Matrix Algebra

This document is a summary of different stats courses:

- Introduction to Linear Models and Matrix Algebra (HarvardX PH525.2x via Edx) There is a free e-book which contains the full course: dataanalysisforthelifesciences.pdf and all the code can be found in the git of the author.

Introduction

Scalars: are numbers, for example 3, or -9.898

Vectors: are series of numbers. For example (3,5,6.7,-1)

Matrices: are a series of vectors. We generally use `X` to represent a matrix, and a matrix will have `N` rows and `p` columns A square matrix has the same number of rows as columns. For example a matrix of 3 rows and 3 columns (3x3):

$$\begin{pmatrix} 1 & 1 & 1.6 \\ 3 & -2 & 0 \\ 2 & 1 & -1 \end{pmatrix}$$

Magnitude of a vector

The magnitude of a vector represents its length in space. The magnitude (also called the Euclidean norm) is calculated using the square root of the sum of the squares of its components.

For a vector: $\vec{v} = [v_1, v_2, \cdots, v_n]$

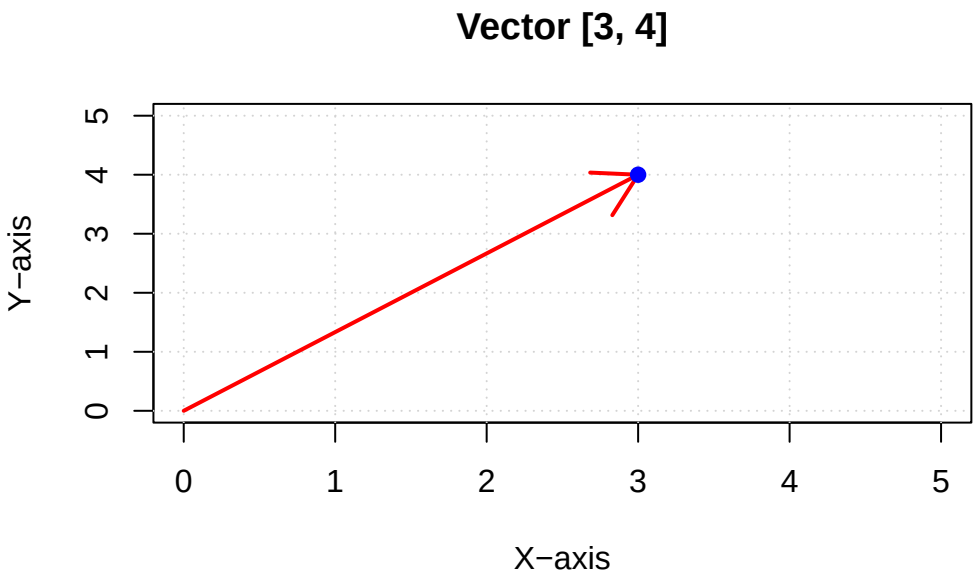
$$\|\vec{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Example:

$$\vec{v} = [3, 4]$$

Then:

$$\|\vec{v}\| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$




```
# Define the vector
v <- c(3, 4)

# Calculate the magnitude (Euclidean norm)
(magnitude <- sqrt(sum(v^2)))
```

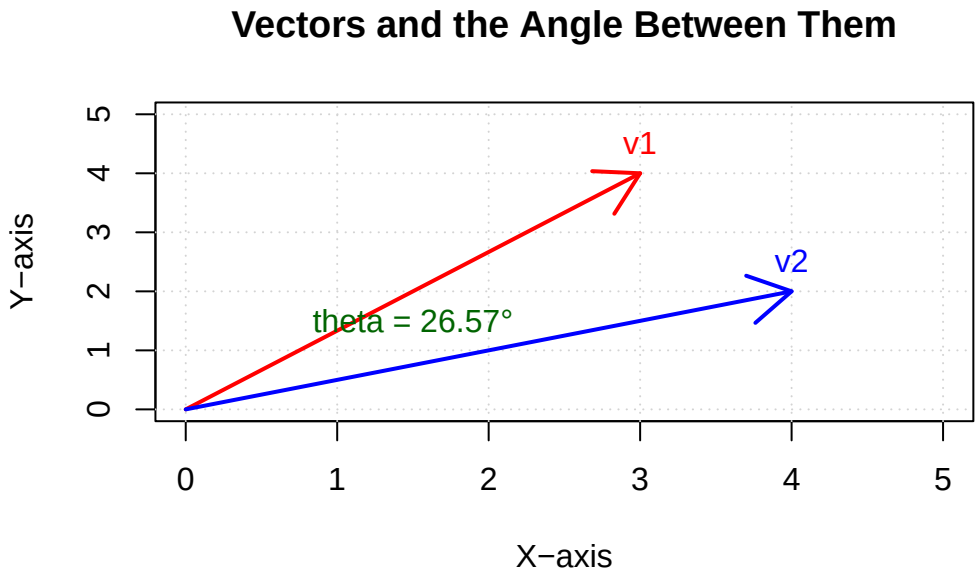
```
[1] 5
```

Dot product of vectors

The dot product is one way of multiplying two or more vectors. The resultant of the dot product of vectors is a scalar quantity. Thus, the dot product is also known as a scalar product.

Geometrically, the dot product of two vectors is the product of their Euclidean magnitudes and the cosine of the angle between them.

$$\vec{a} \cdot \vec{b} = \|\vec{a}\|\|\vec{b}\| \cos(\theta)$$



Algebraically, it is the sum of the products of the corresponding entries of two sequences of numbers.

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Example:

Let:

$$\vec{a} = [1, 2, 3], \quad \vec{b} = [4, 5, 6]$$

Then:

$$\vec{a} \cdot \vec{b} = 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32$$

in r

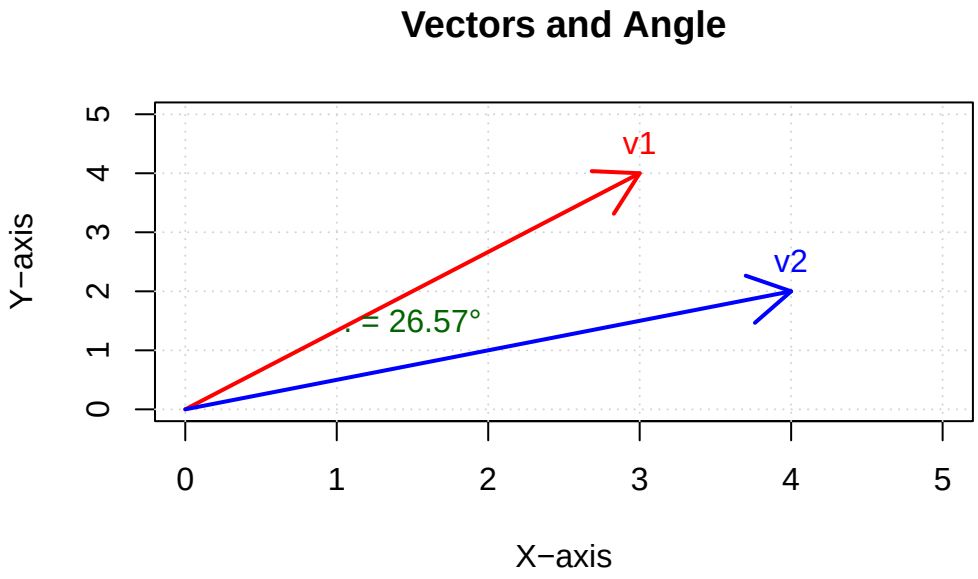
```r
# Define the vectors
a <- c(1, 2, 3)
b <- c(4, 5, 6)

# Compute the dot product
dot_product <- sum(a * b)
dot_product
```

```
[1] 32
```

Cosine Similarity

Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space. It is defined as the cosine of the angle between the two vectors. This measure is particularly used in text analysis to measure the similarity between documents.

```r
# Define vectors
v1 <- c(3, 4)
v2 <- c(4, 2)

# Calculate angle
dot <- sum(v1 * v2)
mag1 <- sqrt(sum(v1^2))
mag2 <- sqrt(sum(v2^2))
angle <- acos(dot / (mag1 * mag2)) * 180 / pi

# Plot
plot(0, 0, xlim = c(0, 5), ylim = c(0, 5), type = "n",
     xlab = "X-axis", ylab = "Y-axis", main = "Vectors and Angle")
grid()
arrows(0, 0, v1[1], v1[2], col = "red", lwd = 2)
arrows(0, 0, v2[1], v2[2], col = "blue", lwd = 2)
text(v1[1], v1[2], "v1", pos = 3, col = "red")
text(v2[1], v2[2], "v2", pos = 3, col = "blue")
text(1.5, 1.5, paste0("  = ", round(angle, 2), "°"), col = "darkgreen")
```



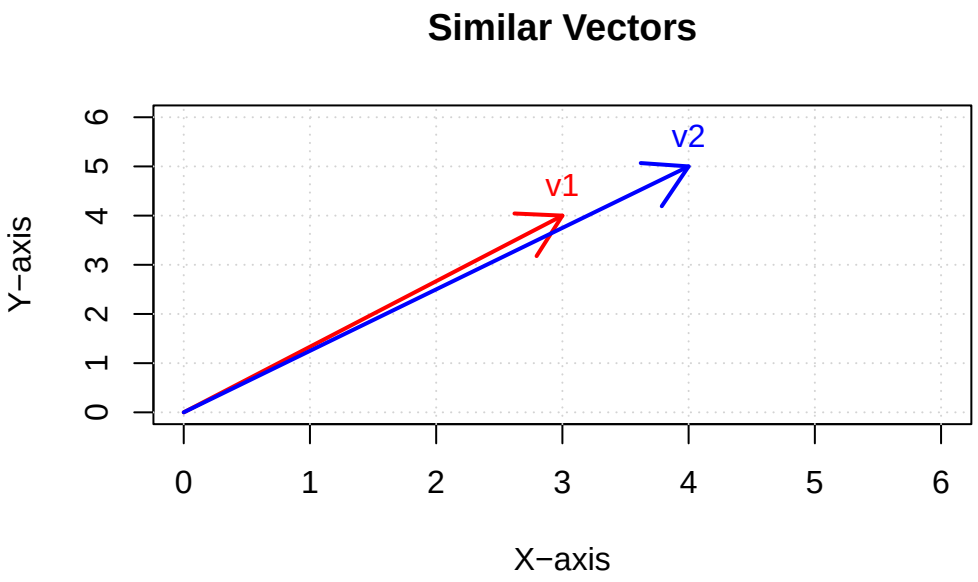
The cosine similarity between two vectors **a** and **b** is calculated as:

$$\text{cosine similarity} = \cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

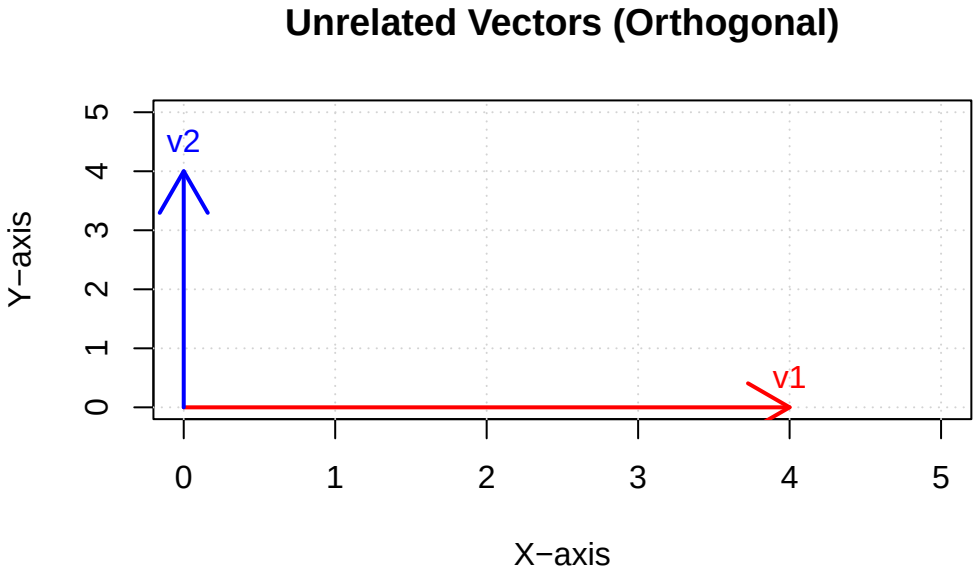
Where: - $\vec{a} \cdot \vec{b}$ is the dot product of vectors **a** and **b** - $\|\vec{a}\|$ and $\|\vec{b}\|$ are the magnitudes (Euclidean norms) of vectors **a** and **b** - θ is the angle between the vectors

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

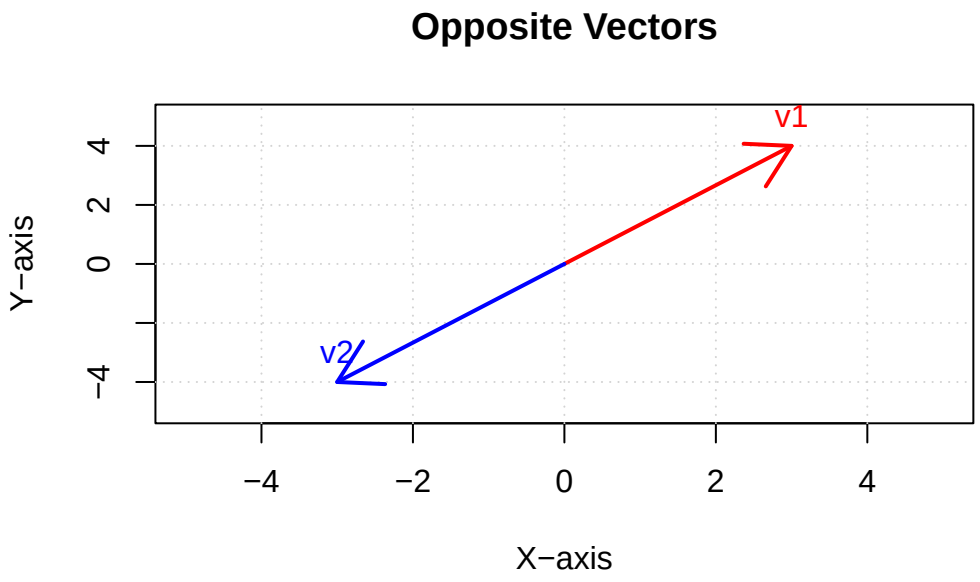
$$\|\vec{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} = \sqrt{\sum_{i=1}^{n} v_i^2}$$



These vectors point in nearly the same direction, forming a small angle between them. This results in a cosine similarity close to 1, indicating high similarity.



These vectors are perpendicular to each other, forming a 90° angle. Their cosine similarity is 0, meaning they are completely unrelated in direction.



These vectors point in exactly opposite directions, forming a 180° angle. Their cosine similarity is -1, indicating they are completely dissimilar in direction.

