Introduction to Machine Learning

Working Group "Computational Statistics" – Bernd Bischl et al.

```
##   |                                                                      |
```

Logistic regression

Idea

Logistic regression is a discriminant approach, i.e., we are modeling the posterior probabilities $\pi(x)$ of the labels directly using the logistic function, s.t.

$$\pi(x) = \mathbb{P}(y = 1|x) = \frac{1}{1 + \exp(-\theta^T x)}.$$

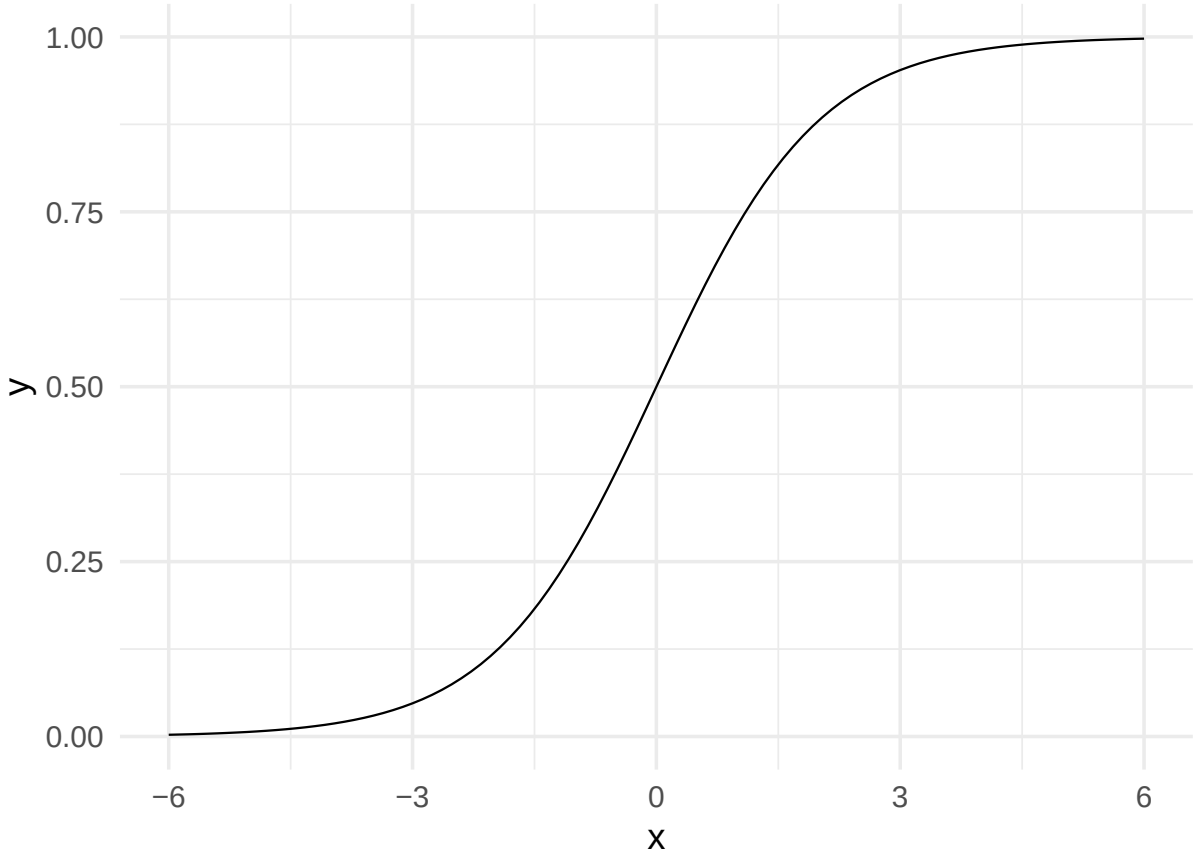
Note As in the lecture we suppress the intercept in notation, i.e., $\theta^T x \equiv \theta_0 + \theta^T x$.

This can be easily implemented as:

```
logistic <- function(X, theta) {
  1 / (1 + exp(-X %*% theta))
}
```

And we can observe that logistic regression "squashes" the estimated linear scores $\theta^T x$ to $[0, 1]$:

```
library(ggplot2)
num_data <- 100
theta <- c(0, 1)

ggplot(data = data.frame(x = seq(-6, 6, length.out = num_data)), aes(x)) +
  stat_function(fun = function(x) logistic(cbind(rep(1, length(x)), x), theta))
```



From the definition of accuracy and the encoding of the labels to $\{-1, 1\}$, one can derive the **Bernoulli loss** (see lecture slides):

