Aplicación del aprendizaje automático al laboratorio de diagnóstico clínico: Detección temprana de series analíticas fuera de control en análisis inmunoquímicos de muestras de sangre mediante algoritmos de Machine Learning

Sergio García Muñoz

2023-06-22

LSTM_samples

Instalación y carga de paquetes requeridos:

```
require(readr)
require(readxl)
require(purrr)
require(dplyr)
require(knitr)
require(filesstrings)
require(stringr)
require(tidyr)
require(lubridate)
require(ggplot2)
require(wavelets)
require(tictoc)
require(reticulate)
require(abind)
require(tensorflow)
require(tfdatasets)
require(tfaddons)
require(keras)
require(ModelMetrics)
require(ROCR)
```

Definición de rutas y directorios

```
# Definir el directorio donde se encuentran los archivos de datos:
workingdir <- getwd()
datadir <- file.path(workingdir, "Datos/daily_s")
eventdir <- file.path(workingdir, "Datos/daily_s/Event")
```

```
eventdir_old <- file.path(workingdir, "Datos/daily_s/Event/old")
lotdir <- file.path(workingdir, "Datos/daily_s/Lot")
lotdir_old <- file.path(workingdir, "Datos/daily_s/Lot/old")
qcdir <- file.path(workingdir, "Datos/daily_s/qc")
qcdir_old <- file.path(workingdir, "Datos/daily_s/qc/old")
resultsdir <- file.path(workingdir, "Resultados")
figuresdir <- file.path(workingdir, "Resultados/Figuras")

# Definir el intervalo de tiempo en segundos entre las lecturas
interval <- 60
```

Definición de funciones:

Función `read_lot`.

Lee los archivos csv de lotes y crear un data frame.

```
read_lot <- function(CLC=NULL) {
  # Crea una lista de archivos existentes en datadir
  # correspondientes a los lotes diarios:
  files_list <- list.files(lotdir, pattern = "Lote_",
                           full.names = T)

  # Especifica los tipos de datos que contiene cada columna:
  col_types <- cols(
  PACIENTE_EDAD = col_integer(),
  RESULTADO = col_double()
  )

  # Especifica que el separador decimal es ",":
  locale_decimal_comma <- locale(decimal_mark = ",")

  # Crea una lista de dataframes leyendo cada archivo:
  df_list <- map(files_list,
                 ~read_delim(.x,
                             delim = "|",
                             col_select = -8,
                             col_types = col_types,
                             trim_ws = T,
                             locale
                             =locale_decimal_comma))

  # Crea el dataframe final:
  df <- bind_rows(df_list)

  # Convierte múltiples formatos de fecha y hora de la columna
  # FECHA_RECEPCIÓN en formato POSIXct, cambia formato fecha y
  # nombre de columna:

  df[[1]] <- dmy_hms(df[[1]])
  colnames(df)[1] <- "TIEMPO_MUESTRA"
```

```
  # Elimina una parte del string del nombre del analizador no utilizable:
  df[[2]] <- substring(df[[2]],10, 21)

  # Filtra los resultados según el CLC introducido si es distinto de nulo:
  if (!is.null(CLC)){
    df %<>% filter(., CODIGO_PRUEBA == CLC)
  }

  # Mueve el archivo leído a la carpeta old:
  # file.move(files_list, lotdir_old, overwrite = TRUE)

  return(df)
}
```

Función `read_lotX`.

Lee los archivos csv de lotes con distintos formatos de fecha y crear un data frame.

```
read_lotX <- function(CLC=NULL) {
  # Crea una lista de archivos existentes en datadir
  # correspondientes a los lotes diarios:
  files_list <- list.files(lotdir, pattern = "LoteX_",
                           full.names = T)

  # Especifica los tipos de datos que contiene cada columna:
  col_types <- cols(
  PACIENTE_EDAD = col_integer(),
  RESULTADO = col_double()
  )

  # Especifica que el separador decimal es ",":
  locale_decimal_comma <- locale(decimal_mark = ",")

  # Crea una lista de dataframes leyendo cada archivo:
  df_list <- map(files_list,
                 ~read_delim(.x,
                             delim = "|",
                             col_select = -8,
                             col_types = col_types,
                             trim_ws = T,
                             locale
                             =locale_decimal_comma))

  # Crea el dataframe final:
  df <- bind_rows(df_list)

  # Convierte múltiples formatos de fecha y hora de la columna
  # FECHA_RECEPCIÓN en formato POSIXct, cambia formato fecha y
  # nombre    de columna:

  df[[1]] <- mdy_hm(df[[1]], tz=Sys.timezone())
  colnames(df)[1] <- "TIEMPO_MUESTRA"
```

```
  # Elimina una parte del string del nombre del analizador   no utilizable:
  df[[2]] <- substring(df[[2]],10, 21)

  # Filtra los resultados según el CLC introducido si es distinto de nulo:
  if (!is.null(CLC)){
    df %<>% filter(., CODIGO_PRUEBA == CLC)
  }

  # Mueve el archivo leído a la carpeta old:
  # file.move(files_list, lotdir_old, overwrite = TRUE)

  return(df)
}
```