We can calculate this in r manually:

```r
# Define vectors
a <- c(1, 2, 3)
b <- c(4, 5, 6)

# Dot product
dot_product <- sum(a * b)

# Magnitudes
mag_a <- sqrt(sum(a^2))
mag_b <- sqrt(sum(b^2))

# Cosine similarity
cos_sim <- dot_product / (mag_a * mag_b)
cos_sim
```

```
[1] 0.9746318
```

or we can use the `cosine` function from the `lsa` package:

```
x <- c(0.12, 0.44, 0.5, 0.3, 0.7, 0.04, 0.9, 0.8)
y <- c(0.24, 0.5, 0.7, 0.21, 0.69, 0.2, 0.7, 0.5)

lsa::cosine(x, y)
```

```
          [,1]
[1,] 0.9551402
```

Creating a basic search engine

We illustrate a simple example of machine learning by using cosine similarity to determine which product description is most correlated with a given search sentence. This basic technique forms the foundation for many text-based similarity and recommendation systems.

We begin by defining 10 product descriptions:

```
products <- c(
  "Sleek red smartphone with powerful battery and excellent display",
  "Lightweight laptop with fast performance and long battery life",
  "Noise-cancelling wireless headphones with dynamic sound",
  "Durable smartwatch with fitness tracking features and water resistance",
  "Waterproof fitness tracker with heart rate monitoring and step counter",
  "High resolution tablet with slim design and vibrant colors",
  "Compact digital camera with optical image stabilization and zoom",
  "Portable Bluetooth speaker with deep bass and clear audio",
  "Smart home hub connecting all smart devices seamlessly",
  "Lightweight e-reader with adjustable backlight and user friendly interface"
)
products
```

```
 [1] "Sleek red smartphone with powerful battery and excellent display"
 [2] "Lightweight laptop with fast performance and long battery life"
 [3] "Noise-cancelling wireless headphones with dynamic sound"
 [4] "Durable smartwatch with fitness tracking features and water resistance"
 [5] "Waterproof fitness tracker with heart rate monitoring and step counter"
 [6] "High resolution tablet with slim design and vibrant colors"
 [7] "Compact digital camera with optical image stabilization and zoom"
 [8] "Portable Bluetooth speaker with deep bass and clear audio"
```

```
 [9] "Smart home hub connecting all smart devices seamlessly"
[10] "Lightweight e-reader with adjustable backlight and user friendly interface"
```

1. Tokenization and Vectorization

We define a helper function to tokenize the text (by converting to lowercase, removing punctuation, and splitting into words). We then build a vocabulary from both the product descriptions and the search sentence, and create bag-of-words vectors that count occurrences of each vocabulary word.

```
# Function to tokenize a text string
tokenize <- function(text) {
  # Convert to lower case and remove punctuation (keeping only a-z and space)
  words <- gsub("[^a-z ]", "", tolower(text))
  unlist(strsplit(words, "\\s+"))
}

# Tokenize the product descriptions
tokenized_products <- lapply(products, tokenize)

#print one example:
tokenized_products[1]
```

```
[[1]]
[1] "sleek"      "red"        "smartphone" "with"       "powerful"
[6] "battery"    "and"        "excellent"  "display"
```

```
# Define a search sentence
search_sentence <- "Looking for a smartphone with a powerful battery"
tokenized_search <- tokenize(search_sentence)

# Create a vocabulary from all tokens in products and search sentence
vocab <- unique(c(unlist(tokenized_products), tokenized_search))

head(vocab,10)
```

```
 [1] "sleek"       "red"         "smartphone"  "with"        "powerful"
 [6] "battery"     "and"         "excellent"   "display"     "lightweight"
```

```r
# Function to vectorize a list of tokens given the vocabulary (count words)
vectorize <- function(tokens, vocab) {
  sapply(vocab, function(word) sum(tokens == word))
}

# Create vectors for each product description and the search sentence
product_vectors <- lapply(tokenized_products, vectorize, vocab = vocab)

#see the results of the first product description:
product_vectors[1]
```

```
[[1]]
          sleek             red      smartphone            with        powerful
              1               1               1               1               1
        battery             and       excellent         display     lightweight
              1               1               1               1               0
         laptop            fast     performance            long            life
              0               0               0               0               0
noisecancelling        wireless      headphones         dynamic           sound
              0               0               0               0               0
        durable      smartwatch         fitness        tracking        features
              0               0               0               0               0
          water      resistance      waterproof         tracker           heart
              0               0               0               0               0
           rate      monitoring            step         counter            high
              0               0               0               0               0
     resolution          tablet            slim          design         vibrant
              0               0               0               0               0
         colors         compact         digital          camera         optical
              0               0               0               0               0
          image   stabilization            zoom        portable       bluetooth
              0               0               0               0               0
        speaker            deep            bass           clear           audio
              0               0               0               0               0
          smart            home             hub      connecting             all
              0               0               0               0               0
        devices      seamlessly         ereader      adjustable       backlight
              0               0               0               0               0
           user        friendly       interface         looking             for
              0               0               0               0               0
              a
              0
```

```
search_vector <- vectorize(tokenized_search, vocab)
```

2. Cosine Similarity Calculation We define a function to compute the cosine similarity between two numeric vectors. Recall that the cosine similarity is defined as

$$\text{cosine similarity} = \cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

```
cosine_similarity <- function(vec1, vec2) {
  dot_product <- sum(vec1 * vec2)
  norm_vec1 <- sqrt(sum(vec1^2))
  norm_vec2 <- sqrt(sum(vec2^2))
  if (norm_vec1 == 0 || norm_vec2 == 0) {
    return(0)
  }
  dot_product / (norm_vec1 * norm_vec2)
}
```

3. Computing Similarities and Selecting the Best Match

Now, we loop through each product vector, compute its cosine similarity with the search sentence vector, and then identify the product description with the highest similarity score.

```
# Compute cosine similarity for each product
similarities <- sapply(product_vectors, cosine_similarity, vec2 = search_vector)

# Combine the results into a data frame for clearer viewing
results <- data.frame(
  Product = 1:length(products),
  Description = products,
  Similarity = similarities
)

print(results)
```

```
  Product
1       1
2       2
3       3
4       4
5       5
6       6
```

```
7        7
8        8
9        9
10      10
                                                                  Description
1            Sleek red smartphone with powerful battery and excellent display
2              Lightweight laptop with fast performance and long battery life
3                     Noise-cancelling wireless headphones with dynamic sound
4      Durable smartwatch with fitness tracking features and water resistance
5      Waterproof fitness tracker with heart rate monitoring and step counter
6                   High resolution tablet with slim design and vibrant colors
7            Compact digital camera with optical image stabilization and zoom
8                   Portable Bluetooth speaker with deep bass and clear audio
9                      Smart home hub connecting all smart devices seamlessly
10 Lightweight e-reader with adjustable backlight and user friendly interface
   Similarity
1   0.4216370
2   0.2108185
3   0.1290994
4   0.1054093
5   0.1000000
6   0.1054093
7   0.1054093
8   0.1054093
9   0.0000000
10  0.1054093
```

```r
# Identify which product has the highest similarity
best_match_index <- which.max(similarities)
best_match <- products[best_match_index]

cat("\nThe best matching product is:\n")
```

```
The best matching product is:
```

```r
cat(best_match)
```

```
Sleek red smartphone with powerful battery and excellent display
```

We have created a simple example illustrating how to use cosine similarity in a machine learning context. By tokenizing product descriptions and the search query, building bag-of-words

vectors, and computing cosine similarity, we can determine that the product description most related to the query is selected based on how many words they have in common weighted by their occurrences.

This basic approach can be a stepping stone toward more sophisticated text similarity techniques and machine learning applications.

Matrices operations

to create a matrix in r you can create vectors and bind them together using `cbind` or `rbind` or create a matrix directly for example `matrix(1:60,20,3)`

Linear algebra was developed to solve a system of equations. It gives a general solution to any system of equations. Let's see this example:

$$a+b+c=63a-2b+c=22a+b-c=1$$

$$\begin{pmatrix}1 & 1 & 1\\ 3 & -2 & 1\\ 2 & 1 & -1\end{pmatrix}\begin{pmatrix}a\\ b\\ c\end{pmatrix}=\begin{pmatrix}6\\ 2\\ 1\end{pmatrix}\Rightarrow\begin{pmatrix}a\\ b\\ c\end{pmatrix}=\begin{pmatrix}1 & 1 & 1\\ 3 & -2 & 1\\ 2 & 1 & -1\end{pmatrix}^{-1}\begin{pmatrix}6\\ 2\\ 1\end{pmatrix}$$

Matrix multiplication by scalar

When you have a matrix and you multiply it by a scalar, you multiply each element of the matrix by that scalar: Given a scalar (k) and a matrix (A):

$$k=3,\quad A=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}$$

The result of multiplying the matrix (A) by the scalar (k) is:

$$kA=3\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}=\begin{pmatrix}3\cdot 1 & 3\cdot 2\\ 3\cdot 3 & 3\cdot 4\end{pmatrix}=\begin{pmatrix}3 & 6\\ 9 & 12\end{pmatrix}$$

in r is also very simple:

```
X<- matrix(1:12,4,3)
print(X)
```

```
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

```r
a<-2
print(X*a)
```

```
     [,1] [,2] [,3]
[1,]    2   10   18
[2,]    4   12   20
[3,]    6   14   22
[4,]    8   16   24
```

Matrices multiplication

Matrix multiplication is performed by taking the dot product of rows from the first matrix (◎) with columns of the second matrix (◎). The key steps are:

- Check compatibility: Ensure the number of columns in ◎ matches the number of rows in ◎.
- *Dot Product Computation*: Each element in the resulting matrix is calculated by multiplying corresponding entries from a row of ◎ and a column of◎, summing the results.
- The resulting matrix has dimensions $m \times p$ where A is $m \times n$ and B is $n \times p$.

To multiply a 3×4 matrix A with a 4×2 matrix B, we follow the rule that each row of A interacts with each column of B using the dot product.

Given Matrices:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{pmatrix}, \quad B = \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \\ b_{4,1} & b_{4,2} \end{pmatrix}$$

These matrices are compatible for multiplication because A has **4 columns**, matching B's **4 rows**.

The resulting 3×2 matrix C is computed as follows:

$$C = AB = \begin{pmatrix} a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{2,1} + a_{1,3} \cdot b_{3,1} + a_{1,4} \cdot b_{4,1} & a_{1,1} \cdot b_{1,2} + a_{1,2} \cdot b_{2,2} + a_{1,3} \cdot b_{3,2} + \\ a_{2,1} \cdot b_{1,1} + a_{2,2} \cdot b_{2,1} + a_{2,3} \cdot b_{3,1} + a_{2,4} \cdot b_{4,1} & a_{2,1} \cdot b_{1,2} + a_{2,2} \cdot b_{2,2} + a_{2,3} \cdot b_{3,2} + \\ a_{3,1} \cdot b_{1,1} + a_{3,2} \cdot b_{2,1} + a_{3,3} \cdot b_{3,1} + a_{3,4} \cdot b_{4,1} & a_{3,1} \cdot b_{1,2} + a_{3,2} \cdot b_{2,2} + a_{3,3} \cdot b_{3,2} + \end{pmatrix}$$

Each element in C is derived from the dot product of a row in A and a column in B.

For example, the top-left element of C (i.e., $c_{1,1}$) is calculated as:

$$c_{1,1} = a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{2,1} + a_{1,3} \cdot b_{3,1} + a_{1,4} \cdot b_{4,1}$$

Likewise, every position in C follows the same logic.

Given two matrices (A) and (B):

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \quad B = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$$

The result of multiplying matrix (A) by matrix (B) is:

$$AB = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + 2 \cdot 0 + 3 \cdot (-1) \\ 4 \cdot 1 + 5 \cdot 0 + 6 \cdot (-1) \\ 7 \cdot 1 + 8 \cdot 0 + 9 \cdot (-1) \end{pmatrix} = \begin{pmatrix} -2 \\ -2 \\ -2 \end{pmatrix}$$

and in r we use %*%

```
X<- matrix(c(1,3,2,1,-2,1,1,1,-1),3,3)
X
```

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

```
beta<- c(3,2,1)
X%*%beta
```

```
     [,1]
[1,]    6
[2,]    6
[3,]    7
```

Given two matrices (A) and (B):

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}, \quad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

The result of multiplying matrix (A) by matrix (B) is:

$$AB = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} = \begin{pmatrix} 1 \cdot 7 + 2 \cdot 10 & 1 \cdot 8 + 2 \cdot 11 & 1 \cdot 9 + 2 \cdot 12 \\ 3 \cdot 7 + 4 \cdot 10 & 3 \cdot 8 + 4 \cdot 11 & 3 \cdot 9 + 4 \cdot 12 \\ 5 \cdot 7 + 6 \cdot 10 & 5 \cdot 8 + 6 \cdot 11 & 5 \cdot 9 + 6 \cdot 12 \end{pmatrix} = \begin{pmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{pmatrix}$$

Understanding Matrix Multiplication

Matrix multiplication is an operation where the product of two matrices is obtained by computing the dot product of rows from the first matrix with columns from the second matrix. However, unlike regular arithmetic multiplication, matrix multiplication does **not** follow the commutative property:

$$A \times B \neq B \times A$$

in general. This means swapping the order of multiplication can lead to different results or may even be **undefined**.

Conditions for Matrix Multiplication

For matrices **A** and **B** to be **multipliable**, their dimensions must satisfy: - A is an $m \times n$ matrix. - B is an $n \times p$ matrix. - The resulting matrix C has dimensions $m \times p$.

Example: Demonstrating Non-Commutativity

Let's consider two matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Computing $A \times B$:

$$AB = \begin{pmatrix} (1 \cdot 0 + 2 \cdot 1) & (1 \cdot 1 + 2 \cdot 0) \\ (3 \cdot 0 + 4 \cdot 1) & (3 \cdot 1 + 4 \cdot 0) \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix}$$

Now computing $B \times A$:

$$BA = \begin{pmatrix} (0 \cdot 1 + 1 \cdot 3) & (0 \cdot 2 + 1 \cdot 4) \\ (1 \cdot 1 + 0 \cdot 3) & (1 \cdot 2 + 0 \cdot 4) \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ 1 & 2 \end{pmatrix}$$

Clearly, $A \times B \neq B \times A$, demonstrating non-commutativity.

Additional Matrix Multiplication Properties

1. **Associative Property**:
$$(A \times B) \times C = A \times (B \times C)$$
2. **Distributive Property**:
$$A \times (B + C) = A \times B + A \times C$$
3. **Identity Matrix Property**: If I is the identity matrix:
$$A \times I = I \times A = A$$
4. **Zero Matrix Property**:
$$A \times 0 = 0$$

Matrix multiplication plays a fundamental role in linear algebra, forming the basis for transformations, systems of equations, and numerous applications in data science, physics, and engineering.

Identity matrix

An identity matrix (also known as a *unit matrix*) is a square matrix in which all the elements of the principal diagonal are ones, and all other elements are zeros. It is denoted by (I). The identity matrix plays a crucial role in matrix multiplication, as multiplying any matrix by the identity matrix leaves the original matrix unchanged. he identity matrix (I) of order 3 is:

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

in r we use the function `diag()` with the number of dimensions we want:

```r
diag(5)
```

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

Transpose

Transpose simply turns the rows into columns and vice versa, in r we use `t`

```r
X<- matrix(1:15,5,3)
X
```

```
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
```

```r
t(X)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
```

Inversion

The inverse of a square matrix X is denoted as X^{-1} it has the property that if you multiply a matrix by its inverse, it gives you the identity matrix. $X^{-1}X = I$

Note that not all matrices have an inverse.

In linear algebra, the **determinant** and **adjoint** of a matrix are fundamental concepts used to compute the **inverse** of a matrix. Below, we explain these concepts and demonstrates how to use them to find the inverse of a matrix.

Determinant

The **determinant** of a square matrix is a scalar value that provides important properties of the matrix. It is denoted as `det(A)` for a matrix `A`. A matrix is invertible if and only if its determinant is non-zero.

For a 2×2 matrix:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The determinant is:

$$\det(A) = ad - bc$$

In r we use `det` formula to calculate it.

