Introduction to Machine Learning

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

Linear model and loss minimization

Note

This code demo covers additional material, which is intended for students interested in more in-depth knowledge.

Problem

We can estimate the $\hat{\theta}$ coefficients by minimizing the empirical risk

$$R_{emp}(f) = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)}|\theta))$$

of the model over θ. With quadratic loss, this yields

$$R_{emp}(f) = \frac{1}{n}\sum_{i=1}^{n} (y^{(i)} - (x^{(i)})^T\hat{\theta})^2.$$

This can be written in matrix notation as:

$$R_{emp}(f) = \frac{1}{n}(X\theta - y)^T(X\theta - y) = \frac{1}{n}[\theta^T X^T X\theta - 2\theta^T X^T y + y^T y]$$

and our optimization problem becomes:

$$\hat{\theta} = \arg\min_{\theta} R_{emp}(f)$$

Solution

- we can solve this kind of minimization problem using an iterative technique termed *Gradient Descent*
- this is an extremely important technique that is applied - in many variants - to solve the *optimization* problem when optimizing many kinds of learners.
- *Note*: An analytic solution exists for the quadratic loss, s.t.

$$\hat{\theta} = (X^T X)^{-1} X^T y.$$

Gradient Descent

The Gradient Descent method follows this algorithm:

0. Initialize $\theta^{[0]}$ (randomly) and calculate the gradient of the empirical risk with respect to θ, for example for the squared error loss:

$$\frac{\partial R_{emp}(f)}{\partial \theta} = \frac{\partial}{\partial \theta}\left(\frac{1}{n}[\theta^T X^T X\theta - 2\theta^T X^T y + y^T y]\right) = \frac{2}{n} X^T[X\theta - y]$$

Now iterate these two steps:

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

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

2. update the estimate for θ using this formula:

$$\theta^{[t+1]} = \theta^{[t]} - \lambda \frac{\partial R_{emp}(f)}{\partial \theta}\bigg|_{\theta=\theta^{[t]}}$$

- 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.
- Many variants of gradient descent exist that either
  - develop clever ways on how to choose good stepsizes (maybe even dependent on t),
  - and/or how to compute (approximations to) the gradient effectively,
  - and/or even adapt the direction of the update itself by taking into account, for example, the previously used update directions or the second derivatives of the empirical risk (i.e., the curvature of the risk surface). . . .

Idea of Gradient Descent

Think of the function to minimize as a mountain from which we try to find the way to the valley. In each step, we check for the direction of steepest descent (i.e., the negative gradient [1]) and move in that direction:

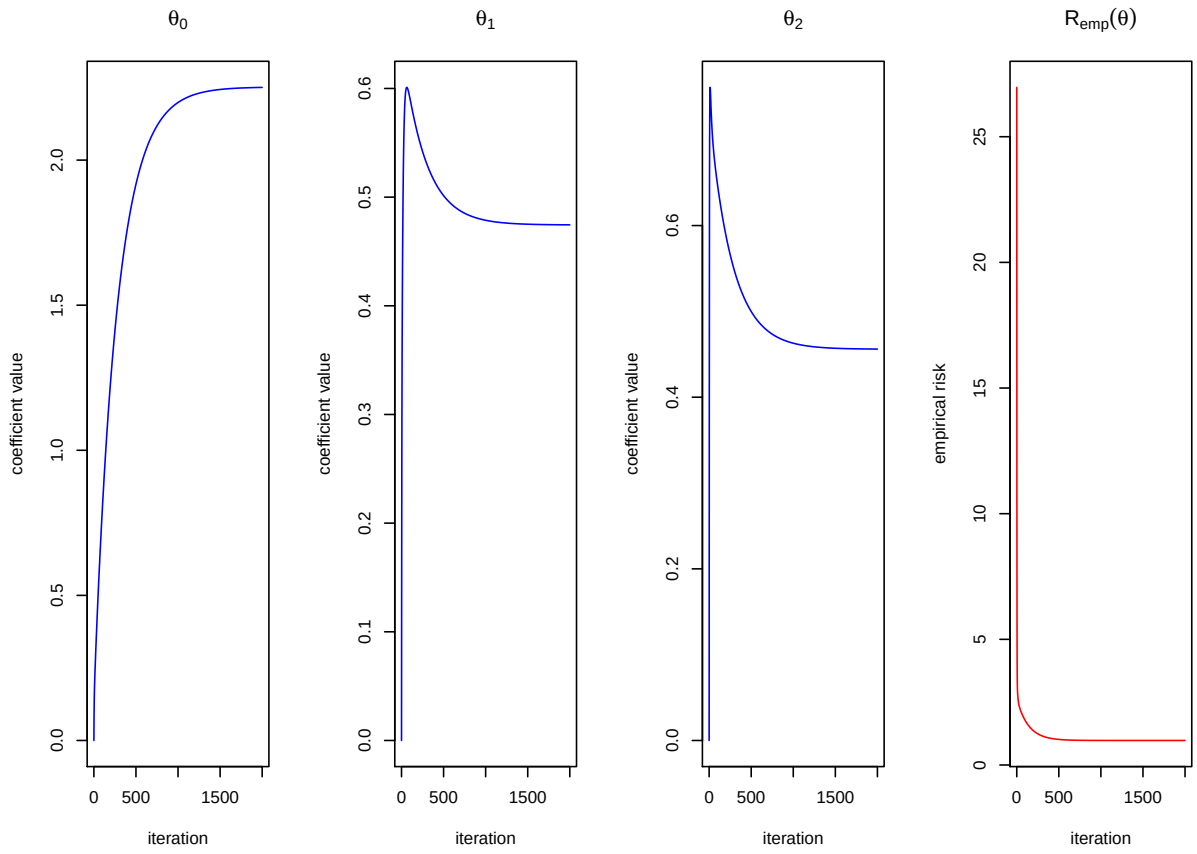
Implementation

```
# function that calculates the risk for squared error loss
risk_quadratic <- function(Y, X, theta) {
  1 / nrow(X) * t(X %*% theta - Y) %*% (X %*% theta - Y)
}
```

