Relational Event Model 2.0

REMs with millions of events

Edoardo Filippi-Mazzola

2024-06-24

```
if (!require(torch)) install.packages('torch')
```

```
## Loading required package: torch
```

```
library(torch)
```

Implementing STREAM

The implementation of the STREAM model is applied to the **Patent Citation Network**, a vast and dynamic network that encompasses more than 7 million US patents, and almost 100 millions citations. In this network, each newly published patent forms connections, or ties, with pre-existing nodes (other patents).

The data provided contains the following features:

- Receiver publication year: the year when the receiving patent was published.
- Time lag: duration between the publication dates of the citing (sender) and cited (receiver) patents.
- Textual similarity: cosine similarity between the embedded abstracts of the sender and receiver patents.
- Jaccard similarity: measure of shared classes between the sender and receiver patents.
- Cumulative citations received (preferential attachment): total number of citations a patent has received up to a given point
- Time from previous citation/event: interval since the last citation/event associated with a receiver patent.
- Receiver outdegree: cumulative number of citations a patent has received.

For the purpose of this example, we are going to use a reduced form of the Network, containing only 100,000 events/citations.

The goal is to estimate the weights of the splines, denoted as θ by maximizing the likelihood function $L_P(\theta)$ using the ADAM optimization algorithm.

The likelihood function is given by:

$$L_P(\theta) = \prod_{i=1}^{n} \left[1 + \exp\left\{ -\sum_{k=1}^{d}\sum_{j=1}^{q} \theta_{jk} \left[B_{j,p}^{k}(x_{s_i r_i k}) - B_{j,p}^{k}(x_{s_i^* r_i^* k}) \right] \right\} \right]^{-1}$$

- $B_{j,p}^{k}$ represents the jth basis transformation of order p, for $j = 1, \ldots, q$ degrees of freedom.
- $x_{s_i r_i k}$ and $x_{s_i^* r_i^* k}$ are the input variables associated with the sender and receiver in the network, respectively.
- θ_{jk} are the spline weights that we aim to estimate.

Phase 1 - Basis transoformation

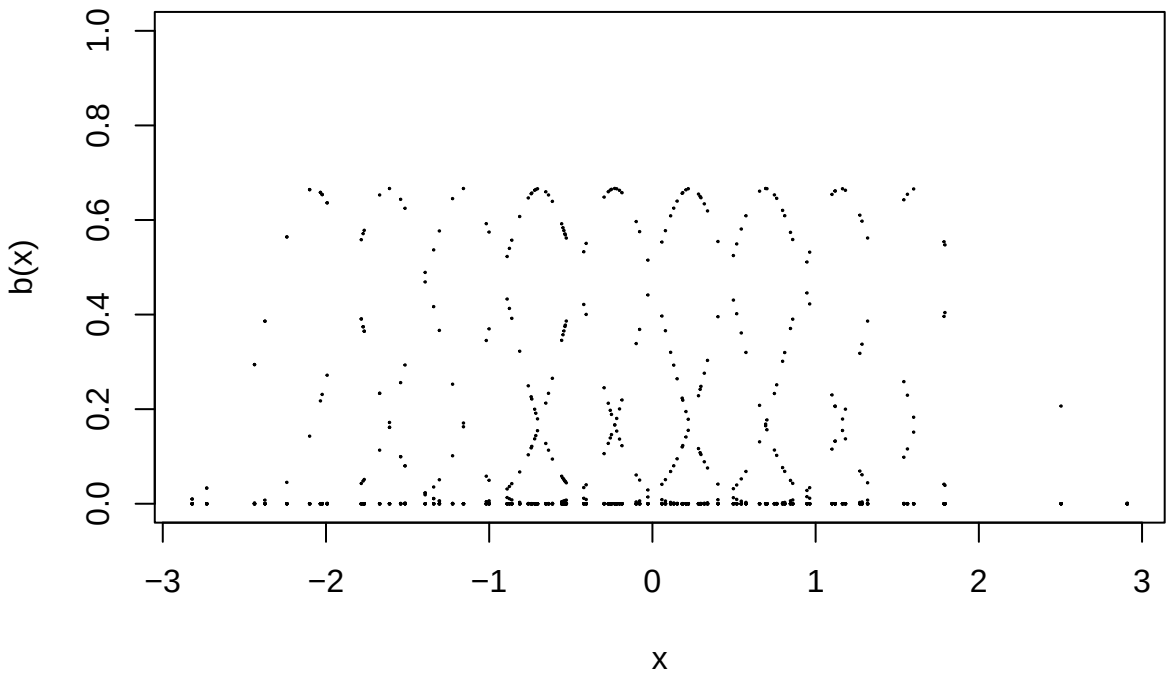
```
bspline <- function(x, k, i, m = 2) {
  # Evaluate the ith B-spline basis function of order m at the values in x
  # given knot locations in k

  if (m == -1) {
    # Base case of the recursion: when m is -1, we are at the lowest order basis function

    res <- as.numeric(x < k[i + 1] & x >= k[i])
    # Returns 1 if x is within the interval [k[i], k[i+1])
  } else {

     # Recursive case: construct the basis function from lower order basis functions
    # Calculate the first term's coefficient
    z0 <- (x - k[i]) / (k[i + m + 1] - k[i])
    # Calculate the second term's coefficient
    z1 <- (k[i + m + 2] - x) / (k[i + m + 2] - k[i + 1])

    # Recursive calls to the lower order basis functions
    res <- z0 * bspline(x, k, i, m - 1) + z1 * bspline(x, k, i + 1, m - 1)
  }

  return(res)  # Return the evaluated B-spline basis function
}
```