Función `read_event`.

Lectura de los archivos csv de eventos y crear un data frame.

```
read_event <- function() {
  # Crea una lista de archivos existentes en datadir
  # correspondientes a los eventos de QC diarios:
  files_list <- list.files(eventdir, pattern = "Evento_",
                           full.names = T)

  # Especifica los tipos de datos que contiene cada columna:
  col_types <- cols(
  FECHA = col_character(),
  LOTE = col_integer())

  # Crea una lista de dataframes leyendo cada archivo:
  df_list <- map(files_list,
                 ~read_delim(.x,
                             delim = "|",
                             col_types = col_types,
                             col_select = -c("LOTE"),
                             trim_ws = T))

  # Crea el dataframe final:
  df <- bind_rows(df_list)

  # Convierte la fecha leída como string en formato POSIXct y cambia
  # nombre a columna:
  df[[1]] <- as.POSIXct(gsub(",", ".", df[[1]]),
              format = "%d/%m/%y %H:%M:%S",
              tz=Sys.timezone())
  colnames(df)[[1]] <- "TIEMPO_EVENTO"

  # Elimina substring no deseado en el nombre del analizador:

  df[[2]] <- substring(df[[2]],10, 21)

  # Mueve el archivo leído a la carpeta old:
  #file.move(files_list, eventdir_old, overwrite = TRUE)
```

```
  return(df)
}
```

Función `read_qc`.

Lee los archivos xls de valores de QC para cada técnica y cada máquina, añade una columna con la id del equipo, otra con el código CLC de la prueba y crea un data frame final. La función contiene una condición if diseñada por el diferente formato de exportación de los archivos del control en uso y el histórico de controles inactivos.

```
read_qc <- function() {
  # Crear una lista de archivos existentes en datadir que contienen los
  # valores de QC, tanto en uso como inactivos:
  files_list <- list.files(qcdir, pattern = "CLC",
                           full.names = T)

  # Lista vacía para contener los dataframes originados en el loop:
  list_df <- list()

  # Loop for para crear una lista de dataframes leyendo cada
  # archivo de qc:
  for(i in 1:length(files_list)){
    data <- read_xls(files_list[i], skip = 4)

    # Buscar y extraer el id del equipo en la primera fila:
    dev_ids <- c("DXI800 num 1", "DXI800 num 2",
             "DXI800 num 3")
    first_rows <- read_xls(files_list[i],
                           col_names = F, n_max = 3)
    match_dev <- which(grepl(paste(dev_ids,
                                   collapse = "|"),
                                   first_rows))
    start_pos <- regexpr("(?<=DxI)\\S+",
                         first_rows[match_dev],
                         perl = TRUE)
    dev <- substring(first_rows[match_dev], start_pos)

    # Extraer el código CLC del nombre del archivo
    start_pos <- regexpr("(?<=CLC)\\S+",
                         files_list[i],
                         perl = TRUE)
    end_pos <- regexpr("\\)", files_list[i], start_pos)
    clc <- substring(files_list[i], start_pos, end_pos - 1)

    # Añadir como columnas "ANALIZADOR" y "CODIGO_PRUEBA" las
    #cadenas extraídas:
    data %<>% mutate("ANALIZADOR"= substr(dev, 13, 24),
                     "CODIGO_PRUEBA"= paste("CLC",clc, sep = "")) %>%
              relocate("ANALIZADOR", "CODIGO_PRUEBA", .after = 5)

    # En caso de que el archivo de datos sea del histórico, la
    # identificación del control no se encuentra como columna, sino en la
```

```
    # cabecera, ya que los archivos han sido generados para cada  nivel de
    # control individualmente.
    # Identificación de este tipo de archivo, extracción de la
    # identificación y creación de la columna "Control":

    if(!("Control" %in% colnames(data))) {
      # Extraer el nombre del control de la fila 3, columna D:
      cont <- read_xls(files_list[i],
                       col_names = F, range = "D2:D2")
      data %<>% mutate("Control"= as.character(cont)) %>%
              relocate("Control", .after = 4)
      }

    # Agrega el dataframe actual a la lista:
    list_df[[i]] <- data
  }

  for (i in seq_along(list_df)) {
  if (!is.numeric(list_df[[i]][[3]])) {
    list_df[[i]][[3]] <- as.numeric(as.character(list_df[[i]][[3]]))
  }
}

  # Unir todos los dataframes en uno solo:
  df <- bind_rows(list_df)

  # Cambia el nombre a la columna Lote de reactivos:
  colnames(df)[10] <- "LOTE_REACTIVO"

  # Fusiona las columnas de los valores de concentraciones
  # encontrados en distintas unidades en una sola columna y cambia el
  # tipo de valor a numérico:
   df <- unite(df, "Resultado", c(3, 13:21),
                          sep = "", na.rm = T)
   df[[3]] %<>% as.numeric()

  # Reordena el dataframe:
   df %<>% relocate("Sup/Inf Media", .after = 4)

  # Cambia el nombre a la columna Lote de reactivos:
   colnames(df)[3] <- "QC_RESULT"

  # Elimina filas con valores NA en LOTE_REACTIVO:

   df <- df[complete.cases(df$LOTE_REACTIVO), ]

   # Mueve el archivo leído a la carpeta old:
  # file.move(files_list, qc_old, overwrite = TRUE)

  return(df)
}
```