[1] if that doesn't mean anything to you or seems mysterious at all, please do read these additional explanations

```r
# function that calculates the gradient of the risk for squared error loss
gradient_quadratic <- function(Y, X, theta) {
  2 / nrow(X) * (t(X) %*% (X %*% theta - Y))
}
# function to perform gradient descent:
gradient_descent <- function(Y, X, theta,
                             risk = risk_quadratic,
                             gradient = gradient_quadratic,
                             lambda = 0.005,
                             max_iterations = 2000,
                             plot = TRUE) {
  # initialize storage for visualizations below (with maxiter + 1 slots)
  loss_storage <- data.frame(
    iterations = seq_len(max_iterations) - 1,
    loss = NA
  )
  theta_storage <- matrix(0, ncol = ncol(X), nrow = max_iterations + 1)

  loss_storage[1, "loss"] <- risk(Y, X, theta)
  theta_storage[1, ] <- theta

  #  loop over gradient updates
  for (i in 1:max_iterations) {
    theta <- theta - lambda * gradient(Y = Y, X = X, theta = theta)
    loss_storage[i + 1, "loss"] <- risk(Y = Y, X = X, theta = theta)
    theta_storage[i + 1, ] <- t(theta)
  }
  if (plot) {
    layout(t(1:(length(theta) + 1)))
    for (i in 1:length(theta)) {
      plot(x = seq_along(theta_storage[, i]) - 1, y = theta_storage[, i],
        ylab = "coefficient value",
        xlab = "iteration", type = "l", col = "blue",
        main = bquote(theta[.(i - 1)])
      )
    }
    plot(x = seq_along(loss_storage[, "loss"]) - 1, y = loss_storage[, "loss"],
      ylab = "empirical risk",
      xlab = "iteration", type = "l", col = "red",
      main = expression(R[emp](theta))
    )
  }
  list(theta = theta, risk = risk(Y, X, theta))
}
set.seed(1337)
n <- 100

#### simulated data with 2 simple features
# design matrix with intercept column:
X <- cbind(
  1,
  runif(n, -3, 5),
  runif(n, -2, 10)
```

```
)
true_theta <- c(2, 0.5, 0.5)
Y <- X %*% true_theta + rnorm(n)

# initialize theta with 0
theta_init <- rep(0, ncol(X))

gradient_descent(Y = Y, X = X, theta = theta_init)
```




```
## $theta
##       [,1]
## [1,] 2.251
## [2,] 0.475
## [3,] 0.456
##
## $risk
##       [,1]
## [1,] 0.976
```

