$\boxed{\text{Notes 9: Optimization}}$

Contents

Intro 1

Calculus Optimization Review 1

Newton's Method 2

Gradient Descent Method 5
- Line Search 6

Optimization in Higher Dimensions 7
- Gradient Descent 7
- Newton's Method 10

R `optimize()` and `optim()` Functions 12

Bonus: Conjugate Gradient Method 14
- The Method 14
- Preconditioning 16

Intro

Optimization problems come up quite frequently in statistics and data science problems. We have encountered some optimization problems already in this class in the form of maximum likelihood maximization and Bayesian estimation.

Calculus Optimization Review

One way to optimize a function is to do so directly using calculus. Suppose we want to optimize $f(x)$ by finding either a global maximum or minimum. The steps to do this include:

1. Find $f'(x)$.
2. Set $f'(x)$ and solve for x to find the critical number(s).
3. Plug the critical number(s) and endpoint(s). If there are no endpoints, take a limit as x approaches the end of the domain.
4. Take the x value that maximizes or minimizes the function, if it exists.

Example 1: If $f(x) = \dfrac{x}{x^2+1}$, find the value of x that maximizes and minimizes the function.

So, we can say that $\underset{x}{\operatorname{argmax}}\left\{\dfrac{x}{x^2+1}\right\}$ is and the $\underset{x}{\operatorname{argmin}}\left\{\dfrac{x}{x^2+1}\right\}$ is .

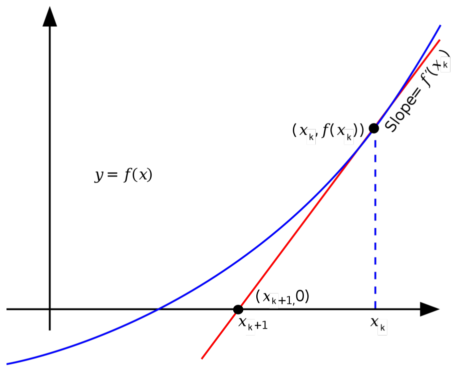
Example 2: Now let's try to maximize this function: $g(x) = x^2 + xe^x$.

This function has a minimizer, but it is not possible to directly solve for it. We would need a numerical method to get an approximation for x.

Newton's Method

One fairly straight forward and powerful method for finding roots (or zeros) of a function is **Newton's Method**. This method is iterative, which means that it typically needs to be run multiple times with each iteration giving a better approximation of the root.

The idea here is to first start with a simple guess of which x is the root. Call this first guess x_k (where $k = 0$ for the initial guess). Then use a tangent line to the curve at $x = x_k$ that intercepts the x-axis at x_{k+1} where x_k is the kth iteration of x.



Since any tangent line that does not have a slope of zero will intercept the x-axis somewhere, we will call the x value were it does x_{k+1}. Then it will be true that $f'(x_k) = \dfrac{f(x_k) - 0}{x_k - x_{k+1}}$. Then we can simply solving this for the x_{k+1}:

Therefore, we have

Example 3: Trying this out for finding the root of $g'(x) = 2x + e^x(x+1)$ from Example 2 above. First, let $f(x) = g'(x)$.

To program this in **R**, it is common to use a tolerance for stopping. That is, if x_{k+1} is very close to x_k, then we should stop.

```r
f <- function(x) {
  2*x + exp(x)*(x+1)
}
f_prime <- function(x) {
  2 + exp(x)*(x+1) + exp(x)
}

xk <- 0 # Initialize our first guess
maxit <- 100 # Set a max iteration number
for(k in 1:maxit) {
  xnew <- xk - f(xk)/f_prime(xk)
  if(abs(xk - xnew) < 1e-10) {
    break
  }
  xk <- xnew
}
xk # Estimate for the zero
```

```
## [1] -0.2752084
```

```r
k  # Number of iterations to converge
```

```
## [1] 5
```

It only took 5 iterations to converge to within 10^{-10}. If we wanted to make it even more precise, we could set the tolerance even lower, like `1e-16`. This would make x_k and x_{k+1} equal to 15 decimal places.

Now, notice that by finding the root of $f(x)$ in this example, we were really finding the critical number (the x value where the derivative is zero) of $g(x)$ since $g'(x) = f(x)$. That means that the iteration sequence for finding the critical number of $g(x)$ is

$$\boxed{x_{k+1} = x_k - \frac{g'(x_k)}{g''(x_k)}}.$$

That gives us our first optimization method. This is sometimes referred to as **Newton's Method for Optimization**.

Example 4: Find the x values that minimizes $f(x) = x^4 - 3x^2 + x$ using Newton's method.

```r
f <- function(x) {
  x^4 - 3*x^2 + x
}
f_prime <- function(x) {
  4*x^3 - 6*x + 1
}
f_double_prime <- function(x) {
  12*x^2 - 6
}

xk <- 0 # Initialize our first guess
maxit <- 100 # Set a max number of iterations
for(k in 1:maxit) {
  xnew <- xk -
    f_prime(xk)/f_double_prime(xk)
  if(abs(xk - xnew) < 1e-10) {
    break
  }
  xk <- xnew
}
xk # Estimate for the zero
```

```
## [1] 0.1699384
```

```r
k  # Number of iterations to converge
```

```
## [1] 4
```

Does $x = 0.1699384$ minimize the function? What if we started at a value different than 0:

```r
xk <- 1 # Initialize our first guess
for(k in 1:maxit) {
  xnew <- xk -
    f_prime(xk)/f_double_prime(xk)
  if(abs(xk - xnew) < 1e-10) {
    break
  }
  xk <- xnew
}
xk # Estimate for the zero
```

```
## [1] 1.130901
```

```r
xk <- -1 # Initialize our first guess
for(k in 1:maxit) {
  xnew <- xk -
    f_prime(xk)/f_double_prime(xk)
  if(abs(xk - xnew) < 1e-10) {
    break
  }
  xk <- xnew
}
xk # Estimate for the zero
```

```
## [1] -1.30084
```