Generate a basis function for a given x.

```
library(torch)

stream_bs <- function(x, df, lower_bound, upper_bound) {
    # Set the order of the B-spline
    m = 2

    # Calculate the number of knots needed for the B-spline basis
    n_knots = df + (m + 1) + 1

    # Generate a sequence of equally spaced knots from lower_bound to upper_bound
    knots = seq(from = lower_bound, to = upper_bound, length.out = n_knots)

    # Initialize a matrix to store the B-spline basis functions
    # The number of rows is the length of x, and the number of columns is df
    basis = torch_zeros(c(length(x), df))

    # Loop over each degree of freedom to calculate the corresponding basis function
    for (i in 1:df) {
        # Compute the i-th B-spline basis function evaluated at x
        basis[, i] = bspline(x, k = knots, i = i)
    }
    return(basis)
}
```

Example of usage

```
x <- torch_tensor(rnorm(100)); lwb <- -3; upb <- 3
bx <- stream_bs(x,df=10,lower_bound = lwb,upper_bound = upb)
plot(y=bx[,1],x=x,ylim=c(0,1),ylab='b(x)',xlab='x',cex=.1)
for (i in 1:10){points(y=bx[,i],x=x,cex=.1)}
```



Phase 2 - Difference between basis for events and non-events

The STREAM routine takes advantage of the computational benefits of the ADAM optimization technique by generating basis functions directly from the observed batches. By knowing the upper and lower bounds of a given covariate in advance (and thus knowing where to place the knots), we need functions that:

1. Generate basis functions for events and non-events.
2. Calculate the difference between the two basis functions.

$$L_P(\theta) = \prod_{i=1}^{n} \left[1 + \exp\left\{ -\sum_{k=1}^{q} \sum_{j=1}^{d} \theta_{jk} \left[\mathbf{B}^{\mathbf{k}}_{\mathbf{j,p}}(\mathbf{x}_{\mathbf{s_i r_i k}}) - \mathbf{B}^{\mathbf{k}}_{\mathbf{j,p}}(\mathbf{x}_{\mathbf{s_i^* r_i^* k}}) \right] \right\} \right]^{-1}$$

```
delta_bs = function(events, non_events, df, lower_bounds, upper_bounds) {

    n_covariates = ncol(events)
    model_mat = NULL

    # Loop over each column
    for (i in 1:n_covariates) {

        ev_tmp = events[, i]; non_ev_tmp = non_events[, i]
        lb = lower_bounds[i]; ub = upper_bounds[i]

        #STEP 1:  Generate B-spline basis for the events and non-events
        basis_ev = stream_bs(x=ev_tmp,df=df,lower_bound=lb, upper_bound=ub)
        basis_non_ev = stream_bs(x=non_ev_tmp,df= df,lower_bound=lb,upper_bound=ub)
```

```
        #STEP 2:  Compute the difference between bases
        model_mat_tmp = basis_ev - basis_non_ev

        # Concatenate the computed basis difference to the model matrix
        if (is.null(model_mat)) {
            model_mat <- model_mat_tmp
        } else {
        model_mat <- torch_cat(list(model_mat, model_mat_tmp), dim = 2)
    }
}
return(model_mat)
}
```

Phase 3 - Define the model in torch

The model and its estimation are built around `torch`, the R-version of PyTorch, a powerful library for deep learning and tensor computation. This setup enables efficient implementation and complex models.

