient vector, $\widehat{\boldsymbol{\beta}}$, in the Gaussian linear model is the *least squares estimate*

$$\widehat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2. \tag{1.7}$$

1.2 Linear algebra of least squares

Because the Gaussian Linear Model, $\mathcal{Y} \sim \mathcal{N}(\boldsymbol{X}\boldsymbol{\beta}_T, \sigma^2\boldsymbol{I}_n)$, is intimately tied to the Euclidean distance, $\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2$, and because the set of all possible fitted values, $\{\boldsymbol{X}\boldsymbol{\beta} : \boldsymbol{\beta} \in \mathbb{R}^p\}$, which is called the *column span* of $\boldsymbol{X}$ and written $\text{col}(\boldsymbol{X})$, is a linear subspace of $\mathbb{R}^n$, linear algebra, especially as related to the model matrix, $\boldsymbol{X}$, and the response vector, $\boldsymbol{y}$, is fundamental to the theory and practice of linear regression analysis.

We will concentrate on the theoretical and computational aspects of linear algebra as related to the linear model and the implementation of such models in R.

1.3 Matrix decompositions

1.3.1 Orthogonal matrices

An *orthogonal* $n \times n$ matrix, $\boldsymbol{Q}$ has the property that its transpose is its inverse,

$$\boldsymbol{Q}'\boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{Q}' = \boldsymbol{I}_n.$$

These properties imply that the columns of $\boldsymbol{Q}$ must be orthogonal to each other and must all have unit length. The same is true for the rows.

An orthogonal matrix has a special property that it preserves lengths.

Preserving lengths

For any $\boldsymbol{x} \in \mathbb{R}^n$

$$\|\boldsymbol{Q}\boldsymbol{x}\|^2 = (\boldsymbol{Q}\boldsymbol{x})'\boldsymbol{Q}\boldsymbol{x} = \boldsymbol{x}'\boldsymbol{Q}'\boldsymbol{Q}\boldsymbol{x} = \boldsymbol{x}'\boldsymbol{x} = \|\boldsymbol{x}\|^2$$

Thus the linear transformation determined by $\boldsymbol{Q}$ or by $\boldsymbol{Q}'$ must be a *rigid* transformation, composed of reflections or rotations.

Orthogonal transformations of the response space, $\mathbb{R}^n$, will be important to us because they preserve lengths and because the likelihood of the parameters, $\boldsymbol{\beta}$, is related to the squared length of the residual vector, $\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2$.

1.3.2 The QR decomposition

Any $n \times p$ matrix $\boldsymbol{X}$ has a QR decomposition consisting of an orthogonal $n \times n$ matrix $\boldsymbol{Q}$ and a $p \times p$ matrix $\boldsymbol{R}$ that is zero below the main diagonal (in other words, it is *upper triangular*). The QR decomposition of the model matrix $\boldsymbol{X}$ is written

$$\boldsymbol{X} = \boldsymbol{Q}\begin{bmatrix}\boldsymbol{R}\\ \boldsymbol{0}\end{bmatrix} = \begin{bmatrix}\boldsymbol{Q}_1 & \boldsymbol{Q}_2\end{bmatrix}\begin{bmatrix}\boldsymbol{R}\\ \boldsymbol{0}\end{bmatrix} = \boldsymbol{Q}_1\boldsymbol{R} \tag{1.8}$$

where $\boldsymbol{Q}_1$ is the first p columns of $\boldsymbol{Q}$ and $\boldsymbol{Q}_2$ is the last $n - p$ columns of $\boldsymbol{Q}$.

That fact that matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ must exist is proved by construction. The matrix $\boldsymbol{Q}$ is the product of p *Householder reflections* (see the Wikipedia page for the QR decomposition). The process of generating the upper triangular matrix $\boldsymbol{R}$ is similar to the Gram-Schmidt orthogonalization process, but more flexible and more numerically stable. If the diagonal elements of $\boldsymbol{R}$ are all non-zero (in practice this means that none of them are very small in absolute value) then $\boldsymbol{X}$ has *full column rank* and the columns of $\boldsymbol{Q}_1$ form an *orthonormal basis* for $\text{col}(\boldsymbol{X})$.

The implementation of the QR decomposition in R guarantees that any elements on the diagonal of $\boldsymbol{R}$ that are considered effectively zero are rearranged by column permutation to occur in the trailing columns. That is, if the rank of $\boldsymbol{X}$ is $k < p$ then the first k columns of $\boldsymbol{Q}$ form an orthonormal basis for $\text{col}(\boldsymbol{X}\boldsymbol{P})$ where $\boldsymbol{P}$ is a $p \times p$ permutation matrix, which means that it is a rearrangement of the columns of $\boldsymbol{I}_p$.

Our text often mentions rank-deficient cases where $\text{rank}(\boldsymbol{X}) = k < p$. In practice the rank deficient case rarely occurs because the process of building the model matrix in R involves a considerable amount of analysis of the model formula to remove the most common cases of rank deficiency. Nevertheless, rank deficiency can occur and is detected and handled in the `lm` function in R.

Because multiplication by an orthogonal matrix like $\boldsymbol{Q}'$ preserves lengths we can write

$$\begin{aligned}\widehat{\boldsymbol{\beta}} &= \arg\min_{\beta} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 \\ &= \arg\min_{\beta} \|\boldsymbol{Q}'(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})\|^2 \\ &= \arg\min_{\beta} \|\boldsymbol{Q}'\boldsymbol{y} - \boldsymbol{Q}'\boldsymbol{X}\boldsymbol{\beta}\|^2 \\ &= \arg\min_{\beta} \|\boldsymbol{c}_1 - \boldsymbol{R}\boldsymbol{\beta}\|^2 + \|\boldsymbol{c}_2\|^2\end{aligned} \tag{1.9}$$

where $\boldsymbol{c}_1 = \boldsymbol{Q}_1'\boldsymbol{y}$ is the first p elements of $\boldsymbol{Q}'\boldsymbol{y}$ and $\boldsymbol{c}_2 = \boldsymbol{Q}_2'\boldsymbol{y}$ is the last $n - p$ elements. If $\text{rank}(\boldsymbol{X}) = p$ then $\text{rank}(\boldsymbol{R}) = p$ and $\boldsymbol{R}^{-1}$ exists so we can write $\widehat{\boldsymbol{\beta}} = \boldsymbol{R}^{-1}\boldsymbol{c}_1$ (although you don't actually calculate $\boldsymbol{R}^{-1}$ to solve the triangular linear system $\boldsymbol{R}\widehat{\boldsymbol{\beta}} = \boldsymbol{c}_1$ for $\widehat{\boldsymbol{\beta}}$).

In a model fit by the `lm()` or `aov()` functions in R there is a component `$effects` which is $\boldsymbol{Q}'\boldsymbol{y}$. The component `$qr` is a condensed form of the QR decomposition of the model matrix $\boldsymbol{X}$. The matrix $\boldsymbol{R}$ is embedded in there but the matrix $\boldsymbol{Q}$ is a virtual matrix represented as a product of Householder reflections and not usually evaluated explicitly.

