R Lesson 3

Federico Reali

2024-10-25

Data visualization

Probability Distributions in R

R allows us to access many probability distributions through specific functions that are contained in the `stats` package, loaded by default.

We can access the distributions through functions that recall their names. For example, with `norm`, `binom`, and `pois`, we access the normal, binomial, and Poisson distributions.

You can access the list of all the distributions in R by searching `distribution` in the help.

To this name is added a *prefix*, which serves to specify the density function (d), the distribution (p), and the function that returns the quantiles (q) or the one that generates random numbers according to the distribution (r).

```r
# Example of accessing normal distribution functions
dnorm(x, mean = 0, sd = 1)  # Density function
pnorm(q, mean = 0, sd = 1)  # Distribution function
qnorm(p, mean = 0, sd = 1)  # Quantile function
rnorm(n, mean = 0, sd = 1)  # Random generation
```

Contingency Tables

The `table()` function builds a contingency table among the combination of factors. `levels()` returns the level attributes of a variable (use the *help* to see what `prop.table()` does).

```
1 library(MASS)
2 df <- MASS::Pima.tr
3 table(df$npreg)
```

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
28 45 30 19 16 11 10 12  9  7  3  1  6  1  2
```

```
1 prop.table(table(df$npreg))
```

```
    0     1     2     3     4     5     6     7     8     9    10    11    12
0.140 0.225 0.150 0.095 0.080 0.055 0.050 0.060 0.045 0.035 0.015 0.005 0.030
   13    14
0.005 0.010
```

Programming in R

In R, we can execute commands for the iteration, the evaluation of conditional expressions and we can define our own functions.

Conditional statements (if)

```
if (condition) command1 else command2
# OR
ifelse(condition, command1, command2)
```

The first expression verifies a *condition* **only** on a single element. In contrast, the `ifelse` command allows you to vectorize the control. If the *condition* is to be evaluated on a vector, the `ifelse` command evaluates it on all entries and applies *command1* if satisfied for that entry and *command2* otherwise. If a vector were passed to the expression `if`, this would evaluate the *condition* only with respect to the first element of the vector and execute the appropriate command.

You can group multiple commands using braces and semicolons. For example, in this way, with the `if` it is possible to perform more operations for each case.

Iterations (for - while - repeat)

We can iterate commands using the `for`, `while`, or `repeat`.

```
1  for (i in sequence) command1
```

`for` allows iterating command1 as a variable varies, in this case *i*. This expression is very useful when visiting vectors and command1 is applied to different entries. Using braces, you can indicate multiple commands.

```
1 while (condition) command1
```

`while` repeats command1 until the condition is true. Using `while` can be risky if the condition is always satisfied and gets stuck in an *infinite loop*.

`repeat` simply repeats a command.

`break` allows you to interrupt any iteration and is the only way to stop a `repeat` loop.

Define functions

R allows the user to define functions, through the `function` command.

```
FunName <- function(arguments) {
  command1
  return(value)
}
```

The previous syntax allows you to define a function called *FunName* that will evaluate command1 and return *value*.

The use of *user-defined* functions allows you to recall the same function more times in different parts of the code and to pass different arguments. Also in this case, using braces, you can specify multiple commands.

```
my_fun <- function(a, b, c) {
  return(a * b + c)
}
```

```
my_fun2 <- function(a, b, c) {
  y <- a^b
  return(y + a * b + c)
}
```

You can save functions and scripts (using the .R extension), which can then be executed using `source("function_name.R")`.

To recall these functions it is essential that they are in the working directory, otherwise the path to reach the file must be indicated.

Data Visualization


50 0 50 100 150 200 Yards
× Pump
• Deaths from cholera
OXFORD STREET
GT. MARLBOROUGH ST.
REGENT STREET
BROAD STREET
WARWICK STREET
DEAN STREET
KING STREET
CONDUIT STREET
GOLDEN SQUARE
SAVILLE ROW
BREWER STREET
NEW BOND STREET
REGENTS QUADRANT
PICCADILLY

Data visualization

Data visualization allows identifying trends, connections, and retrieving information from data that are not obvious in a tabular form. All the commands that follow will help in these investigations and all of them can be customized using different colors, labels of the axes, or placing side by side images.

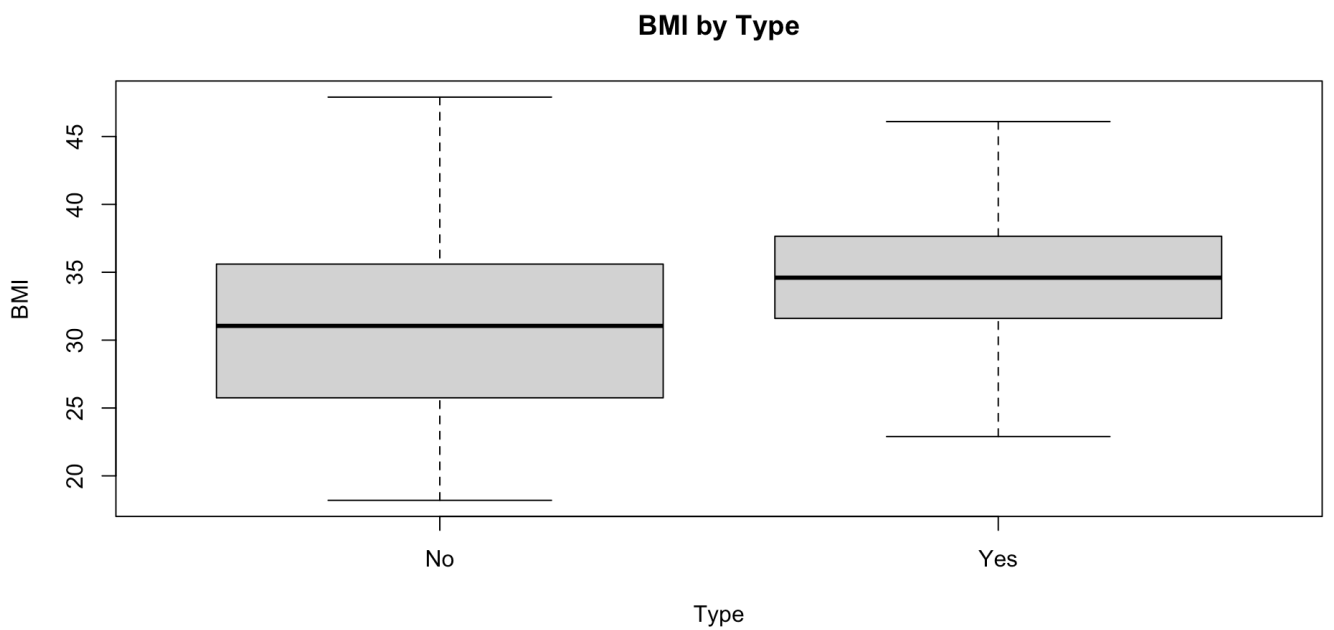
Box Plot

Using R, it is very easy to produce box plots using the `boxplot()` command. We will specify the variable we want to visualize.