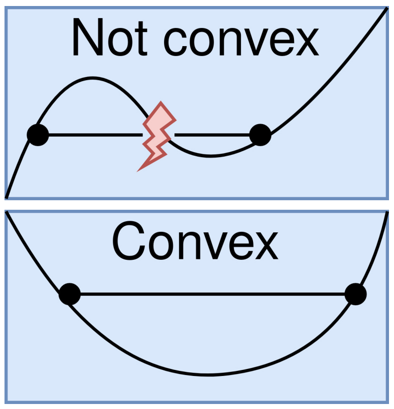
Now we are getting three answers at three different starting points! What is happening?

```r
f(0.1699384); f(1.130901); f(-1.30084)
```

```
## [1] 0.08413522
## [1] -1.07023
## [1] -3.513905
```

This is illustrating one of the most common problems with optimization algorithms: they can converge to local extrema instead of global extrema. In addition, some methods can also converge to the wrong type of extrema. That is, they could find a maximum when a minimum is desired and vise-versa.

This first issue, converging to the wrong point, occurs if the function is not **convex**. A function is convex if the line segment between any two distinct points on the graph of the function lies above the graph between the two points. There are different types of convex functions.



- f is convex if and only if $f''(x) \geq 0$ for all x.
- f is called **strictly convex** if $f''(x) > 0$ for all x.
- f is called **strongly convex** if $f''(x) \geq m > 0$ for all x.

A convex function is shaped like a cup $\cup$. A **concave** function is shaped like a cap $\cap$. If $f(x)$ is convex, then $-f(x)$ would be concave and vise-versa. Many functions are neither convex nor concave.

We can alleviate these problems (to an extent) by

1. Forcing our method to always find a minimum.

- This will make it so we do not find type of extrema we don't want.
- We can always find a maximum of $f(x)$ by minimizing $-f(x)$.

2. Allowing a change in the step size of our method.

- This can (sometimes) get us "unstuck" from a local minimum.
- Truthfully, getting stuck at a local minimum is the biggest problem for any optimization technique and not one we will be able to tackle here. The easiest way to get around this is to choose different starting points.

Gradient Descent Method

One way to remove the potential to converge to a maximum is to remove the second derivative from the equation and instead multiply by a value that is always positive. If we have $x_{k+1} = x_k - \gamma \cdot f'(x_k)$, then if $f'(x_k) > 0$, we will move to the left (toward a minimum) and if $f'(x_k) < 0$, we will move to the right (again, toward a minimum).

So, all we need to tell us which direction to move is $f'(x_k)$. This does not, however, tell us how far to move. One possible value for γ is $\gamma = 1/|f''(x_k)|$. This usually works, and when it does, it often works well, but if the function has a minimum where the second derivative is 0 or if the function is not twice differential, then this will fail. Also, in higher dimensions, the second derivative matrix, called the Hessian, is also much more difficult and time consuming to compute.

A method that does not rely on the second derivative or Hessian is called a first-order optimization method. One that does rely on the second derivative is a second-order method.

One thing we would like to be true about our step size, γ is that it should get smaller as we approach the minimizer. Thus, γ should change at each iteration. So let's call it γ_k.

Line Search

One way to choose γ_k is to make sure we achieve the maximum amount of decrease of the objective function at each individual step. That means we want to minimize $f(x_k - \gamma_k f'(x_k))$. That is, we want

$$\gamma_k = \underset{\gamma \geq 0}{\text{argmin}} \left\{ f\Big(x_k - \gamma f'(x_k)\Big) \right\}.$$

So, we have to find a minimizer (γ) while we are trying to find another minimizer (x)!

One way to find γ_k is to use a line search. One option is a **backtracking line search**. In this method, we fix a line-search parameter β for $0 < \beta < 1$. Then start with $\gamma = 1$ and while $f\Big(x_k - \gamma f'(x_k)\Big) > f(x_k) - \frac{\gamma}{4} f'(x_k)^2$, update $\gamma = \beta \cdot \gamma$ and try again. Once we have a value of γ when that inequality is not true, we set $\gamma_k = \gamma$.

The code below shows the backtracking line search in action!

```r
xk <- 0 # Initialize our first guess
for(k in 1:maxit) {
  gamma <- 1
  beta <- 0.5
  while(f(xk - gamma * f_prime(xk)) > f(xk) - gamma/4 * f_prime(xk)^2) {
    gamma <- beta * gamma
  }
  xnew <- xk - gamma * f_prime(xk)
  if(abs(xk - xnew) < 1e-10) {
    break
  }
  xk <- xnew
}
xk # Estimate for the zero
```

```
## [1] -1.30084
```

```r
k
```

```
## [1] 10
```

Using $1/|f''(x_k)|$ as our step size (Newton's method), $k = 9$ for convergence. So, very close! The time for this method is 0.005 seconds on my computer while Newton's method took 0.003 seconds.

So where does the $f(x_k) - \frac{\gamma}{4} f'(x_k)^2$ come from in the line search inequality? We want to minimize $g(\gamma) = f\Big(x_k - \gamma f'(x_k)\Big)$ with respect to γ. Take the derivative with respect to γ:

$$g'(\gamma) = -f'(x_k) f'\Big(x_k - \gamma f'(x_k)\Big).$$

When $\gamma = 0$, the equation of the tangent line for $f\Big(x_k - \gamma f'(x_k)\Big)$ is $y = g(0) + g'(0)(\gamma - 0)$, which is

$$\begin{aligned} y &= f(x - 0f'(x_k)) - f'(x_k) \cdot f'\Big(x - 0f'(x_k)\Big)(\gamma - 0) \\ &= f(x) - \gamma f'(x_k)^2. \end{aligned}$$