Steps:

1. **Define a `nn_module`**: This step involves creating a custom model by subclassing `nn_module`. Within this module, you describe the *forward passage* of the data into the model, specifying how input data flows through the network layers and how it transforms at each layer.
2. **Initialize the model**: After defining the architecture, you initialize the model by creating an instance of the custom `nn_module`. This step sets up with initial weights and configurations.
3. **Define a loss (likelihood) function**: define the function that the model will use to estimate its parameters. In our case, REM's partial likelihood function.
4. **Define the optimizer with the learning rate**: Set up an optimizer, such as ADAM, which will adjust the model's weights based on the gradients computed during backpropagation.

Step 1:define the *forward passage*

```
model <- nn_module(
  initialize = function(input_dim) {
    self$linear <- nn_linear(in_features = input_dim, out_features = 1,bias = F)
  },
  forward = function(x) {
    x %>% self$linear() %>% torch_sigmoid()
  }
)
```

Step 2: initialize the model

```
# Degrees of freedom for B-splines
df <- 10
# Get the number of event features (columns in the events matrix)
n_covariates <- 2 #ncol(events)

# Initialize the model by setting input size (number of parameters)
# as df * n_covariates
logistic_model <- model(input_dim = df * n_covariates)
```

Step 3: define the *loss function*: binary cross-entropy → negative log-likelihood of logistic regression

$$-\log(L_P) = -\sum_{i=1}^{n_b}\left[\log\left(\frac{e^{\sum_{k=1}^{d}\sum_{j=1}^{q}\theta_{jk}\left[B_{j,p}^{k}(x_{s_i r_i k})-B_{j,p}^{k}(x_{s_i^* r_i^* k})\right]}}{1+e^{\sum_{k=1}^{q}\sum_{j=1}^{d}\theta_{jk}\left[B_{j,p}^{k}(x_{s_i r_i k})-B_{j,p}^{k}(x_{s_i^* r_i^* k})\right]}}\right)\right]$$

```
criterion <- nn_bce_loss()
```

Step 4: define the optimizer for the model and the learning rate.

```
# Define the optimizer with the parameters of the model and a learning rate
optimizer <- optim_adam(logistic_model$parameters, lr = 0.1)
```

Phase 4 - Implementing STREAM

Demo 1

Load the data, shuffle the order of the observations and select the covariates you want to fit. Call `events` the matrix containing your event covariates, and `non_events` the second matrix.

```
library(torch)
dat_original <- read.csv('00-Data/patents_sample_100k.csv')

idx_shuf <- sample(1:nrow(dat_original),size=nrow(dat_original),replace = F)
dat <- dat_original[idx_shuf,]

events <- cbind(dat$rec_pub_year,dat$lag)
non_events <- cbind(dat$rec_pub_year_2,dat$lag_2)
```

```
n_covariates <- ncol(events)
logistic_model <- model(input_dim = df * n_covariates)
optimizer <- optim_adam(logistic_model$parameters, lr = 0.1)
```

Demo 2

Store in two R-vectors the lower and the upper bound of each effect.

```
n_events <- ncol(events)
lower_bounds <- NULL
upper_bounds <- NULL
for (i in 1:n_events){
  lower_bounds <- c(lower_bounds,min(events[,i]))
  upper_bounds <- c(upper_bounds,max(events[,i]))
  }
```

Demo 3

Split the data into train and test with a 80-20 proportion. Transform each new element in a tensor using `torch_tensor(...,dtype=torch.float32())`.

Note: make sure that the input for `torch_tensor` is an R-matrix (i.e. `as.matrix(...)`).

```
#Train-test split
events_train <- events[1:round(0.8*nrow(events)),]
events_test <- events[round(0.8*nrow(events)):nrow(events),]

non_events_train <- non_events[1:round(0.8*nrow(events)),]
non_events_test <-  non_events[round(0.8*nrow(events)):nrow(events),]
```

```
#Tensor conversion
events_train <- torch_tensor(as.matrix(events_train),dtype=torch_float32())
events_test <- torch_tensor(as.matrix(events_test),dtype=torch_float32())

x_events_test <- torch_tensor(as.matrix(events_test),
                              dtype = torch_float32())
x_non_events_test <- torch_tensor(as.matrix(non_events_test),
                                  dtype=torch_float32())
```

Demo 4

Divide the event set and the non-event set into batches. You can choose the size of the batches (16,32,64,128,256,...). Use `torch_split(...,split_size=....)`.