```r
# Example using the MASS dataset Pima.tr
boxplot(df$bmi, main = "BMI boxplot", xlab = "BMI", horizontal = TRUE)
```

In case we are interested in exploring the distribution of a variable according to another factor variable, we can use the notation `boxplot(ColumnName ~ FactorColumnName, data = dataset)`.

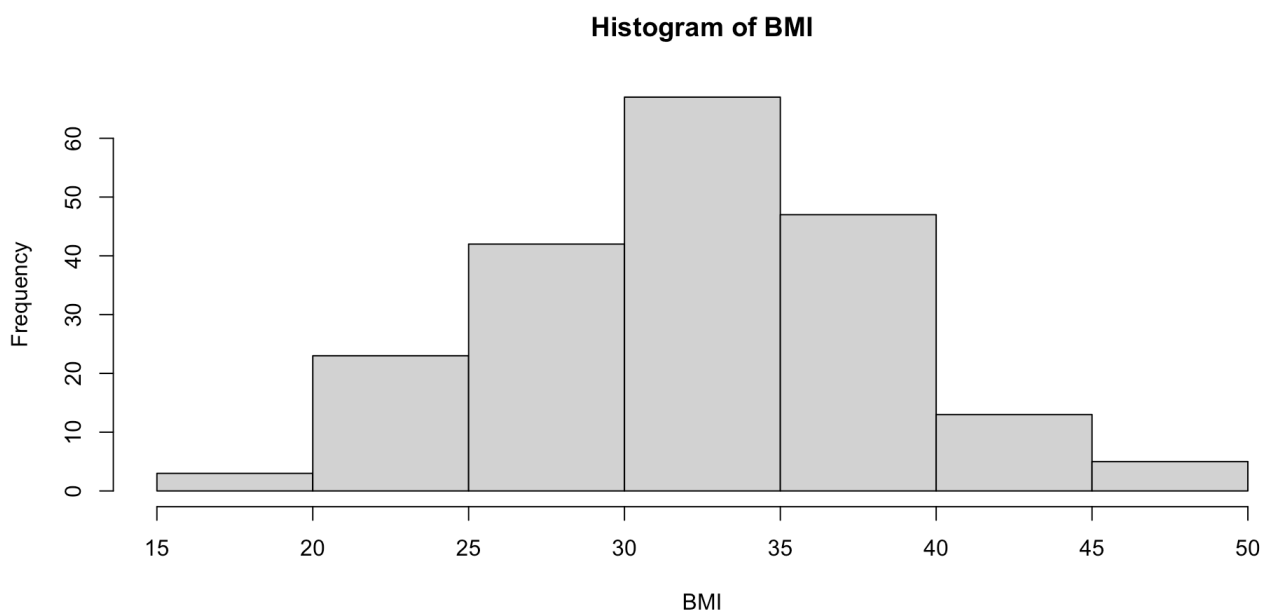
```
1 # Example using the MASS dataset Pima.tr
2 boxplot(bmi ~ type, data = df, main = "BMI by Type", xlab = "Type", ylab =
```



Histograms

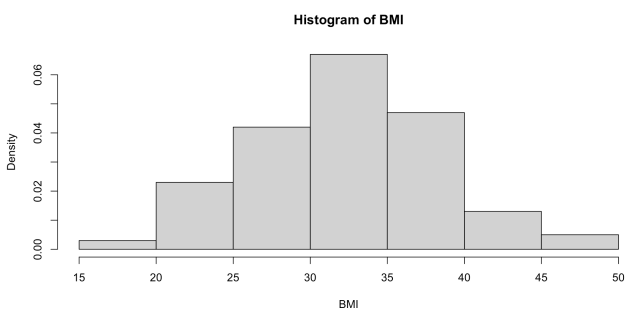
Using the `hist()` command it is possible to get histograms for the variables. This is a simple way to understand the distribution of the data.