R Exercise: To see this theory in action, we will start with a very simple linear model. The `Formaldehyde` data are six observations from a calibration experiment. The response, `optden`, is the optical density. Only one covariate, `carb`, which is the carbohydrate concentration, is included in the data frame.

```
> str(Formaldehyde)

'data.frame':        6 obs. of  2 variables:
 $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
```

The `model.matrix()` function extracts the model matrix, $\boldsymbol{X}$, from a fitted linear model object

```
> (X <- model.matrix(lm1 <- lm(optden ~ 1 + carb, Formaldehyde)))

  (Intercept) carb
1           1  0.1
2           1  0.3
3           1  0.5
4           1  0.6
5           1  0.7
6           1  0.9
attr(,"assign")
[1] 0 1
```

The `$qr` component is an object of class `"qr"`

```
> class(qrlm1 <- lm1$qr)
```

```
[1] "qr"
```

for which there are many extractor functions and methods (see `?qr`).

```
> (R <- qr.R(qrlm1))
```

```
  (Intercept)       carb
1   -2.449490 -1.2655697
2    0.000000  0.6390097
```

produces the $\boldsymbol{R}$ matrix while

```
> (Q1 <- qr.Q(qrlm1))
```

```
           [,1]        [,2]
[1,] -0.4082483 -0.65205066
[2,] -0.4082483 -0.33906635
[3,] -0.4082483 -0.02608203
[4,] -0.4082483  0.13041013
[5,] -0.4082483  0.28690229
[6,] -0.4082483  0.59988661
```

by default produces $\boldsymbol{Q}_1$. First we should check that their product is indeed $\boldsymbol{X}$

```
> (Q1R <- Q1 %*% R)
```

```
     (Intercept) carb
[1,]           1  0.1
[2,]           1  0.3
[3,]           1  0.5
[4,]           1  0.6
[5,]           1  0.7
[6,]           1  0.9
```

It seems to be the same, although as in all floating point calculations on a computer, there may be some small imprecision caused by round-off error in the calculations. This is why we don't use exact comparisons on the results of floating point calculations

```
> all(X == Q1R)
```

```
[1] FALSE
```

but instead compare results using

```
> all.equal(X, Q1R, check.attr = FALSE)
```

```
[1] TRUE
```

(As a model matrix, `X` has some additional attributes that are not present in the product `Q1R`, which is why we turn off checking of the attributes of the two objects.)

Notice that $\boldsymbol{X}$ and $\boldsymbol{y}$ are not explicitly part of the fitted model object, `lm1`.

```
> names(lm1)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"          "fitted.values"
 [6] "assign"        "qr"            "df.residual"   "xlevels"       "call"
[11] "terms"         "model"
```

Both are generated from the *model.frame*, which is stored as the component `$model`. Althought this is getting into more detail than is needed at present, the reason for introducing the model frame is to say that the safe way of extracting the response vector, $\boldsymbol{y}$, is

```
> (y <- model.response(model.frame(lm1)))
```

```
    1     2     3     4     5     6
0.086 0.269 0.446 0.538 0.626 0.782
```

We have already seen that `model.matrix` returns the matrix $\boldsymbol{X}$ from the fitted model object.

We can produce the full $n \times n$ orthogonal matrix $\boldsymbol{Q}$ from `qr.Q()` by setting the optional argument `complete=TRUE`. We do this for illustration only. In practice the matrix $\boldsymbol{Q}$ is never explicitly created — it is a "virtual" matrix in the sense that it is a product of Householder reflections that are stored much more compactly than $\boldsymbol{Q}$ would be stored.

```
> (Q <- qr.Q(qrlm1, complete=TRUE))
```

```
           [,1]        [,2]        [,3]       [,4]       [,5]       [,6]
[1,] -0.4082483 -0.65205066 -0.37370452 -0.3405290 -0.3073534 -0.2410023
[2,] -0.4082483 -0.33906635  0.05460995  0.2207196  0.3868293  0.7190487
[3,] -0.4082483 -0.02608203  0.86857638 -0.1439791 -0.1565346 -0.1816455
[4,] -0.4082483  0.13041013 -0.15359661  0.8125532 -0.2212971 -0.2889976
[5,] -0.4082483  0.28690229 -0.17576960 -0.2309146  0.7139404 -0.3963496
[6,] -0.4082483  0.59988661 -0.22011559 -0.3178501 -0.4155847  0.3889463
```

The `$effects` vector should be the product $\boldsymbol{Q}'\boldsymbol{y}$ but it happens that they are stored differently. The `$effects` vector is a vector of length n and the product $\boldsymbol{Q}'\boldsymbol{y}$ is an $n \times 1$ matrix. To compare them, we need to make `$effects` an $n \times 1$ matrix or make $\boldsymbol{Q}'\boldsymbol{y}$ into a vector. A convenient way of making an $n \times 1$ matrix from an n-vector is the function `cbind()`, which creates matrices or data frames by binding columns together. If we give it a single vector argument it creates an $n \times 1$ matrix

```
> str(cbind(lm1$effects))
```

```
 num [1:6, 1] -1.12146 0.55996 0.00514 0.00992 0.01069 ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:6] "(Intercept)" "carb" "" "" ...
  ..$ : NULL

> str(crossprod(Q, y))

 num [1:6, 1] -1.12146 0.55996 0.00514 0.00992 0.01069 ...

> all.equal(cbind(lm1$effects), crossprod(Q, y), check.attr=FALSE)

[1] TRUE
```

(The function `crossprod(A,B)` creates $\boldsymbol{A}'\boldsymbol{B}$ directly, without creating $\boldsymbol{A}'$ from $\boldsymbol{A}$. It is most commonly used to create matrices like $\boldsymbol{X}'\boldsymbol{X}$ as

```
> crossprod(X)

            (Intercept) carb
(Intercept)         6.0 3.10
carb                3.1 2.01
```

The companion function, `tcrossprod`, creates $\boldsymbol{X}\boldsymbol{X}'$.)

If we wish to do the comparison by converting $\boldsymbol{Q}'\boldsymbol{y}$ to a vector, we can use

```
> all.equal(lm1$effects, as.vector(crossprod(Q, y)), check.attr=FALSE)

[1] TRUE
```

I find `cbind` easier to type than `as.vector`.