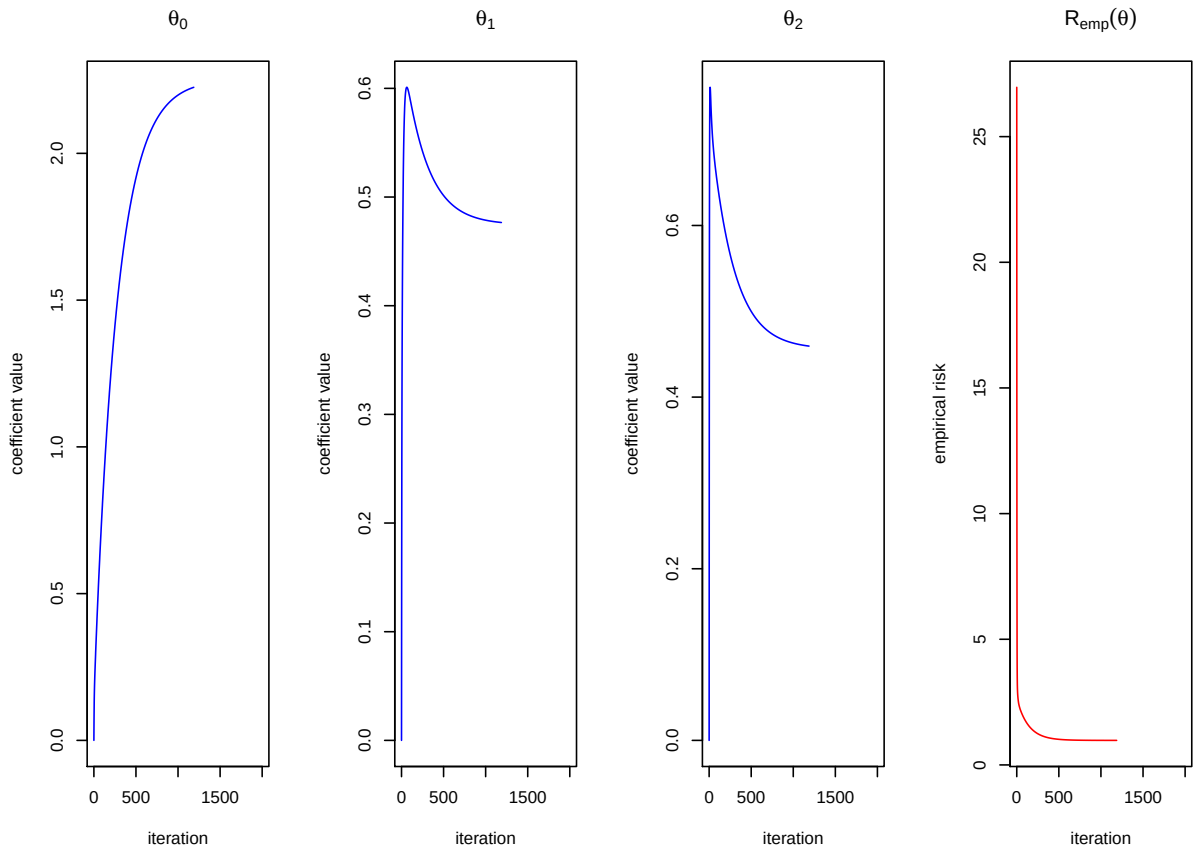
Stopping criterion

By looking at the figures we see that the empirical risk quickly starts to stagnate and is not improving much more, i.e., we are doing unnecessary computations. Because of this it makes sense to use a stopping criterion. One commonly used stopping criterion consists in checking if θ is not changing notably, e.g. that