```
1 # Example using the MASS dataset Pima.tr
2 hist(df$bmi, main = "Histogram of BMI", xlab = "BMI", freq = TRUE)
```



It is important to notice that we can specify the optional argument *freq*. If *freq* is TRUE, the histogram shows the counts of the component; if FALSE, it shows the relative frequencies, *i.e.*, the probability densities. The latter is useful if we are comparing variables with different numbers of observations. The default value for *freq* is TRUE.

```
1  # Example using the MASS dataset Pima.tr
2  hist(df$bmi, main = "Histogram of BMI", xlab = "BMI", freq = FALSE)
```

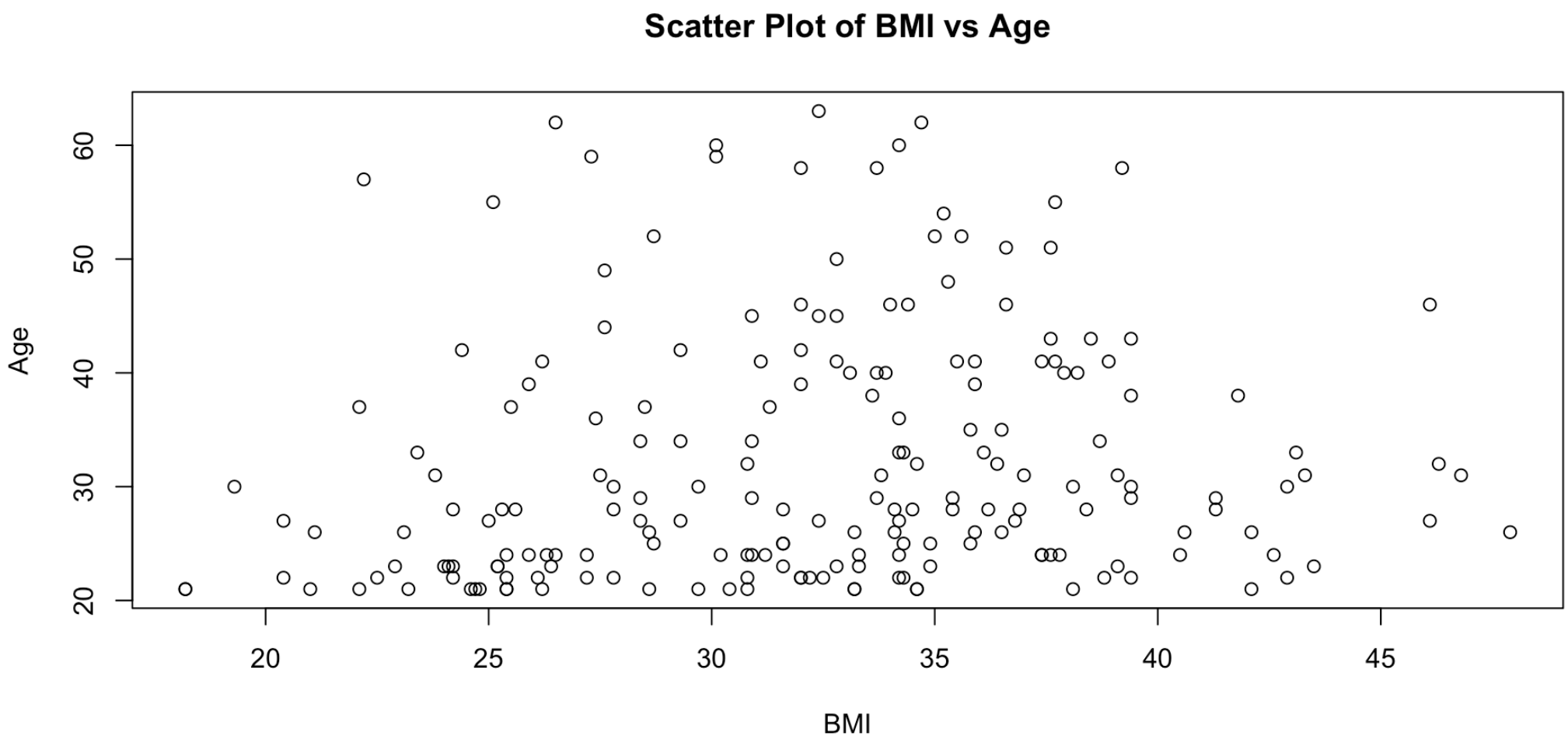


Note that `hist()` does not support the `~` notation.

Scatter Plots

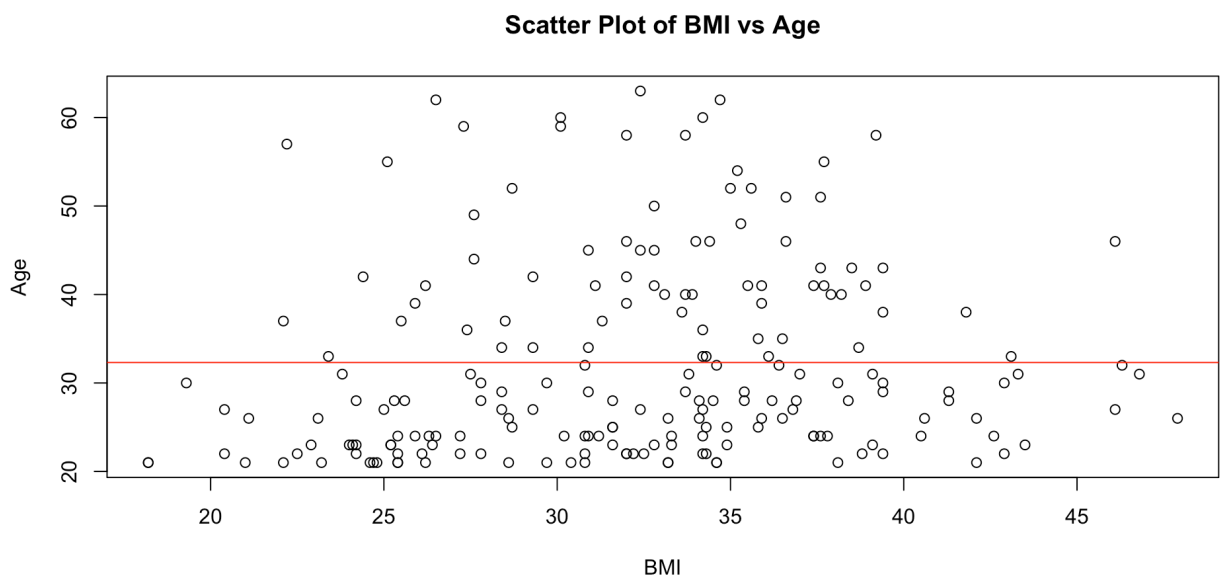
With the commands `plot(x, y)` or `scatterplot(x, y)` it is possible to represent the data as points. We have already used plot and we noticed that it can be very useful for exploring the relationships between variables.