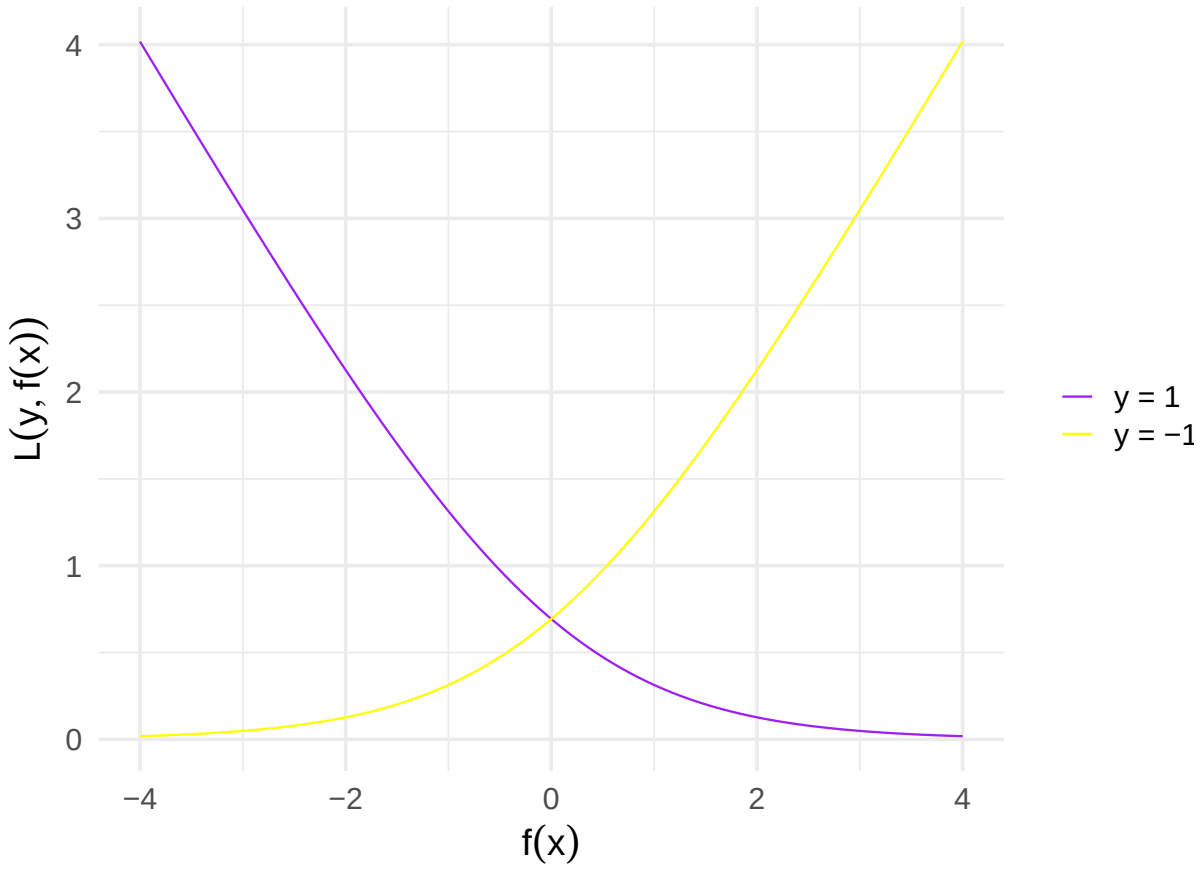
$$L(y, f(x)) = \log(1 + \exp(-yf(x)))$$

For logistic regression, we have $f(x) = \theta^T x$.

```
loss_bernoulli <- function(Y, X, theta) {
  log(1 + 1 / exp(Y * (X %*% theta)))
}
```

We can plot the loss for $y = 1$ and $y = -1$:

```
library(reshape2)
num_data <- 100
theta <- c(0, 1)
y <- c(1, -1)

x <- seq(-4, 4, length.out = num_data)
L <- sapply(y, function(y) loss_bernoulli(y, cbind(rep(1, length(x)), x), theta))
plot_data <- melt(data.frame(x = x, L1 = L[, 1], L2 = L[, 2]), id.vars = "x")

ggplot(plot_data, aes(x = x, y = value, colour = variable)) +
  geom_line() +
  labs(x = expression(f(x)), y = expression(L(y, f(x)))) +
  scale_color_manual(name = "", values = c("purple", "yellow"),
                     labels = c("y = 1", "y = -1"))
```



This means that, in order to minimize the loss, we should predict $y = 1$ if

$$\theta^T x \geq 0 \iff \pi(x) = \mathbb{P}(y = 1|x) = \frac{1}{1 + \exp(-\theta^T x)} \geq 0.5.$$

```
classify_logreg <- function(X, theta, conv = as.numeric) {
  probability <- logistic(X, theta)
  list(
    prediction = conv(ifelse(probability > .5, 1, -1)),
    levels = c("1", "-1")
  )
}
```

With the help of the loss function, we can get the empirical risk as usual:

$$R_{emp} = \sum_{i=1}^{n} L\left(y^{(i)}, f\left(x^{(i)}\right)\right)$$

```
risk_bernoulli <- function(Y, X, theta) {
  sum(loss_bernoulli(Y, X, theta))
}
```

Logistic regression - example

To investigate how the parameter θ of the logistic regression can be estimated, we create some artificial data:

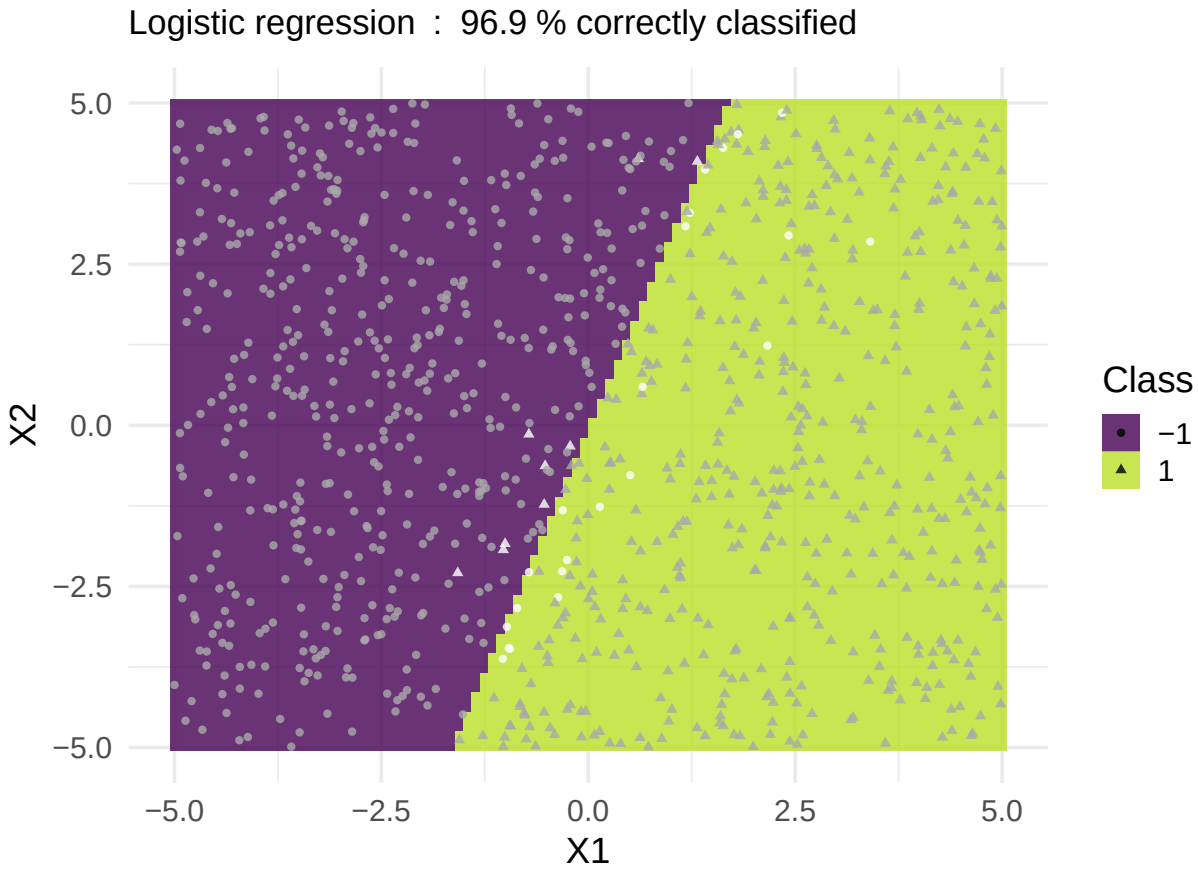
```r
set.seed(1337)

n <- 1000
p <- 2

# model without intercept: x ~ U[-5, 5]
X <- matrix(runif(n * p, -5, 5), nrow = n)
theta <- c(3, -1)

# generate y ~ B(1, p) and recode from {0, 1} to {-1, 1}
Y <- 2 * rbinom(n = n, size = 1, prob = logistic(X, theta)) - 1
```