```r
# Define the matrix
A <- matrix(c(1, 0, 1, 2, 4, 0, 3, 5, 6), nrow = 3, byrow = TRUE)

# Compute the determinant
det_A <- det(A)
```

Adjoint

The **adjoint** (or adjugate) of a matrix is the transpose of the cofactor matrix. For a 2×2 matrix:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The adjoint is:

$$\text{adj}(A) = \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Cofactor Matrix

To compute the **adjoint** of a matrix, we first need the **cofactor matrix**.

The **cofactor** of an element a_{ij} in a matrix is calculated as:

$$C_{ij} = (-1)^{i+j} \cdot M_{ij}$$

Where: - M_{ij} is the **minor** of the element a_{ij}, i.e., the determinant of the submatrix formed by removing the i-th row and j-th column from the original matrix. - $(-1)^{i+j}$ gives the correct sign based on the position.

The **cofactor matrix** is the matrix of all C_{ij} values.

Adjoint of a Matrix (General Case)

The **adjoint** of a matrix is the **transpose** of its cofactor matrix:

$$\text{adj}(A) = \text{Cofactor}(A)^T$$

This method works for any square matrix, not just 2×2.

Calculating Inverse of a Matrix manually

$$A = \begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix}$$

```
# Define the matrix
A <- matrix(c(2, 1, 3, 4), nrow = 2, byrow = TRUE)

# Compute the determinant
det_A <- det(A)

# Compute the adjoint manually
adj_A <- matrix(c(4, -3, -1, 2), nrow = 2, byrow = TRUE)

# Compute the inverse using the formula
A_inv <- (1 / det_A) * adj_A

# Display the result
A_inv
```

```
     [,1] [,2]
[1,]  0.8 -0.6
[2,] -0.2  0.4
```

In r there is no formula to calculate the adjoint of a matrix directly, so if you need to calculate the adjoint of a matrix of more than 2x2, you can use the package `matlib`

```r
# Define a matrix
A <- matrix(c(1, 0, 1, 2, 4, 0, 3, 5, 6), nrow = 3, byrow = TRUE)

# Compute the adjoint
adj_A <- matlib::adjoint(A)

# Display the result
adj_A
```

```
     [,1] [,2] [,3]
[1,]   24    5   -4
[2,]  -12    3    2
[3,]   -2   -5    4
```

Calculating Inverse of a Matrix using software

We rarely need to get the adjoint outside of the scope of calculating the inverse of a matrix, and r gives us a formula for directly calculating the inverse of a matrix, the determinant and the adjoint are calculated internally

Example: Adjoint and Inverse of a 3×3 Matrix in R

Let's compute the inverse of:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 1 & 0 & 6 \end{bmatrix}$$

```r
# Define the matrix
A <- matrix(c(1, 0, 1, 2, 4, 0, 3, 5, 6), nrow = 3, byrow = TRUE)

# Compute the inverse using solve (R handles adjoint and cofactors internally)
A_inv <- solve(A)
```

```r
# Display the result
A_inv
```

```
            [,1]       [,2]        [,3]
[1,]  1.09090909  0.2272727 -0.18181818
[2,] -0.54545455  0.1363636  0.09090909
[3,] -0.09090909 -0.2272727  0.18181818
```

In r we use the function `solve` to get the inverse, and we use it to solve equations: it gives us the values for a, b and c to resolve the system of equations:

$$\begin{aligned} a + b + c &= 6 \\ 3a - 2b + c &= 2 \\ 2a + b - c &= 1 \end{aligned}$$

$$\begin{pmatrix} 1 & 1 & 1 \\ 3 & -2 & 1 \\ 2 & 1 & -1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 6 \\ 2 \\ 1 \end{pmatrix}$$

```r
X <- matrix(c(1,3,2,1,-2,1,1,1,-1),3,3)
y <- matrix(c(6,2,1),3,1)
solve(X)%*%y
```

```
     [,1]
[1,]    1
[2,]    2
[3,]    3
```

Example

A small factory produces two products: **Chairs** and **Tables**. Each product requires a certain amount of **wood** and **labor hours**:

- A **Chair** requires 2 units of wood and 3 hours of labor.
- A **Table** requires 5 units of wood and 2 hours of labor.

The factory has **available resources** of:

- 40 units of wood

- 30 hours of labor

We want to determine how many **Chairs (x)** and **Tables (y)** the factory can produce using all available resources.

Step 1: Represent the System as Equations

We can write the constraints as:

$$\begin{aligned} 2x + 5y &= 40 \quad \text{(wood constraint)} \\ 3x + 2y &= 30 \quad \text{(labor constraint)} \end{aligned}$$

Step 2: Matrix Form

This system can be written in matrix form:

$$AX = B$$

Where:

$$A = \begin{bmatrix} 2 & 5 \\ 3 & 2 \end{bmatrix}, \quad X = \begin{bmatrix} x \\ y \end{bmatrix}, \quad B = \begin{bmatrix} 40 \\ 30 \end{bmatrix}$$

Step 3: Solve in R

```r
# Coefficient matrix A
A <- matrix(c(2, 3, 5, 2), nrow = 2, byrow = TRUE)

# Resource vector B
B <- matrix(c(40, 30), nrow = 2)

# Solve for X (number of chairs and tables)
X <- solve(A, B)

# Display the result
X
```

```
           [,1]
[1,]  0.9090909
[2,] 12.7272727
```

Calculate an average using matrices

```
y<- father.son$fheight
mean(y)
```

```
[1] 67.6871
```

```
#using matrices:
N<- length(y)
Y<- matrix(y,N,1)
A<- matrix(1,N,1)
barY<- t(A)%*%Y/N
##equivalent to
barY<- crossprod(A,Y)/N
print(barY)
```

```
         [,1]
[1,] 67.6871
```

Sample variance

First, remember that the residuals are given by $e = Y - \hat{Y}$ where e is the nX1 vector of residuals, Y is the nx1 vector of observed values and $\hat{Y}$ is the nx1 vector of predicted values from our model. The formula for the sample variance s^2 of the residuals is

$$s^2 = \frac{\mathbf{e}^T\mathbf{e}}{n-p}$$

This gives you the average squared deviation of the residuals from their mean, which is an estimate of the variance of the errors in your model. in r:

```
e<- y -barY
crossprod(e)/N
```

```
          [,1]
[1,] 7.527313
```

Example:

```
# Sample data
Y <- matrix(c(2, 3, 5, 7, 9), ncol = 1)
Y
```

```
     [,1]
[1,]    2
[2,]    3
[3,]    5
[4,]    7
[5,]    9
```

```
X <- matrix(c(1, 1, 1, 1, 1, 1, 2, 3, 4, 5), ncol = 2)
X
```

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

```
# Calculate the coefficients (beta_hat)
beta_hat <- solve(crossprod(X)) %*% crossprod(X, Y)
beta_hat
```

```
     [,1]
[1,] -0.2
[2,]  1.8
```

```
# Calculate the predicted values (Y_hat)
Y_hat <- X %*% beta_hat

# Calculate the residuals
residuals <- Y - Y_hat

# Number of observations and parameters
n <- nrow(Y)
p <- ncol(X)
```

```r
# Calculate the sum of squared residuals using crossprod
ss_res <- crossprod(residuals)

# Calculate the sample variance
s_squared <- ss_res / (n - p)

s_squared
```

```
          [,1]
[1,] 0.1333333
```

Linear models represented by matrices

We can represent a linear model mathematically like this:

$$Y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_2 x_{i,p} + \varepsilon_i, i = 1, \ldots, n$$

$$Y_i = \beta_0 + \sum_{j=1}^{p} \beta_j x_{ij} + \varepsilon_i, \quad i = 1, \ldots, N$$

but using matrices we can simplify the formula to:

$$Y = X\beta + \epsilon$$

where: **Y** is the vector of data, **X** is a matrix with columns representing the different covariates or predictors, β represents the unknown parameters, and ε represents the vector of error terms.

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_N \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_{1,1} & \cdots & x_{1,P} \\ 1 & x_{2,1} & \cdots & x_{2,P} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N,1} & \cdots & x_{N,P} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_P \end{bmatrix}, \quad \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_N \end{bmatrix}$$

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_N \end{pmatrix} = \begin{pmatrix} 1 & x_{1,1} & \cdots & x_{1,p} \\ 1 & x_{2,1} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N,1} & \cdots & x_{N,p} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_N \end{pmatrix}$$

Residual sum of squares

Writing it this way we can calculate the values to minimize the *residual sum of squares (RSS)*. The RSS equation now looks like this:

$$(Y - X\beta)^T(Y - X\beta)$$

Least Squares Estimator (LSE) LSE is a method used to estimate the parameters of a linear model by minimizing the sum of the squared differences (errors) between observed and predicted values.

To find the $\hat{\beta}$ that minimizes this we solve by taking the derivative:

$$2X^T(Y - X\hat{\beta}) = 0X^TX\hat{\beta} = X^TY\hat{\beta} = (X^TX^{-1}X^TY)$$

In r:

```
x= father.son$fheight
y= father.son$sheight
X<- cbind(1,x)
betahat <- solve(t(X)%*%X)%*%t(X)%*%y
betahat
```

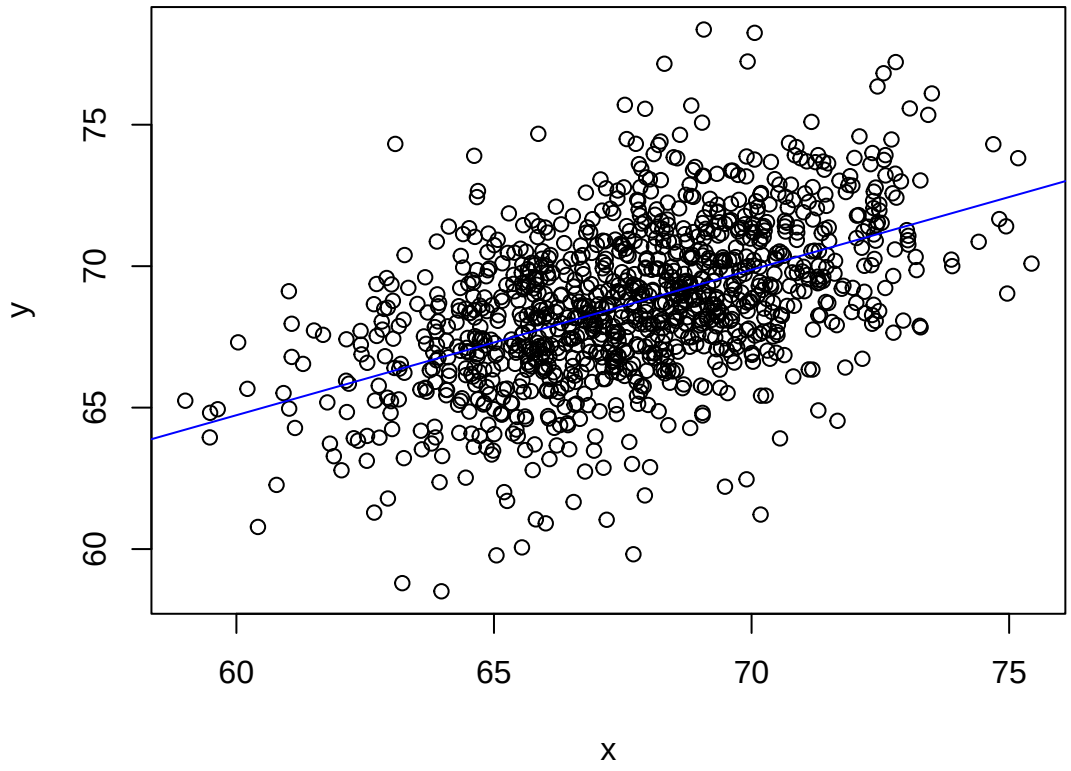
```
       [,1]
  33.886604
x  0.514093
```

```
# or equivalent code:
betahat <- solve(crossprod((X)))%*%crossprod(X,y)
betahat
```

```
       [,1]
  33.886604
x  0.514093
```

so now with $\hat{\beta}$ we can draw the linear model line.