Another way of generating $\boldsymbol{Q}'\boldsymbol{y}$ is with the function `qr.qty()`

```
> all.equal(lm1$effects, qr.qty(qrlm1, y), check.attr=FALSE)

[1] TRUE
```

We should check that $\boldsymbol{Q}$ is indeed orthogonal and that $\boldsymbol{Q}_1'\boldsymbol{Q}_1 = \boldsymbol{I}_p$. The matrix $\boldsymbol{I}_k$ is generated by `diag(nrow=k)`.

```
> all.equal(crossprod(Q1), diag(nrow=ncol(Q1)))

[1] TRUE

> all.equal(crossprod(Q), diag(nrow=nrow(Q)))

[1] TRUE

> all.equal(tcrossprod(Q), diag(nrow=nrow(Q)))
```

```
[1] TRUE
```

When we print a matrix that may have negligibly small non-zero values in it

```
> crossprod(Q1)
```

```
             [,1]         [,2]
[1,] 1.000000e+00 1.197637e-16
[2,] 1.197637e-16 1.000000e+00
```

we can clean up the output with `zapsmall()` which, as the name suggests, zeros the very small values.

```
> zapsmall(crossprod(Q1))
```

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

```
> zapsmall(crossprod(Q))
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    0    1    0    0    0    0
[3,]    0    0    1    0    0    0
[4,]    0    0    0    1    0    0
[5,]    0    0    0    0    1    0
[6,]    0    0    0    0    0    1
```

Because the diagonal elements of $\boldsymbol{R}$ are all safely non-zero, we can solve the system $\boldsymbol{R}\widehat{\boldsymbol{\beta}} = \boldsymbol{Q}_1'\boldsymbol{y}$ for the coefficient estimates, $\widehat{\boldsymbol{\beta}}$. We could use the function `solve()` to do this but it us better to use `backsolve()` for the solution to upper triangular systems,

```
> coef(lm1)
```

```
(Intercept)        carb
0.005085714 0.876285714
```

```
> backsolve(R, crossprod(Q1, y))
```

```
            [,1]
[1,] 0.005085714
[2,] 0.876285714
```

```
> all.equal(coef(lm1), as.vector(backsolve(R, crossprod(Q1, y))), check.attr=FALSE)
```

```
[1] TRUE
```

The function `qr.coef` combines the multiplication of $\boldsymbol{y}$ by $\boldsymbol{Q}_1'$ and the backsolve step

```
> qr.coef(qrlm1, y)
```

```
(Intercept)        carb
0.005085714 0.876285714
```

R Exercise: As seen above, a linear model is specified as a model formula and the data frame in which to evaluate the formula. Because the formula is analyzed for conditions that may introduce rank deficiency and consequently removes those conditions, rank deficient cases occur infrequently. Of course, it is possible to artificially generate data with a built-in rank dependency

```
> set.seed(1234)                      # allow for reproducible "random" numbers
> badDat <- within(data.frame(x1=1:20, x2=rnorm(20,mean=6,sd=0.2),
+                             x4=rexp(20,rate=0.02),
+                             y=runif(20,min=18,max=24)),
+                  x3 <- x1 + 2*x2)   # create linear combination
> (summary(lm2 <- lm(y ~ x1 + x2 + x3 + x4, badDat)))


Call:
lm(formula = y ~ x1 + x2 + x3 + x4, data = badDat)
Residuals:
    Min      1Q  Median      3Q     Max
-2.3444 -1.7670 -0.3585  1.6159  3.0292

Coefficients: (1 not defined because of singularities)
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.793e+01  1.390e+01   1.290    0.215
x1          4.140e-02  8.553e-02   0.484    0.635
x2          3.822e-01  2.358e+00   0.162    0.873
x3                 NA         NA      NA       NA
x4          3.901e-05  8.680e-03   0.004    0.996

Residual standard error: 2.02 on 16 degrees of freedom
Multiple R-squared: 0.021,        Adjusted R-squared: -0.1626
F-statistic: 0.1144 on 3 and 16 DF,  p-value: 0.9504

> (lm2qr <- lm2$qr)$rank

[1] 4

> qr.R(lm2qr)                         # the columns are rearranged

  (Intercept)        x1          x2         x4            x3
1   -4.472136 -46.95743 -26.6086150 -182.87421 -1.001747e+02
2    0.000000  25.78759   0.1721295   83.85993  2.613185e+01
3    0.000000   0.00000  -0.8668932   35.62409 -1.733786e+00
4    0.000000   0.00000   0.0000000  232.75139  2.005660e-15
5    0.000000   0.00000   0.0000000    0.00000  5.179752e-15

> lm2qr$pivot                         # the permutation vector

[1] 1 2 3 5 4
```

but we don't do this in practice.

The common case of an analysis of variance model which, when written as

$$y_{i,j} = \mu + \alpha_i + \epsilon_{i,j} \quad i = 1, \ldots, I,\ j = 1, \ldots, n_i$$

would generate linearly dependent columns for μ and the α_i, $i = 1, \ldots, I$ is analyzed and represented by the intercept column and $I - 1$ columns for the factor.

```
> str(InsectSprays)

'data.frame':        72 obs. of  2 variables:
 $ count: num  10 7 20 14 14 12 10 23 17 20 ...
 $ spray: Factor w/ 6 levels "A","B","C","D",..: 1 1 1 1 1 1 1 1 1 1 ...

> unique(mm <- model.matrix(lm3 <- lm(count ~ spray, InsectSprays)))

   (Intercept) sprayB sprayC sprayD sprayE sprayF
1            1      0      0      0      0      0
13           1      1      0      0      0      0
25           1      0      1      0      0      0
37           1      0      0      1      0      0
49           1      0      0      0      1      0
61           1      0      0      0      0      1

> attr(mm, "assign")

[1] 0 1 1 1 1 1

> qr.R(lm3[["qr"]])

  (Intercept)    sprayB     sprayC     sprayD     sprayE     sprayF
1   -8.485281 -1.414214 -1.4142136 -1.4142136 -1.4142136 -1.4142136
2    0.000000  3.162278 -0.6324555 -0.6324555 -0.6324555 -0.6324555
3    0.000000  0.000000  3.0983867 -0.7745967 -0.7745967 -0.7745967
4    0.000000  0.000000  0.0000000  3.0000000 -1.0000000 -1.0000000
5    0.000000  0.000000  0.0000000  0.0000000  2.8284271 -1.4142136
6    0.000000  0.000000  0.0000000  0.0000000  0.0000000  2.4494897
```

This shows only the unique rows in the model matrix. The six levels of the `spray` factor are represented by 5 indicator columns.

Because we are discussing an analysis of variance model we also show the `"assign"` attribute of the model matrix. This indicates that the first column is associated with the 0th term, which is the intercept, and the second through sixth columns are associated with the first term, which is `spray`.

In general, a factor with I levels is converted to a set of $I-1$ columns. These are called *contrasts*, but be warned that these do not fulfill the definition of contrasts as used in some texts. You should think of them as being a set of columns representing changes between levels of the factor.

The type of contrasts generated is controlled by the option called `"contrasts"`.

```
> getOption("contrasts")

        unordered           ordered
"contr.treatment"      "contr.poly"
```

(By the way, plotting these data first would show that this is not a good model. The `count` variable is, not surprisingly, a count and does not have constant variance. A better model would use the square root of the count as a response.)

The determinant of an orthogonal matrix

As described on its Wikipedia page, the *determinant*, $|\boldsymbol{A}|$, of the $k \times k$ square matrix, $\boldsymbol{A}$, is the volume of the parallelepiped spanned by its columns (or, equivalently, the volume spanned by its rows). Because we can consider either the rows or the columns when evaluating the determinant, we must have

$$|\boldsymbol{A}| = |\boldsymbol{A}'|.$$

We can regard $|\boldsymbol{A}|$ as the magnification factor in the transformation $\boldsymbol{x} \rightarrow \boldsymbol{A}\boldsymbol{x}$ from $\mathbb{R}^k$ to $\mathbb{R}^k$. This transformation takes the unit cube to a parallelopiped with volume $|\boldsymbol{A}|$. Composing transformations will just multiply the magnification factors so we must have

$$|\boldsymbol{A}\boldsymbol{B}| = |\boldsymbol{A}|\,|\boldsymbol{B}|$$

We know that the columns of an orthogonal matrix $\boldsymbol{Q}$ are *orthonormal* hence they span a unit volume. That is, for an $n \times n$ matrix $\boldsymbol{Q}$

$$\boldsymbol{Q}'\boldsymbol{Q} = \boldsymbol{I}_n \quad \Rightarrow \quad |\boldsymbol{Q}| = \pm 1.$$

The sign indicates whether the transformation preserves orientation. In two dimensions a rotation preserves orientation and a reflection reverses orientation.