for an $\epsilon > 0$

$$||\theta^{[t+1]} - \theta^{[t]}||_2 < \epsilon.$$

```
gradient_descent_threshold <- function(Y, X, theta,
                                       risk = risk_quadratic,
                                       gradient = gradient_quadratic,
                                       lambda = 0.005,
                                       epsilon = 0.0001,
                                       max_iterations = 2000,
                                       plot = TRUE) {
  # initialize storage for visualizations below (with maxiter + 1 slots)
  loss_storage <- data.frame(
    iterations = seq_len(max_iterations) - 1,
    loss = NA
  )
  theta_storage <- matrix(NA, ncol = ncol(X), nrow = max_iterations + 1)

  loss_storage[1, "loss"] <- risk(Y, X, theta)
  theta_storage[1, ] <- theta

  #  loop over gradient updates
  for (i in 1:max_iterations) {
    theta <- theta - lambda * gradient(Y = Y, X = X, theta = theta)
    loss_storage[i + 1, "loss"] <- risk(Y = Y, X = X, theta = theta)
    theta_storage[i + 1, ] <- t(theta)
    # check if convergence has been reached
    if (i > 1 && sqrt(sum((theta_storage[i, ] - theta_storage[i + 1, ])^2)) < epsilon) {
      break
    }
  }
  if (plot) {
    layout(t(1:(length(theta) + 1)))
    for (i in 1:length(theta)) {
      plot(x = seq_along(theta_storage[, i]) - 1, y = theta_storage[, i],
        ylab = "coefficient value",
        xlab = "iteration", type = "l", col = "blue",
        main = bquote(theta[.(i - 1)])
      )
    }
    plot(x = seq_along(loss_storage[, "loss"]) - 1, y = loss_storage[, "loss"],
      ylab = "empirical risk",
      xlab = "iteration", type = "l", col = "red",
      main = expression(R[emp](theta))
    )
  }
  list(theta = theta, risk = risk(Y, X, theta))
}
set.seed(1337)

gradient_descent_threshold(Y = Y, X = X, theta = theta_init)
```




```
## $theta
##       [,1]
## [1,] 2.225
## [2,] 0.477
## [3,] 0.459
##
## $risk
##       [,1]
## [1,] 0.976
```