Note: bigger batches will make the algorithm work slower. Smaller batches could cause unstable gradients.

```
batch_size <- 128
x_events <- torch_split(events_train,split_size = batch_size)
x_non_events <- torch_split(non_events_train,split_size = batch_size)
```

Models defined by `torch` can operate in either a training state or a validation state.

- **Training:** the model is actively learning from the data. This involves adjusting its weights based on the input data and the loss function to minimize errors. The training state can be set using the command `logistic_model$train()`.
- **Validation/Test:** model is used to evaluate its performance on a separate set of data that it has not seen during training. The validation state can be set using the command `logistic_model$eval()`.

Demo 5

Next, we need to write the optimization routine. `torch` allows for customization in this process, thus allowing to generate the basis directly on the batches. We will initialize a nested for-loop structure: the outer loop will iterate over each epoch, while the inner loop will iterate over each batch. Start by setting your model to training mode. Reset the gradients computed by the `optimizer` to zero using `optimizer$zero_grad()`. Set the maximum number of epochs to 10.

```
num_epochs <- 10
for (epoch in 1:num_epochs) {
  for (batch in 1:n_batches) {
    logistic_model$train()
    optimizer$zero_grad()

    ...
}
```

Demo 6

Inside the nested for loop, take the first batch from the splitted event set and non event set. Give them as input to the `delta_bs(...)` function we defind earlier. This is going to be the input of your model. Set 10 degrees of freedom for the basis functions.

```
for (epoch in 1:num_epochs) {
  for (batch in 1:n_batches) {
    ...
```

```
    x_tmp_ev <- x_events[[batch]]
        x_tmp_nv <- x_non_events[[batch]]

        input <- delta_bs(events = x_tmp_ev,
                          non_events = x_tmp_nv,
                                          df = 10,
                                          lower_bounds = lower_bounds,
                                          upper_bounds = upper_bounds)

        ...
            }
```

Demo 7

We will now have to compute the "forward pass" and the "backword pass".

The forward pass is the process of feeding the input data through the model. The backward pass is also known as backpropagation. It consists in the process of calculating the gradients from the likelihood with respect to the model's parameters. These gradients indicate the direction and magnitude of changes needed for each parameter to minimize the loss. Finally, an optimizer (ADAM in this case) uses these gradients to update the model's parameters, adjusting them slightly in the direction maximizes the likelihood.

1. give your `input` to the model (`logistic_model`)
2. generate a tensor of 1s, using `torch_ones(nrow(input),dtype=torch_float32())`,
3. compute the loss using the tensor of ones and the output of your model
4. compute the gradient using "your loss"`$backward()`
5. update the weights of the model using `optimizer$step()`

```
for (epoch in 1:num_epochs) {
  for (batch in 1:n_batches) {
    ...

    # Forward pass
    outputs <- logistic_model(input)
    y <- torch_ones(nrow(input),dtype=torch_float32())
    loss <- criterion(outputs, y)

    # Backward pass and optimization
    loss$backward()
    optimizer$step()

    ...
            }
```

Demo 8

Compute the validation step.

1. set the model to validation using the `eval` command from demo 4.
2. repeat the same steps from demo 5 and 6, without doinf the backward and optimization steps-
3. print the current "train loss" and "validation loss" at each step.

```
for (epoch in 1:num_epochs) {
  for (batch in 1:n_batches) {

    ...

    #Validation step
    logistic_model$eval()
      input <- delta_bs(events = x_events_test, non_events = x_non_events_test,
                        df = 10,
                        lower_bounds = lower_bounds,
                        upper_bounds = upper_bounds)

      val_out <- logistic_model(input)
      y <- torch_ones(c(nrow(input)),dtype=torch_float32())

      val_loss <- criterion(val_out,y)

      #Printing the current step information
      cat(sprintf("Epoch: [%d/%d] | Batch: [%d/%d] | Loss: %.4f\n | Val. Loss: %.4f\n",
                  epoch,num_epochs,batch,n_batches,loss$item(),val_loss$item()))

      ...
      }
  }
```

Demo 9

Implement an *early stopping* criterion.