Furthermore, the determinant of a *diagonal* matrix or a *triangular* matrix is simply the product of its diagonal elements. (For a triangular matrix, first consider the 2×2 case and the shape of the parallelogram spanned by the columns. The width of the parallelogram is the (1,1) element and the height is the (2,2) element so the area is the product of the diagonal elements (up to sign). Then convince yourself that this property scales to a parellelopiped in k dimensions.)

From these properties we can formally derive

$$1 = |\boldsymbol{I}_n| = |\boldsymbol{Q}\boldsymbol{Q}'| = |\boldsymbol{Q}||\boldsymbol{Q}'| = |\boldsymbol{Q}|^2 \quad \Rightarrow \quad |\boldsymbol{Q}| = \pm 1.$$

Interestingly, one way that the determinant, $|\boldsymbol{A}|$ is evaluated in practice is by forming the QR decomposition of $\boldsymbol{A}$, taking the product of the diagonal elements of $\boldsymbol{R}$, and determining whether $|\boldsymbol{Q}|$ has a plus or a minus sign.

R Exercise: The `det()` function evaluates the determinant of a square matrix (although if you check its definition you will find that it just calls another function `determinant`, which is the preferred approach).

```
> all.equal(det(R), prod(diag(R)))

[1] TRUE

> det(crossprod(Q1))

[1] 1

> det(crossprod(Q))

[1] 1
```

1.3.3 Comparison to the usual text-book formulas

Most text books state that the least squares estimates are

$$\widehat{\boldsymbol{\beta}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y} \tag{1.10}$$

giving the impression that $\widehat{\boldsymbol{\beta}}$ is calculated this way. It isn't.

If you substitute $\boldsymbol{X} = \boldsymbol{Q}_1\boldsymbol{R}$ in eqn. 1.10 you get

$$(\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y} = (\boldsymbol{R}'\boldsymbol{R})^{-1}\boldsymbol{R}'\boldsymbol{Q}_1'\boldsymbol{y} = \boldsymbol{R}^{-1}(\boldsymbol{R}')^{-1}\boldsymbol{R}'\boldsymbol{Q}_1'\boldsymbol{y} = \boldsymbol{R}^{-1}\boldsymbol{Q}_1'\boldsymbol{y},$$

our previous result.

Whenever you see $\boldsymbol{X}'\boldsymbol{X}$ in a formula you should mentally replace it by $\boldsymbol{R}'\boldsymbol{R}$ and similarly replace $(\boldsymbol{X}'\boldsymbol{X})^{-1}$ by $\boldsymbol{R}^{-1}(\boldsymbol{R}')^{-1}$ then see if you can simplify the result.

For example, the variance of the least squares estimator $\widehat{\boldsymbol{\beta}}$ is

$$\mathrm{Var}(\widehat{\boldsymbol{\beta}}) = \sigma^2(\boldsymbol{X}'\boldsymbol{X}) = \sigma^2\boldsymbol{R}^{-1}(\boldsymbol{R}^{-1})'$$

The R function `chol2inv` calculates $\boldsymbol{R}^{-1}(\boldsymbol{R}^{-1})'$ directly from $\boldsymbol{R}$ without evaluating $\boldsymbol{R}^{-1}$ explicitly (not a big deal in most cases but when p is very large it should be faster and more accurate than evaluating $\boldsymbol{R}^{-1}$ explicitly).

Also, the determinant of $\boldsymbol{X}'\boldsymbol{X}$ is

$$|\boldsymbol{X}'\boldsymbol{X}| = |\boldsymbol{R}'\boldsymbol{R}| = |\boldsymbol{R}|^2 = \left(\prod_{i=1}^{p} r_{i,i}\right)^2$$

The fitted values $\widehat{\boldsymbol{y}}$ are $\boldsymbol{Q}_1\boldsymbol{Q}_1'\boldsymbol{y}$ and thus the *hat matrix* (which puts a "hat" on $\boldsymbol{y}$ by transforming it to $\widehat{\boldsymbol{y}}$) is the $n \times n$ matrix $\boldsymbol{Q}_1\boldsymbol{Q}_1'$. Often we are interested in the diagonal elements of the hat matrix, which are the sums of the squares of rows of $\boldsymbol{Q}_1$. (In practice you don't want to calculate the entire $n \times n$ hat matrix just to get the diagonal elements when n could be very large.)

The residuals, $\widehat{\boldsymbol{e}} = \boldsymbol{y} - \widehat{\boldsymbol{y}}$, are calculated as $\widehat{\boldsymbol{e}} = \boldsymbol{Q}_2\boldsymbol{Q}_2'\boldsymbol{y}$.

The matrices $\boldsymbol{Q}_1\boldsymbol{Q}_1'$ and $\boldsymbol{Q}_2\boldsymbol{Q}_2'$ are *projection matrices*, which means that they are symmetric and *idempotent*. (A square matrix $\boldsymbol{A}$ is idempotent if $\boldsymbol{A}\boldsymbol{A} = \boldsymbol{A}$.) When $\text{rank}(\boldsymbol{X}) = p$, the hat matrix $\boldsymbol{Q}_1\boldsymbol{Q}_1'$ projects any vector in $\mathbb{R}^n$ onto the column span of $\boldsymbol{X}$. The other projection, $\boldsymbol{Q}_2\boldsymbol{Q}_2'$, is onto the subspace orthogonal to the column span of $\boldsymbol{X}$ (see the figure on the front cover of the text).

R Exercise: We have already seen that $\widehat{\boldsymbol{\beta}}$ can be calculated as

```
> backsolve(R, crossprod(Q1, y))

            [,1]
[1,] 0.005085714
[2,] 0.876285714
```

or as

```
> qr.coef(qrlm1, y)

(Intercept)        carb
0.005085714 0.876285714
```

The functions `qr.fitted()` and `qr.fitted()` perform projection onto $\text{col}(\boldsymbol{X})$ and onto its orthogonal complement, respectively.

The fitted values are, unsurprisingly, calculated as

```
> qr.fitted(qrlm1, y)

         1          2          3          4          5          6
0.09271429 0.26797143 0.44322857 0.53085714 0.61848571 0.79374286

> all.equal(qr.fitted(qrlm1, y), fitted(lm1))

[1] TRUE
```

and the residuals as

```
> qr.resid(qrlm1, y)

           1           2           3           4           5           6
-0.006714286  0.001028571  0.002771429  0.007142857  0.007514286 -0.011742857

> all.equal(qr.resid(qrlm1, y), residuals(lm1))

[1] TRUE
```

We use the explicit calculations for illustration only. In practice, use of the "extractor" methods, `fitted()` and `residuals()`, is preferred.