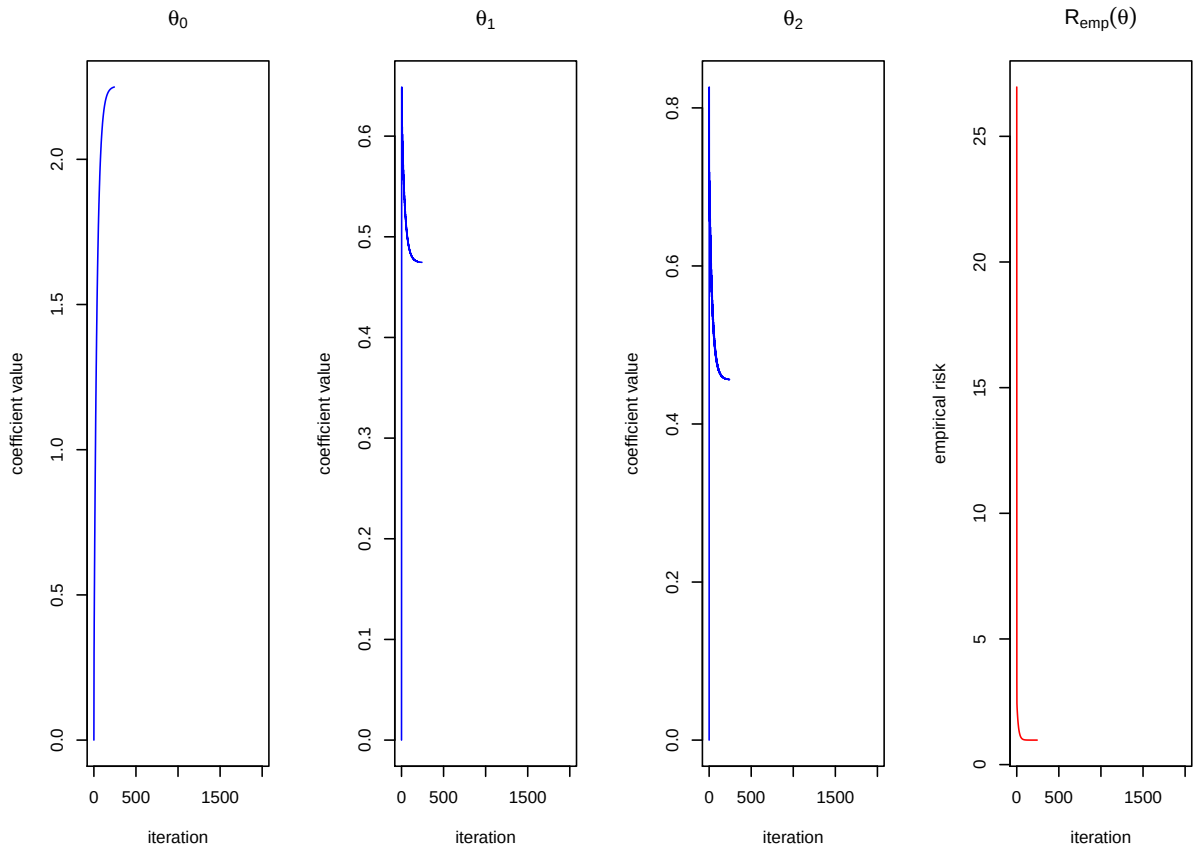
Stepsize

We notice that our implementation depends on the hyperparameter stepsize λ. Our implementation could be improved by doing an inner optimization for the stepsize λ, s.t. we follow the direction of the gradient until the function starts to increase again. For this task we can use the `optim` function. This is called a *line search* algorithm, because we're looking for a (local) minimum along the line of steepest descent.

```
gradient_descent_opt_stepsize <- function(Y, X, theta,
                                          risk = risk_quadratic,
                                          gradient = gradient_quadratic,
                                          lambda = 0.005,
                                          epsilon = 0.0001,
                                          max_iterations = 2000,
                                          min_learning_rate = 0,
                                          max_learning_rate = 1000,
```

```
                                          plot = TRUE,
                                          include_storage = FALSE) {
  # initialize storage for visualizations below (with maxiter + 1 slots)
  loss_storage <- data.frame(
    iterations = seq_len(max_iterations) - 1,
    loss = NA
  )
  theta_storage <- matrix(NA, ncol = length(theta), nrow = max_iterations + 1)

  loss_storage[1, "loss"] <- risk(Y, X, theta)
  theta_storage[1, ] <- theta

  #  loop over gradient updates
  for (i in 1:max_iterations) {
    grad <- gradient(Y = Y, X = X, theta = theta)
    lambda_opt <- optim(
      par = lambda, # start value
      fn = function(lambda) risk(Y = Y, X = X, theta = theta - lambda * grad), # to min
      method = "Brent", # 1d minimization
      lower = min_learning_rate,
      upper = max_learning_rate,
    )$par

    theta <- theta - lambda_opt * grad
    loss_storage[i + 1, "loss"] <- risk(Y = Y, X = X, theta = theta)
    theta_storage[i + 1, ] <- t(theta)
    # check if convergence has been reached
    if (i > 1 && sqrt(sum((theta_storage[i, ] - theta_storage[i + 1, ])^2)) < epsilon) {
      break
    }
  }
  if (plot) {
    layout(t(1:(length(theta) + 1)))
    for (i in 1:length(theta)) {
      plot(x = seq_along(theta_storage[, i]) - 1, y = theta_storage[, i],
        ylab = "coefficient value",
        xlab = "iteration", type = "l", col = "blue",
        main = bquote(theta[.(i - 1)])
      )
    }
    plot(x = seq_along(loss_storage[, "loss"]) - 1, y = loss_storage[, "loss"],
      ylab = "empirical risk",
      xlab = "iteration", type = "l", col = "red",
      main = expression(R[emp](theta))
    )
  }
  if(include_storage)
     list(theta = theta, risk = risk(Y, X, theta),
          storage = list(loss = loss_storage, theta = theta_storage))
  else
    list(theta = theta, risk = risk(Y, X, theta))
}
```

```r
set.seed(1337)
gradient_descent_opt_stepsize(Y = Y, X = X, theta = theta_init)
```




```
## $theta
##       [,1]
## [1,] 2.249
## [2,] 0.475
## [3,] 0.456
##
## $risk
##       [,1]
## [1,] 0.976
```