The *early stopping* criterion monitors the validation loss and stops the training if the validation loss does not improve for a specified number of epochs (patience).

```
patience <- 10  # Number of epochs to wait before stopping
best_val_loss <- Inf  # Initialize the best validation loss to infinity
epochs_no_improve <- 0  # Counter for epochs with no improvement

for (epoch in 1:num_epochs) {
  for (batch in 1:n_batches) {
    ...

        # Check if validation loss has improved
        if (val_loss$item() < best_val_loss) {
          best_val_loss <- val_loss$item()  # Update the best validation loss
          epochs_no_improve <- 0  # Reset the counter
          } else {
            epochs_no_improve <- epochs_no_improve + 1  # Increment the counter
            }
    # Check if early stopping criterion is met
    if (epochs_no_improve >= patience) {
      cat("Early stopping triggered. Training stopped.\n")
      break
    }
  }
  if (stop_routine){
    break
```

```
  }
}
```

Full routine

```
n_batches <- length(x_events)

batch_size <- 128
num_epochs <- 10

patience <- 10  # Number of epochs to wait before stopping
best_val_loss <- Inf  # Initialize the best validation loss to infinity
epochs_no_improve <- 0  # Counter for epochs with no improvement
stop_routine <- FALSE

for (epoch in 1:num_epochs) {
  for (batch in 1:n_batches) {
    logistic_model$train()
    optimizer$zero_grad()

    x_tmp_ev = x_events[[batch]]
    x_tmp_nv = x_non_events[[batch]]

    input = delta_bs(events = x_tmp_ev,
                     non_events = x_tmp_nv,
                     df = 10,
                     lower_bounds = lower_bounds,
                     upper_bounds = upper_bounds)

    # Forward pass
    outputs <- logistic_model(input)
    y <- torch_ones(c(nrow(input)),dtype=torch_float32())
    loss <- criterion(outputs, y)

    # Backward pass and optimization
    loss$backward()
    optimizer$step()

    #Validation step
    logistic_model$eval()

    input <- delta_bs(events = x_events_test,
                     non_events = x_non_events_test,
                     df = 10,
                     lower_bounds = lower_bounds,
                     upper_bounds = upper_bounds)
    val_out <- logistic_model(input)

    y <- torch_ones(c(nrow(input)),dtype=torch_float32())
    val_loss <- criterion(val_out,y)

    cat(sprintf("Epoch: [%d/%d] | Batch: [%d/%d] | Loss: %.4f\n | Val. Loss: %.4f\n",
                epoch,num_epochs,batch,n_batches,loss$item(),val_loss$item()))
```

```
    # Check if validation loss has improved
    if (val_loss$item() < best_val_loss) {
      best_val_loss <- val_loss$item()  # Update the best validation loss
      epochs_no_improve <- 0  # Reset the counter
    } else {
      epochs_no_improve <- epochs_no_improve + 1  # Increment the counter
    }

    # Check if early stopping criterion is met
    if (epochs_no_improve >= patience) {
      cat("Early stopping triggered. Training stopped.\n")
      stop_routine <- TRUE
      break

    }
  }
  if (stop_routine){
    break
  }
}
```

```
## Epoch: [1/10] | Batch: [1/625] | Loss: 0.7022
##  | Val. Loss: 0.6644
## Epoch: [1/10] | Batch: [2/625] | Loss: 0.6629
##  | Val. Loss: 0.6383
## Epoch: [1/10] | Batch: [3/625] | Loss: 0.6498
##  | Val. Loss: 0.6202
## Epoch: [1/10] | Batch: [4/625] | Loss: 0.6295
##  | Val. Loss: 0.6089
## Epoch: [1/10] | Batch: [5/625] | Loss: 0.6272
##  | Val. Loss: 0.6017
## Epoch: [1/10] | Batch: [6/625] | Loss: 0.5610
##  | Val. Loss: 0.5965
## Epoch: [1/10] | Batch: [7/625] | Loss: 0.6269
##  | Val. Loss: 0.5929
## Epoch: [1/10] | Batch: [8/625] | Loss: 0.6167
##  | Val. Loss: 0.5901
## Epoch: [1/10] | Batch: [9/625] | Loss: 0.6203
##  | Val. Loss: 0.5878
## Epoch: [1/10] | Batch: [10/625] | Loss: 0.5507
##  | Val. Loss: 0.5862
## Epoch: [1/10] | Batch: [11/625] | Loss: 0.5768
##  | Val. Loss: 0.5851
## Epoch: [1/10] | Batch: [12/625] | Loss: 0.6176
##  | Val. Loss: 0.5841
## Epoch: [1/10] | Batch: [13/625] | Loss: 0.5805
##  | Val. Loss: 0.5832
## Epoch: [1/10] | Batch: [14/625] | Loss: 0.6556
##  | Val. Loss: 0.5819
## Epoch: [1/10] | Batch: [15/625] | Loss: 0.6079
##  | Val. Loss: 0.5807
## Epoch: [1/10] | Batch: [16/625] | Loss: 0.6071
##  | Val. Loss: 0.5799
```