If we wanted the projection matrices $\boldsymbol{P}_1$ for projection onto $\text{col}\,\boldsymbol{X}$ and onto the residual space we could form them as

```
> (P1 <- tcrossprod(Q1))
```

```
            [,1]        [,2]      [,3]       [,4]        [,5]        [,6]
[1,]  0.59183673  0.38775510 0.1836735 0.08163265 -0.02040816 -0.22448980
[2,]  0.38775510  0.28163265 0.1755102 0.12244898  0.06938776 -0.03673469
[3,]  0.18367347  0.17551020 0.1673469 0.16326531  0.15918367  0.15102041
[4,]  0.08163265  0.12244898 0.1632653 0.18367347  0.20408163  0.24489796
[5,] -0.02040816  0.06938776 0.1591837 0.20408163  0.24897959  0.33877551
[6,] -0.22448980 -0.03673469 0.1510204 0.24489796  0.33877551  0.52653061
```

and

```
> (P2 <- tcrossprod(Q[, -(1:2)]))
```

```
            [,1]        [,2]       [,3]        [,4]        [,5]        [,6]
[1,]  0.40816327 -0.38775510 -0.1836735 -0.08163265  0.02040816  0.22448980
[2,] -0.38775510  0.71836735 -0.1755102 -0.12244898 -0.06938776  0.03673469
[3,] -0.18367347 -0.17551020  0.8326531 -0.16326531 -0.15918367 -0.15102041
[4,] -0.08163265 -0.12244898 -0.1632653  0.81632653 -0.20408163 -0.24489796
[5,]  0.02040816 -0.06938776 -0.1591837 -0.20408163  0.75102041 -0.33877551
[6,]  0.22448980  0.03673469 -0.1510204 -0.24489796 -0.33877551  0.47346939
```

respectively (the expression `Q[, -(1:2)]` drops the first two columns of $\boldsymbol{Q}$ producing $\boldsymbol{Q}_2$). (And, of course, we don't do this in practice, especially if n is large. Instead we use `qr.fitted` or `qr.resid` if we want to project vectors other than $\boldsymbol{y}$.)

We can check that `P1` and `P2` are projection matrices. They are obviously symmetric by construction so we check the idempotent property

```
> all.equal(P1 %*% P1, P1)
```

```
[1] TRUE
```

```
> all.equal(P2 %*% P2, P2)
```

```
[1] TRUE
```

Because `P1` is projection onto col($\boldsymbol{X}$) it should take $\boldsymbol{X}$ to itself

```
> all.equal(P1 %*% X, X, check.attr=FALSE)
```

```
[1] TRUE
```

and `P2` should take $\boldsymbol{X}$ to zeros, although in practice we expect very small but possibly non-zero values.

```
> all.equal(P2 %*% X, 0 * X, check.attr=FALSE)
```

```
[1] TRUE
```

(The weird construction, `0 * X`, create a matrix of zeros that is the same size as $\boldsymbol{X}$.)

Because `P1` is the hat matrix, we can get its diagonal elements as

```
> diag(P1)

[1] 0.5918367 0.2816327 0.1673469 0.1836735 0.2489796 0.5265306
```

As mentioned, the alternative calculation is

```
> rowSums(Q1^2)

[1] 0.5918367 0.2816327 0.1673469 0.1836735 0.2489796 0.5265306
```

and a third way, preferred in practice, is

```
> hatvalues(lm1)

        1         2         3         4         5         6
0.5918367 0.2816327 0.1673469 0.1836735 0.2489796 0.5265306
```

rank($\boldsymbol{X}$), which is the number of linearly independent columns in $\boldsymbol{X}$, is calculated during the decomposition and also stored as the `$rank` component of the fitted model

```
> lm1$rank

[1] 2

> qrlm1$rank

[1] 2
```

1.3.4 R functions related to the QR decomposition

To review, every time you fit a linear model with `lm` or `aov` or `lm.fit`, the returned object contains a `$qr` component. This is a condensed form of the QR decomposition of $\boldsymbol{X}$, only slightly larger than $\boldsymbol{X}$ itself. Its class is `"qr"`.

There are several extractor functions for a `"qr"` object: `qr.R()`. `qr.Q()` and `qr.X()`, which regenerates the original matrix. By default `qr.Q()` returns the matrix called $\boldsymbol{Q}_1$ above with p columns but you can specify the number of columns desired. Typical alternative choices are n or rank($\boldsymbol{X}$).

The `$rank` component of a `"qr"` object is the computed rank of $\boldsymbol{X}$ (and, hence, of $\boldsymbol{R}$). The `$pivot` component is the permutation applied to the columns. It will be `1:p` when rank($\boldsymbol{X}$) $= p$ but when rank($\boldsymbol{X}$) $< p$ it may be other than the identity permutation.

Several functions are applied to a `"qr"` object and a vector or matrix. These include `qr.coef()`, `qr.qy()`, `qr.qty()`, `qr.resid()` and `qr.fitted()`. The `qr.qy()` and `qr.qty()` functions multiply an n-vector or an $n \times m$ matrix by $\boldsymbol{Q}$ or $\boldsymbol{Q}'$ without ever forming $\boldsymbol{Q}$. Similarly, `qr.fitted()` creates $\boldsymbol{Q}_1\boldsymbol{Q}_1'\boldsymbol{x}$ and `qr.resid()` creates $\boldsymbol{Q}_2\boldsymbol{Q}_2'\boldsymbol{x}$ without forming $\boldsymbol{Q}$.

The `is.qr()` function tests an object to determine if it is of class `"qr"`.

1.4 Related matrix decompositions

1.4.1 The Cholesky decomposition

The Cholesky decomposition of a positive definite symmetric matrix, which means a $p \times p$ symmetric matrix $\boldsymbol{A}$ such that $\boldsymbol{x}'\boldsymbol{A}\boldsymbol{x} > 0$ for all non-zero $\boldsymbol{x} \in \mathbb{R}^p$ is of the form

$$\boldsymbol{A} = \boldsymbol{R}'\boldsymbol{R} = \boldsymbol{L}\boldsymbol{L}'$$

where $\boldsymbol{R}$ is an upper triangular $p \times p$ matrix and $\boldsymbol{L} = \boldsymbol{R}'$ is lower triangular. The two forms are the same decomposition: it is just a matter of whether you want $\boldsymbol{R}$, the factor on the right, or $\boldsymbol{L}$, the factor on the left. Generally statisticians write the decomposition as $\boldsymbol{R}'\boldsymbol{R}$.

The decomposition is only determined up to changes in sign of the rows of $\boldsymbol{R}$ (or, equivalently, the columns of $\boldsymbol{L}$). For definiteness we require positive diagonal elements in $\boldsymbol{R}$.