So, we are sure that $f\Big(x_k - \gamma f'(x_k)\Big) = f(x_k) - \gamma f'(x_k)^2$ when $\gamma = 0$. When γ is small we have,

$$f\Big(x_k - \gamma f'(x_k)\Big) \approx f(x_k) - \gamma f'(x_k)^2$$

due to a linear approximation. Since $\gamma > 0$ and $f'(x_k)^2 > 0$, we know that when γ is small, for $0 < \alpha < 1$,

$$f(x_k) - \gamma f'(x_k)^2 < f(x_k) - \alpha \cdot \gamma f'(x_k)^2.$$

Then that means

$$f\Big(x_k - \gamma f'(x_k)\Big) < f(x_k) - \alpha \cdot \gamma f'(x_k)^2$$

as well since $f\Big(x_k - \gamma f'(x_k)\Big) \approx f(x_k) - \gamma f'(x_k)^2 < f(x_k) - \alpha \cdot \gamma f'(x_k)^2$.

Therefore, specifying a larger γ value (like $\gamma = 1$) and then gradually making it smaller will eventually get it down to a point where

$$f\Big(x_k - \gamma f'(x_k)\Big) < f(x_k) - \alpha \cdot \gamma f'(x_k)^2.$$

That will help us find one of the largest γ values that will work as a step size. It has been found that α between 0.01 and 0.3 works well, so we will use $\alpha = 0.25$. For more info on this, see *Convex Optimization* by Boyd and Vandenberghe page 465.

If we are able to perfectly optimize γ_k at each iteration to take the largest possible step, we would be performing the **steepest descent** optimization method. Steepest descent is a special case of **gradient descent**. Gradient descent is any method whose step direction is equal to $-f'(x_k)$ regardless of step size while steepest descent takes the largest step possible in that direction at each iteration. So while we are probably not actually performing **steepest descent** (since we are estimating the step size), we are attempting to do so. Gradient descent is by definition a first-order optimization method. So even though Newton's method involves $f'(x_k)$, it is not a gradient descent method since it relies on the second derivative.

Optimization in Higher Dimensions

Gradient Descent

Now that we have some intuition about these optimization algorithms, we should discuss them in higher dimensions. A mutli-dimensional derivative is known as a **gradient** (hence the name of gradient descent). A gradient gives the direction of fastest change (increase or decrease). These gradients can be computed using **partial derivatives**. This is a Calculus III topic, but they are pretty simple. When taking the partial derivatives of a function with respect to a variable, we just treat all other variables as constants.

Example 5 Let $f(x, y, z) = 12x^2y + y^2z + z^2$.

$\dfrac{\partial f}{\partial x} =$ $\qquad\qquad$ $\dfrac{\partial f}{\partial y} =$ $\qquad\qquad$ $\dfrac{\partial f}{\partial z} =$

The gradient of $f(x_1, x_2, \ldots, x_n) = f(\boldsymbol{x})$ is $\nabla f(\boldsymbol{x}) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \dfrac{\partial f}{\partial x_2} \\ \vdots \\ \dfrac{\partial f}{\partial x_n} \end{bmatrix}$. So the gradient of a function of n variables is an n directional vector. The gradient of the function in Example 5 is $f(x, y, z)$ is $\nabla f(\boldsymbol{x}) = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \\ \vdots \\ \dfrac{\partial f}{\partial z} \end{bmatrix} = \begin{bmatrix} 24x \\ 12x^2 + 2yz \\ \vdots \\ y^2 + 2z \end{bmatrix}$. That means the slope at a given point (x, y, z) is greatest when we go $24x$ in the x-direction, $12x^2 + 2yz$ in the y-direction, and $y^2 + 2x$ in the z-direction.

For using the gradient descent method in higher dimensions, we need to replace
$f'(x)$ with $\nabla f(\boldsymbol{x})$. So, we have

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \gamma_k \nabla f(\boldsymbol{x}).$$

We can find γ_k again using a line search. We still fix a line-search parameter β for $0 < \beta < 1$. Then start with $\gamma = 1$ and while $f\Big(\boldsymbol{x}_k - \gamma \nabla f(\boldsymbol{x}_k)\Big) > f(\boldsymbol{x}_k) - \frac{\gamma}{4} \|\nabla f(\boldsymbol{x}_k)\|^2$, update $\gamma = \beta\gamma$ and try again. Once we have a value of γ when that inequality is not true, we set $\gamma_k = \gamma$.

Notice that $f'(x_k)^2$ has been replaced with $\|\nabla f(\boldsymbol{x}_k)\|^2$, which is the squared **Euclidean norm** of the gradient vector. That is, $\|\nabla f(\boldsymbol{x}_k)\|^2 = \sum_{i=1}^{n} \left(\dfrac{\partial f}{\partial x_i}\Big|_{\boldsymbol{x}_k} \right)^2$. We can find this in **R** using either `norm(gradient, type = "2")` or `sum(gradient^2)`.

Also, when we check for convergence, the norm of the difference in the vectors can be used.

Example 6: Recall in Notes 7 when we discussed the Poisson regression, we used maximum likelihood to find the estimates for the intercept and slope. In that example we had the log likelihood was

$$\ell(\beta_0, \beta_1) = \sum_{i=1}^{n} y_i(\beta_0 + \beta_1 x_i) - \sum_{i=1}^{n} \exp(\beta_0 + \beta_1 x_i) - \sum_{i=1}^{n} \ln(y_i!)$$

However, we want to maximize this function and our algorithm only works to minimize functions. Easy fix! Maximizing a function is the same as minimizing its negative. So, we want to minimize

$$n\ell(\beta_0, \beta_1) = -\left(\sum_{i=1}^{n} y_i(\beta_0 + \beta_1 x_i) - \sum_{i=1}^{n} \exp(\beta_0 + \beta_1 x_i) - \sum_{i=1}^{n} \ln(y_i!) \right)$$