We use the *plot_2D_classify* function from the KNN code demo as we want to visualize how accurately we can classify the data:

```r
# function for logistic regression with two features to visualize class
# boundaries X1 and X2 are the names of the two features to use.

plot_2D_logreg <- function(theta, X1 = "X1", X2 = "X2",
                           # by default, we "predict" the class of the training data:
                           to_classify_labels = as.character(Y),
                           to_classify_data = data.frame(
                             "X1" = X[, 1],
                             "X2" = X[, 2]
                           ),
                           lengthX1 = 100, lengthX2 = 100,
                           title = '"Logistic regression" ~',
                           plot_class_rate = TRUE) {
  plot_2D_classify(
    to_classify_labels = to_classify_labels,
    to_classify_data = to_classify_data,
    classify_method = function(to_classify_data)
      classify_logreg(
        X = cbind(as.matrix(to_classify_data)),
        theta = theta, conv = as.character
      ),
    X1, X2,
    lengthX1 = lengthX1, lengthX2 = lengthX2,
    title = title,
    plot_class_rate = plot_class_rate
  )
}
```

Let's classify the data with our *real* parameter θ:

```r
plot_2D_logreg(theta)
```




```
risk_bernoulli(Y, X, theta)
```

```
## [1] 99.9
```

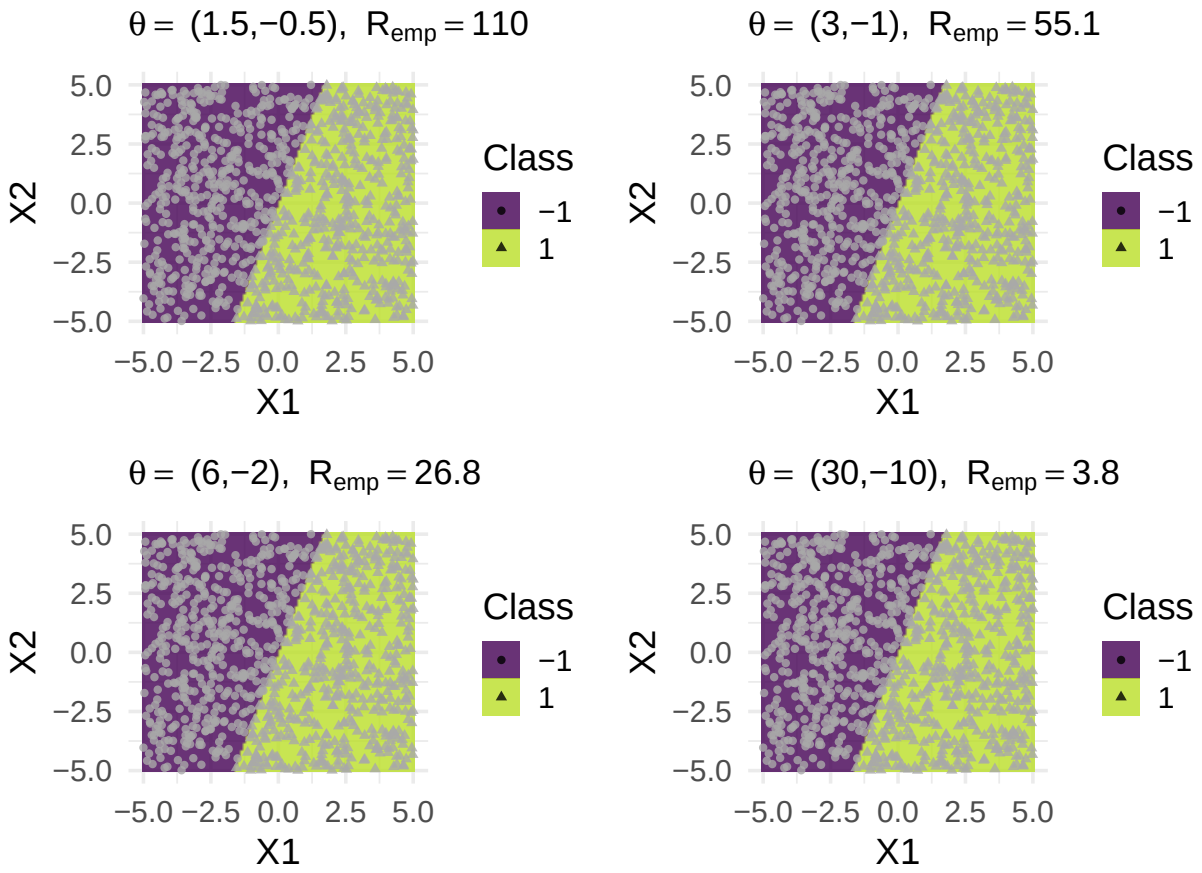
We notice that our data is linearly separated as one would expect, since our decision boundary is defined by