```r
# Example using the MASS dataset Pima.tr
plot(df$bmi, df$age, main = "Scatter Plot of BMI vs Age", xlab = "BMI", yla
```



There are other commands that can be combined with all the previous, for example `abline()`, `text()`, `points()`, and `lines()` that are very useful and we will use them in the examples.

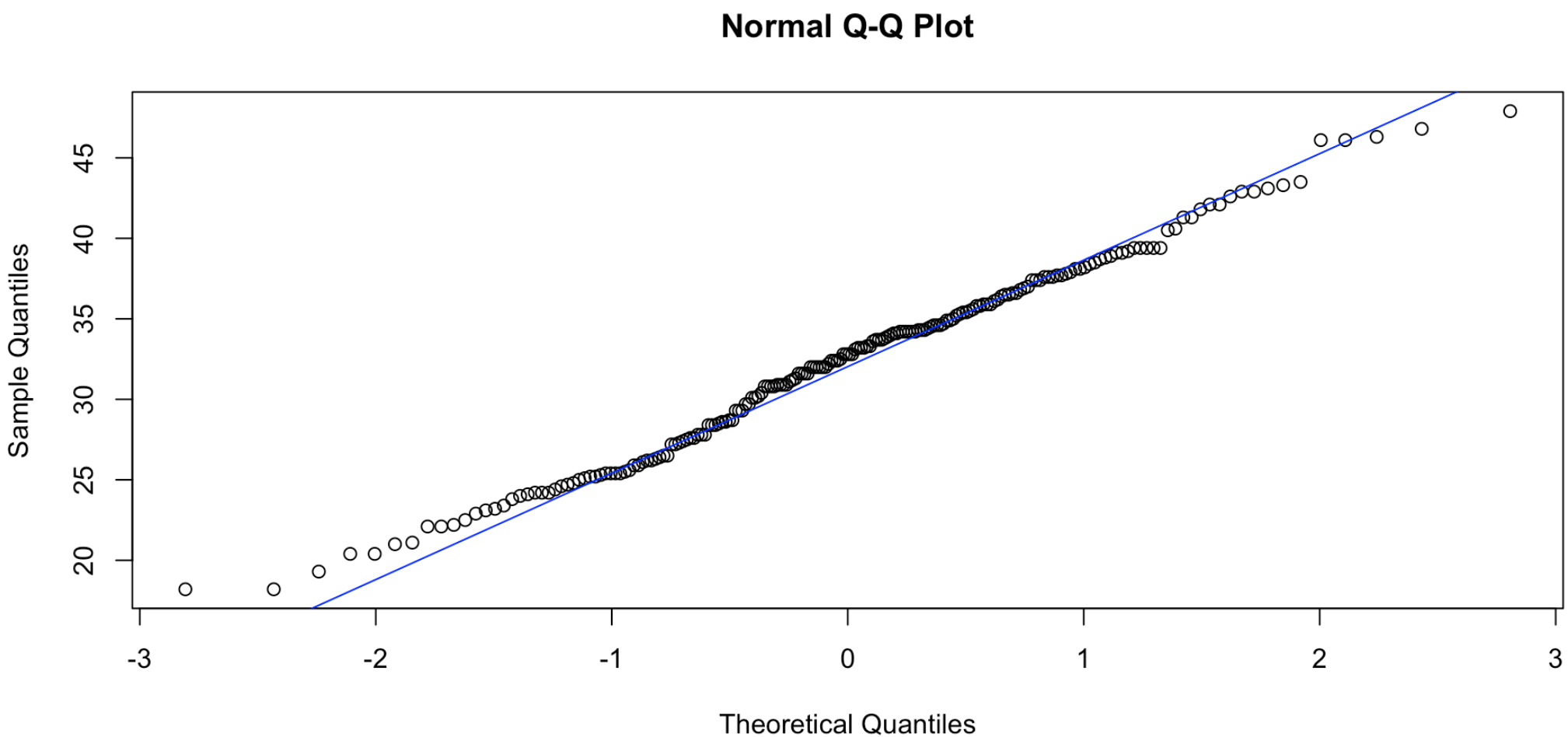
```
1 # Example using the MASS dataset Pima.tr
2 plot(df$bmi, df$age, main = "Scatter Plot of BMI vs Age", xlab = "BMI", yla
3 abline(h = mean(df$bmi), col = "red")
```



It should be pointed out, at least for the normal distribution, there are the `qqnorm` and `qqline` functions, which graphically compare the quantiles of the data with those of a normal distribution.

This is a graphical way (and therefore not formal) to verify if your data follows a normal distribution. Together with the requirement that the measurements be independent, these two are hypotheses that we have seen very often, especially in the use of limit theorems.