Now obtain the gradient:

$$\nabla n\ell(\beta_0, \beta_1) = \begin{bmatrix} \dfrac{\partial n\ell}{\partial \beta_0} \\[2ex] \dfrac{\partial n\ell}{\partial \beta_1} \end{bmatrix} = \begin{bmatrix} -\left(\sum_{i=1}^n y_i - \sum_{i=1}^n \exp(\beta_0 + \beta_1 x_i)\right) \\ -\left(\sum_{i=1}^n x_i y_i - \sum_{i=1}^n x_i \exp(\beta_0 + \beta_1 x_i)\right) \end{bmatrix}$$

In this case, the variables we are trying to find are β_0 and β_1. Let's code an implementation of this in **R**.

```r
set.seed(2024)
x <- rnorm(5) # Simulate 5 random values for x
# Simulate 5 random values for y (with a couple different means)
y <- c(rpois(3, 2), rpois(2, 6))
mod <- glm(y ~ x, family = "poisson") # Fit the Poisson model
b0 <- coef(mod)[1] # Extract the intercept
b1 <- coef(mod)[2] # Extract the slope
b0; b1
```

```
## (Intercept)
##    1.527908
##          x
## 0.08645586
```

```r
neg_likelihood <- function(beta) {
  # Make it negative since we are trying to maximize it.
  #   That is the same as minimizing the negative.
  -(sum(y*(beta[1] + beta[2]*x)) - sum(exp(beta[1] + beta[2]*x)) -
      sum(log(factorial(y))))
}
gradient <- function(beta) {
  # These should be the partial derivatives of the function we are
  #   minimizing. So, their signs are also opposites.
  dl_db0 <- -(sum(y) - sum(exp(beta[1] + beta[2]*x)))
  dl_db1 <- -(sum(x*y) - sum(x*exp(beta[1] + beta[2]*x)))
  c(dl_db0, dl_db1) # Return the gradient vector
}

maxit <- 100
betak <- c(0, 0) # Initialize our first guess
for(k in 1:maxit) {
  gamma <- 1
  line_beta <- 0.5 # Line search parameter, not a beta from the model!
  while(neg_likelihood(betak - gamma * gradient(betak)) >
        neg_likelihood(betak) - gamma/4 * sum(gradient(betak)^2)) {
    gamma <- line_beta * gamma
  }
  newbeta <- betak - gamma * gradient(betak)
  if(norm(betak - newbeta, type = "2") < 1e-10) {
    break
  }
  betak <- newbeta
}
betak # Estimate for the zero
```

```
## [1] 1.52790762 0.08645585
```

```
k
```

```
## [1] 39
```

Newton's Method

Now, for one input variable, Newton's method looks like

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

We already saw that a multi-dimensional version of $f'(x_k)$ is the gradient. A multi-dimensional version of a second derivative is called a Hessian. Here is how the Hessian of a function $f(x_1, x_2, \ldots, x_n) = f(\boldsymbol{x})$ is defined:

$$\mathbf{H}_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Note that when we take the second partial derivatives with more than one variable, $\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial}{\partial x_i}\left[\frac{\partial f}{\partial x_j}\right]$. So, we take the partial derivative with respect to the variable listed second in the denominator first, and then take the other partial derivative. However, this usually does not matter since $\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$ in almost every practical case.

Notice also that these operations are getting larger. $f(\boldsymbol{x})$ is a scalar, $\nabla f(\boldsymbol{x})$ is an $n \times 1$ vector, and $\mathbf{H}_f$ is an $n \times n$ matrix.

Getting back to the equation, we are dividing by $f''(x_k)$ in Newton's method. We don't divide matrices, instead, we take inverses and multiply. So, Newton's method becomes

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - [\mathbf{H}_f]^{-1} \nabla f(\boldsymbol{x}).$$

Because an inverse of a matrix exists here, this method can become **very** computational expensive when n is large.

Example 6: Let's try Newton's method on the Poisson regression problem. Recall the gradient:

$$\nabla n\ell(\beta_0, \beta_1) = \begin{bmatrix} \dfrac{\partial n\ell}{\partial \beta_0} \\ \\ \dfrac{\partial n\ell}{\partial \beta_1} \end{bmatrix} = \begin{bmatrix} -\left(\displaystyle\sum_{i=1}^{n} y_i - \sum_{i=1}^{n} \exp(\beta_0 + \beta_1 x_i)\right) \\ -\left(\displaystyle\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \exp(\beta_0 + \beta_1 x_i)\right) \end{bmatrix}$$

To complete the Hessian, we need to get more partial derivatives:

$$\mathbf{H}_f = \begin{bmatrix} \dfrac{\partial^2 n\ell}{\partial \beta_0^2} & \dfrac{\partial^2 n\ell}{\partial \beta_0 \partial \beta_1} \\ \\ \dfrac{\partial^2 n\ell}{\partial \beta_1 \partial \beta_0} & \dfrac{\partial^2 n\ell}{\partial \beta_1^2} \end{bmatrix} = \begin{bmatrix} \displaystyle\sum_{i=1}^{n} \exp(\beta_0 + \beta_1 x_i) & \displaystyle\sum_{i=1}^{n} x_i \exp(\beta_0 + \beta_1 x_i) \\ \displaystyle\sum_{i=1}^{n} x_i \exp(\beta_0 + \beta_1 x_i) & \displaystyle\sum_{i=1}^{n} x_i^2 \exp(\beta_0 + \beta_1 x_i) \end{bmatrix}$$