$$\theta^T x = 0,$$

which defines a hyperplane.
Furthermore, we note that θ represents a vector normal to this plane and is pointing to the "1"-side. But doesn't that mean that for any positive $\lambda \in \mathbb{R}^+$, a rescaled coefficient vector $\lambda \cdot \theta$, which defines the same "direction" of decision boundary as θ, would separate the data equally well. . . ?
Check the classification and the empirical risk for $\lambda \in \{0.5, 2, 3\}$. Can you figure out what determines the *optimal* length of θ?
To gain further understanding of this behavior, let's look at a data scenario where we encounter so-called **complete separation**, i.e., a situation in which the data can be
classified without error by our classification method. We can simply use the predictions from our model as if they were the observed labels to create such a dataset:

```
library(gridExtra)
Y_compl_sep <- classify_logreg(X, theta)$prediction

thetas <- list(0.5 * theta, 1 * theta, 2 * theta, 10 * theta)
ggplot_list <- lapply(
  thetas,
```

```
  function(theta) {
    plot_2D_logreg(theta,
      to_classify_labels = as.character(Y_compl_sep),
      title = paste0(
        "theta ==",
        paste0('~ "(', paste0(theta, collapse = ","), '), " ~ '),
        "R[emp] == ",
        risk_bernoulli(Y_compl_sep, X, theta = theta)
      ),
      plot_class_rate = FALSE
    )
  }
)
do.call(grid.arrange, ggplot_list)
```



Remember the Bernoulli loss was derived, s.t.

$$L(y, f(x)) = \log\left(1 + \exp\left(-y\theta^T x\right)\right).$$

In the case of complete separation, the loss for every observation is

$$L(y, f(x)) = \log\left(1 + \exp(-|\theta^T x|\right),$$

since every observation is classified correctly, i.e. $\theta^T x < 0$ for $y = -1$ and $\theta^T x > 0$ for $y = 1$.
This means the empirical risk is monotonously decreasing as the length of theta increases, which means that we never converge on a finite solution by minimizing the empirical risk. In this situation, additional

artificial constraints – such as regularization – can be introduced to find a solution. In paractice, complete separation can happen fairly easily in datasets with a small number of observations compared to the number of available features.

Since we don't need to deal with complete separation in our example, we can simply apply the `gradient_descent_opt_stepsize` function of the code demo to the linear model directly to the empirical risk we derived from the Bernoulli loss. This type of gradient descent mimization can be applied to any risk for which we can figure out the gradient, in this case this is[1]

$$\frac{\partial R_{emp}(f)}{\partial \theta} = \sum_{i=1}^{n} \frac{-y^{(i)}}{1 + \exp\left(y^{(i)} \theta^T x^{(i)}\right)} x^{(i)}.$$

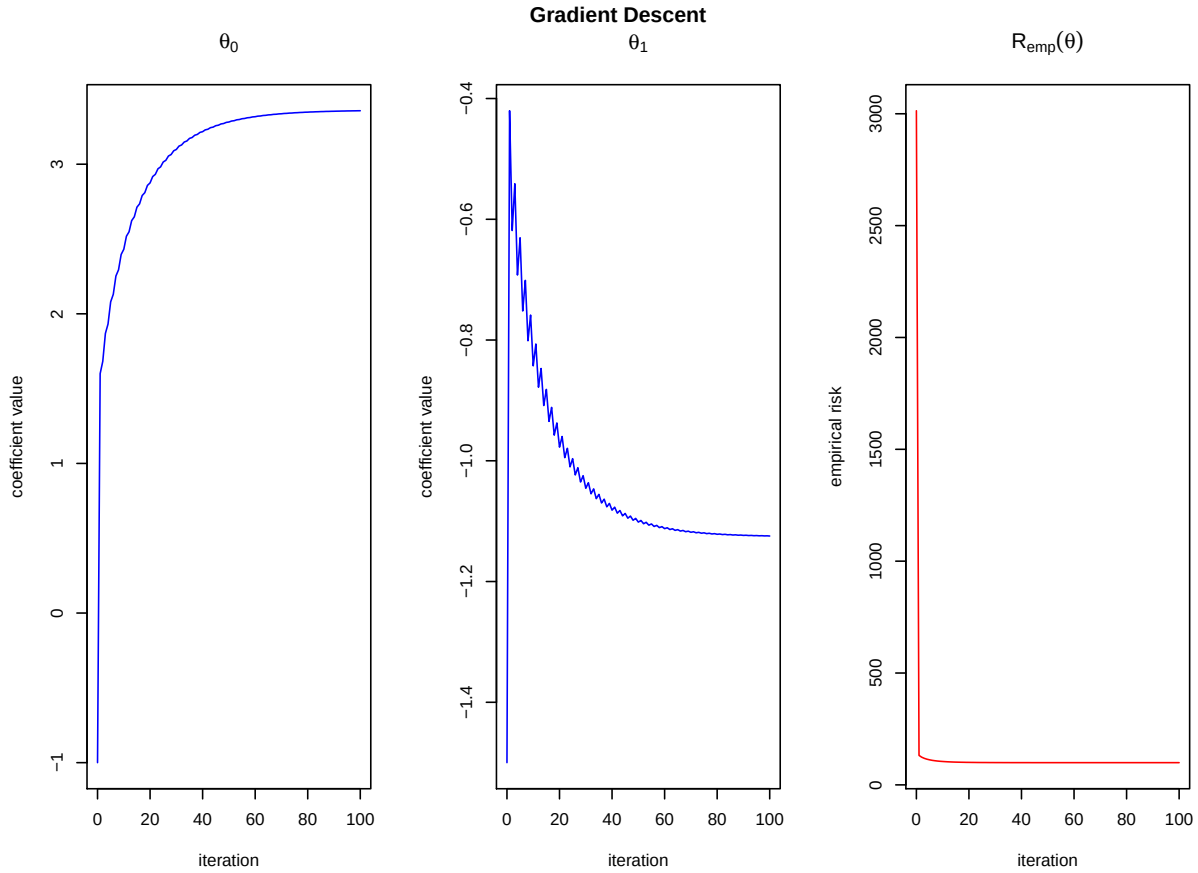
Which can be implemented as:

```
gradient_bernoulli <- function(Y, X, theta) {
  colSums((1 / (1 + exp(Y * (X %*% theta))) * -Y)[, 1] * X)
}
```

Now we can apply the gradient descent method to find a numerical solution:

```
res <- gradient_descent_opt_stepsize(Y, X, c(-1, -1.5),
  risk_bernoulli,
  gradient_bernoulli,
  max_learning_rate = 0.1,
  max_iterations = 100,
  epsilon = 0.00001,
  include_storage = TRUE
)
title(main = "Gradient Descent", outer = TRUE, line = -1)
```