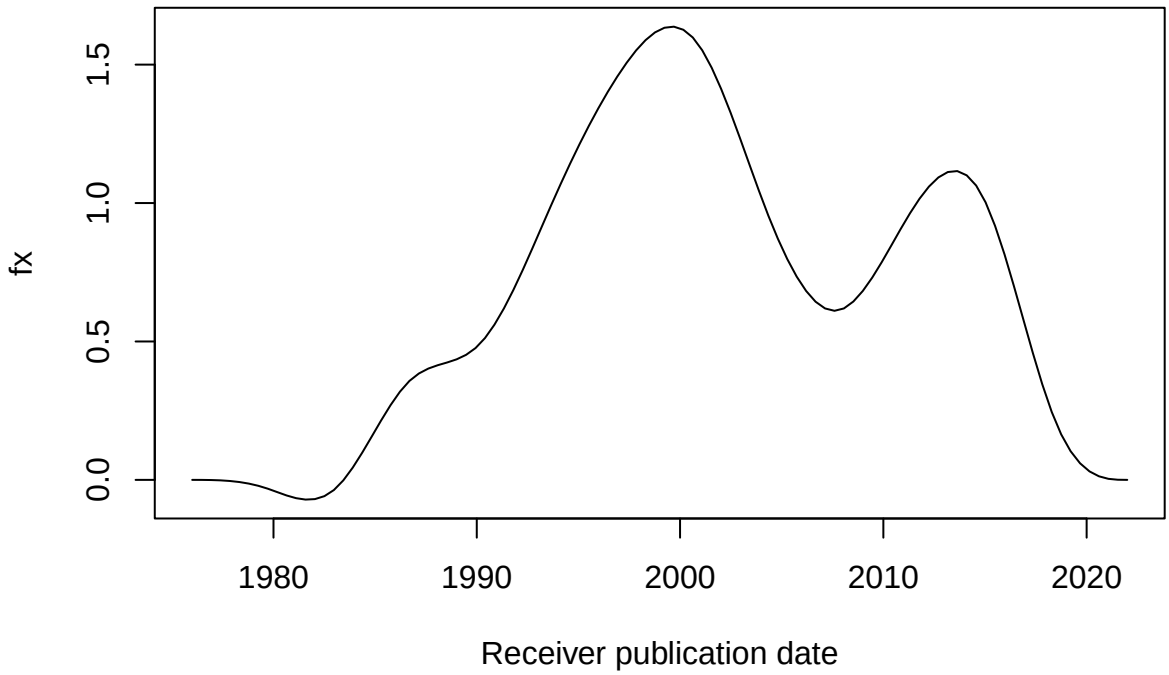
```
## Epoch: [1/10] | Batch: [17/625] | Loss: 0.4682
##  | Val. Loss: 0.5800
## Epoch: [1/10] | Batch: [18/625] | Loss: 0.5832
##  | Val. Loss: 0.5799
## Epoch: [1/10] | Batch: [19/625] | Loss: 0.6497
##  | Val. Loss: 0.5795
## Epoch: [1/10] | Batch: [20/625] | Loss: 0.6505
##  | Val. Loss: 0.5786
## Epoch: [1/10] | Batch: [21/625] | Loss: 0.6134
##  | Val. Loss: 0.5778
## Epoch: [1/10] | Batch: [22/625] | Loss: 0.5503
##  | Val. Loss: 0.5769
## Epoch: [1/10] | Batch: [23/625] | Loss: 0.5866
##  | Val. Loss: 0.5759
## Epoch: [1/10] | Batch: [24/625] | Loss: 0.6447
##  | Val. Loss: 0.5752
## Epoch: [1/10] | Batch: [25/625] | Loss: 0.6485
##  | Val. Loss: 0.5745
## Epoch: [1/10] | Batch: [26/625] | Loss: 0.6136
##  | Val. Loss: 0.5739
## Epoch: [1/10] | Batch: [27/625] | Loss: 0.6107
##  | Val. Loss: 0.5734
## Epoch: [1/10] | Batch: [28/625] | Loss: 0.5823
##  | Val. Loss: 0.5734
## Epoch: [1/10] | Batch: [29/625] | Loss: 0.5884
##  | Val. Loss: 0.5731
## Epoch: [1/10] | Batch: [30/625] | Loss: 0.5639
##  | Val. Loss: 0.5727
## Epoch: [1/10] | Batch: [31/625] | Loss: 0.6565
##  | Val. Loss: 0.5729
## Epoch: [1/10] | Batch: [32/625] | Loss: 0.6017
##  | Val. Loss: 0.5727
## Epoch: [1/10] | Batch: [33/625] | Loss: 0.6316
##  | Val. Loss: 0.5725
## Epoch: [1/10] | Batch: [34/625] | Loss: 0.5847
##  | Val. Loss: 0.5725
## Epoch: [1/10] | Batch: [35/625] | Loss: 0.5838
##  | Val. Loss: 0.5722
## Epoch: [1/10] | Batch: [36/625] | Loss: 0.6012
##  | Val. Loss: 0.5720
## Epoch: [1/10] | Batch: [37/625] | Loss: 0.6218
##  | Val. Loss: 0.5718
## Epoch: [1/10] | Batch: [38/625] | Loss: 0.5587
##  | Val. Loss: 0.5713
## Epoch: [1/10] | Batch: [39/625] | Loss: 0.6081
##  | Val. Loss: 0.5710
## Epoch: [1/10] | Batch: [40/625] | Loss: 0.5928
##  | Val. Loss: 0.5707
## Epoch: [1/10] | Batch: [41/625] | Loss: 0.5477
##  | Val. Loss: 0.5699
## Epoch: [1/10] | Batch: [42/625] | Loss: 0.5785
##  | Val. Loss: 0.5691
## Epoch: [1/10] | Batch: [43/625] | Loss: 0.6063
##  | Val. Loss: 0.5684
```

```
## Epoch: [1/10] | Batch: [44/625] | Loss: 0.5428
##  | Val. Loss: 0.5677
## Epoch: [1/10] | Batch: [45/625] | Loss: 0.5089
##  | Val. Loss: 0.5672
## Epoch: [1/10] | Batch: [46/625] | Loss: 0.4880
##  | Val. Loss: 0.5669
## Epoch: [1/10] | Batch: [47/625] | Loss: 0.5931
##  | Val. Loss: 0.5670
## Epoch: [1/10] | Batch: [48/625] | Loss: 0.5869
##  | Val. Loss: 0.5673
## Epoch: [1/10] | Batch: [49/625] | Loss: 0.5958
##  | Val. Loss: 0.5676
## Epoch: [1/10] | Batch: [50/625] | Loss: 0.6229
##  | Val. Loss: 0.5677
## Epoch: [1/10] | Batch: [51/625] | Loss: 0.5536
##  | Val. Loss: 0.5678
## Epoch: [1/10] | Batch: [52/625] | Loss: 0.5095
##  | Val. Loss: 0.5681
## Epoch: [1/10] | Batch: [53/625] | Loss: 0.5637
##  | Val. Loss: 0.5684
## Epoch: [1/10] | Batch: [54/625] | Loss: 0.5279
##  | Val. Loss: 0.5687
## Epoch: [1/10] | Batch: [55/625] | Loss: 0.5803
##  | Val. Loss: 0.5691
## Epoch: [1/10] | Batch: [56/625] | Loss: 0.5730
##  | Val. Loss: 0.5692
## Early stopping triggered. Training stopped.
```