When $\text{rank}(\boldsymbol{X}) = p$ the Cholesky decomposition $\boldsymbol{R}$ of $\boldsymbol{X}'\boldsymbol{X}$ is the equal to the matrix $\boldsymbol{R}$ from the QR decomposition up to changes in sign of rows. The matrix $\boldsymbol{X}'\boldsymbol{X}$ matrix is obviously symmetric and it is positive definite because

$$\boldsymbol{x}'(\boldsymbol{X}'\boldsymbol{X})\boldsymbol{x} = \boldsymbol{x}'(\boldsymbol{R}'\boldsymbol{R})\boldsymbol{x} = \|\boldsymbol{R}\boldsymbol{x}\|^2 \geq 0$$

with equality only when $\boldsymbol{R}\boldsymbol{x} = \boldsymbol{0}$, which, when $\text{rank}(\boldsymbol{R}) = p$, implies that $\boldsymbol{x} = \boldsymbol{0}$.

1.4.2 Evaluation of the Cholesky decomposition

The R function `chol()` evaluates the Cholesky decomposition. As mentioned above `chol2inv()` creates $(\boldsymbol{X}'\boldsymbol{X})^{-1}$ directly from the Cholesky decomposition of $\boldsymbol{X}'\boldsymbol{X}$.

Generally the QR decomposition is preferred to the Cholesky decomposition for least squares problems because there is a certain loss of precision when forming $\boldsymbol{X}'\boldsymbol{X}$. However, when n is very large you may want to build up $\boldsymbol{X}'\boldsymbol{X}$ using blocks of rows. Also, if $\boldsymbol{X}$ is *sparse* it is an advantage to use sparse matrix techniques to evaluate and store the Cholesky decomposition.

The `Matrix` package for R provides even more capabilities related to the Cholesky decomposition, especially for sparse matrices.

For everything we will do in Statistics 849 the QR decomposition should be the method of choice.

R Exercises:

```
> chol(crossprod(X))
```

```
            (Intercept)      carb
(Intercept)    2.449490 1.2655697
carb           0.000000 0.6390097
```

1.4.3 The singular value decomposition

Another decomposition related to orthogonal matrices is the singular value decomposition (or SVD) in which the matrix $\boldsymbol{X}$ is reduced to a diagonal form

$$\boldsymbol{X} = \boldsymbol{U}_1 \boldsymbol{D} \boldsymbol{V}' = \boldsymbol{U} \begin{bmatrix} \boldsymbol{D} \\ \boldsymbol{0} \end{bmatrix} \boldsymbol{V}' \tag{1.11}$$

where $\boldsymbol{U}$ is an $n \times n$ orthogonal matrix, $\boldsymbol{D}$ is a $p \times p$ diagonal matrix with non-negative diagonal elements (which are called the *singular values* of $\boldsymbol{X}$) and $\boldsymbol{V}$ is a $p \times p$ orthogonal matrix. As for $\boldsymbol{Q}$ and $\boldsymbol{Q}_1$, $\boldsymbol{U}_1$ consists of the first p columns of $\boldsymbol{U}$. For definiteness we order the diagonal elements of $\boldsymbol{D}$, which must be non-negative, in decreasing order.

Just like $\boldsymbol{Q}_1$, the columns of $\boldsymbol{U}_1$ form an orthonormal basis for $\text{col}(\boldsymbol{X})$ when $\boldsymbol{X}$ has full column rank (which means that the singular values are all safely positive). If $\text{rank}(X) = r < p$ then the first r columns of $\boldsymbol{U}$ form the orthonormal basis.

One way to visualize the singular value decomposition of $\boldsymbol{X}$ is to remember that a p-sphere in $\mathbb{R}^p$ will get mapped to an ellipsoid in $\text{col}(\boldsymbol{X})$ by $\boldsymbol{X}$. The singular values are the lengths of the principal axes of this ellipsoid. The right singular vectors (columns of $\boldsymbol{V}$) are the directions in the parameter space that map onto the principal axes of the ellipsoid. The first $\text{rank}(\boldsymbol{X})$ left singular vectors (columns of $\boldsymbol{U}$) are the principal axes of the ellipsoid.

The singular value decomposition of $\boldsymbol{X}$ is related to the eigendecomposition or spectral decomposition of $\boldsymbol{X}'\boldsymbol{X}$ because

$$\boldsymbol{X}'\boldsymbol{X} = \boldsymbol{V}\boldsymbol{D}\boldsymbol{U}_1'\boldsymbol{U}_1\boldsymbol{D}\boldsymbol{V}' = \boldsymbol{V}\boldsymbol{D}^2\boldsymbol{V}'$$

implying that the eigenvalues of $\boldsymbol{X}'\boldsymbol{X}$ are the squares of the singular values of $\boldsymbol{X}$ and the right singular vectors, which are the columns of $\boldsymbol{V}$, are also the eigenvectors of $\boldsymbol{X}'\boldsymbol{X}$

Calculation of the SVD is an iterative (as opposed to a direct) computation and potentially more computing intensive than the QR decomposition, although modern methods for evaluating the SVD are very good indeed.

Symbolically we can write the least squares solution in the full-rank case as

$$\widehat{\boldsymbol{\beta}} = \boldsymbol{V}\boldsymbol{D}^{-1}\boldsymbol{U}_1'\boldsymbol{y}$$

where $\boldsymbol{D}^{-1}$ is a diagonal matrix whose diagonal elements are the inverses of the diagonal elements of $\boldsymbol{D}$.

The pseudo-inverse or *generalized inverse* of $\boldsymbol{X}$, written $\boldsymbol{X}^-$, is calculated from the pseudo-inverse of the diagonal matrix, $\boldsymbol{D}$. In theory the diagonal elements of $\boldsymbol{D}^-$ are $1/d_{i,i}$ when $d_{i,i} \neq 0$ and 0 when $d_{i,i} = 0$. However, we can't count on $d_{i,i}$ being 0 even when, in theory, it should be. We need to decide when the singular values are close to zero, which is actually a very difficult problem. At best we can use some heuristics, based on the ratio of $d_{i,i}/d_{1,1}$, to decide when a diagonal element is "effectively zero".

The use of the pseudo-inverse seems to be a convenient way to handle rank-deficient $\boldsymbol{X}$ matrices but, as mentioned above, the best way to handle rank-deficient $\boldsymbol{X}$ matrices is not to produce them in the first place. Even when a rank-deficient $\boldsymbol{X}$ is produced we use a pivoted QR decomposition rather than a pseudo-inverse.