[1] take a couple of minutes to verify this yourself with pen & paper. Remember that the derivative of $\exp(f(x))$ for x is $\exp(f(x))f'(x)$ and that the derivative of $\log(x)$ is $\frac{1}{x}$.




```
print(paste("Risk: ", res$risk))
```

```
## [1] "Risk:  99.2996208092343"
```

```
res$theta
```

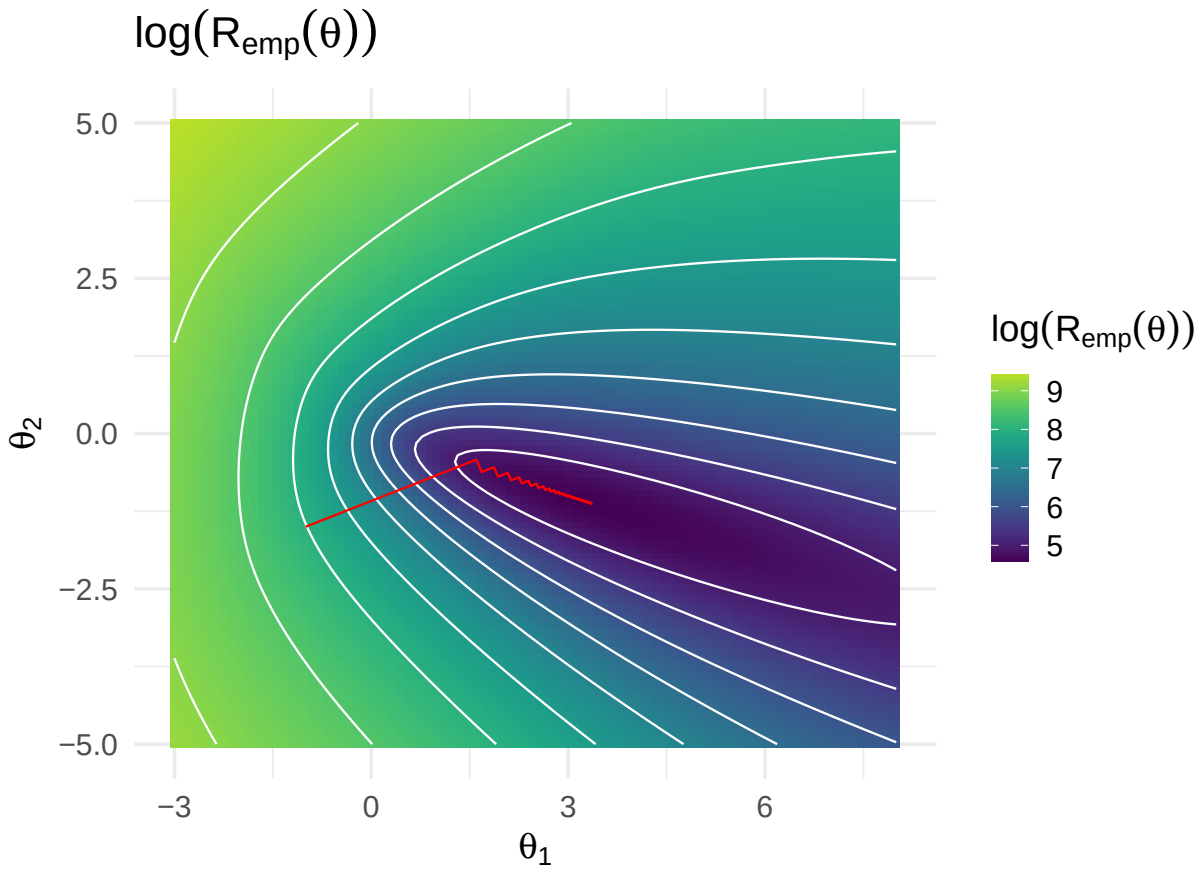
```
## [1]  3.36 -1.12
```

To see what happens when we use gradient descent, we write a function which plots how we *traverse* the parameter space as the gradient descent algorithm proceeds:

```
plot2D_emp_risk <- function(min_theta = c(0, 0),
                            max_theta = c(1, 1),
                            theta_length = c(100, 100),
                            emp_risk,
                            thetas,
                            log_emp_risk = FALSE) {
  grid_thetas <- sapply(1:2, function(i)
    seq(min_theta[i],
      max_theta[i],
      length.out = theta_length[i]
    ))

  # compute grid coordinates with cartesian product
```

```
  grid_theta <- expand.grid(grid_thetas[, 1], grid_thetas[, 2])

  emp_risk <- if (log_emp_risk) {
    apply(grid_theta, 1, function(theta)
      log(emp_risk(theta)))
  } else {
    apply(grid_theta, 1, function(theta)
      emp_risk(theta))
  }

  grid_theta$emp_risk <- emp_risk



  ggplot() +
    geom_raster(
      data = grid_theta,
      aes(x = Var1, y = Var2, fill = emp_risk)
    ) +
    geom_contour(
      data = grid_theta, colour = "white",
      aes(x = Var1, y = Var2, z = emp_risk)
    ) +
    geom_line(
      data = data.frame(thetas), aes(x = X1, y = X2),
      colour = "red"
    ) +
    labs(
      fill = if (log_emp_risk) {
        expression(log(R[emp](theta)))
      } else {
        expression(R[emp](theta))
      },
      title = if (log_emp_risk) {
        expression(log(R[emp](theta)))
      } else {
        expression(R[emp](theta))
      },
      x = expression(theta[1]),
      y = expression(theta[2])
    )
}
```

```
plot2D_emp_risk(
  emp_risk = function(theta) (risk_bernoulli(Y, X, theta)),
  max_theta = c(8, 5), min_theta = c(-3, -5),
  log_emp_risk = TRUE,
  thetas = res$storage$theta
)
```



With gradient descent we are able to find the minimum, but also we note that the algorithm needs a lot of zig-zagging steps in the "valley" of the empirical risk function. It can be shown that, for gradient descent with exact line search, two consecutive search directions are orthogonal to each other, which is what we see here. In the next chapter, we look at another gradient-based approach that can avoid this zig-zagging behavior.