Función `figure_ext`.

Extracción de datos numéricos de la columna media y sd.

```
figure_ext <- function(df, col_number) {
  # Extrae números presentes en la columna especificada:
  mean_val <- c()
  sd_val <- c()
  for (i in 1:nrow(df[col_number])){
    numbers <- str_extract_all(df[i, col_number],
                               "\\d+(\\.\\d+)?")

    # Convertir a decimal y hallar media y sd:
    numbers <- as.double(unlist(numbers))
    mean_val <- rbind(mean_val,
                      (numbers[1] + numbers[2])/2)
    sd_val <- rbind(sd_val,
                    (numbers[2] - numbers[1])/2)
  }

  # Crear nuevas columnas media y sd:

  df %<>% mutate("Media" = mean_val, "SD" = sd_val) %>%
          relocate("Media", "SD", .after =3)

  # Borrar columna original
  df[[col_number]] <- NULL

  # Devolver el dataframe modificado
  return(df)
}
```

Función `merge_date`.

Fusiona columnas Fecha y Hora en una nueva columna Fecha_hora y borra las columnas individuales.

```
  merge_date <- function(df){
    df %<>%
      mutate("TIEMPO_QC" =
               as.POSIXct(paste(Fecha, Hora),
                          format = "%d/%m/%y %H:%M:%S",
                          tz=Sys.timezone()),
             .before = 1)
    # Borrar columnas originales
    df[["Fecha"]] <- NULL
    df[["Hora"]] <- NULL

  return(df)
}
```

Función `merge_feat_qc`.

Función que une los dataframes features día D y qc_results día D+1 en uno solo mediante Left outer join.

```
merge_feat_qc <- function(df1, df2){
  # Crear timestamps con solo la fecha:
  df1$rounded_date <- floor_date(as.Date(df1[[1]])+1,
                                 unit = "day")
  df2$rounded_date <- floor_date(as.Date(df2[[1]]),
                                 unit = "day")

  # Crear un nuevo dataframe con merge (equivale a left outer join):
  merged_df <- merge(df1, df2,
                     by = c("rounded_date", "ANALIZADOR",
                            "CODIGO_PRUEBA", "LOTE_REACTIVO"))

  # Borrar los timestamps con solo fecha:
  merged_df$rounded_date <- NULL


  return(merged_df)

}
```

Función `merge_feat_qc_event`.