```r
# Example using the MASS dataset Pima.tr
qqnorm(df$bmi)
qqline(df$bmi, col = "blue")
```



Violin Plots

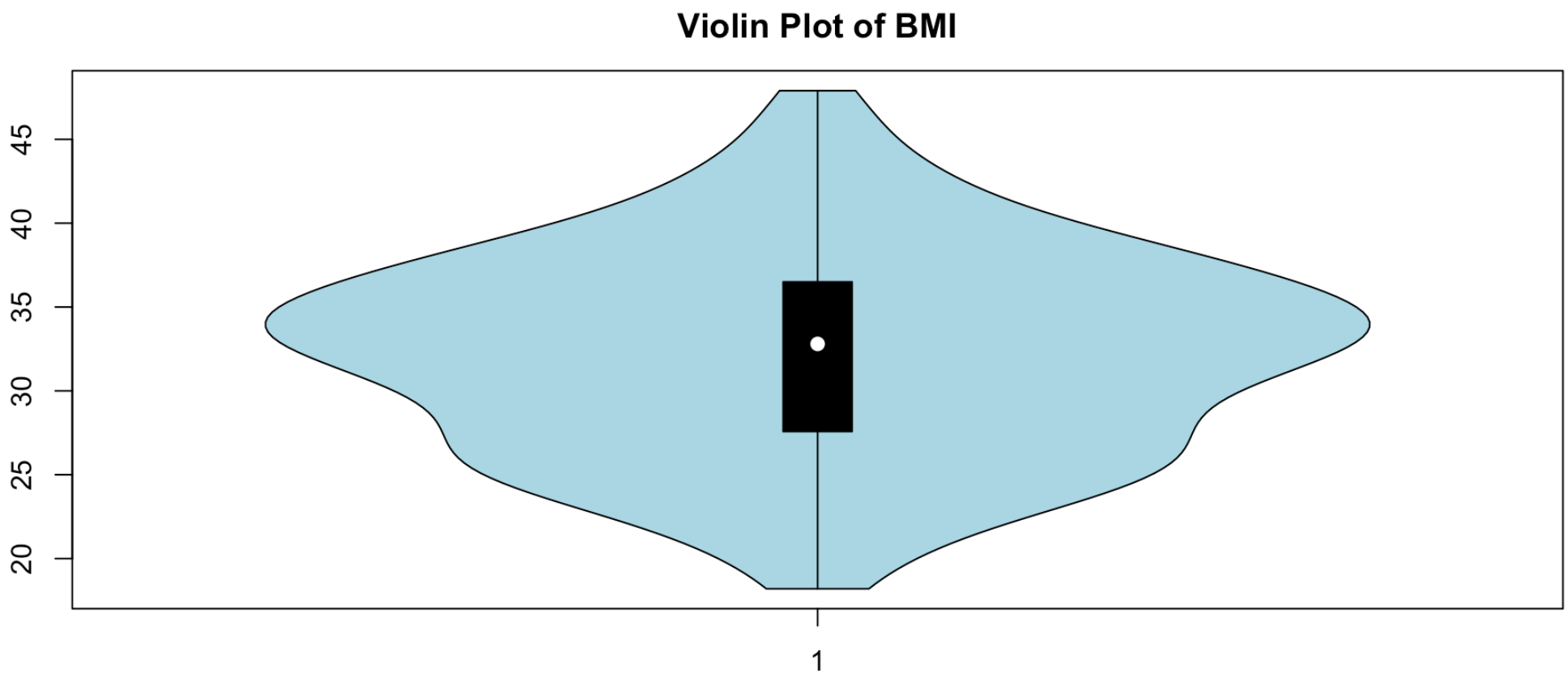
Violin plots are a method of plotting numeric data and can be understood as a combination of a box plot and a kernel density plot. They show the distribution of the data across different categories.

In R, violin plots can be created using the `vioplot` package. First, you need to install and load the package.

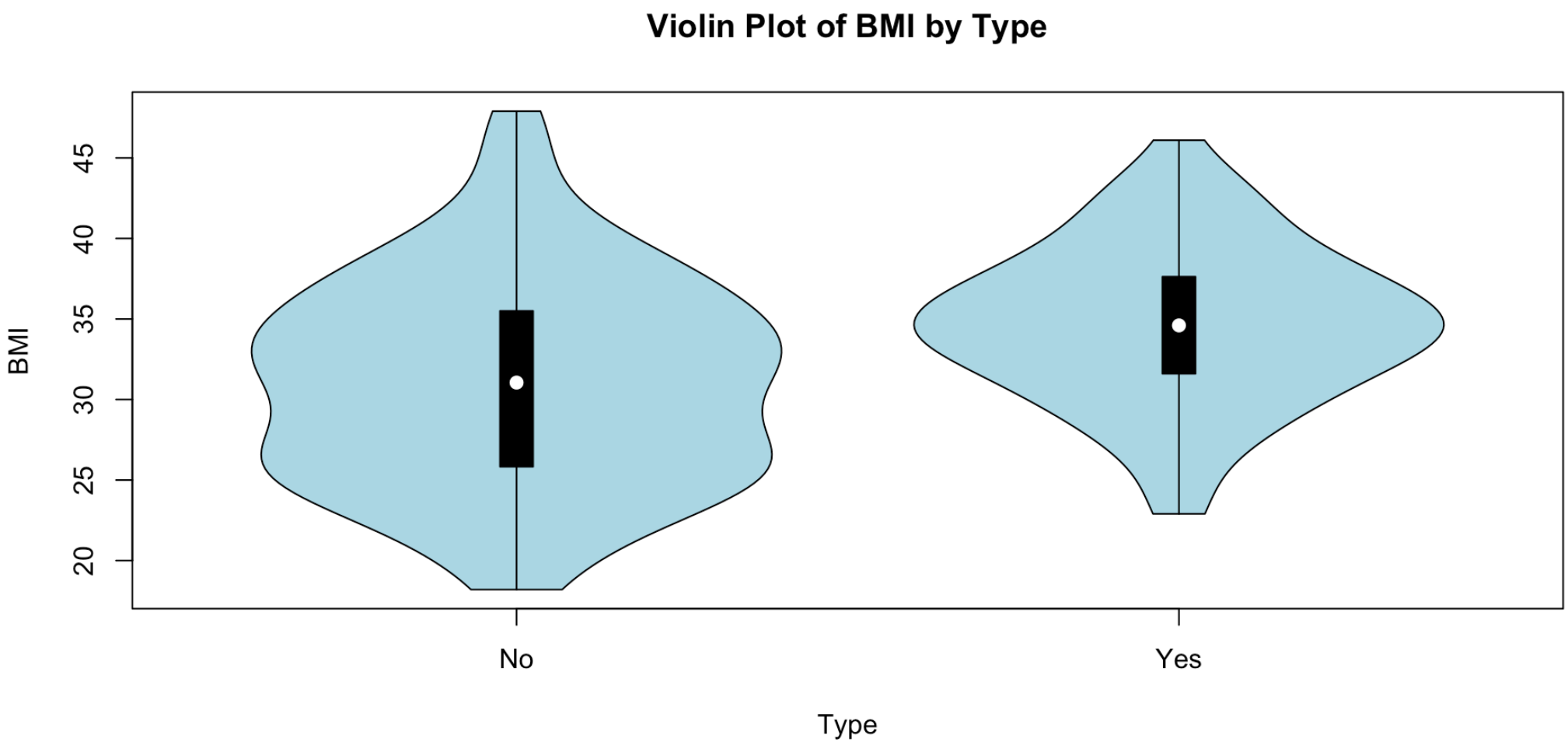
```
# Install and load the vioplot package
# install.packages("vioplot")
library(vioplot)
```

Violin Plot of BMI