Let's code it up!

```r
neg_likelihood <- function(beta) {
  # Make it negative since we are trying to maximize it.
  #   That is the same as minimizing the negative.
  -(sum(y*(beta[1] + beta[2]*x)) - sum(exp(beta[1] + beta[2]*x)) -
      sum(log(factorial(y))))
}
gradient <- function(beta) {
  # These should be the partial derivatives of the function we are
  #   minimizing. So, their signs are also opposites.
  dl_db0 <- -(sum(y) - sum(exp(beta[1] + beta[2]*x)))
  dl_db1 <- -(sum(x*y) - sum(x*exp(beta[1] + beta[2]*x)))
  c(dl_db0, dl_db1) # Return the gradient vector
}
hessian <- function(beta) {
  matrix(
    c(sum(exp(beta[1] + beta[2]*x)), sum(x*exp(beta[1] + beta[2]*x)),
      sum(x*exp(beta[1] + beta[2]*x)), sum(x^2*exp(beta[1] + beta[2]*x))),
    nrow = 2
  )
}

maxit <- 100
betak <- c(0, 0) # Initialize our first guess
for(k in 1:maxit) {
  newbeta <- betak - solve(hessian(betak)) %*% gradient(betak)
  if(norm(betak - newbeta, type = "2") < 1e-10) {
    break
  }
  betak <- newbeta
}
betak # Estimate for the zero
```

```
##            [,1]
## [1,] 1.52790761
## [2,] 0.08645586
```

```
k
```

```
## [1] 9
```

This converges in far fewer iterations than the gradient descent method and a line search is not needed.

The bottleneck here is usually finding the Hessian, since that needs to be done by hand first, and then inverting the Hessian if n is big. However, we don't actually need to compute and store the Hessian matrix. We just need to know what $[\mathbf{H}_f]^{-1}\nabla f(\boldsymbol{x})$ is equal to. If we say $[\mathbf{H}_f]^{-1}\nabla f(\boldsymbol{x}) = \boldsymbol{r}$, it can often be much quicker to use numerical methods to determine what $\boldsymbol{r}$ is by setting $\mathbf{H}_f\boldsymbol{r} = \nabla f(\boldsymbol{x})$ and then applying a method like **conjugate gradient** (if applicable) to solve for $\boldsymbol{r}$ numerically at each iteration. We won't go over the conjugate gradient method, but I have included information about it on the last few pages of these notes in case you are interested.

R `optimize()` and `optim()` Functions

R has a couple of nice optimization functions built into the `stats` library that is a part of base **R**. For one-dimensional optimization, we can use the `optimize()` function. For general purpose optimization, including in multi-dimensional space, the `optim()` function is the one that should be used. These functions use algorithms that are more advanced than the ones we discussed here, but they are similar.