```r
intercept = betahat[1,1]
slope= betahat[2, 1]

plot(x,y)
abline(intercept, slope, col = "blue")
```



Motivating Examples

Falling objects

Imagine you are Galileo in the 16th century trying to describe the velocity of a falling object. An assistant climbs the Tower of Pisa and drops a ball, while several other assistants record the position at different times. Let's simulate some data using the equations we know today and adding some measurement error:

```r
set.seed(1)
g <- 9.8 ##meters per second
n <- 25
tt <- seq(0,3.4,len=n) ##time in secs, note: we use tt because t is a base function
d <- 56.67  - 0.5*g*tt^2 + rnorm(n,sd=1) ##meters
```

The assistants hand the data to Galileo and this is what he sees:

```r
mypar()

plot(tt,d,ylab="Distance in meters",xlab="Time in seconds")
```

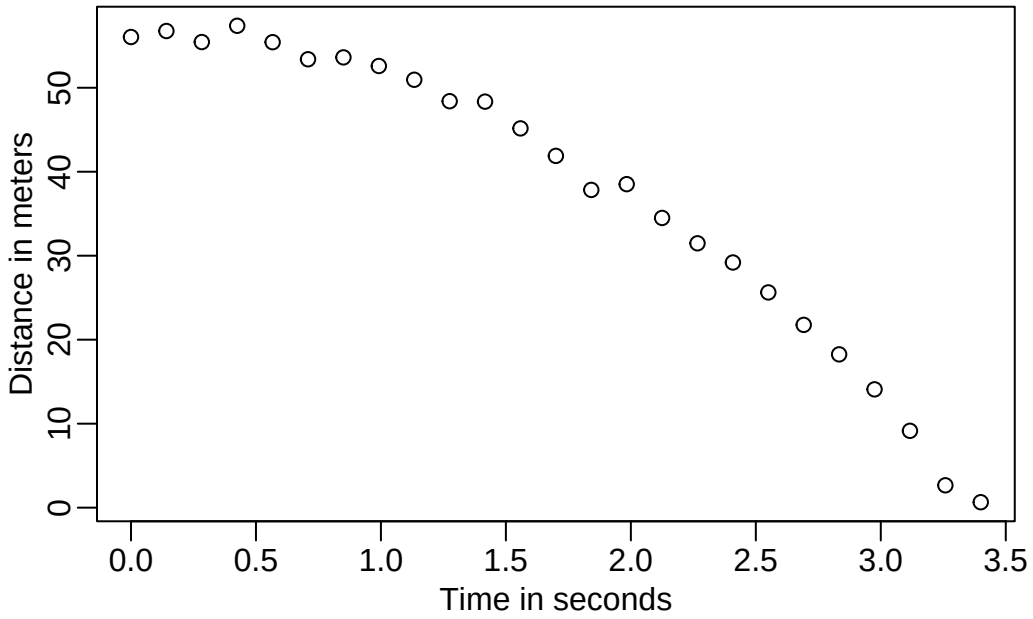


Figure 1: Simulated data for distance travelled versus time of falling object measured with error.

He does not know the exact equation, but by looking at the plot above he deduces that the position should follow a parabola. So he models the data with:

$$Y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \varepsilon_i, i = 1, \dots, n$$

With Y_i representing location, x_i representing the time, and ε_i accounting for measurement error. This is a linear model because it is a linear combination of known quantities (the x's) referred to as predictors or covariates and unknown parameters (the β's).

so we are have our measures `d` and we want to calculate the unknown parameters or betas: `h` is the hight of the Tower of Pisa and should result in a value similar to 56.67 `g` is the acceleration due to gravity, but we will actually get $\frac{1}{2}g$ due to physics. and we will have some errors due to measurament errors that we introduced in the formula above using `rnorm(n,sd=1)`

we want to find the values of beta that minimize the sum square of errors (RSS) Our first step is to create a matrix with `tt` and tt^2 and we add a column of 1s:

```
X<- cbind(1,tt,tt^2)
X
```

```
                tt
 [1,] 1 0.0000000  0.00000000
 [2,] 1 0.1416667  0.02006944
 [3,] 1 0.2833333  0.08027778
 [4,] 1 0.4250000  0.18062500
 [5,] 1 0.5666667  0.32111111
 [6,] 1 0.7083333  0.50173611
 [7,] 1 0.8500000  0.72250000
 [8,] 1 0.9916667  0.98340278
 [9,] 1 1.1333333  1.28444444
[10,] 1 1.2750000  1.62562500
[11,] 1 1.4166667  2.00694444
[12,] 1 1.5583333  2.42840278
[13,] 1 1.7000000  2.89000000
[14,] 1 1.8416667  3.39173611
[15,] 1 1.9833333  3.93361111
[16,] 1 2.1250000  4.51562500
[17,] 1 2.2666667  5.13777778
[18,] 1 2.4083333  5.80006944
[19,] 1 2.5500000  6.50250000
[20,] 1 2.6916667  7.24506944
[21,] 1 2.8333333  8.02777778
[22,] 1 2.9750000  8.85062500
[23,] 1 3.1166667  9.71361111
[24,] 1 3.2583333 10.61673611
[25,] 1 3.4000000 11.56000000
```

Now we choose a random matrix for beta of 3 rows (so we can multiply by X) Note that the values chosen for the matrix are arbitrary:

```
Beta <- matrix(c(55,0,5),3,1)
Beta
```

```
     [,1]
[1,]   55
[2,]    0
[3,]    5
```

the residuals will be y - X times Beta.

```
r<- d - X%*%Beta
r
```

```
               [,1]
 [1,]    1.04354619
 [2,]    1.65495582
 [3,]    0.03962139
 [4,]    1.47709330
 [5,]   -1.17949223
 [6,]   -4.11765588
 [7,]   -4.99532095
 [8,]   -7.32736279
 [9,]  -10.47021865
[10,]  -14.72907589
[11,]  -16.68696883
[12,]  -21.98134426
[13,]  -27.56224058
[14,]  -34.12288739
[15,]  -36.14781908
[16,]  -43.07962111
[17,]  -49.21019026
[18,]  -54.80685129
[19,]  -61.88352880
[20,]  -69.46228618
[21,]  -76.88602263
[22,]  -85.16905120
[23,]  -94.42018502
[24,] -105.42503920
[25,] -112.15417425
```

and the *Residual Sum of Squares (RSS)* will be:

```
RSS<- crossprod(r)
RSS
```

```
         [,1]
[1,] 66131.18
```

now to get the values for our unknown parameters we solve the *least squares estimate (LSE)* for those

```
betahat <- solve(crossprod(X))%*% crossprod(X,d)
betahat
```

```
         [,1]
   56.5317368
tt  0.5013565
   -5.0386455
```

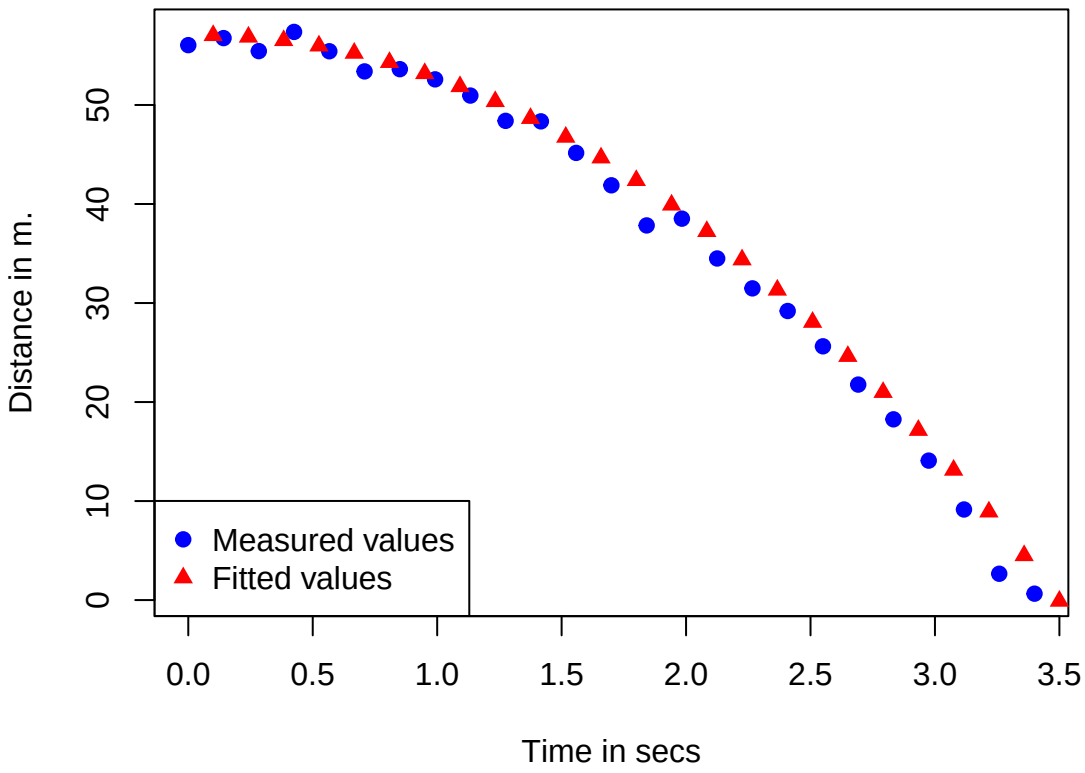
which gives us: 57.0212322 is the hight of the tower of Pisa -0.4223921 is the starting velocity (should be 0) -4.8175119 is half of the gravity acceleration.

which will result in our formula: `d <- 57.0212322 - 0.4223921 tt - 4.8175119 tt^2`

```
fun <- function(x){
  57.0212322 - (0.4223921*x) - (4.8175119*x^2)}
y_1 <- fun(tt)
```

Now we can plot the measured values along with the calculated values using the equation (note that I have slightly displaced the calculated values to avoid overlapping)

```
# Plot the measured values
plot(tt, d, xlab = "Time in secs", ylab = "Distance in m.", col = "blue", pch = 19)

# Add the fitted values to the plot
points(tt+0.1, y_1, col = "red", pch = 17)

# Add a legend to differentiate between the two lines
legend("bottomleft", legend = c("Measured values", "Fitted values"), col = c("blue", "red"),
```



We could have solved this without matrices using the linear model formula in r:

```
tt2<- tt^2
fit <- lm(d~tt+tt2)
summary(fit)
```

```
Call:
lm(formula = d ~ tt + tt2)

Residuals:
    Min      1Q  Median      3Q     Max
-2.5295 -0.4882  0.2537  0.6560  1.5455
```

```
Coefficients:
            Estimate Std. Error t value            Pr(>|t|)
(Intercept)  56.5317     0.5451 103.701 <0.0000000000000002 ***
tt            0.5014     0.7426   0.675               0.507
tt2          -5.0386     0.2110 -23.884 <0.0000000000000002 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9822 on 22 degrees of freedom
Multiple R-squared:  0.9973,    Adjusted R-squared:  0.997
F-statistic:  4025 on 2 and 22 DF,  p-value: < 0.00000000000000022
```

Standard Error in the context of linear models

We have shown how to find the least squares estimates with matrix algebra. These estimates are random variables as they are linear combinations of the data. For these estimates to be useful we also need to compute the standard errors.

It is useful to think about where randomness comes from. In our falling object example, randomness was introduced through measurement errors. Every time we rerun the experiment a new set of measurement errors will be made which implies our data will be random. This implies that our estimate of the gravitational constant will change. The constant is fixed, but our estimates are not. To see this we can run a Monte Carlo simulation. Specifically we will generate the data repeatedly and compute the estimate for the quadratic term each time.

```
g = 9.8 ## meters per second
h0 = 56.67
v0 = 0
n = 25
tt = seq(0,3.4,len=n) ##time in secs, t is a base function
y = h0 + v0 *tt  - 0.5* g*tt^2 + rnorm(n,sd=1)
```

now we act as if we didn't know `h0`, `v0` and `-0.5g` and use regression to estimate these. We can rewrite the models as `y=b0+b1 t+ b2 t^2 +e` and obtain LSE. Note that g will be `g=-2*b2`

To obtain the LSE in r

```
X = cbind(1,tt,tt^2)
A = solve(crossprod(X))%*%t(X)%*%y
```

so g will be measured after this experiment as:

```
-2*A[3]
```

```
[1] 9.788724
```

now we are going to repeat the experiment 100,000 times and calculate the standard deviation for the estimate g.

```
g = 9.8 ## meters per second
h0 = 56.67
v0 = 0
n = 25
tt = seq(0,3.4,len=n) ##time in secs, t is a base function
set.seed(1)
myfunc <- function(){
y = h0 + v0 *tt  - 0.5* g*tt^2 + rnorm(n,sd=1)

X = cbind(1,tt,tt^2)
A = solve(crossprod(X))%*%t(X)%*%y
A
g<- -2*A[3]
return (g)
}

gs<- replicate(100000,myfunc())
sd(gs)
```

```
[1] 0.429747
```

Now we are going to use matrix algebra to compute standard errors of regression coefficients. We will start by defining the *variance covariance matrix*.

Variance-covariance matrix (Advanced)

As a first step we need to define the *variance-covariance matrix*, Σ. For a vector of random variables, $\mathbf{Y}$, we define Σ as the matrix with the i, j entry:

$$\Sigma_{i,j} \equiv \mathrm{Cov}(Y_i, Y_j)$$

The covariance is equal to the variance if $i = j$ and equal to 0 if the variables are independent. In the kinds of vectors considered up to now, for example, a vector $\mathbf{Y}$ of individual observations Y_i sampled from a population, we have assumed independence of each observation and assumed the Y_i all have the same variance σ^2, so the variance-covariance matrix has had only two kinds of elements:

$$\text{Cov}(Y_i, Y_i) = \text{var}(Y_i) = \sigma^2$$

$$\text{Cov}(Y_i, Y_j) = 0, \text{ for } i \neq j$$

which implies that $\Sigma = \sigma^2\mathbf{I}$ with $\mathbf{I}$, the identity matrix.

Later, we will see a case, specifically the estimate coefficients of a linear model, $\hat{\beta}$, that has non-zero entries in the off diagonal elements of Σ. Furthermore, the diagonal elements will not be equal to a single value σ^2.

Variance of a linear combination

A useful result provided by linear algebra is that the variance covariance-matrix of a linear combination $\mathbf{AY}$ of $\mathbf{Y}$ can be computed as follows:

$$\text{var}(\mathbf{AY}) = \mathbf{A}\text{var}(\mathbf{Y})\mathbf{A}^\top$$

For example, if Y_1 and Y_2 are independent both with variance σ^2 then:

$$\text{var}\{Y_1 + Y_2\} = \text{var}\left\{(1 \quad 1)\begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix}\right\}$$

$$= (1 \quad 1)\,\sigma^2\mathbf{I}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = 2\sigma^2$$

as we expect. We use this result to obtain the standard errors of the LSE (least squares estimate).

Least Squares Estimates (LSE) standard errors (Advanced)

Note that the LSE $\widehat{\beta}$ is a linear combination of $\mathbf{Y}$: $\mathbf{AY}$ with $\mathbf{A} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$, so we can use the equation above to derive the variance of our estimates:

$$\text{var}(\widehat{\beta}) = \text{var}((\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{Y}) =$$

$$(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\text{var}(Y)((\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top)^\top =$$

$$(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\sigma^2\mathbf{I}((\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top)^\top =$$

$$\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1} =$$

$$\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}$$

The diagonal of the square root of this matrix contains the standard error of our estimates.

Estimating σ^2

To obtain an actual estimate in practice from the formulas above, we need to estimate σ^2. Previously we estimated the standard errors from the sample. However, the sample standard deviation of Y is not σ because Y also includes variability introduced by the deterministic part of the model: $\mathbf{X}\beta$. The approach we take is to use the residuals.

We form the residuals like this:

$$\mathbf{r} \equiv \widehat{\varepsilon} = \mathbf{Y} - \mathbf{X}\widehat{\beta}$$

Both $\mathbf{r}$ and $\widehat{\varepsilon}$ notations are used to denote residuals.

Then we use these to estimate, in a similar way, to what we do in the univariate case:

$$s^2 \equiv \widehat{\sigma}^2 = \frac{1}{N-p}\mathbf{r}^\top\mathbf{r} = \frac{1}{N-p}\sum_{i=1}^{N} r_i^2$$

Here N is the sample size and p is the number of columns in $\mathbf{X}$ or number of parameters (including the intercept term β_0). The reason we divide by $N - p$ is because mathematical theory tells us that this will give us a better (unbiased) estimate.

Let's try this in R and see if we obtain the same values as we did with the Monte Carlo simulation above:

```
n <- nrow(father.son)
N <- 50
index <- sample(n,N)
sampledat <- father.son[index,]
x <- sampledat$fheight
y <- sampledat$sheight
X <- model.matrix(~x)

N <- nrow(X)
p <- ncol(X)

XtXinv <- solve(crossprod(X))

resid <- y - X %*% XtXinv %*% crossprod(X,y)

s <- sqrt(sum(resid^2)/(N-p))
ses <- sqrt(diag(XtXinv))*s
```

Let's compare to what `lm` provides:

```
summary(lm(y~x))$coef[,2]
```

```
(Intercept)           x
  8.7487832   0.1284109
```

```
ses
```

```
(Intercept)           x
  8.7487832   0.1284109
```

They are identical because they are doing the same thing. Also, note that we approximate the Monte Carlo results:

```
apply(betahat,2,sd)
```

```
[1] 34.06124
```

Linear combination of estimates

Imagine that you estimated the effects of several treatments and now you are interested in the difference in the effects of two of those treatments. You already have the $\widehat{\beta}$ and want to calculate $\widehat{\beta}_2 - \widehat{\beta}_1$

If we want to compute the standard deviation of a linear combination of estimates such as $\widehat{\beta}_2 - \widehat{\beta}_1$, this is a linear combination of $\widehat{\beta}$:

$$\widehat{\beta}_2 - \widehat{\beta}_1 = (0 \quad -1 \quad 1 \quad 0 \quad \dots \quad 0) \begin{pmatrix} \widehat{\beta}_0 \\ \widehat{\beta}_1 \\ \widehat{\beta}_2 \\ \vdots \\ \widehat{\beta}_p \end{pmatrix}$$

Using the above, we know how to compute the variance covariance matrix of $\widehat{\beta}$.

CLT and t-distribution

We have shown how we can obtain standard errors for our estimates. However, as we learned in the first chapter, to perform inference we need to know the distribution of these random variables. The reason we went through the effort to compute the standard errors is because the CLT applies in linear models. If N is large enough, then the LSE will be normally distributed with mean β and standard errors as described. For small samples, if the ε are normally distributed, then the $\widehat{\beta} - \beta$ follow a t-distribution. We do not derive this result here, but the results are extremely useful since it is how we construct p-values and confidence intervals in the context of linear models.

Code versus math

The standard approach to writing linear models either assume the values in $\mathbf{X}$ are fixed or that we are conditioning on them. Thus $\mathbf{X}\beta$ has no variance as the $\mathbf{X}$ is considered fixed. This is why we write $\text{var}(Y_i) = \text{var}(\varepsilon_i) = \sigma^2$. This can cause confusion in practice because if you, for example, compute the following:

```
x =  father.son$fheight
beta =  c(34,0.5)
var(beta[1]+beta[2]*x)
```

```
[1] 1.883576
```

it is nowhere near 0. This is an example in which we have to be careful in distinguishing code from math. The function `var` is simply computing the variance of the list we feed it, while the mathematical definition of variance is considering only quantities that are random variables. In the R code above, `x` is not fixed at all: we are letting it vary, but when we write $\text{var}(Y_i) = \sigma^2$ we are imposing, mathematically, `x` to be fixed. Similarly, if we use R to compute the variance of Y in our object dropping example, we obtain something very different than $\sigma^2 = 1$ (the known variance):

```
n <- length(tt)
y <- h0 + v0*tt  - 0.5*g*tt^2 + rnorm(n,sd=1)
var(y)
```

```
[1] 334.0487
```

Again, this is because we are not fixing `tt`.

Exercise

We are going to calculate the standard error of $\hat{\beta}$ for the heights of father and son.

$$SE(\hat{\beta}) = sqrtVar(\hat{\beta})$$

$$Var(\hat{\beta}) = \sigma^2(X^TX)^{-1}$$

First, we want to estimate σ^2, the variance of Y. As we have seen in the previous unit, the random part of Y is only coming from ϵ, because we assume $X\beta$ is fixed. So we can try to estimate the variance of the epsilons from the residuals: the Y minus the fitted values from the linear model.

```
x = father.son$fheight
y = father.son$sheight
n = length(y)
N = 50
set.seed(1)
index = sample(n,N)
sampledat = father.son[index,]
x = sampledat$fheight
y = sampledat$sheight
betahat = lm(y~x)$coef
```

first we calculate the $\widehat{Y}$ values from a linear model:

```
fit = lm(y ~ x)
Yhat<- fit$fitted.values
```

and now we can calculate the sum of the squares residuals SSR that will be the diffeence between y_i and $\widehat{y}_i$

```
res2 <- (y- Yhat)^2
SSR <- sum(res2)
SSR
```

```
[1] 331.2952
```

Our estimate of σ^2 will be the sum of squared residuals divided by (N-p), the sample size minus the number of terms in the model. Since we have a sample of 50 and 2 terms in the model (the intercept and a slope), our estimate will be:

```
sigma2 <- SSR/(N-2)
sigma2
```

```
[1] 6.901984
```

Now we form the matrix X, this can be done by combining a column of 1s with a column with the father's heights.

```
X = cbind(rep(1,N),x)
```

And now we calculate $(X^T X)^{-1}$

```
I<-solve(t(X)%*%X)
I
```

```
                      x
  14.5639039 -0.214704345
x -0.2147043  0.003169572
```

Now we are only one step away from getting the standard error of $\hat{\beta}$ Take the diagonals of $(X^T X)^{-1}$ and multiply by our estimate of σ^2. This is the estimated variance of $\hat{\beta}$

```
varianceBeta <- diag(I)*sigma2
varianceBeta
```

```
                      x
100.51983131   0.02187634
```

and finally the SE will be the square root of this

```
SE<- sqrt(varianceBeta)
SE
```

```
                   x
10.0259579  0.1479065
```

this gives us two numbers, the standard error of the intercept and the standard error for the slope. (Note that the standard error estimate is also printed int eh second column of `summary(fit)`)

```
summary(fit)
```

```

Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-7.8912 -1.3527  0.3593  1.5703  4.4794

Coefficients:
```

```
            Estimate Std. Error t value  Pr(>|t|)
(Intercept)  44.3377    10.0260   4.422 0.0000558 ***
x             0.3530     0.1479   2.387     0.021 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.627 on 48 degrees of freedom
Multiple R-squared:  0.1061,    Adjusted R-squared:  0.08747
F-statistic: 5.697 on 1 and 48 DF,  p-value: 0.02098
```

Linear models as matrix multiplication.

Suppose we have a linear model with 4 variables

$$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \beta_3 X_{i3} + \beta_4 X_{i4} + \epsilon_i$$

that can be written as:

$$\mathbf{Y} = \mathbf{X}\beta + \epsilon$$

where

$$\mathbf{Y} = \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_N \end{pmatrix}$$

$$\mathbf{X} = \begin{pmatrix} 1 & X_{11} & X_{12} & X_{13} & X_{14} \\ 1 & X_{21} & X_{22} & X_{23} & X_{24} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & X_{N1} & X_{N2} & X_{N3} & X_{N4} \end{pmatrix}$$

$$\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \end{pmatrix}$$

and

$$\epsilon = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_N \end{pmatrix}$$

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \end{bmatrix} = \begin{bmatrix} 1 & X_{11} & X_{12} & X_{13} & X_{14} \\ 1 & X_{21} & X_{22} & X_{23} & X_{24} \\ 1 & X_{31} & X_{32} & X_{33} & X_{34} \\ 1 & X_{41} & X_{42} & X_{43} & X_{44} \\ 1 & X_{51} & X_{52} & X_{53} & X_{54} \\ 1 & X_{61} & X_{62} & X_{63} & X_{64} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \end{bmatrix}$$

the sum of least squares will be:

$$\sum_{i=1}^{N} (Y_i - \beta_0 - \beta_1 X_{i1} - \beta_2 X_{i2} - \beta_3 X_{i3} - \beta_4 X_{i4})^2$$

in matrix notation that would be

$$(Y - \beta X)^T (Y - \beta X)$$

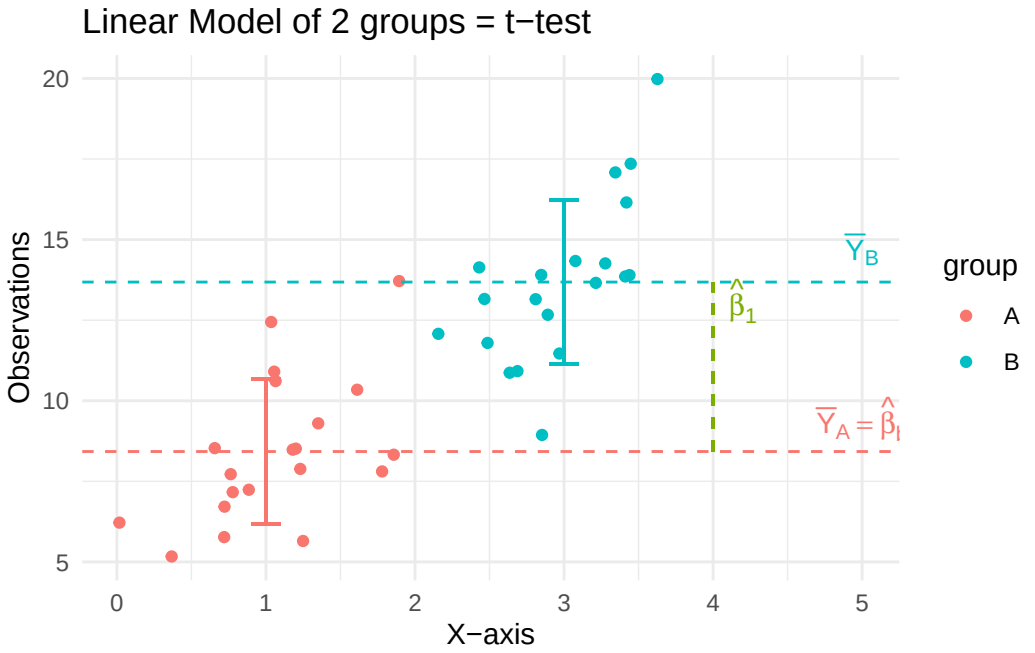
t-test

When we are doing a t-test we calculate the average in one group and substract from the average in the other group and that is an estimate of the population average.

We are going to make something difference. We will get a baseline, $\hat{\beta}_0$ and we are going to calculate the difference between the two groups $\hat{\beta}_1$

```
set.seed(123)

# Generate data for two groups
group <- rep(c("A", "B"), each = 20)
x <- c(rnorm(20, mean = 1, sd = 0.5), rnorm(20, mean = 3, sd = 0.5))  # Shift x-values for gr
y <- 5 + 3 * x + rnorm(40, sd = 2)

data <- data.frame(group, x, y)

mean_A <- mean(data$y[data$group == "A"])
mean_B <- mean(data$y[data$group == "B"])
sd_A <- sd(data$y[data$group == "A"])
sd_B <- sd(data$y[data$group == "B"])

# Plot the data
ggplot(data, aes(x = x, y = y, color = group)) +
```

```r
  geom_point() +
  geom_hline(aes(yintercept = mean_A), color = "#f8766d", linetype = "dashed") +
  geom_hline(aes(yintercept = mean_B), color = "#00bfc4", linetype = "dashed") +
  geom_segment(aes(x = 4, y = mean_A, xend = 4, yend = mean_B), linetype = "dashed", color="#
  annotate("text", x = 5, y = mean_A + 1, label = expression(bar(Y)[A] == hat(beta)[b]), colo
  annotate("text", x = 5, y = mean_B + 1, label = expression(bar(Y)[B]), color = "#00bfc4") +
  annotate("text", x = 4.2, y = (mean_A + mean_B) / 2, label = expression(hat(beta)[1]), vjus
  geom_errorbar(aes(x = 1, ymin = mean_A - sd_A, ymax = mean_A + sd_A), width = 0.2, color =
  geom_errorbar(aes(x = 3, ymin = mean_B - sd_B, ymax = mean_B + sd_B), width = 0.2, color =
  labs(title = "Linear Model of 2 groups = t-test",
       x = "X-axis",
       y = "Observations") +
  theme_minimal()
```



$$\bar{Y_A} = \hat{\beta}_0$$

$$\bar{Y_B} = \hat{\beta}_0 + \hat{\beta}_1$$

$$\bar{Y_A} = 1 \times \hat{\beta}_0 + 0 \times \hat{\beta}_1$$

$$\bar{Y_B} = 1 \times \hat{\beta}_0 + 1 \times \hat{\beta}_1$$

And if we had three groups:

$$\bar{Y_A} = \hat{\beta_0}$$
$$\bar{Y_B} = \hat{\beta_0} + \hat{\beta_1}$$
$$\bar{Y_C} = \hat{\beta_0} + \hat{\beta_2}$$
$$\bar{Y_A} = 1 \times \hat{\beta_0} + 0 \times \hat{\beta_1} + 0 \times \hat{\beta_2}$$

$$\bar{Y_B} = 1 \times \hat{\beta_0} + 1 \times \hat{\beta_1} + 0 \times \hat{\beta_2}$$
$$\bar{Y_C} = 1 \times \hat{\beta_0} + 0 \times \hat{\beta_1} + 1 \times \hat{\beta_2}$$

how can we create this using r:

```r
x<- factor(c(1,1,2,2,3,3))
model.matrix(~ x)
```

```
  (Intercept) x2 x3
1           1  0  0
2           1  0  0
3           1  1  0
4           1  1  0
5           1  0  1
6           1  0  1
attr(,"assign")
[1] 0 1 1
attr(,"contrasts")
attr(,"contrasts")$x
[1] "contr.treatment"
```

with two parameters:

```r
x<- factor(c(1,1,1,1,2,2,2))
y<- factor(c("a","a","b","b","a","a","b"))
cbind(as.character(x),as.character(y))
```

```
     [,1] [,2]
[1,] "1"  "a"
[2,] "1"  "a"
[3,] "1"  "b"
[4,] "1"  "b"
[5,] "2"  "a"
[6,] "2"  "a"
[7,] "2"  "b"
```

```
model.matrix(~ x+y)
```

```
  (Intercept) x2 yb
1           1  0  0
2           1  0  0
3           1  0  1
4           1  0  1
5           1  1  0
6           1  1  0
7           1  1  1
attr(,"assign")
[1] 0 1 2
attr(,"contrasts")
attr(,"contrasts")$x
[1] "contr.treatment"

attr(,"contrasts")$y
[1] "contr.treatment"
```

and we can compute some calculations if we want using `I()`