Newton-Raphson

The Newton-Raphson method follows this algorithm:

0. Initialize $\theta^{[0]}$ (randomly) and calculate the gradient and the Hessian matrix, which is a square matrix of second-order partial derivatives, of the empirical risk with respect to θ, for example for the Bernoulli loss:

$$\frac{\partial R_{emp}(f)}{\partial \theta} = \sum_{i=1}^{n} \frac{-y^{(i)}}{1 + \exp\left(y^{(i)} \theta^T x^{(i)}\right)} x^{(i)},$$

$$\frac{\partial^2 R_{emp}(f)}{\partial \theta \partial \theta^T} = \sum_{i=1}^{n} \frac{\exp\left(y^{(i)} \theta^T x^{(i)}\right)}{\left(1 + \exp\left(y^{(i)} \theta^T x^{(i)}\right)\right)^2} x^{(i)} {x^{(i)}}^T.$$

Now iterate these two steps:

1. Evaluate the gradient and the Hessian matrix at the current value of the parameter vector $\theta^{[t]}$:

$$\left.\frac{\partial R_{emp}(f)}{\partial \theta}\right|_{\theta=\theta^{[t]}} = \sum_{i=1}^{n} \frac{-y^{(i)}}{1+\exp\left(y^{(i)}{\theta^{[t]}}^T x^{(i)}\right)} x^{(i)},$$

$$\left.\frac{\partial^2 R_{emp}(f)}{\partial \theta \partial \theta^T}\right|_{\theta=\theta^{[t]}} = \sum_{i=1}^{n} \frac{\exp\left(y^{(i)}{\theta^{[t]}}^T x^{(i)}\right)}{\left(1+\exp\left(y^{(i)}{\theta^{[t]}}^T x^{(i)}\right)\right)^2} x^{(i)} {x^{(i)}}^T.$$

2. update the estimate for θ using this formula:

$$\theta^{[t+1]} = \theta^{[t]} - \lambda \left(\left.\frac{\partial^2 R_{emp}(f)}{\partial \theta \partial \theta^T}\right|_{\theta=\theta^{[t]}}\right)^{-1} \left.\frac{\partial R_{emp}(f)}{\partial \theta}\right|_{\theta=\theta^{[t]}}$$

- The *stepsize* or *learning rate* parameter λ controls the size of the updates per iteration t.
- We stop if the differences between successive updates of θ are below a certain threshold or once a maximum number of iterations is reached.
- Note: The classic Newton-Raphson method uses $\lambda = 1$, but we can enlarge its radius of convergence by optimizing λ in every step.

We can implement this quite easily by realizing that we are doing something very similar to the gradient descent method. The only difference is that now we use a direction for the update that is given by the gradient vector (i.e., the first derivatives) weighted with the inverse of second derivatives.

Why does this make sense? (very rough intuition building):

a) What does the inverse Hessian $\left(\left.\frac{\partial^2 R_{emp}(f)}{\partial \theta \partial \theta^T}\right|_{\theta=\theta^{[t]}}\right)^{-1}$ mean?

The second derivatives represent how curved the risk surface is at the current point $\theta^{[t]}$.
Where the surface is very strongly curved (i.e., has large second derivatives), the direction of steepest descent changes very quickly. Where the surface is barely curved at all (i.e., has small second derivatives), the direction of steepest descent changes very slowly.
So, if we are at a point where the surface is very strongly curved (i.e., has large second derivatives), we need to take relatively small steps in the currently steepest direction so that we don't go *past* the minimum and back uphill. Conversely, if we are at a spot where there is little curvature, we can take bigger steps in the currently steepest direction without overshooting the minimum.
That's the reason why we use the *inverse* second derivatives to rescale the gradient.

b) Why multiply the inverse Hessian matrix with the negative gradient?

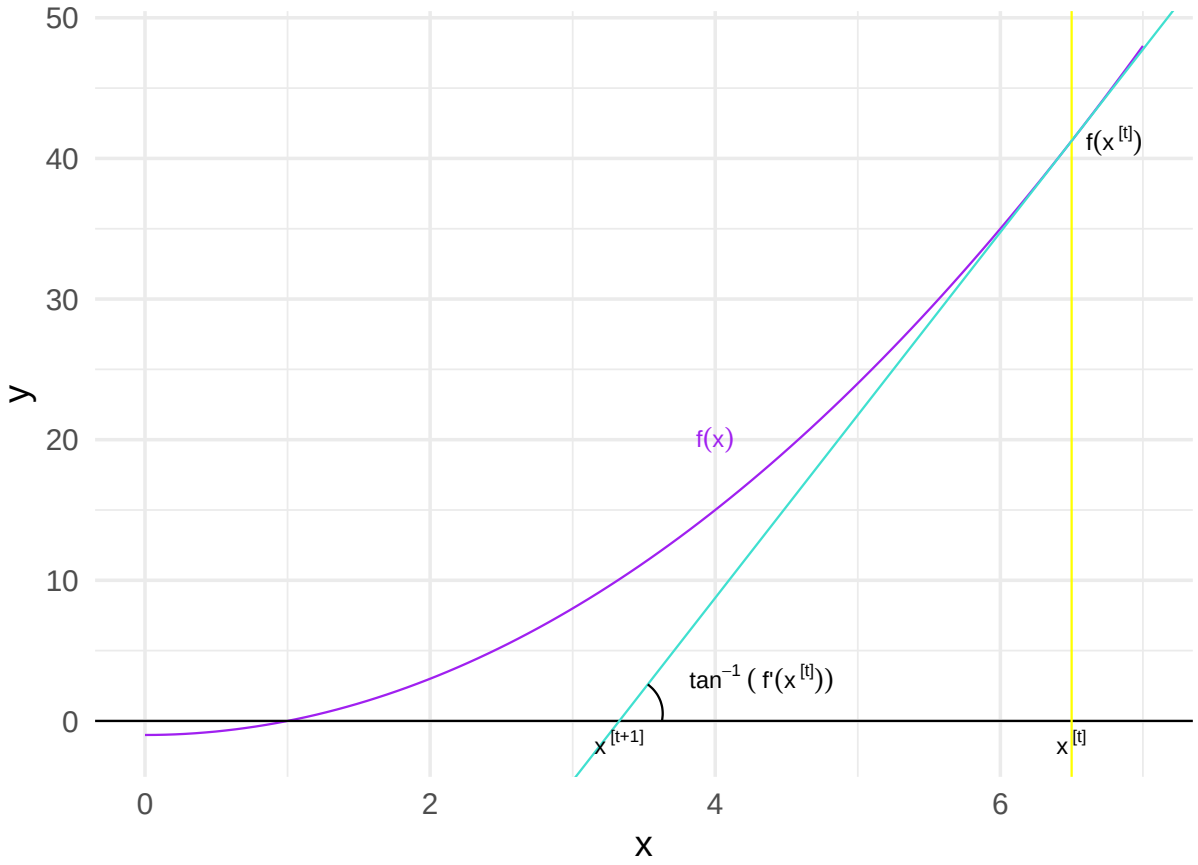
For vector-valued θ, the second derivative is a matrix. In some sense this matrix tells us, for every possible direction, whether the surface gets more steep (positive gradient becomes more positive, negative gradient becomes more negative) or more flat (gradient becomes smaller in absolute value) as we go into that direction.
We can use this information to pick an even better direction "downhill" than simply the negative gradient: instead of using the one that's currently the steepest (i.e, the negative gradient), we use one that is already steep and that becomes even more steep as we follow it (negative gradient times the inverse hessian).

```
library(MASS)

newton_opt_stepsize <- function(Y, X, theta,
                                risk,
```

```
                                gradient,
                                hessian,
                                lambda = 0.005,
                                epsilon = 0.0001,
                                max_iterations = 2000,
                                min_learning_rate = 0,
                                max_learning_rate = 1000,
                                plot = TRUE,
                                include_storage = FALSE) {
  gradient_descent_opt_stepsize(Y, X, theta,
    risk = risk,
    gradient = function(Y, X, theta)
      ginv(hessian(Y, X, theta)) %*%
        gradient(Y, X, theta),
    lambda = lambda,
    epsilon = epsilon,
    max_iterations = max_iterations,
    min_learning_rate = min_learning_rate,
    max_learning_rate = max_learning_rate,
    plot = plot,
    include_storage = include_storage
  )
}
```