```
1 vioplot(df$bmi, main = "Violin Plot of BMI", col = "lightblue")
```



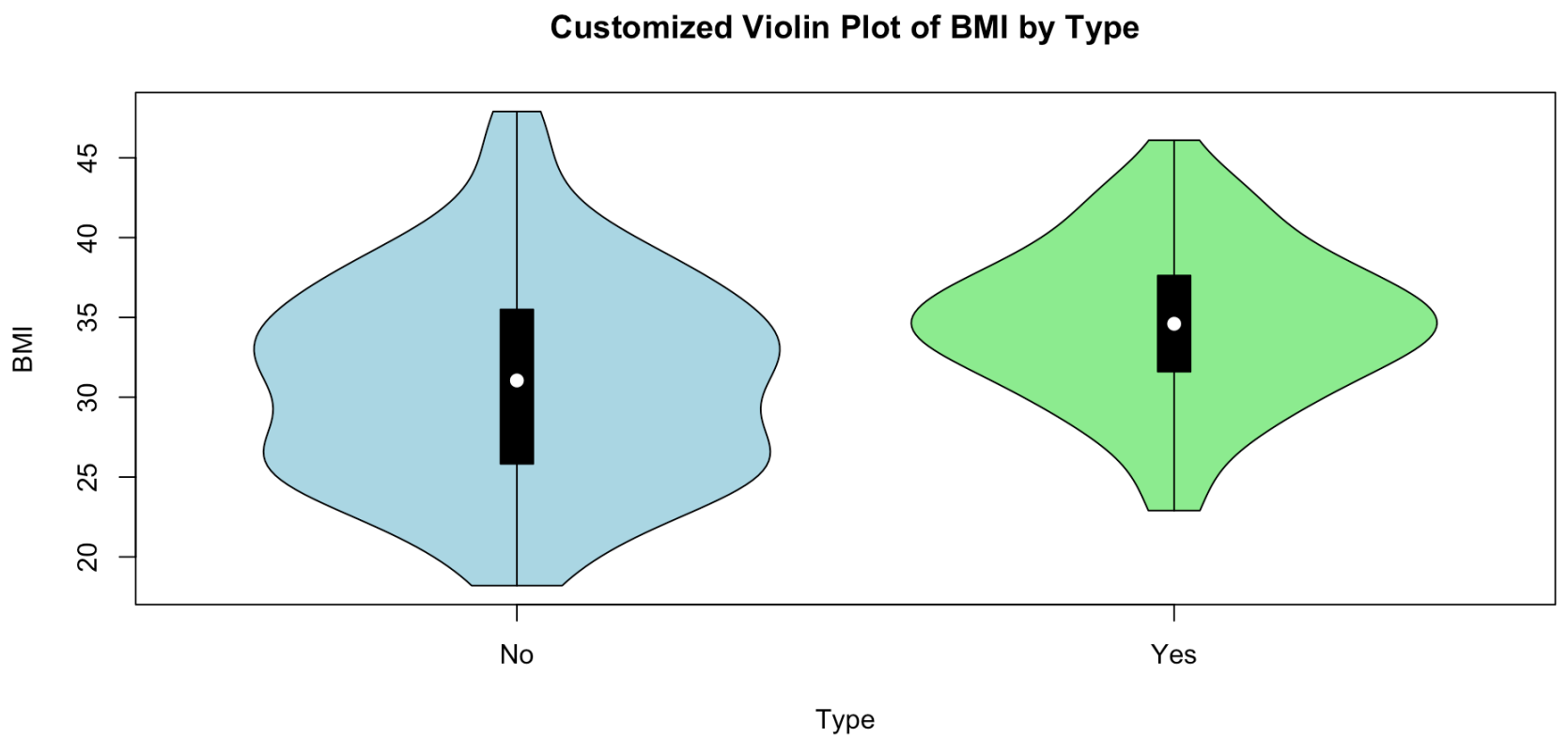
```r
vioplot(bmi ~ type, data = df, main = "Violin Plot of BMI by Type", xlab =
```



Customizing Violin Plots

Violin plots can be customized with different colors, labels, and additional features to enhance the visualization.