```
x<-c(1,2,3,4,5)
model.matrix(~x +I(x^2))
```

```
  (Intercept) x I(x^2)
1           1 1      1
2           1 2      4
3           1 3      9
4           1 4     16
5           1 5     25
attr(,"assign")
[1] 0 1 2
```

example: Suppose we have an experiment with the following design: on three different days, we perform an experiment with two treated and two control samples. We then measure some outcome y_i and we want to test the effect of condition, while controlling for whatever differences might have occurred due to the different day. Assume that the true condition effect is the same for each day:

```
day<- c("A","B","C")
treated<- c(2,2,2)
control<-c(2,2,2)
```

produce a design matrix that let's us analyse the effect of condition, controlling for the different days:

```
condition<- cbind(factor(treated),factor(control))
model.matrix(~factor(day)+condition)
```

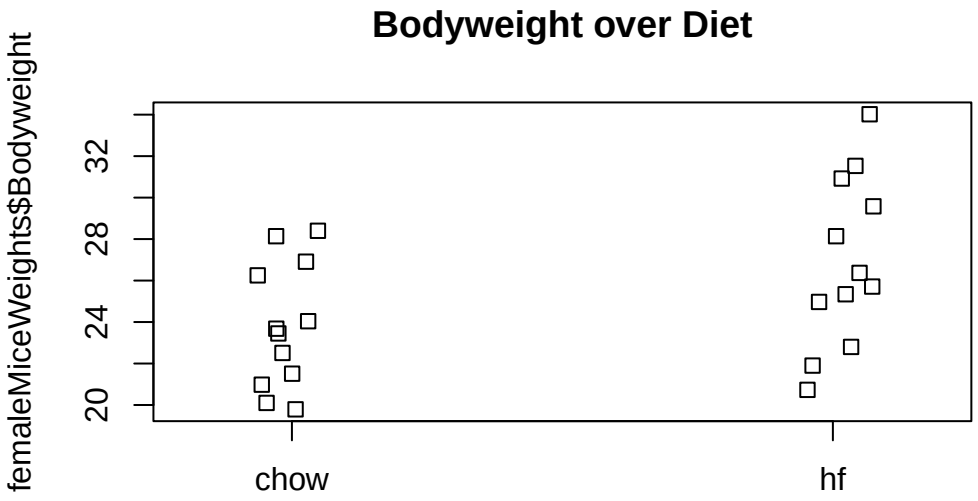
```
  (Intercept) factor(day)B factor(day)C condition1 condition2
1           1            0            0          1          1
2           1            1            0          1          1
3           1            0            1          1          1
attr(,"assign")
[1] 0 1 1 2 2
attr(,"contrasts")
attr(,"contrasts")$`factor(day)`
[1] "contr.treatment"
```

Example:

we are going to use the dataset of female mice bodyweight to see how matrices apply to linear models.

We load the dataset and create the matrix. Notice how we can change the reference level using relevel or factor, this will change which values for the variable diet get 0 and which ones get a 1. We will usually use the control group as our reference group.

```
femaleMiceWeights <- read_csv("data/femaleMiceWeights.csv")
View(femaleMiceWeights)


stripchart(femaleMiceWeights$Bodyweight ~ femaleMiceWeights$Diet,
  vertical= TRUE, method = "jitter",
  main = "Bodyweight over Diet")
```




```
model.matrix(~ Diet, data= femaleMiceWeights)
```

```
   (Intercept) Diethf
1            1      0
2            1      0
3            1      0
4            1      0
5            1      0
6            1      0
7            1      0
8            1      0
9            1      0
10           1      0
11           1      0
12           1      0
13           1      1
14           1      1
15           1      1
16           1      1
17           1      1
18           1      1
19           1      1
```

```
20           1      1
21           1      1
22           1      1
23           1      1
24           1      1
attr(,"assign")
[1] 0 1
attr(,"contrasts")
attr(,"contrasts")$Diet
[1] "contr.treatment"
```

if we run a linear model over the data:

```
fit <- lm(Bodyweight ~ Diet, data= femaleMiceWeights)
summary(fit)
```

```
Call:
lm(formula = Bodyweight ~ Diet, data = femaleMiceWeights)

Residuals:
    Min      1Q  Median      3Q     Max
-6.1042 -2.4358 -0.4138  2.8335  7.1858

Coefficients:
            Estimate Std. Error t value            Pr(>|t|)
(Intercept)   23.813      1.039  22.912 <0.0000000000000002 ***
Diethf         3.021      1.470   2.055              0.0519 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.6 on 22 degrees of freedom
Multiple R-squared:  0.1611,    Adjusted R-squared:  0.1229
F-statistic: 4.224 on 1 and 22 DF,  p-value: 0.05192
```

we see we get in the coefficients part we get a value for each column in the matrix, in our case one for the intercept and one for the non reference Diet.
The estimate for the Diet hf is 3.021 which means that the difference in weight between the control group (chow diet) and the high fat diet is 3.021 grams.

The second column gives us the standard error.

The third column gives us a t-value that is the estimate divided by the standard error `23.813/1.039` for the intercept and `3.021/1.470`

The fourth column gives us a p.value that is the probability of seeing such a large t-value in absolute value.

The fifth column gives you starts or a dot that symbolizes whether that p-value is below the usual cut-off significant levels α

if we want to get only the coefficients we can do it by using `coef(fit)`

We know that the maths behind the linear model are

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

and to prove it with our data:

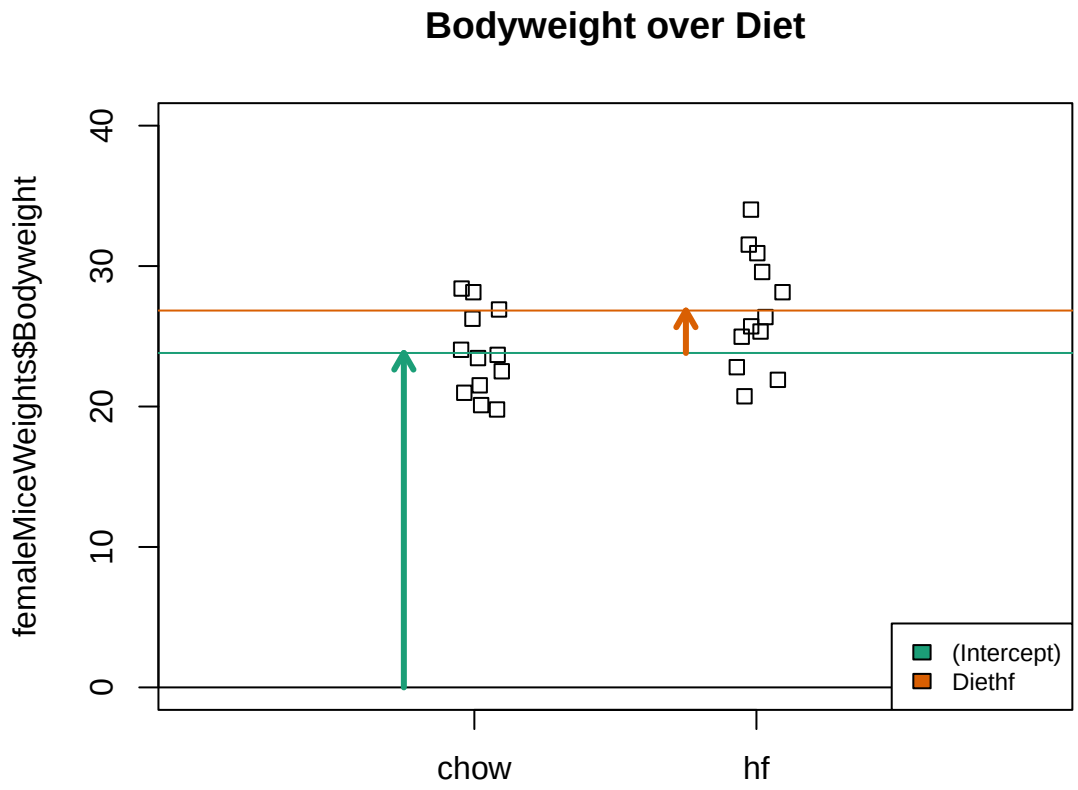
```
y<- femaleMiceWeights$Bodyweight
X<- model.matrix(~ Diet, data= femaleMiceWeights)
solve(t(X) %*% X) %*% t(X) %*% y
```

```
                [,1]
(Intercept) 23.813333
Diethf       3.020833
```

visualizing it:

```
stripchart(femaleMiceWeights$Bodyweight ~ femaleMiceWeights$Diet,
  vertical= TRUE, method = "jitter",
  main = "Bodyweight over Diet",
  ylim=c(0,40),
  xlim=c(0,3)
 )
a<- -0.25
lgth<- .1

cols<- brewer.pal(3,"Dark2")
abline(h=0)
arrows(1+a,0,1+a, #position
  coef(fit)[1], #where the arrow ends (intercept)
  lwd=3, #
  col=cols[1], #color
  length= lgth #arrow point size.
```

```
 )
abline(h=coef(fit)[1], col=cols[1]) #intercept (mean of chow)
arrows(2+a,coef(fit)[1],2+a,
  coef(fit)[1]+coef(fit)[2],
  lwd=3,
  col=cols[2],
  length= lgth)
abline(h=coef(fit)[1]+coef(fit)[2], col=cols[2])
legend("bottomright", names(coef(fit)), fill=cols, cex=.75, bg="white")
```



Finally, we can check that we obtain the same results if we run a t-test:

```
s<- split(femaleMiceWeights$Bodyweight, femaleMiceWeights$Diet)
s
```

```
$chow
 [1] 21.51 28.14 24.04 23.45 23.68 19.79 28.40 20.98 22.51 20.10 26.91 26.25

$hf
 [1] 25.71 26.37 22.80 25.34 24.97 28.14 29.58 30.92 34.02 21.90 31.53 20.73
```

```
testResult <- t.test(s[["hf"]],s[["chow"]], var.equal=TRUE)
testResult
```

```
    Two Sample t-test

data:  s[["hf"]] and s[["chow"]]
t = 2.0552, df = 22, p-value = 0.05192
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.02748516  6.06915183
sample estimates:
mean of x mean of y
 26.83417  23.81333
```

```
summary(fit)$coefficients[2,3]
```

```
[1] 2.055174
```

```
testResult$statistic
```

```
       t
2.055174
```

Though the linear model in this case is equivalent to a t-test, we will soon explore more complicated designs, whre the linear model is a useful extension (cofounging variables, testing contrast of terms, testing interactions, testing many terms at once etc.) but for now we can review the mathematics on why these produce the same t-value and therefore the same p-value.

We already know that the numerator of the t-value is the difference between the average of the groups, so we only have to see that the denominator is also the same. In the linear model, we saw how to calculate the standard error using the design matrix X and the estimate of σ^2 from the residuals. The estimate of the variance σ^2 was the sum of the squared residuals divided by (N-p) where N is the total number of samples and p is the number of terms (in our example an intercept and a group indicator so p=2).

In the t-test, the denominator of the t-value is the standard error of the difference.

The t-test formula for the standard error of the difference, if we assume equal variance in the two groups is:

$$SE = \sqrt{var(diff)} = \sqrt{\sigma^2_\Delta}$$

and we know that:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

now the *variance of the difference between the means of two samples*:

$$var(diff) = \left(\frac{1}{n_x} + \frac{1}{n_y}\right)\frac{\sum(x_i - \mu_x)^2 + \sum(y_i - \mu_y)^2}{(n_x + n_y - 2)}$$

this formula calculates the variance of the difference between two sample means where - n_x and n_y are the sample sizes of the two groups. - x_i and y_i are individual data points within each group

- μ_x and μ_y are the means of the two groups.
- $(\frac{1}{n_x} + \frac{1}{n_y})$ adjusts for the sample sizes.
- $\sum(x_i - \mu_x)^2 + \sum(y_i - \mu_y)^2$ this sums up the squared deviations from the mean for each group.
- $\frac{\sum(x_i-\mu_x)^2+\sum(y_i-\mu_y)^2}{(n_x+n_y-2)}$ this calculates the pooled variance, considering both groups together.
- 2 is the degrees of freedom for the pooled variance calculation.

The variance of the difference tells you how spread out the difference between the group means are likely to be.

The standard Error of the estimated coefficient $\hat{\beta}$ is :

$$se(\hat{\beta}) = \sqrt{s^2(X^T X)^{-1}_{ii}}$$

where $(X^T X)^{-1}_{ii}$ is the i-th diagonal element of this inverse matrix, which corresponds to the variance of the i-th coefficient estimate.

Crossed designs

imagine we are running an experiment and we have two different treatments, group A is the control group and receives no treatment (a), group B receives treatment 1 (b), group C receives treatment 2 (c) and a fourth group receives treatment 1 and 2 (d). In this case we consider the effects of receiving both treatments additive. If we write down a model it will look like this:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}$$

we can see that the first two rows are no treatment, the rows 3rd and 4th receive treatment 1 but no treatment 2 , the 5th and the 6th rows receive treatment 2 but no treatment 1 and the last two rows receive both treatments.

If the effects of treatment 1 + treatment 2 are not additive, then we need to plug in a fourth element that will gives us a different mean for each group and we call that fourth term *interaction*

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_{1:2} \end{pmatrix}$$

Interactions and Contrast

As a running example to learn about more complex linear models, we will be using a dataset which compares the different frictional coefficients on the different legs of a spider. Specifically, we will be determining whether more friction comes from a pushing or pulling motion of the leg.

To remind ourselves how the simple two-group linear model looks, we will subset the data to include only the L1 leg pair, and run `lm`:

```
spider <- read.csv2("data/spider_wolff_gorb_2013.csv")
spider$friction <- as.numeric(spider$friction)
spider.sub <- spider[spider$leg == "L1",]
fit <- lm(friction ~ type, data=spider.sub)
summary(fit)
```

```
Call:
lm(formula = friction ~ type, data = spider.sub)

Residuals:
     Min       1Q   Median       3Q      Max
-0.33147 -0.10735 -0.04941 -0.00147  0.76853

Coefficients:
            Estimate Std. Error t value             Pr(>|t|)
(Intercept)  0.92147    0.03827  24.078 < 0.0000000000000002 ***
typepush    -0.51412    0.05412  -9.499    0.000000000000057 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2232 on 66 degrees of freedom
Multiple R-squared:  0.5776,    Adjusted R-squared:  0.5711
F-statistic: 90.23 on 1 and 66 DF,  p-value: 0.00000000000005698
```

```
(coefs <- coef(fit))
```

```
(Intercept)    typepush
  0.9214706  -0.5141176
```

These two estimated coefficients are the mean of the pull observations (the first estimated coefficient) and the difference between the means of the two groups (the second coefficient). We can show this with R code:

```
s <- split(spider.sub$friction, spider.sub$type)
mean(s[["pull"]])
```

```
[1] 0.9214706
```

```
mean(s[["push"]]) - mean(s[["pull"]])
```

```
[1] -0.5141176
```

We can form the design matrix, which was used inside `lm`:

```
X <- model.matrix(~ type, data=spider.sub)
colnames(X)
```

```
[1] "(Intercept)" "typepush"
```

```
head(X)
```

```
  (Intercept) typepush
1           1        0
2           1        0
3           1        0
4           1        0
5           1        0
6           1        0
```

```
tail(X)
```

```
   (Intercept) typepush
63           1        1
64           1        1
65           1        1
66           1        1
67           1        1
68           1        1
```

Now we'll make a plot of the **X** matrix by putting a black block for the 1's and a white block for the 0's. This plot will be more interesting for the linear models later on in this script. Along the y-axis is the sample number (the row number of the `data`) and along the x-axis is the column of the design matrix **X**. If you have installed the *rafalib* library, you can make this plot with the `imagemat` function:

```
imagemat(X, main="Model matrix for linear model with one variable")
```

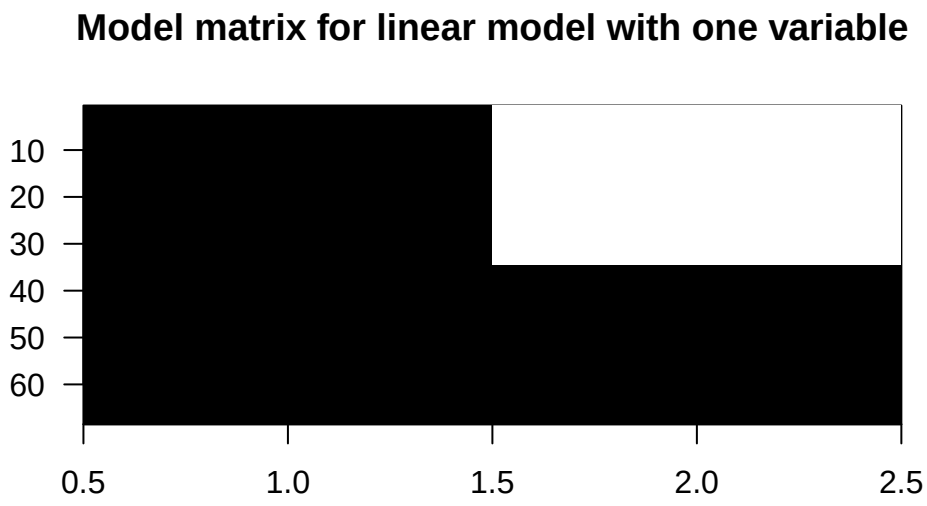


Figure 2: Model matrix for linear model with one variable.

The black represent a 1 and the white represents 0 in the matrix.

We can visualize the data in a strip plot as well

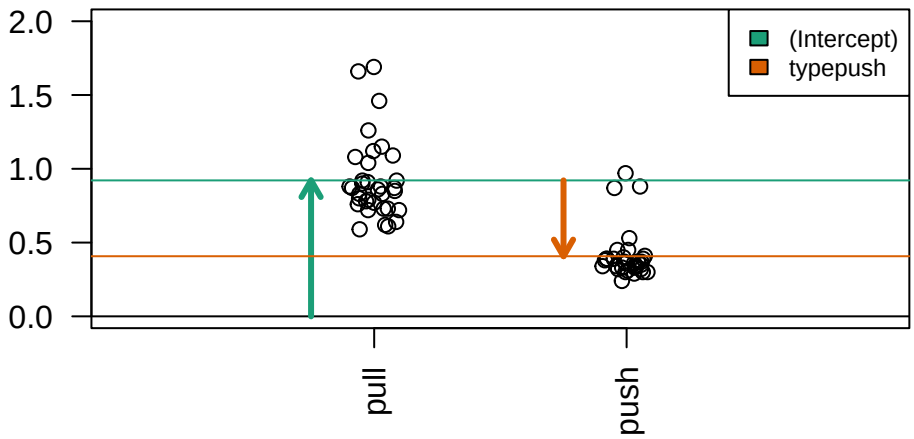


Figure 3: Diagram of the estimated coefficients in the linear model. The green arrow indicates the Intercept term, which goes from zero to the mean of the reference group (here the 'pull' samples). The orange arrow indicates the difference between the push group and the pull group, which is negative in this example. The circles show the individual samples

linear model with two variables

Now we are going to use the 4 pairs of legs to show a more complex linear model.

```
X <- model.matrix(~ type + leg, data=spider)
colnames(X)
```

```
[1] "(Intercept)" "typepush"    "legL2"       "legL3"       "legL4"
```

```
head(X)
```

```
  (Intercept) typepush legL2 legL3 legL4
1           1        0     0     0     0
2           1        0     0     0     0
3           1        0     0     0     0
```

```
4           1        0     0     0     0
5           1        0     0     0     0
6           1        0     0     0     0
```

```
imagemat(X, main="Model matrix for linear model with two factors")
```

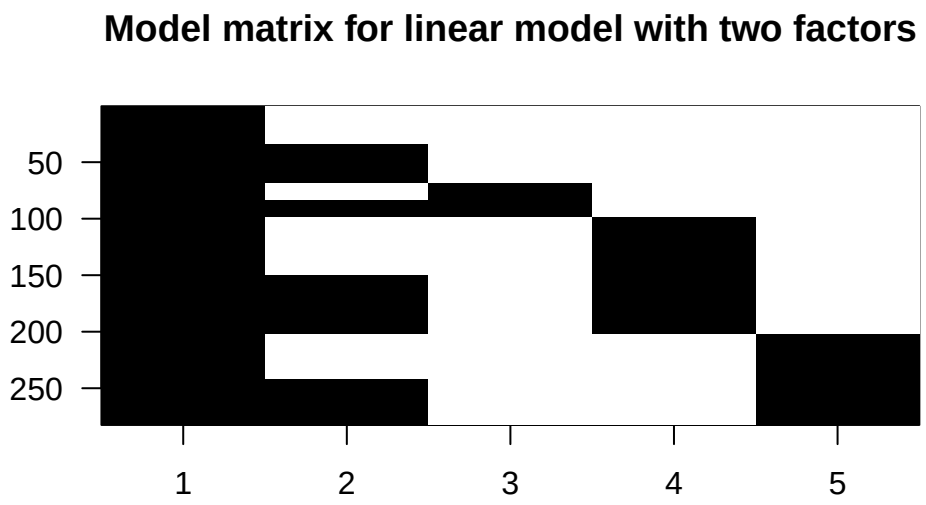


Figure 4: Image of the model matrix for a formula with type + leg

We have a row for each data point. The first column is the intercept, and so it has 1's (black) for all samples. The second column expresses if the data is for a pull or push: it has 1's for the push samples, and we can see that there are four groups of them (one for each pair of legs). Finally, the third, fourth and fifth columns expresses what leg pair we are referencing and have 1's for the L2, L3 and L4 samples. The L1 samples do not have a column, because *L1* is the reference level for `leg`. Similarly, there is no *pull* column, because *pull* is the reference level for the `type` variable.

To estimate coefficients for this model, we use `lm` with the formula `~ type + leg`. We'll save the linear model to `fitTL` standing for a *fit* with *Type* and *Leg*.

```
fitTL <- lm(friction ~ type + leg, data=spider)
summary(fitTL)
```

```
Call:
lm(formula = friction ~ type + leg, data = spider)

Residuals:
     Min       1Q   Median       3Q      Max
-0.46392 -0.13441 -0.00525  0.10547  0.69509

Coefficients:
            Estimate Std. Error t value             Pr(>|t|)
(Intercept)  1.05392    0.02816  37.426 < 0.0000000000000002 ***
typepush    -0.77901    0.02482 -31.380 < 0.0000000000000002 ***
legL2        0.17192    0.04569   3.763             0.000205 ***
legL3        0.16049    0.03251   4.937   0.0000013710214077 ***
legL4        0.28134    0.03438   8.183   0.0000000000000101 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2084 on 277 degrees of freedom
Multiple R-squared:  0.7916,    Adjusted R-squared:  0.7886
F-statistic:   263 on 4 and 277 DF,  p-value: < 0.00000000000000022
```

```
(coefs <- coef(fitTL))
```

```
(Intercept)    typepush       legL2       legL3       legL4
  1.0539153  -0.7790071   0.1719216   0.1604921   0.2813382
```

R uses the name `coefficient` to denote the component containing the least squares **estimates**. It is important to remember that the coefficients are parameters that we do not observe, but only estimate.

We can make the same plot as before, with arrows for each of the estimated coefficients in the model.

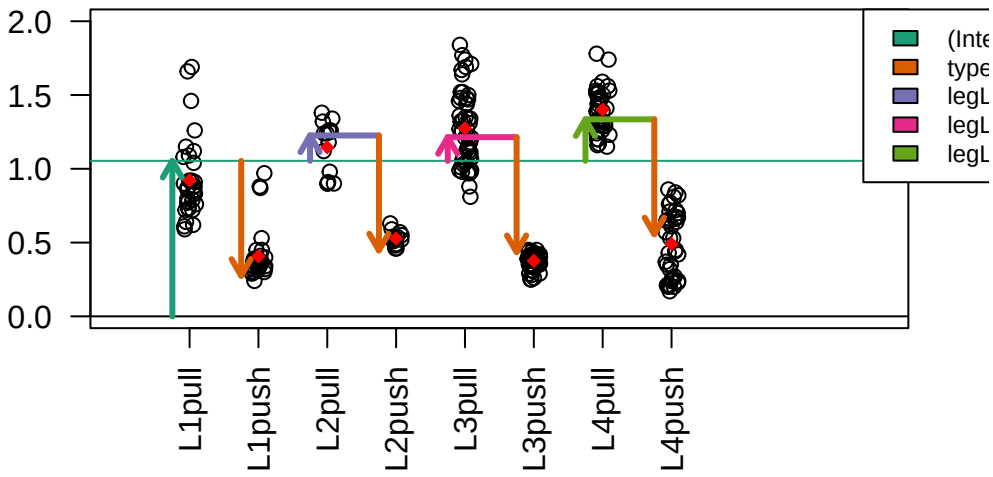


Figure 5: Diagram of the estimated coefficients in the linear model. As before, the teal-green arrow represents the Intercept, which fits the mean of the reference group (here, the pull samples for leg L1). The purple, pink, and yellow-green arrows represent differences between the three other leg groups and L1. The orange arrow represents the difference between the push and pull samples for all groups.

The intercept is the coefficient for L1 Pull. (dark green) The typepush is the difference between pull and push. (orange) The LegL2 is the difference between L1 Pull and L2 Pull The LegL3 is the difference between L1 Pull and L3 Pull The LegL4 is the difference between L1 Pull and L4 Pull

The red diamonds represent the mean of each group.

Notice that the intercept is no longer exactly equal to the mean of the L1 Pull values:

```
s <- split(spider$friction, spider$group)
cat('mean:' , mean(s[["L1pull"]]))
```

```
mean: 0.9214706
```

and same for the other coefficients, the linear model with 8 groups does not manage to accurately fit the LSE in a way that the coefficients match the exact mean for each group.

Here we can demonstrate that the push vs. pull estimated coefficient, `coefs[2]`, is a *weighted average of the difference of the means for each group*. Furthermore, the weighting is determined by the sample size of each group. The math works out simply here because the sample size is equal for the push and pull subgroups within each leg pair. If the sample sizes were not equal for push and pull within each leg pair, the weighting is more complicated.

```
(means <- sapply(s, mean))
```

```
   L1pull    L1push    L2pull    L2push    L3pull    L3push    L4pull    L4push
0.9214706 0.4073529 1.1453333 0.5273333 1.2738462 0.3759615 1.4007500 0.4907500
```

```
##the sample size of push or pull groups for each leg pair
ns <- sapply(s, length)[c(1,3,5,7)]
(w <- ns/sum(ns))
```

```
   L1pull    L2pull    L3pull    L4pull
0.2411348 0.1063830 0.3687943 0.2836879
```

```
sum(w * (means[c(2,4,6,8)] - means[c(1,3,5,7)]))
```

```
[1] -0.7790071
```

```
coefs[2]
```

```
  typepush
-0.7790071
```

Contrasting coefficients

So all these coefficients are comparing each leg to Leg 1. What do we do if we want to compare one group to another group that is not the reference level, for example we want to compare L2 vs L3? We can use the library `contrast`

If we want to compare leg pairs L3 and L2, this is equivalent to contrasting two coefficients from the linear model because, in this contrast, the comparison to the reference level *L1* cancels out:

$$(\text{L3} - \text{L1}) - (\text{L2} - \text{L1}) = \text{L3} - \text{L2}$$

An easy way to make these contrasts of two groups is to use the `contrast` function from the *contrast* package. We just need to specify which groups we want to compare. We have to pick one of *pull* or *push* types, although the answer will not differ, as we will see below.

```
L3vsL2 <- contrast::contrast(fitTL,list(leg="L3",type="pull"),list(leg="L2",type="pull"))
L3vsL2
```

```
lm model parameter contrast

    Contrast       S.E.      Lower      Upper     t  df Pr(>|t|)
 -0.01142949 0.04319685 -0.0964653 0.07360632 -0.26 277   0.7915
```

The first column `Contrast` gives the L3 vs. L2 estimate from the model we fit above.

```
L3vsL2 <- contrast::contrast(fitTL,list(leg="L3",type="push"),list(leg="L2",type="push"))
L3vsL2
```

```
lm model parameter contrast

    Contrast       S.E.      Lower      Upper     t  df Pr(>|t|)
 -0.01142949 0.04319685 -0.0964653 0.07360632 -0.26 277   0.7915
```

we can check the matrix of the contrast like this:

```
L3vsL2$X
```

```
  (Intercept) typepush legL2 legL3 legL4
1           0        0    -1     1     0
attr(,"assign")
[1] 0 1 2 2 2
attr(,"contrasts")
attr(,"contrasts")$type
[1] "contr.treatment"

attr(,"contrasts")$leg
[1] "contr.treatment"
```

and we see that it gives us 0,0,-1,1,0 which means that to find the contrast we are interested in we need to go take the third coefficient from the linear model times -1 and sum the 4th coefficient. Let's check if that applies:

```
coefs<- coef(fitTL)
-coefs[3]+coefs[4]
```

```
      legL2
-0.01142949
```

Linear model with interactions

In the previous linear model, we assumed that the push vs. pull effect was the same for all of the leg pairs (the same orange arrow). You can easily see that this does not capture the trends in the data that well. That is, the tips of the arrows did not line up perfectly with the group averages. For the L1 leg pair, the push vs. pull estimated coefficient was too large, and for the L3 leg pair, the push vs. pull coefficient was somewhat too small.

Interaction terms will help us overcome this problem by introducing additional coefficients to compensate for differences in the push vs. pull effect across the 4 groups. As we already have a push vs. pull term in the model, we only need to add three more terms to have the freedom to find leg-pair-specific push vs. pull differences. As we will see, interaction terms are added to the design matrix by multiplying the columns of the design matrix representing existing terms.

We can rebuild our linear model with an interaction between `type` and `leg`, by including an extra term in the formula `type:leg`. The `:` symbol adds an interaction between the two variables surrounding it. An equivalent way to specify this model is `~ type*leg`, which will expand to the formula `~ type + leg + type:leg`, with main effects for `type`, `leg` and an interaction term `type:leg`.