Une los dataframes feat_qc y events a día D en uno solo mediante Left outer join. Añade la posibilidad de filtrar por códigos de prueba para seguir trabajando con un subconjunto de las mismas.

```
merge_featqc_event <- function(df1, df2, CLC=NULL){

  # Crear rounded_date sin hora, solo fecha:
  df1$rounded_date <- floor_date(as.Date(df1[[10]]),
                                 unit = "day")
  df2$rounded_date <- floor_date(as.Date(df2[[1]]),
                                 unit = "day")

  # Crear un nuevo dataframe con merge (equivale a left outer join):
  merged_df <- merge(df1, df2,
                     by = c("rounded_date", "ANALIZADOR",
                            "CODIGO_PRUEBA"), all.x = T)  %>%
               unique()

  # Borrar rounded_date:
  merged_df$rounded_date <- NULL

  # Filtrado en caso de que se requiera por CODIGO_PRUEBA:
  if (!is.null(CLC)){
    merged_df %<>% filter(., CODIGO_PRUEBA == CLC)
  }

  return(merged_df)

}
```

Función `wt`.

Función para la aplicación de la transformación wavelet de tipo Maximal Overlap Discrete Wavelet Transformation (MODWT) que devuelve los coeficientes de wavelet y los coeficientes de escalado.

```
wt <- function(x, wavefun){
  mod <- modwt(x, wavefun, boundary = "periodic")
return(mod)
}
```

Función `wavelet_tr`.

Aplica la transformación wavelet a un dataframe df agrupado según variables y devuelve un dataframe con los coeficientes y escalas para cada observación.

```
wavelet_tr <- function(df) {
  # Aplicar la función wt a cada grupo por separado
  df_wavelet <- df %>%
    group_by(ANALIZADOR, CODIGO_PRUEBA) %>%
    mutate(
      across(RESULTADO, ~ cbind(.x, as.data.frame(wt(.x, "la20")@W),
                                as.data.frame(wt(.x, "la20")@V)))
    ) %>%
    ungroup() %>%
    unnest(cols = RESULTADO)

  # Reordenar columnas
  df_wavelet <- df_wavelet %>%
    relocate(starts_with("RESULTADO"), .after = 1) %>%
    relocate(starts_with("W"), .after = 2) %>%
    relocate(starts_with("V"), .before = PACIENTE_SEXO)

  # Sustituir valores NA en los coeficientes wavelet por un valor
  # arbitrario -99999 para su posterior filtrado
  wavelet_cols <- grep("^[WV]", colnames(df_wavelet), value = TRUE)

  return(df_wavelet)
}
```

Función `sub_sample`

Realiza un submuestreo estratificado aleatorio de la clase mayoritaria CTRL = 0, sustituyendo las entradas de esta clase por las seleccionadas aleatoriamente después de agrupar por fecha y código de prueba.

```
sub_sample <- function(ds, prop){
  ss0 <- ds %>%
  mutate(FECHAS = as.Date(TIEMPO_MUESTRA)) %>%
  group_by(FECHAS, CODIGO_PRUEBA, CTRL) %>%
  filter(CTRL == 0) %>%
  slice_sample(prop = prop) %>%
  ungroup() %>%
  select(-FECHAS)
```

```
  ds_ss <- rbind(ds %>% filter(CTRL == 1), ss0)
  ds_ss %<>% arrange(TIEMPO_MUESTRA)

  return(ds_ss)
}
```

Función `normalize_train`.

Para la normalización de las variables continuas del dataset de entrenamiento y sustitución de valores NA en los coeficientes wavelet por un valor numérico arbitrario fácilmente enmascarable. Devuelve también una lista de vectores de medias y de desviaciones estándar que luego se usan como argumentos en para las funciones de normalización de los datasets de validación y test. De esta forma, el escalado y normalización se realiza con la misma media y desviación del set de entrenamiento para los otros dos datasets.

```
normalize_train <- function(data) {

  # Identificar las columnas que empiezan por W y la columna 'RESULTADO'
  wavelet_cols <- grep("^[WV]", colnames(data), value = TRUE)
  result_col <- "RESULTADO"

  # Calcular medias y desviaciones estándar por grupos de pruebas y   analizador:
  means <- data %>%
    group_by(ANALIZADOR, CODIGO_PRUEBA) %>%
    summarise(across(c(result_col, wavelet_cols),
                     ~mean(., na.rm = T)))

  std_devs <- data %>%
    group_by(ANALIZADOR, CODIGO_PRUEBA) %>%
    summarise(across(c(result_col, wavelet_cols),
                     ~sd(., na.rm = T)))

  # Normalizar las columnas de resultado y coeficientes wavelet
  norm_data <- data %>%
    group_by(ANALIZADOR, CODIGO_PRUEBA) %>%
    mutate(across(c(result_col, wavelet_cols), scale))%>%
    mutate(across(wavelet_cols,
                            ~ifelse(is.na(.), -99999, .))) %>%
    ungroup()

  # Devolver el dataframe escalado y las medias y desviaciones estándar:

  return(list(scaled_data = as.data.frame(norm_data),
              means = means, stdev = std_devs))
}
```

Función `norm_test_val`.