Idea of Newton-Raphson

Another way to look at this is to view Newton-Raphson as a method for finding a root of a function. To understand this, let's consider a scalar function f whose root we are looking for:



The idea for finding an approximate root consists in following the negative gradient from the starting point until it crosses zero. By applying simple trigonometry one derives that

$$f'(x^{[t]}) = \frac{f(x^{[t]})}{x^{[t]} - x^{[t+1]}}.$$

By solving for $x^{[t+1]}$ we get that

$$x^{[t+1]} = x^{[t]} - f'(x^{[t]})^{-1} \cdot f(x^{[t]}),$$

which is the one-dimensional Newton-Raphson method. But how does finding the root of a function help in minimizing the empirical risk function? Remember that a necessary condition for a minimum is, that the gradient is zero, i.e., for the empirical risk that

$$\frac{\partial R_{emp}(f)}{\partial \theta} = 0.$$

So, by applying the multidimensional version of the Newton-Raphson method to the gradient function (i.e, the derivative of the empirical risk function), we are minimizing the empirical risk function.
This is also the reason why the second derivatives (the Hessian matrix) appears: to perform Newton-Raphson on the gradient, we need to take the derivative of the derivative of the risk.

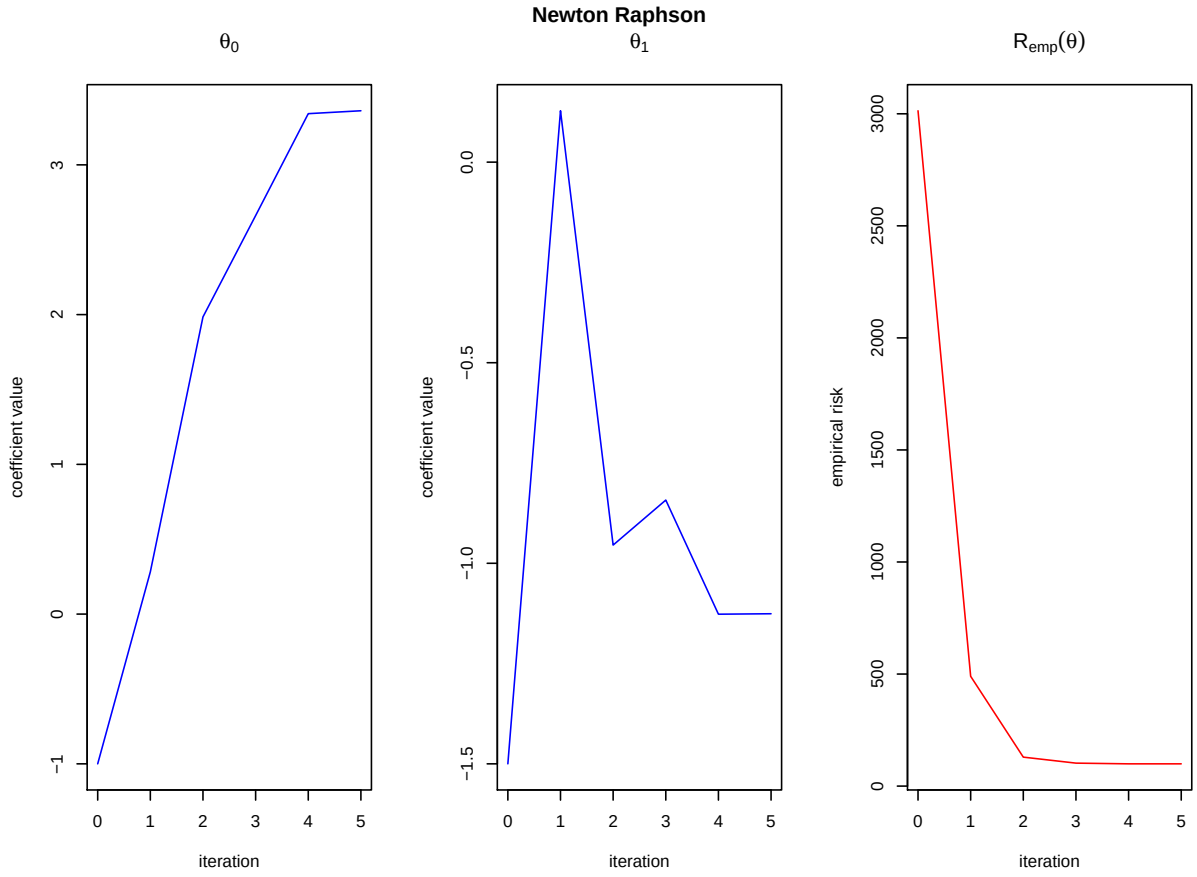
Example continued with Newton-Raphson

As already mentioned for using Newton-Raphson we need the Hessian matrix:

```r
hessian_bernoulli <- function(Y, X, theta) {
  # note that in X, x^(i) is a row vector, so the order of
  # transposing stuff is reversed here compared to the formula
  # notation above ...
  exp_yxt <- exp(Y * (X %*% theta))
  t((exp_yxt / (1 + exp_yxt)^2)[, 1] * X) %*% X
}
```