R Exercises: The SVD of our model matrix $\boldsymbol{X}$ is

```
> str(Xsv <- svd(X))

List of 3
 $ d: num [1:2] 2.773 0.564
 $ u: num [1:6, 1:2] 0.334 0.368 0.403 0.42 0.437 ...
 $ v: num [1:2, 1:2] 0.878 0.479 -0.479 0.878
```

We see that, by default, the `svd()` function produces the diagonal of $\boldsymbol{D}$, the matrix $\boldsymbol{U}_1$ and the matrix $\boldsymbol{V}$. We should check that $\boldsymbol{X} = \boldsymbol{U}_1\boldsymbol{D}\boldsymbol{V}'$, as advertised. We could form a diagonal matrix $\boldsymbol{D}$ from the `$d` component of `Xsv` but multiplication of a matrix on the left by a diagonal matrix corresponds to scaling its rows, so we write the reconstruction as

```
> Xsv$u %*% (Xsv$d * t(Xsv$v))

     [,1] [,2]
[1,]    1  0.1
[2,]    1  0.3
[3,]    1  0.5
[4,]    1  0.6
[5,]    1  0.7
[6,]    1  0.9

> all.equal(Xsv$u %*% (Xsv$d * t(Xsv$v)), X, check.attr=FALSE)

[1] TRUE
```

We check that the matrix $\boldsymbol{U}_1$ has orthonormal columns and that $\boldsymbol{V}$ is orthogonal

```
> zapsmall(crossprod(Xsv$u))

     [,1] [,2]
[1,]    1    0
[2,]    0    1

> zapsmall(crossprod(Xsv$v))

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

The squares of the singular values should be the eigenvalues of $\boldsymbol{X}'\boldsymbol{X}$ and the eigenvectors of $\boldsymbol{X}'\boldsymbol{X}$ should be the columns of $\boldsymbol{V}$, up to changes in sign along columns. (The eigenvectors, which are really just directions, are only determined up to changes in sign, and in the case of repeated eigenvalues, only up to orthogonal transformation within the repeated eigenvalue's eigenspace.)

```
> str(ev <- eigen(crossprod(X)))
```

```
List of 2
 $ values : num [1:2] 7.691 0.319
 $ vectors: num [1:2, 1:2] -0.878 -0.479 0.479 -0.878

> Xsv$d^2

[1] 7.6914651 0.3185349

> all.equal(ev$values, Xsv$d^2)

[1] TRUE

> ev$vectors

           [,1]       [,2]
[1,] -0.8778294  0.4789735
[2,] -0.4789735 -0.8778294

> Xsv$v

          [,1]       [,2]
[1,] 0.8778294 -0.4789735
[2,] 0.4789735  0.8778294

> all.equal(-Xsv$v, ev$vectors)

[1] TRUE
```

In practice, you never need to calculate the eigenvalues and eigenvectors of $\boldsymbol{X}'\boldsymbol{X}$. It is more effective and more stable to calculate the singular value decomposition of $\boldsymbol{X}$ and use the squares of the singular values and the `$v` component (assuming that you really do need the eigenvalues and eigenvectors which, most of the time, you don't).

The reason that it is preferable to work with decompositions of $\boldsymbol{X}$ rather than forming $\boldsymbol{X}'\boldsymbol{X}$ is related to the condition number of these matrices. As described on the Wikipedia page, the condition number of a matrix, written $\kappa(\boldsymbol{X})$, is the ratio of its largest and smallest singular values. Obviously we must have $\kappa(\boldsymbol{X}) \geq 1$. A matrix with κ close to 1 is well-conditioned. A matrix with a very large condition number is close to being singular, in that spheres are mapped to highly elongated ellipsoids.

An orthogonal matrix or a rectangular matrix with orthonormal columns must have a condition number of 1 because it maps a sphere to a sphere. (Recall that, for us, rectangular matrices like $\boldsymbol{X}$ have more rows than columns. In the opposite case, more columns than rows, it would be the rows that are orthonormal.) In fact, all the singular values of an orthogonal matrix must be unity because it preserves lengths so the unit sphere gets mapped to the unit sphere.

We can check that matrices like $\boldsymbol{Q}$, $\boldsymbol{Q}_1$ and $\boldsymbol{U}_1$ have a condition number of 1.

```
> svd(Q, nu=0, nv=0)$d
```

```
[1] 1 1 1 1 1 1

> kappa(Q)

[1] 1

> svd(Q1, nu=0, nv=0)$d

[1] 1 1

> kappa(Q1)

[1] 1

> kappa(Xsv$u)

[1] 1
```

The condition number of $\boldsymbol{X}$ can be explicitly calculated as

```
> Xsv$d

[1] 2.773349 0.564389

> (kappaX <- Xsv$d[1]/Xsv$d[length(Xsv$d)])

[1] 4.913897
```

(The complicated expression in the last line is to generalize the calculation. It will give the correct answer when there are more than two singular values.) The `kappa()` function, by default, produces an upper bound on the condition number, because this upper bound can be calculated directly. To get the exact value set the optional argument `exact=TRUE`.

```
> kappa(X)

[1] 5.1073

> kappa(X, exact=TRUE)

[1] 4.913897
```

In practice we usually calculate the reciprocal of the condition number because its value is in $[0, 1]$ and it is easier to decide when it is close to zero instead of trying to decide when $\kappa(\boldsymbol{X})$ is "close to" ∞. We compare the reciprocal condition number to the relative machine precision,

```
> .Machine$double.eps

[1] 2.220446e-16
```

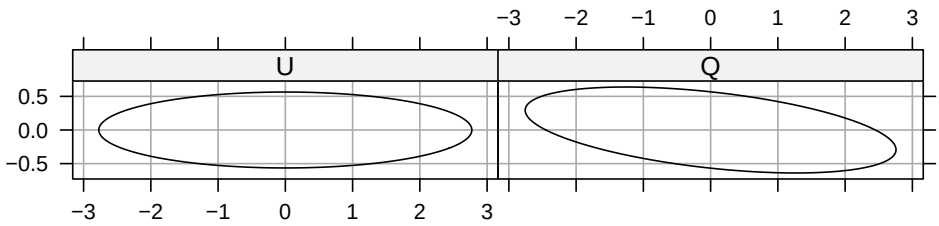

Figure 1.1: The image of the unit circle in $\mathbb{R}^2$ after mapping by $\boldsymbol{U}_1'\boldsymbol{X}$ (left panel) and by $\boldsymbol{Q}_1'\boldsymbol{X}$ (right panel)

A matrix is considered computationally singular when its reciprocal condition number is within some multiple, typical values are 10 or 100, of this number.

Getting back to the question of why we prefer to work with $\boldsymbol{X}$ directly, instead of forming $\boldsymbol{X}'\boldsymbol{X}$, it is because $\kappa(\boldsymbol{X}'\boldsymbol{X}) = \kappa(\boldsymbol{X})^2$. If $\kappa(\boldsymbol{X}) = 10^6$, which is large but not catastrophically so, then $\kappa(\boldsymbol{X}'\boldsymbol{X})$ will be 10^{12}, which means it is very close to being singular.

Finally, let's revisit the idea of the singular values being the lengths of the principal axes of the image of the unit sphere in the map $\boldsymbol{\beta} \to \boldsymbol{X}\boldsymbol{\beta}$. When $p = 2$ the unit sphere is the circle of radius 1 centered at the origin and the ellipsoid mentioned above will be an ellipse.

A convenient way of creating a $2 \times N$ matrix whose columns are the points on the unit circle is to start with a sequence of values from 0 to 2π and use its sines and cosines

```
> str(rad <- seq(0, 2*pi, len=201))

 num [1:201] 0 0.0314 0.0628 0.0942 0.1257 ...