```
X <- model.matrix(~ type + leg + type:leg, data=spider)
colnames(X)
```

```
[1] "(Intercept)"    "typepush"       "legL2"          "legL3"
[5] "legL4"          "typepush:legL2" "typepush:legL3" "typepush:legL4"
```

```
head(X)
```

```
  (Intercept) typepush legL2 legL3 legL4 typepush:legL2 typepush:legL3
1           1        0     0     0     0              0              0
2           1        0     0     0     0              0              0
3           1        0     0     0     0              0              0
4           1        0     0     0     0              0              0
```

```
5            1        0     0     0     0                0                0
6            1        0     0     0     0                0                0
  typepush:legL4
1              0
2              0
3              0
4              0
5              0
6              0
```

```
imagemat(X, main="Model matrix for linear model with interactions")
```

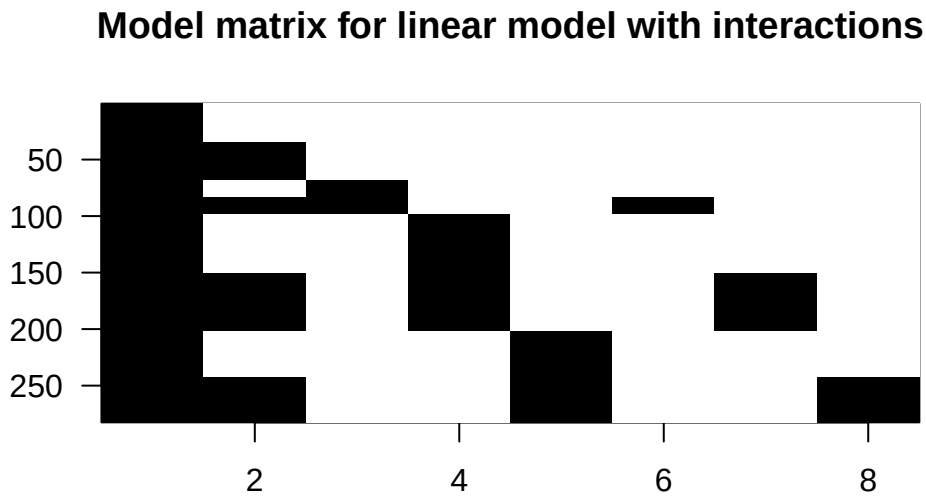


Figure 6: Image of model matrix with interactions.

Columns 6-8 (`typepush:legL2`, `typepush:legL3`, and `typepush:legL4`) are the product of the 2nd column (`typepush`) and columns 3-5 (the three `leg` columns). Looking at the last column, for example, the `typepush:legL4` column is adding an extra coefficient $\beta_{\text{push,L4}}$ to those samples which are both push samples and leg pair L4 samples. This accounts for a possible difference when the mean of samples in the L4-push group are not at the location which would be predicted by adding the estimated intercept, the estimated push coefficient `typepush`, and the estimated L4 coefficient `legL4`.

We can run the linear model using the same code as before:

```r
fitX <- lm(friction ~ type + leg + type:leg, data=spider)
summary(fitX)
```

```
Call:
lm(formula = friction ~ type + leg + type:leg, data = spider)

Residuals:
     Min       1Q   Median       3Q      Max
-0.46385 -0.10735 -0.01111  0.07848  0.76853

Coefficients:
               Estimate Std. Error t value             Pr(>|t|)
(Intercept)     0.92147    0.03266  28.215 < 0.0000000000000002 ***
typepush       -0.51412    0.04619 -11.131 < 0.0000000000000002 ***
legL2           0.22386    0.05903   3.792             0.000184 ***
legL3           0.35238    0.04200   8.390  0.00000000000000262 ***
legL4           0.47928    0.04442  10.789 < 0.0000000000000002 ***
typepush:legL2 -0.10388    0.08348  -1.244             0.214409
typepush:legL3 -0.38377    0.05940  -6.461  0.00000000047335813 ***
typepush:legL4 -0.39588    0.06282  -6.302  0.00000000117063701 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1904 on 274 degrees of freedom
Multiple R-squared:  0.8279,    Adjusted R-squared:  0.8235
F-statistic: 188.3 on 7 and 274 DF,  p-value: < 0.00000000000000022
```

```r
coefs <- coef(fitX)
```

Here is where the plot with arrows really helps us interpret the coefficients. The estimated interaction coefficients (the yellow, brown and silver arrows) allow leg-pair-specific differences in the push vs. pull difference. The orange arrow now represents the estimated push vs. pull difference only for the reference leg pair, which is L1. If an estimated interaction coefficient is large, this means that the push vs. pull difference for that leg pair is very different than the push vs. pull difference in the reference leg pair.

Now, as we have eight terms in the model and eight parameters, you can check that the tips of the arrowheads are exactly equal to the group means (code not shown).

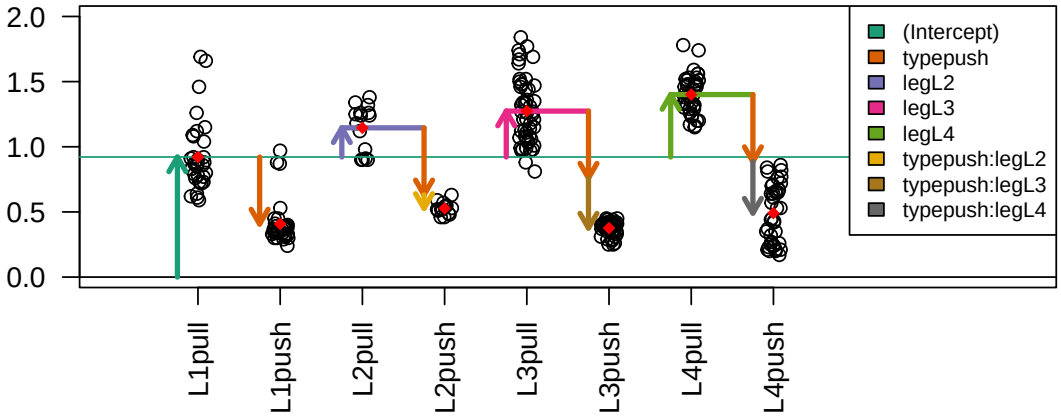


Figure 7: Diagram of the estimated coefficients in the linear model. In the design with interaction terms, the orange arrow now indicates the push vs. pull difference only for the reference group (L1), while three new arrows (yellow, brown and grey) indicate the additional push vs. pull differences in the non-reference groups (L2, L3 and L4) with respect to the reference group.

Now we want to compare push vs pull in L2:

```
L2push.vs.pull <- contrast::contrast(fitX,
  list(leg="L2", type = "push"),
  list(leg="L2", type = "pull")
)
L2push.vs.pull
```

```
lm model parameter contrast

 Contrast      S.E.      Lower      Upper     t  df Pr(>|t|)
   -0.618 0.0695372 -0.7548951 -0.4811049 -8.89 274        0
```

we can look at the contrast vector that will be :

```
(C<- L2push.vs.pull$X)
```

```
  (Intercept) typepush legL2 legL3 legL4 typepush:legL2 typepush:legL3
1           0        1     0     0     0              1              0
  typepush:legL4
1              0
attr(,"assign")
[1] 0 1 2 2 2 3 3 3
attr(,"contrasts")
attr(,"contrasts")$type
[1] "contr.treatment"

attr(,"contrasts")$leg
[1] "contr.treatment"
```

wich is: 0,1,0,0,0,1,0 and means we need to add the 2nd coefficient to the 6th coefficient

```
coefs[2]+coefs[6]
```

```
typepush
  -0.618
```

Now we are interested in comparing if the difference between push and pull from one leg with the difference between push and pull from another leg, let's say L3 and L4. This is a differences of differences and we cannot use the same contrast package. We will use the library `multcomp`. Remember we had 8 coefficients from the linear model:

```
coefs
```

```
   (Intercept)       typepush          legL2          legL3          legL4
     0.9214706     -0.5141176      0.2238627      0.3523756      0.4792794
typepush:legL2 typepush:legL3 typepush:legL4
    -0.1038824     -0.3837670     -0.3958824
```

If we look in the plot we are interested in the difference between the brown line and the yellow line and those are represented by the coefficients 6 and 7. So we have to construct a matrix with 1 on the coefficients we are interested in and 0 in the rest:

```
C<- matrix(c(0,0,0,0,0,-1,1,0),1)
L3vsL2interaction <- multcomp::glht(fitX, linfct=C)
summary(L3vsL2interaction)
```

```
     Simultaneous Tests for General Linear Hypotheses

Fit: lm(formula = friction ~ type + leg + type:leg, data = spider)

Linear Hypotheses:
       Estimate Std. Error t value Pr(>|t|)
1 == 0 -0.27988    0.07893  -3.546  0.00046 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
(Adjusted p values reported -- single-step method)
```

and we see that it is the same as subtracting the coefficients, but the function above gives us also a t-statistic and a p-value.

```
coefs[7]-coefs[6]
```

```
typepush:legL3
    -0.2798846
```

Finally we can ask if the pull vs push difference is different for each pair of legs, and this can be answered using anova.

```
anova(fitX)
```

```
Analysis of Variance Table

Response: friction
           Df Sum Sq Mean Sq  F value                Pr(>F)
type        1 42.783  42.783 1179.713 < 0.00000000000000022 ***
leg         3  2.921   0.974   26.847  0.000000000000002972 ***
type:leg    3  2.098   0.699   19.282  0.000000000022555601 ***
Residuals 274  9.937   0.036
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

the `Sum Sq` column in the anova results is the variance of the aggregated coefficients and it is telling us what variables are responsible for the variance, so for example in our results the `Sum Sq` for the `type` (push vs pull) is `42.783` it's the highest of them all, which means that this is most responsible for the variance in the coefficients. Then we have a `Sum Sq` for the leg (in our graph are the purple, pink and green arrows) with a value of of `2.921` so they also explain

part of the variance. Finally we also have a `2.098` value in `Sum Sq` for the interaction type:leg (push vs pull by Leg pair) (in our graph the yellow, brown and grey arrows) which means that there is also a difference attributed to that. The f-value is like the t-value in a t-test. The p-value works the same as in a t-test, in our case it is smaller than 0.05 so it means that the difference we are seeing in those values are more than what we would expect by chance.

Collinearity

If an experiment is designed incorrectly we may not be able to estimate the parameters of interest. Similarly, when analyzing data we may incorrectly decide to use a model that can't be fit. If we are using linear models then we can detect these problems mathematically by looking for collinearity in the design matrix.

Some system of equations can have more than one solution:

$$\begin{aligned} a + c &= 1 \\ b - c &= 1 \\ a + b &= 2 \end{aligned}$$

there are an infinite number of triplets that satisfy $a = 1 - c, b = 1 + c$.

The system of equations above can be written like this:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}$$

and we can notice that the third column is a linear combination of the first two:

$$\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + -1 \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}$$

The third column does not add a constraint and what we really have are three equations and two unknowns: $a + c$ and $b - c$. Once we have values for those two quantities, there are an infinity number of triplets that can be used.

Collinearity and Least Squares

Consider a design matrix $\mathbf{X}$ with two collinear columns. Here we create an extreme example in which one column is the opposite of another:

$$\mathbf{X} = (\mathbf{1} \quad \mathbf{X}_1 \quad \mathbf{X}_2 \quad \mathbf{X}_3) \text{ with, say, } \mathbf{X}_3 = -\mathbf{X}_2$$

This means that we can rewrite the residuals like this:

$$\mathbf{Y} - \{\mathbf{1}\beta_0 + \mathbf{X}_1\beta_1 + \mathbf{X}_2\beta_2 + \mathbf{X}_3\beta_3\} = \mathbf{Y} - \{\mathbf{1}\beta_0 + \mathbf{X}_1\beta_1 + \mathbf{X}_2\beta_2 - \mathbf{X}_2\beta_3\} = \mathbf{Y} - \{\mathbf{1}\beta_0 + \mathbf{X}_1\beta_1 + \mathbf{X}_2(\beta_2 -$$

so if we have a solution, adding one to both beta2 and beta3 will also be a solution, so there is not a single value that minimizes the error.

and if $\hat{\beta}_1$, $\hat{\beta}_2$, $\hat{\beta}_3$ is a least squares solution, then, for example, $\hat{\beta}_1$, $\hat{\beta}_2 + 1$, $\hat{\beta}_3 + 1$ is also a solution.

Confounding as an example

Now we will demonstrate how collinearity helps us determine problems with our design using one of the most common errors made in current experimental design: confounding. To illustrate, let's use an imagined experiment in which we are interested in the effect of four treatments A, B, C and D. We assign two mice to each treatment. After starting the experiment by giving A and B to female mice, we realize there might be a sex effect. We decide to give C and D to males with hopes of estimating this effect. But can we estimate the sex effect? The described design implies the following design matrix:

$$\begin{pmatrix} Sex & A & B & C & D \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Here we can see that sex and treatment are confounded. Specifically, the sex column can be written as a linear combination of the C and D matrices.

$$\begin{pmatrix} Sex \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} C \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} D \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

This implies that a unique least squares estimate is not achievable.

It can be difficult to perceive that just looking at the matrix. In r we have a function that will help us with this:

Rank

The *rank* of a matrix columns is the number of columns that are independent of all the others. If the rank is smaller than the number of columns, then the LSE are not unique. In R, we can obtain the rank of matrix with the function `qr`, which we will describe in more detail in a following section.

```
Sex <- c(0,0,0,0,1,1,1,1)
A <-   c(1,1,0,0,0,0,0,0)
B <-   c(0,0,1,1,0,0,0,0)
C <-   c(0,0,0,0,1,1,0,0)
D <-   c(0,0,0,0,0,0,1,1)
X <- model.matrix(~Sex+A+B+C+D-1)
cat("ncol=",ncol(X),"rank=", qr(X)$rank,"\n")
```

```
ncol= 5 rank= 4
```

This particular experiment could have been designed better. Using the same number of male and female mice, we can easily design an experiment that allows us to compute the sex effect as well as all the treatment effects. Specifically, when we balance sex and treatments, the confounding is removed as demonstrated by the fact that the rank is now the same as the number of columns:

```
Sex <- c(0,1,0,1,0,1,0,1)
A <-   c(1,1,0,0,0,0,0,0)
B <-   c(0,0,1,1,0,0,0,0)
C <-   c(0,0,0,0,1,1,0,0)
D <-   c(0,0,0,0,0,0,1,1)
X <- model.matrix(~Sex+A+B+C+D-1)
cat("ncol=",ncol(X),"rank=", qr(X)$rank,"\n")
```

```
ncol= 5 rank= 5
```

Here we will not be able to estimate the effect of sex.