Now we can use the Newton-Raphson for solving our example:

```r
res <- newton_opt_stepsize(Y, X, c(-1, -1.5),
  risk_bernoulli,
  gradient_bernoulli,
  hessian_bernoulli,
  min_learning_rate = 0.0,
  max_learning_rate = 10,
  max_iterations = 5,
  include_storage = TRUE
)
title(main = "Newton Raphson", outer = TRUE, line = -1)
```




```r
print(paste("Risk: ", res$risk))
```

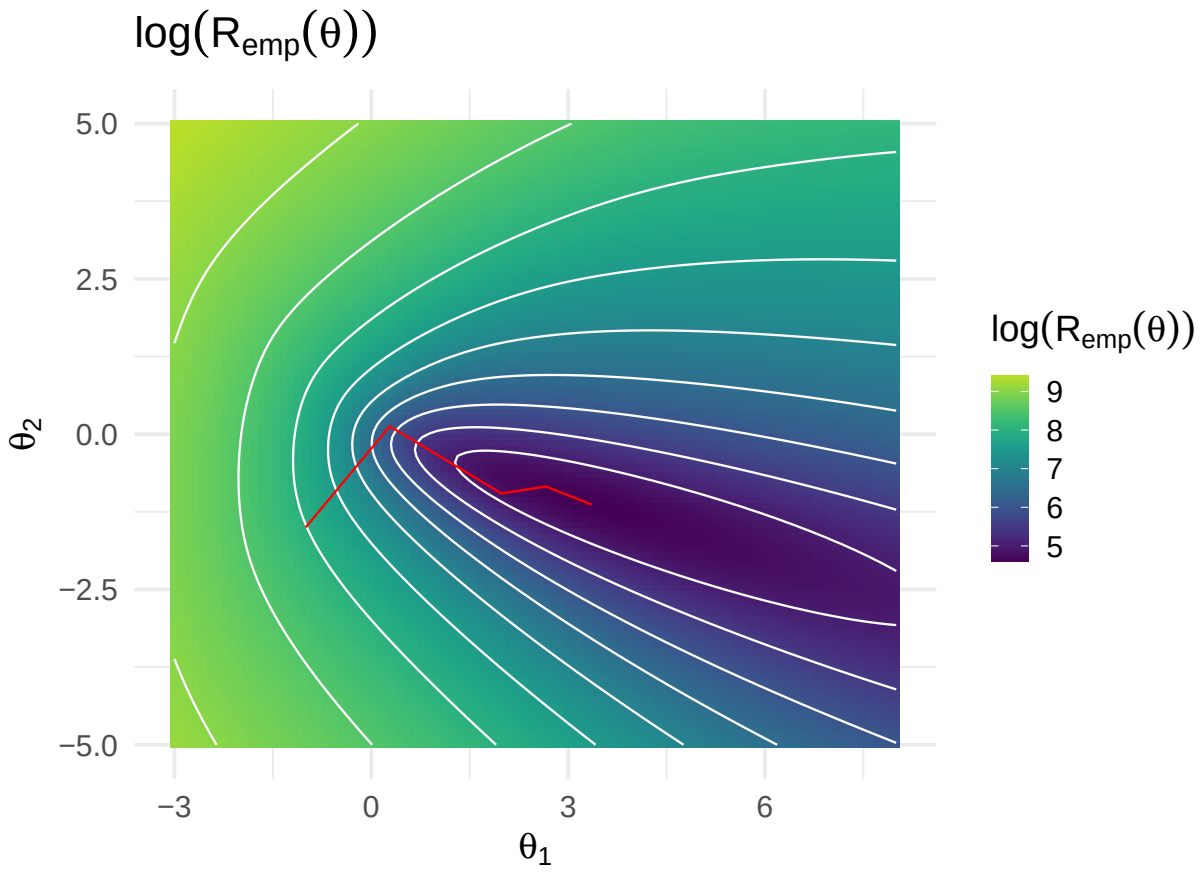
```
## [1] "Risk:  99.2995442059162"
```

```
res$theta
```

```
##         [,1]
## [1,]  3.36
## [2,] -1.13
```

We see that we need less steps to achieve a similar good result as when we used the gradient descent method.

```
plot2D_emp_risk(
  emp_risk = function(theta) (risk_bernoulli(Y, X, theta)),
  max_theta = c(8, 5), min_theta = c(-3, -5),
  log_emp_risk = TRUE,
  thetas = res$storage$theta
)
```



We see that by using Newton-Raphson we are incorporating additional knowledge about our objective function, since the Hessian matrix represents the curvature of a function. This enables us to take bigger steps in the right direction. Gradient descent is a so-called first-order iterative optimization algorithm (because it only uses first derivatives), while Newton-Raphson is a second-order method (... it uses second derivatives, duh).
The behavior we see here are typical for how these classes of optimization algorithms perform:

- First-order iterative optimization algorithms usually need (many) more steps than second-order ones.

- However, the calculation for each step for a first-order one is (much) less *expensive* than for a second-order method, since there is no need to compute the Hessian matrix and invert it.

Note: In our toy example we did not use an intercept so that we could visualize the algorithms in 2D. For real data sets, an intercept should definitively be included – it represents something like the base-line probability for class "1"[2].

[2] i.e, $\pi(x) = \frac{1}{1+\exp(\theta_0+\theta_1 x)}$, so $\pi(0) = \frac{1}{1+\exp(\theta_0)}$. If we didn't have a θ_0, we would always have $\pi(0) = \frac{1}{1+\exp(0)} = \frac{1}{2}$. That will rarely make sense.