Demo 10

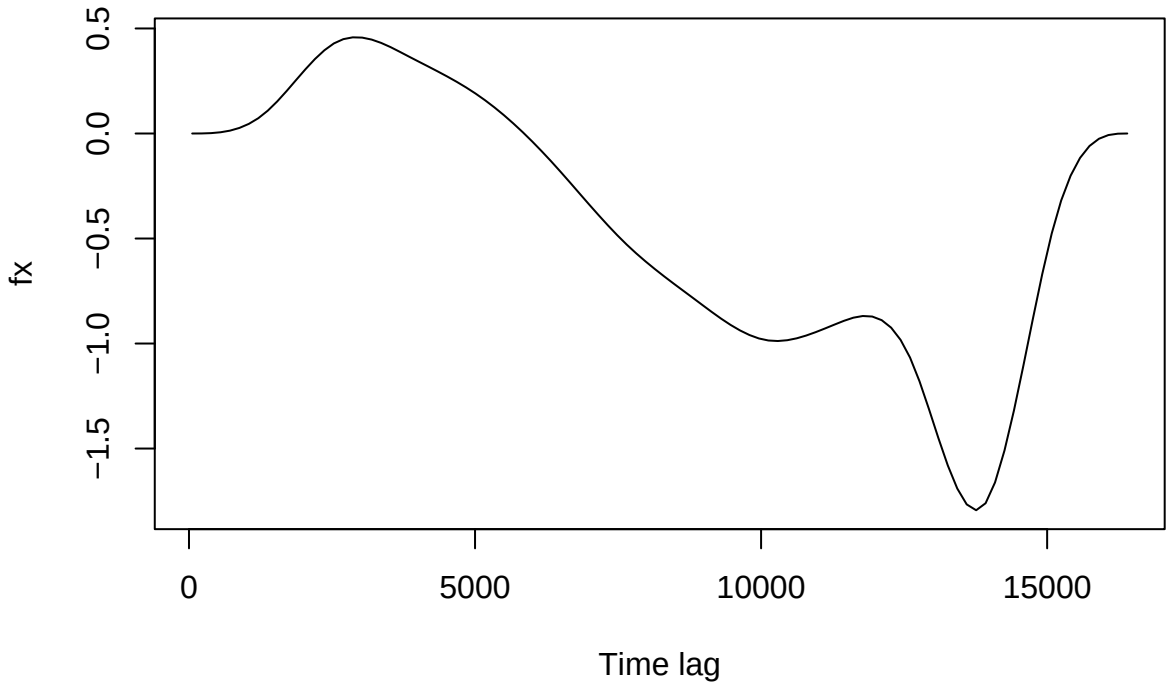
Plot the effects that have been estimated.

```r
#Extract the weights for the first covariate
w <- as.matrix(logistic_model$linear$weight[,1:df])

#Generate a reference vector
test <- seq(from =lower_bounds[1],to = upper_bounds[1],length.out = 100)

#Apply the basis transformation
bs <- stream_bs(x=test,
                            df = 10,
                            lower_bound = lower_bounds[1],
                            upper_bound = upper_bounds[1])

bs <- as.matrix(bs)

#compute the effect
fx <- bs%*%t(w)

plot(x=test,y=fx,type='l',xlab='Receiver publication date')
```



From the plot, we can observe the relationship between the publication year of patents and their propensity to be cited. The plot indicates that patents published around the years 2000 and 2015 have a higher likelihood of being cited compared to patents from other years. This trend suggests that patents from these periods might contain innovations or technological advancements that are particularly influential or relevant.

```
# Plot the time-lag spline
w = as.matrix(logistic_model$linear$weight[,(df+1):(df*2)])

test = seq(from =lower_bounds[2],
           to = upper_bounds[2],length.out = 100)

bs = stream_bs(x=torch_tensor(test),
               df=10,
               lower_bound = lower_bounds[2],
               upper_bound = upper_bounds[2])

bs = as.matrix(bs)
fx = bs%*%t(w)

plot(x=test,y=fx,type='l',xlab='Time lag')
```



From the second effect, we can observe that the chance of being cited is higher approximately during the first 2500 days following the publication date. This trend indicates that patents are more likely to receive citations within the first 7 years after they are published.

Exercise 1

Include the *textual similarity* covariate (`sim` and `sim_2`) in the model and plot the effects. (Note: If training is taking too long, either manually stop it when the validation loss is below 0.15 or reduce the patience to 1).

Exercise 2

Reduce the learning rate to 0.01, and run the model again.