Let's see a couple examples:

```
f <- function(x) {
  x^4 - 3*x^2 + x
}
optimize(f, c(-10, 10)) # Finds the minimum
```

```
## $minimum
## [1] -1.30082
##
## $objective
## [1] -3.513905
```

```
optimize(f, c(-10, 10), maximum = T) # Finds the maximum (not helpful here)
```

```
## $maximum
## [1] 9.999926
##
## $objective
## [1] 9709.71
```

```r
set.seed(2024)
x <- rnorm(5)
y <- c(rpois(3, 2), rpois(2, 6))
likelihood <- function(beta) {
  # Define the positive likelihood
  sum(y*(beta[1] + beta[2]*x)) - sum(exp(beta[1] + beta[2]*x)) -
      sum(log(factorial(y)))
}
starting_values <- c(0, 0)
# Putting "fnscale" to a negative number makes us maximize instead of minimize
optim(starting_values, likelihood, control = list(fnscale = -1))
```

```
## $par
## [1] 1.52786041 0.08639149
##
## $value
## [1] -9.241506
##
## $counts
## function gradient
##       65       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Bonus: Conjugate Gradient Method

The conjugate gradient (CG) and preconditioned conjugate gradient (PCG) methods are algorithms for finding numerical solutions for systems of linear equations like those in the form

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}, \tag{1}$$

where $\mathbf{A}$ and $\boldsymbol{b}$ are known and we are interested in finding the solution $\boldsymbol{x}$. There are some restrictions on $\mathbf{A}$ in that it must be symmetric (i.e. $\mathbf{A}^T = \mathbf{A}$) and positive definite (i.e. $\boldsymbol{x}^T\mathbf{A}\boldsymbol{x} > 0$ for all nonzero $\boldsymbol{x} \in \mathbb{R}^n$). A matrix with both of these properties is known as symmetric positive definite (SPD). If these are satisfied, CG is one of the most useful numerical techniques for solving large linear systems of equations.

The Method

We will first explain the CG method before discussing the PCG method. The key to CG is the fact that solving the expression (1) is equivalent to the following minimization problem:

$$\arg\min_{\boldsymbol{x}} J(\boldsymbol{x}) := \frac{1}{2}\boldsymbol{x}^T\mathbf{A}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x},$$

since both give the same solution, which we will denote $\boldsymbol{x}_*$. Using vector derivatives, the gradient of J is equal to

$$\nabla J(\boldsymbol{x}) = \mathbf{A}\boldsymbol{x} - \boldsymbol{b} := \boldsymbol{r}(\boldsymbol{x}),$$

which is also known as the residual of the linear system. The only use of $\boldsymbol{r}$ in this section will refer to the residual as opposed to a vector of distances, as it is elsewhere in the thesis. Since CG is an iterative method, when $\boldsymbol{x} = \boldsymbol{x}_k$, the kth iteration of $\boldsymbol{x}$, the kth iteration of the residual is given by

$$\boldsymbol{r}_k = \mathbf{A}\boldsymbol{x}_k - \boldsymbol{b}. \tag{2}$$

Therefore, the direction of steepest descent of the function $J(\boldsymbol{x})$ at iteration k is $-\nabla J(\boldsymbol{x}) = -\boldsymbol{r}_k$.

Now, the conjugacy in the name of the CG method comes from the fact that it uses the property that two vectors, $\boldsymbol{u}$ and $\boldsymbol{v}$, are called *conjugate* with respect to a SPD matrix $\mathbf{A}$ if $\boldsymbol{u}^T\mathbf{A}\boldsymbol{v} = 0$ for $\boldsymbol{u} \neq \boldsymbol{v}$. Take a set of nonzero, conjugate vectors $\{\boldsymbol{p}_1, \dots, \boldsymbol{p}_n\}$. Since they are conjugate, it is simple to show they are also linearly independent, and hence, form a basis in $\mathbb{R}^n$. This means the solution can be written as a linear combination of these vectors,

$$\boldsymbol{x}_* = \boldsymbol{x}_0 + \sum_{i=1}^{n} \alpha_i \boldsymbol{p}_i.$$

Therefore, if we let

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k, \tag{3}$$

where the subscript denotes iteration number. We will obtain the solution $\boldsymbol{x}_*$ in at most n iterations since it is certain that $\boldsymbol{x}_n = \boldsymbol{x}_*$. Ideally, we will choose these $\boldsymbol{p}_k$ vectors in such a way that we can approximate $\boldsymbol{x}_*$ in fewer than n iterations, which is necessary for large n.

We need an expression for α_k and $\boldsymbol{p}_k$. To determine the α_k values, we will minimize $J(\boldsymbol{x}_{k+1})$ with respect to α_k:

$$\begin{aligned} J(\boldsymbol{x}_{k+1}) = J(\boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k) &= \frac{1}{2}(\boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k)^T\mathbf{A}(\boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k) - \boldsymbol{b}^T(\boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k) \\ &= \frac{1}{2}\boldsymbol{x}_k^T\mathbf{A}\boldsymbol{x}_k + \alpha_k\boldsymbol{x}_k^T\mathbf{A}\boldsymbol{p}_k + \frac{1}{2}\alpha_k^2\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k - \boldsymbol{b}^T\boldsymbol{x}_k - \alpha_k\boldsymbol{b}^T\boldsymbol{p}_k, \end{aligned}$$

and so

$$\frac{d}{d\alpha_k}J(\boldsymbol{x}_{k+1}) = 0 \implies \boldsymbol{x}_k^T\mathbf{A}\boldsymbol{p}_k + \alpha_k\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k - \boldsymbol{b}^T\boldsymbol{p}_k = 0.$$

Solving this for α_k gives

$$\begin{aligned}\alpha_k &= \frac{\boldsymbol{b}^T\boldsymbol{p}_k - \boldsymbol{x}_k^T\mathbf{A}\boldsymbol{p}_k}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k} = \frac{(\boldsymbol{b}^T - \boldsymbol{x}_k^T\mathbf{A})\boldsymbol{p}_k}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k}\\ &= -\frac{\boldsymbol{r}_k^T\boldsymbol{p}_k}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k}. \qquad (4)\end{aligned}$$

Since $\boldsymbol{p}_k$ can be thought of as the step direction, and we showed $-\boldsymbol{r}_k$ is the steepest descent direction, we will update the $\boldsymbol{p}_k$ by using the residuals and the previous $\boldsymbol{p}$ vector:

$$\boldsymbol{p}_{k+1} = -\boldsymbol{r}_{k+1} + \beta_k\boldsymbol{p}_k. \qquad (5)$$

The choice of β_k will be discussed below and $\boldsymbol{r}_{k+1}$ is updated by using (2) and (3). Specifically,

$$\begin{aligned}\boldsymbol{r}_{k+1} &= \mathbf{A}\boldsymbol{x}_{k+1} - \boldsymbol{b} \quad \text{by (2)}\\ &= \mathbf{A}(\boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k) - \boldsymbol{b} \quad \text{by (3)}\\ &= \mathbf{A}\boldsymbol{x}_k - \boldsymbol{b} + \alpha_k\mathbf{A}\boldsymbol{p}_k\\ &= \boldsymbol{r}_k + \alpha_k\mathbf{A}\boldsymbol{p}_k. \qquad (6)\end{aligned}$$

β_k must be chosen so that the conjugacy of the $\boldsymbol{p}_i$ vectors with respect to $\mathbf{A}$ is retained, i.e. $\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_{k+1} = 0$. Left-multiplying (5) by $\boldsymbol{p}_k^T\mathbf{A}$ yields

$$\begin{aligned}\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_{k+1} &= \boldsymbol{p}_k^T\mathbf{A}(-\boldsymbol{r}_{k+1} + \beta_k\boldsymbol{p}_k) = 0\\ \implies &-\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{r}_{k+1} + \beta_k\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k = 0.\end{aligned}$$

Solving this for β_k gives us

$$\beta_k = \frac{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{r}_{k+1}}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k}.$$

To slightly increase the efficiency of this algorithm, we will now make a few adjustments. Firstly, it is possible to show that $\boldsymbol{r}_i^T\boldsymbol{r}_j = 0$ for $i \neq j$ and $\boldsymbol{r}_i^T\boldsymbol{p}_j = 0$ for $i > j$. Using this fact and (5), we can change the expression for α_k in (4) to be

$$\alpha_k = -\frac{\boldsymbol{r}_k^T\boldsymbol{p}_k}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k} = -\frac{\boldsymbol{r}_k^T(-\boldsymbol{r}_k + \beta_k\boldsymbol{p}_{k-1})}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k} = \frac{\boldsymbol{r}_k^T\boldsymbol{r}_k - \beta_k\boldsymbol{r}_k^T\boldsymbol{p}_{k-1}}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k} = \frac{\boldsymbol{r}_k^T\boldsymbol{r}_k}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k}.$$

Now, using the orthogonality of the $\boldsymbol{r}_i$ vectors and the fact in (6) that $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k\mathbf{A}\boldsymbol{p}_k$ and thus $\mathbf{A}\boldsymbol{p}_k = (\boldsymbol{r}_k - \boldsymbol{r}_{k+1})/\alpha_k$, β_k can be written

$$\begin{aligned}\beta_k &= \frac{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{r}_{k+1}}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k} = \frac{(\boldsymbol{r}_k - \boldsymbol{r}_{k+1})^T\boldsymbol{r}_{k+1}/\alpha_k}{\boldsymbol{p}_k^T(\boldsymbol{r}_k - \boldsymbol{r}_{k+1})/\alpha_k} = \frac{\boldsymbol{r}_k^T\boldsymbol{r}_{k+1} - \boldsymbol{r}_{k+1}^T\boldsymbol{r}_{k+1}}{(-\boldsymbol{r}_k + \beta_{k-1}\boldsymbol{p}_{k-1})^T(\boldsymbol{r}_k - \boldsymbol{r}_{k+1})}\\ &= \frac{\boldsymbol{r}_{k+1}^T\boldsymbol{r}_{k+1}}{\boldsymbol{r}_k\boldsymbol{r}_k}.\end{aligned}$$

Algorithm 1 Conjugate Gradient Method