Comparison with `mlr3` / `stats::lm`

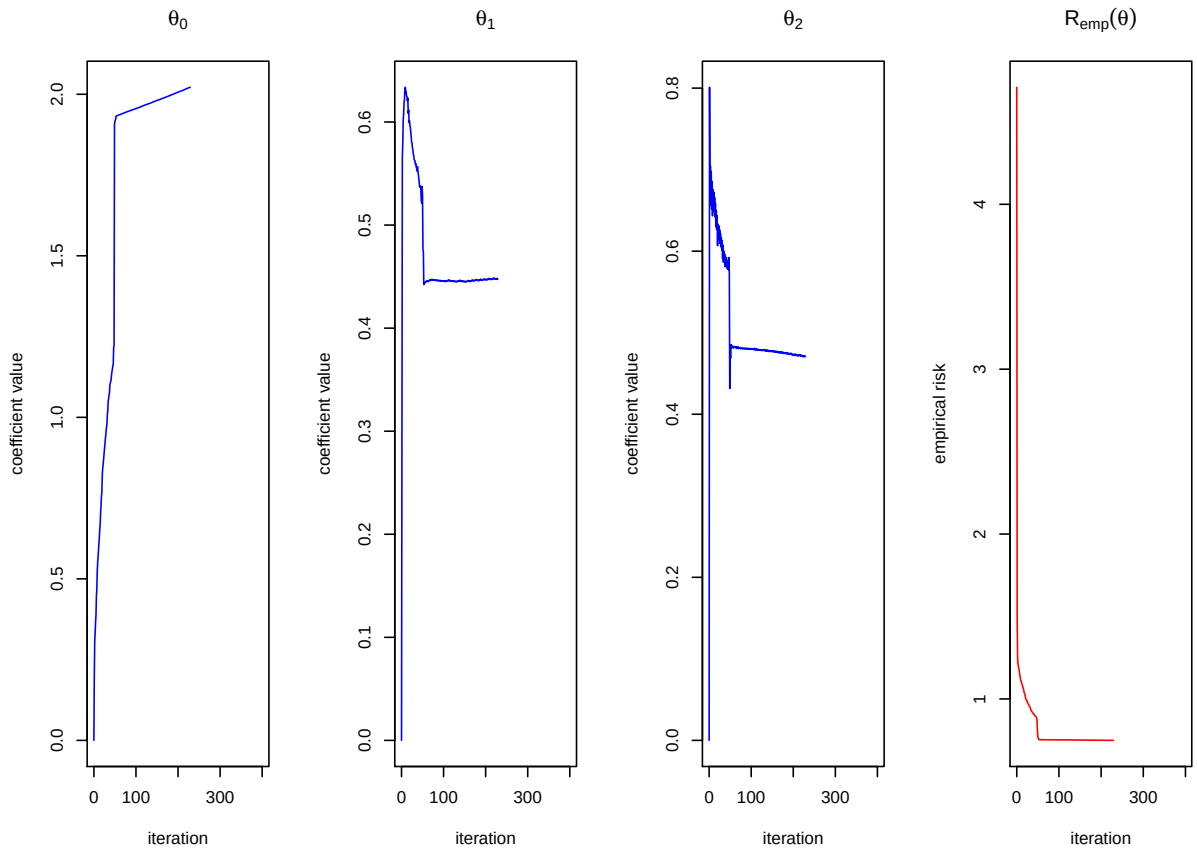
```r
library(mlr3)
library(mlr3learners)

df <- data.frame(Y= Y, X = X[, -1 ])
sim_task <- TaskRegr$new(id = "sim", backend = df, target = "Y")

sim_lm <- lrn("regr.lm")
sim_lm$train(sim_task)
sim_lm$model
```

```
##
## Call:
## stats::lm(formula = task$formula(), data = task$data())
##
## Coefficients:
## (Intercept)          X.1          X.2
##       2.252        0.474        0.456
```

Absolute Loss

As already mentioned, there exists an analytic solution to the empirical risk minimization with quadratic risk. Obviously, the gradient descent method is useful primarily in the case where no such easy analytic solution exists, such as for absolute (L1) loss-

```
risk_absolute <- function(Y, X, theta) {
  mean(abs(X %*% theta - Y))
}
# function that calculates the gradient of the risk for absolute error loss
gradient_absolute <- function(Y, X, theta) {
  1/nrow(X) * t(X) %*% sign(X %*% theta - Y)
}

set.seed(1337)

gradient_descent_opt_stepsize(
  Y = Y, X = X, theta = theta_init,
  risk = risk_absolute,
  gradient = gradient_absolute,
  epsilon = 0.0005,
  max_iterations = 4e2,
  min_learning_rate = 0.01
)
```




```
## $theta
##       [,1]
## [1,] 2.022
## [2,] 0.448
## [3,] 0.471
##
## $risk
## [1] 0.749
```

```
# Further thoughts:
# Compare the risk we have got with the risk we would get if the minimal learning rate is
# set to zero. Set the mininmal learning rate back to 0.01:
# What happens if you set epsilon to its default value? Does it converge for a higher
# number of iterations with this smaller epsilon value?
# Is the same problem encountered with the quadratic loss function, if a minimal learning
# rate is set?
# Hint: Does the gradient of the quadratic loss function shrink as the minimum is
# approached? Does this also hold for the gradient of the absolute loss?
```

Note: The gradient of the empirical risk for absolute loss is not defined at its minimum, nor at many other configurations of θ (think about why...?). That is often not a problem in practice, but this naive implementation runs into problems, as you can see if you play around with the settings of the code above (and in the rather wild looking optimization traces in the figure above)...