> str(circ <- rbind(cos(rad), sin(rad)))

 num [1:2, 1:201] 1 0 0.9995 0.0314 0.998 ...
```

The n-dimensional response vectors corresponding to these points on the circle are

```
> fits <- X %*% circ
```

To plot the this image in two dimensions (Fig. 1.1) we need to represent these points with respect to an orthogonal basis for col($\boldsymbol{X}$). Fortunately we have two such bases: the columns of $\boldsymbol{Q}_1$ and of $\boldsymbol{U}_1$. In the $\boldsymbol{U}_1$ basis the principal axes of the ellipse correspond to the coordinate axes. In the $\boldsymbol{Q}_1$ basis the principal axes are skewed.

1.5 Theoretical results on the eigendecomposition

1.5.1 Eigenvalues and Eigenvectors

For any $k \times k$ matrix $\boldsymbol{A}$, the roots of the k^{th} degree polynomial equation in λ, $|\lambda \boldsymbol{I}_k - \boldsymbol{A}| = 0$, which we will write as $\lambda_1, \ldots, \lambda_k$ are called the *eigenvalues* of $\boldsymbol{A}$. The polynomial is called the

characteristic polynomial of $\boldsymbol{A}$.

Any nonzero $n \times 1$ vector $\boldsymbol{v}_i \neq \boldsymbol{0}$ such that $\boldsymbol{A}\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i$ is an *eigenvector* of $\boldsymbol{A}$ corresponding to the eigenvalue λ_i.

For any diagonal matrix $\boldsymbol{D} = \text{diag}(d_1, \ldots, d_k)$, $|\lambda \boldsymbol{I}_k - \boldsymbol{D}| = \prod_{i=1}^{k}(\lambda - d_i) = 0$ has roots d_i, therefore the diagonal elements $d_i,\ i = 1, \ldots, n$ are the eigenvalues of $\boldsymbol{D}$

If $\boldsymbol{Q}$ is an orthogonal matrix, then $\boldsymbol{Q}\boldsymbol{A}\boldsymbol{Q}'$ and $\boldsymbol{A}$ have the same eigenvalues.

Proof.

$$\begin{aligned} |\lambda \boldsymbol{I} - \boldsymbol{Q}\boldsymbol{A}\boldsymbol{Q}'| &= |\lambda \boldsymbol{Q}\boldsymbol{Q}' - \boldsymbol{Q}\boldsymbol{A}\boldsymbol{Q}'| \\ &= |\boldsymbol{Q}||\lambda \boldsymbol{Q}' - \boldsymbol{A}\boldsymbol{Q}'| \\ &= |\boldsymbol{Q}||\lambda \boldsymbol{I} - \boldsymbol{A}||\boldsymbol{Q}'| \\ &= |\boldsymbol{Q}|^2|\lambda \boldsymbol{I} - \boldsymbol{A}| = 1|\lambda \boldsymbol{I} - \boldsymbol{A}| \\ &= |\lambda \boldsymbol{I} - \boldsymbol{A}| \end{aligned} \tag{1.12}$$

□

Note: Although the eigenvalues are defined as the roots of the characteristic polynomial, in practice they are not calculated this way. In fact, if you check the documentation for function `solve.polynomial()` in the `polynom` package for R you will find that it uses the numerical methods for evaluating the eigenvalues of the *companion matrix* of a polynomial to solve for the polynomial's roots.

1.5.2 Diagonalization of a Symmetric Matrix

For any $k \times k$ symmetric matrix $\boldsymbol{A}$ (i.e. $\boldsymbol{A}' = \boldsymbol{A}$), there exists an orthogonal matrix $\boldsymbol{Q}$ such that $\boldsymbol{Q}\boldsymbol{A}\boldsymbol{Q}'$ is a diagonal matrix $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, \ldots, \lambda_n)$ where λ_i are the eigenvalues of $\boldsymbol{A}$. The corresponding eigenvectors of $\boldsymbol{A}$ are the column vectors of $\boldsymbol{Q}$

Proof. Let $\boldsymbol{e}_i,\ i = 1, \ldots, n$ be $n \times 1$ unit vectors that form the canonical basis of $\mathbb{R}^n$, (i.e. $\boldsymbol{e}_i = (0, \ldots, 1, \ldots, 0)'$ where the 1 is in the i^{th} position) and $\boldsymbol{q}_i$ be the ith column of $\boldsymbol{Q}$. That is, $\boldsymbol{q}_i = \boldsymbol{Q}\boldsymbol{e}_i$. Then

$$\boldsymbol{Q}'\boldsymbol{A}\boldsymbol{Q} = \boldsymbol{\Lambda} \Rightarrow \boldsymbol{Q}'\boldsymbol{A}\boldsymbol{Q}\boldsymbol{e}_i = \boldsymbol{\Lambda}e_i = \lambda_i \boldsymbol{e}_i$$

Multiplying on the left by $\boldsymbol{Q}$ produces

$$\boldsymbol{A}\boldsymbol{q}_i = \underbrace{\boldsymbol{Q}\boldsymbol{Q}'}_{\boldsymbol{I}}\boldsymbol{A}\boldsymbol{Q}\boldsymbol{e}_i = \lambda_i \boldsymbol{Q}\boldsymbol{e}_i = \lambda_1 \boldsymbol{q}_i.$$

□

1.5.3 Spectral Decomposition

From the relationship $\boldsymbol{Q}'\boldsymbol{A}\boldsymbol{Q} = \boldsymbol{\Lambda}$ just established for a $k \times k$ symmetric matrix $\boldsymbol{A}$ we can compute its spectral decomposition,

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}' = \sum_{i=1}^{k} \lambda_i \boldsymbol{q}_i \boldsymbol{q}_i'$$

where $\boldsymbol{q}_i$ is the i^{th} column of $\boldsymbol{Q}$.

$$\boldsymbol{Q}\boldsymbol{Q}' = \sum_{i=1}^{k} \boldsymbol{q}_i \boldsymbol{q}_i' = \boldsymbol{I}$$

1.5.4 Trace and Determinant of A

The relationship $\boldsymbol{Q}'\boldsymbol{A}\boldsymbol{Q} = \boldsymbol{\Lambda}$ for symmetric $\boldsymbol{A}$ implies that the trace, $\mathrm{tr}(\boldsymbol{A})$, and the determinant, $|\boldsymbol{A}|$, are the same as those of $\boldsymbol{\Lambda}$.

$$\mathrm{tr}(\boldsymbol{A}) = \mathrm{tr}(\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}') = \mathrm{tr}(\boldsymbol{\Lambda}\boldsymbol{Q}\boldsymbol{Q}') = \mathrm{tr}(\boldsymbol{\Lambda}) = \sum_{i=1}^{k} \lambda_i$$

where we have used the property that $\mathrm{tr}(\boldsymbol{C}\boldsymbol{D}) = \mathrm{tr}(\boldsymbol{D})$ for any conformable matrices $\boldsymbol{C}$ and $\boldsymbol{D}$ (meaning that if $\boldsymbol{C}$ is $m \times n$ then $\boldsymbol{D}$ must be $n \times m$).

$$|\boldsymbol{A}| = |\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}'| = |\boldsymbol{Q}||\boldsymbol{\Lambda}||\boldsymbol{Q}'| = |\boldsymbol{Q}|^2|\boldsymbol{\Lambda}| = \prod_{i=1}^{k} \lambda_i$$