```
0. Set $k = 0$, $\boldsymbol{r}_0 = \mathbf{A}\boldsymbol{x}_0 - \boldsymbol{b}$, and $\boldsymbol{p}_0 = -\boldsymbol{r}$;
1. Set $\alpha_k = \dfrac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \mathbf{A}\boldsymbol{p}_k}$;
2. Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$;
3. Set $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k + \alpha_k \mathbf{A}\boldsymbol{p}_k$;
4. Set $\beta_k = \dfrac{\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{r}_k}$;
5. Set $\boldsymbol{p}_{k+1} = -\boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$;
6. Return to step 1 and repeat until $\boldsymbol{r}_{k+1}$ is sufficiently small.
```

Now that we have more economical representations of α_k and β_k, we can write the CG algorithm in full as Algorithm 1.

We have stated that it will take at most n iterations for CG to reach the solution, $\boldsymbol{x}_*$. While that is true, it often reaches an acceptable solution much more quickly. For instance, if $\mathbf{A}$ has r distinct eigenvectors, the CG algorithm will obtain the solution in at most r iterations. Additionally, there are two error bounds that can be useful in assessing convergence. If we denote the eigenvalues of $\mathbf{A}$ as $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$, then the following two bounds hold:

$$(1) \quad (\boldsymbol{x}_{k+1} - \boldsymbol{x}_*)^T \mathbf{A} (\boldsymbol{x}_{k+1} - \boldsymbol{x}_*) \leq \left(\frac{\lambda_{n-k} - \lambda_1}{\lambda_{n-k} + \lambda_1} \right)^2 (\boldsymbol{x}_0 - \boldsymbol{x}_*)^T \mathbf{A} (\boldsymbol{x}_0 - \boldsymbol{x}_*)$$

$$(2) \quad (\boldsymbol{x}_{k+1} - \boldsymbol{x}_*)^T \mathbf{A} (\boldsymbol{x}_{k+1} - \boldsymbol{x}_*) \leq \left(\frac{\sqrt{\lambda_n/\lambda_1} - 1}{\sqrt{\lambda_n/\lambda_1} + 1} \right)^k (\boldsymbol{x}_0 - \boldsymbol{x}_*)^T \mathbf{A} (\boldsymbol{x}_0 - \boldsymbol{x}_*).$$

Therefore, it is clear that the speed of the CG convergence is dependent on the eigenvalues of $\mathbf{A}$.

Preconditioning

Since the eigenvalues of $\mathbf{A}$ are important in the convergence speed of the CG algorithm, we can increase the convergence rate by using a preconditioner matrix, $\mathbf{G} = \mathbf{E}\mathbf{E}^T$, so that the CG depends on the eigenvalues of $\mathbf{E}^{-T}\mathbf{A}\mathbf{E}^{-1}$ instead of $\mathbf{A}$. Using a preconditioner will yield the preconditioned conjugate gradient (PCG) method. The best preconditioner is one in which $\mathbf{E}^{-T}\mathbf{A}\mathbf{E}^{-1} \approx \mathbf{I}$, the identity matrix, and $\mathbf{G}$ is efficient to invert. Using a preconditioner is equivalent to making the change of variables $\boldsymbol{y} = \mathbf{E}\boldsymbol{x}$ and then solving the linear system

$$(\mathbf{E}^{-T}\mathbf{A}\mathbf{E}^{-1})\boldsymbol{y} = \mathbf{E}^{-T}\boldsymbol{b},$$

which is equivalent to (1). The resulting PCG algorithm is given as Algorithm 2.

Note that $\mathbf{E}$ is not used in the implementation of PCG; $\mathbf{G}$ is the only new matrix needed. Although each iteration of PCG is more costly than each iteration of CG, the total number of iterations needed for PCG is significantly lower with a good choice of a preconditioner. PCG is the method we will use in this thesis to solve for our MAP estimator.

An illustration of the CG and PCG methods is presented in Figure 1. The left side shows the way $\boldsymbol{x}_k$ converges to the solution in a small-scale example when using CG. For this example,

$$\mathbf{A} = \begin{bmatrix} 3.5 & -1 \\ -1 & 1.5 \end{bmatrix} \text{ and } \boldsymbol{b} = \begin{bmatrix} 2.75 \\ 2.25 \end{bmatrix}, \text{ which means } \boldsymbol{x}_* = \begin{bmatrix} 1.5 \\ 2.5 \end{bmatrix}.$$

Algorithm 2 Preconditioned Conjugate Gradient Method