```
1  vioplot(bmi ~ type, data= df, main = "Customized Violin Plot of BMI by Type
```

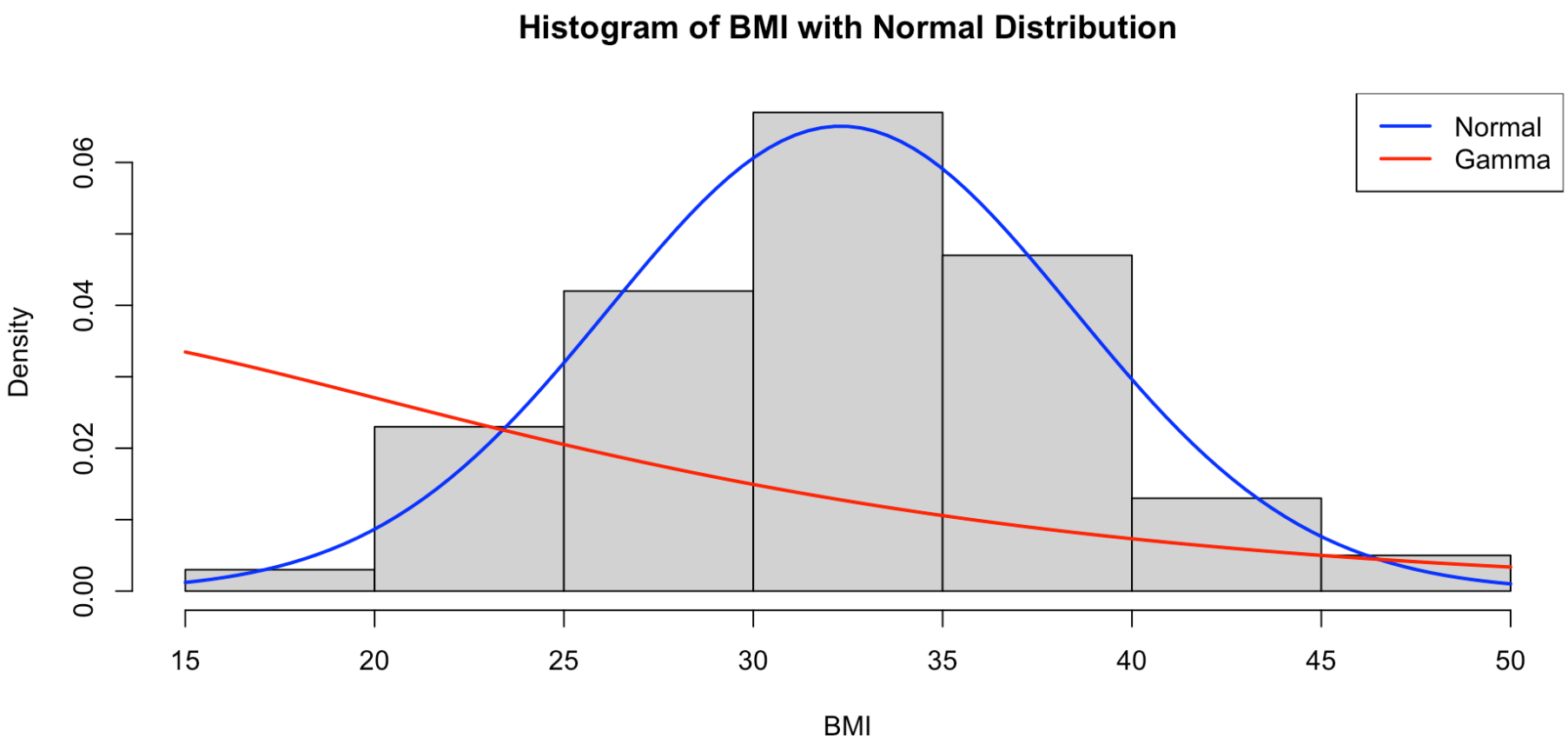


Comparing data and distributions

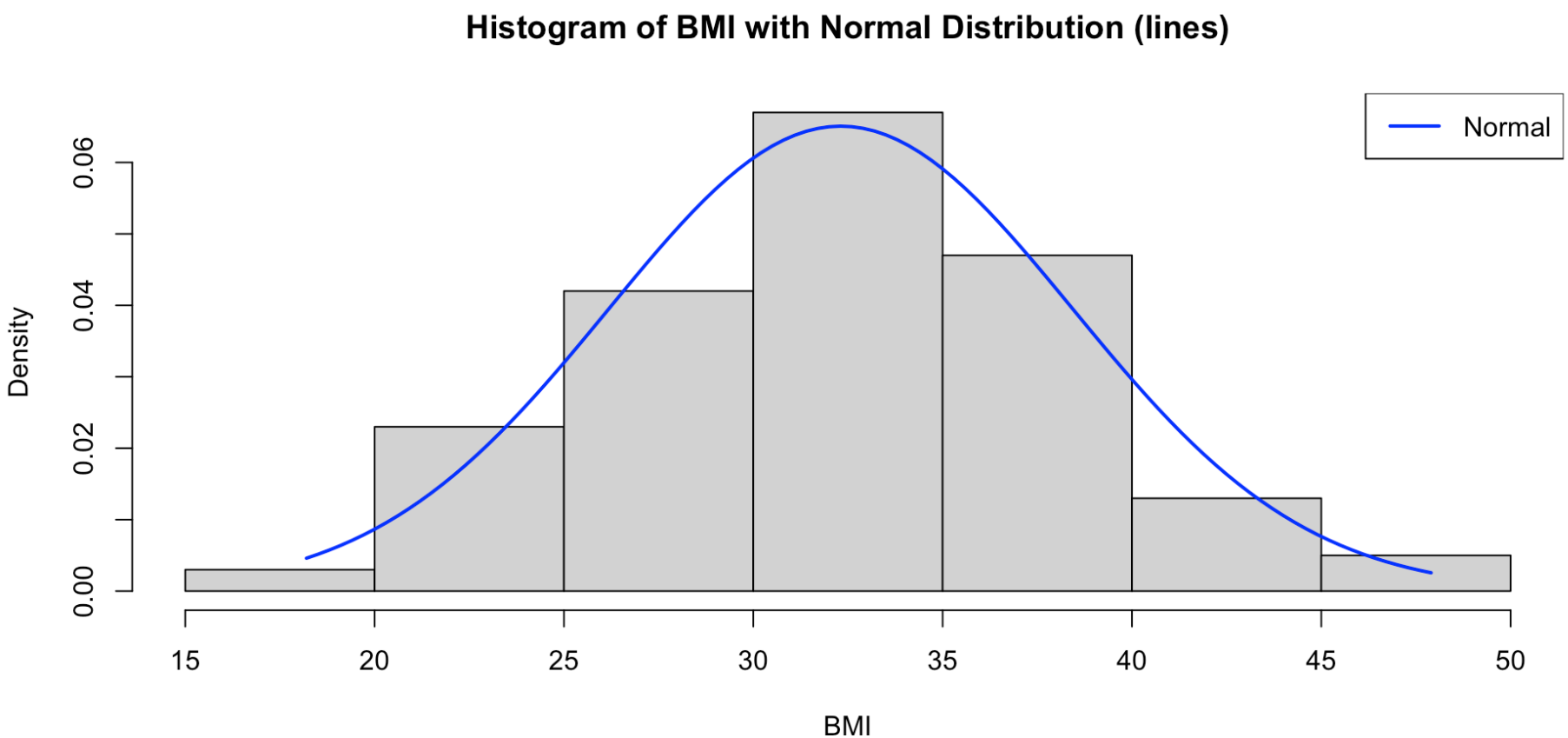
We can overlay histograms with known distributions to compare the data distribution with theoretical distributions.

We can first use violin plots to have hints on the shape and select an appropriate distribution.

```r
# Example using the MASS dataset Pima.tr
hist(df$bmi, probability = TRUE, main = "Histogram of BMI with Normal Distr
curve(dnorm(x, mean = mean(df$bmi), sd = sd(df$bmi)), add = TRUE, col = "bl
# Overlay with a different distribution, e.g., Gamma
curve(dgamma(x, shape = 2, rate = 0.1), add = TRUE, col = "red", lwd = 2)
legend("topright", legend = c("Normal", "Gamma"), col = c("blue", "red"), l
```



```r
# Histogram with Normal Distribution Overlay using lines
hist(df$bmi, probability = TRUE, main = "Histogram of BMI with Normal Distr
x_vals <- seq(min(df$bmi), max(df$bmi), length.out = 100)
y_vals <- dnorm(x_vals, mean = mean(df$bmi), sd = sd(df$bmi))
lines(x_vals, y_vals, col = "blue", lwd = 2)
legend("topright", legend = "Normal", col = "blue", lwd = 2)
```



Repeat all the visualization of another numerical variable of the Pima dataset