```
0. Set $k = 0$, $\boldsymbol{r}_0 = \mathbf{A}\boldsymbol{x}_0 - \boldsymbol{b}$, $\boldsymbol{z}_0 = \mathbf{G}^{-1}\boldsymbol{r}_0$, and $\boldsymbol{p}_0 = -\boldsymbol{z}_0$;
1. Set $\alpha_k = \dfrac{\boldsymbol{r}_k^T\boldsymbol{r}_k}{\boldsymbol{p}_k^T\mathbf{A}\boldsymbol{p}_k}$;
2. Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$;
3. Set $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k + \alpha_k\mathbf{A}\boldsymbol{p}_k$;
4. Set $\boldsymbol{z}_{k+1} = \mathbf{G}^{-1}\boldsymbol{r}_{k+1}$;
5. Set $\beta_k = \dfrac{\boldsymbol{z}_{k+1}^T\boldsymbol{r}_{k+1}}{\boldsymbol{z}_k^T\boldsymbol{r}_k}$;
6. Set $\boldsymbol{p}_{k+1} = -\boldsymbol{z}_{k+1} + \beta_k\boldsymbol{p}_k$;
7. Return to step 1 and repeat until $\boldsymbol{r}_{k+1}$ is sufficiently small.
```

Beginning with $\boldsymbol{x}_0 = (0,0)^T$, the first iteration of CG yields $\boldsymbol{x}_1 = (1.60, 1.30)^T$ and the second iteration $\boldsymbol{x}_2 = \boldsymbol{x}_*$ arrives at the true solution, as it should since $n = 2$. The advantage PCG has over CG is illustrated on the right side of Figure 1. We now use the preconditioner

$$\mathbf{G} = \begin{bmatrix} 3.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

that has just the diagonal entries of $\mathbf{A}$ and is therefore easier to invert. This time, $\boldsymbol{x}_1 = (1.37, 2.61)^T$, which is substantially closer to $\boldsymbol{x}_*$ than $\boldsymbol{x}_1$ was in the CG case and $\boldsymbol{x}_2 = \boldsymbol{x}_* = (1.5, 2.5)^T$. For this example, both the CG and PCG converged to the true solution in two iterations, but it is clear that PCG has the potential to converge in fewer iterations for a problem with much larger n.

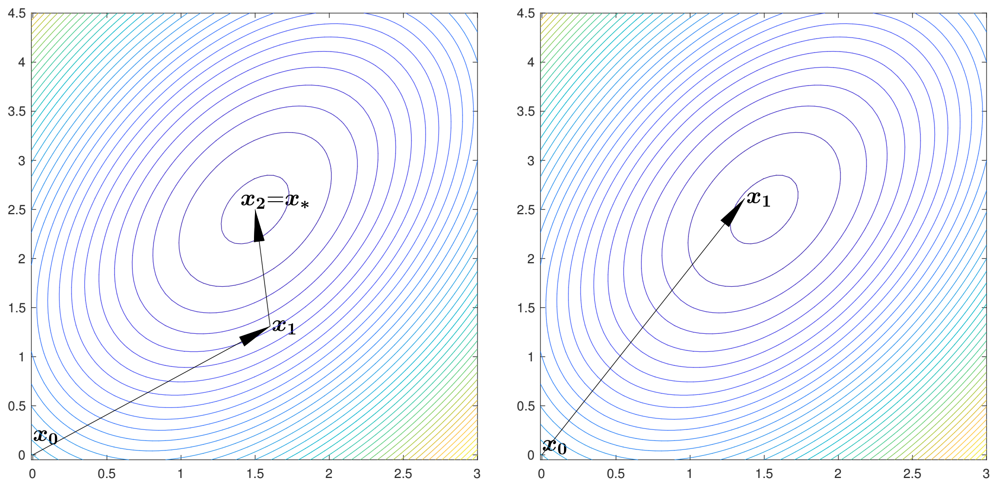


Figure 1: Conjugate gradient methods. The image on the left tracks the convergence of the CG algorithm and the image on the right shows PCG. Both methods converged in two iterations.

Session Info

```r
sessionInfo()
```

```
## R version 4.3.3 (2024-02-29)
## Platform: aarch64-apple-darwin20 (64-bit)
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3.11.0
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] lubridate_1.9.3 forcats_1.0.0   stringr_1.5.1   dplyr_1.1.4
##  [5] purrr_1.0.2     readr_2.1.5     tidyr_1.3.1     tibble_3.2.1
##  [9] tidyverse_2.0.0 caret_6.0-94    lattice_0.22-5  ggplot2_3.5.0
##
## loaded via a namespace (and not attached):
##  [1] gtable_0.3.4         xfun_0.42            recipes_1.0.9
##  [4] tzdb_0.4.0           vctrs_0.6.5          tools_4.3.3
##  [7] generics_0.1.3       stats4_4.3.3         parallel_4.3.3
## [10] fansi_1.0.6          pkgconfig_2.0.3      ModelMetrics_1.2.2.2
## [13] Matrix_1.6-5         data.table_1.15.0    lifecycle_1.0.4
## [16] compiler_4.3.3       munsell_0.5.0        codetools_0.2-19
## [19] htmltools_0.5.7      class_7.3-22         yaml_2.3.8
## [22] prodlim_2023.08.28   pillar_1.9.0         MASS_7.3-60.0.1
## [25] gower_1.0.1          iterators_1.0.14     rpart_4.1.23
## [28] foreach_1.5.2        nlme_3.1-164         parallelly_1.36.0
## [31] lava_1.7.3           tidyselect_1.2.0     digest_0.6.34
## [34] stringi_1.8.3        future_1.33.1        reshape2_1.4.4
## [37] listenv_0.9.0        splines_4.3.3        fastmap_1.1.1
## [40] grid_4.3.3           colorspace_2.1-0     cli_3.6.2
## [43] magrittr_2.0.3       survival_3.5-8       utf8_1.2.4
## [46] future.apply_1.11.1  withr_3.0.0          scales_1.3.0
## [49] timechange_0.3.0     rmarkdown_2.25       globals_0.16.2
## [52] nnet_7.3-19          timeDate_4032.109    hms_1.1.3
## [55] evaluate_0.23        knitr_1.45           hardhat_1.3.0
## [58] rlang_1.1.3          Rcpp_1.0.12          glue_1.7.0
## [61] pROC_1.18.5          ipred_0.9-14         rstudioapi_0.15.0
## [64] R6_2.5.1             plyr_1.8.9
```