Análoga a `normalize_train`, pero con la diferencia de que usa la media y desviación estándar del dataset de entrenamiento para validar los datasets de validación y test:

```r
norm_test_val <- function(data, means, stds) {

  # Identificar las columnas numéricas y seleccionarlas
  wavelet_cols <- grep("^[WV]", colnames(data), value = TRUE)
  result_col <- c("RESULTADO")
  test_cols <- "CODIGO_PRUEBA"
  cols <- c(result_col, wavelet_cols, test_cols)

  # Seleccionar medias y desviaciones de resultados y coeficientes   wavelet:
  means_r <- train_n$means
  stds_r <- train_n$stdev
  means_wl <- train_n$means[wavelet_cols]
  stds_wl <- train_n$stdev[wavelet_cols]

  # Normalizar por cada prueba las columnas de resultado y coeficientes
  # wavelet usando las medias y desviaciones estándar de train
  # suministradas:
  means <- data %>%
    group_by(ANALIZADOR, CODIGO_PRUEBA) %>%
    left_join(means_r, by = c("ANALIZADOR", "CODIGO_PRUEBA"),
              suffix = c(".val", ".means")) %>%
     ungroup()

  stds <- data %>%
    group_by(ANALIZADOR, CODIGO_PRUEBA) %>%
     left_join(stds_r, by = c("ANALIZADOR", "CODIGO_PRUEBA"),
               suffix = c(".val", ".sd")) %>%
    ungroup()


  norm_data <- data %>%
    mutate(., RESULTADO.st =
             (.[,result_col]-
                select(means,
                       starts_with(result_col) &
                       ends_with(".means"))) /
                select(stds,
                       starts_with(result_col) &
                       ends_with(".sd"))
           ) %>%
    mutate(WAVELETS.st =
             across(all_of(wavelet_cols),
                    ~ (. - select(means,
                                  starts_with(cur_column()) &
                                    ends_with(".means"))[[1]]) /
                      select(stds,
                             starts_with(cur_column()) &
                               ends_with(".sd"))[[1]])
           ) %>%
    select(-c(result_col, wavelet_cols)) %>%
    unnest(c(WAVELETS.st, RESULTADO.st)) %>%
    relocate(starts_with("RESULTADO"), .after = 1) %>%
    relocate(starts_with("W"), .after = 2) %>%
```

```
    relocate(starts_with("V"), .before = PACIENTE_SEXO) %>%
    mutate(across(wavelet_cols,
                          ~ifelse(is.na(.), -99999, .)))

  names(norm_data)[2]<-result_col

  return(as.data.frame(norm_data))
}
```

Función `code_cat_var`.

Convierte las variables categóricas tipo string en vectores numéricos y realiza una codificción one-hot, creando una lista que retiene en un vector la información de los niveles de los factores.

```
code_cat_var <- function(data) {
  # Eliminación de la columna CODIGO_PRUEBA si solo tiene una          categoría:
  if(length(unique(data$CODIGO_PRUEBA))==1){
    data %<>% select(., -CODIGO_PRUEBA)
    }

  # Eliminación de los casos con sexo y/o indeterminadoz, debido a que
  # son muy minoritarios y crean una variable desequilibrada entre
  # datasets:

  data %<>% filter(PACIENTE_SEXO != "U", EDAD_BIN != "Desconocido")

  # Detección de strings:
  cat_vars <- which(sapply(data, is.character))
  # Conversión a factores:
  data_cat <- lapply(data[,cat_vars], as.factor)
  # Extracción de niveles:
  levels <- lapply(data_cat, levels)

  # Codificar one-hot todas las variables factor:
  encoded_cols <- lapply(seq_along(data_cat), function(i) {
    cols <- model.matrix(~ factor(data_cat[[i]]) - 1)
    colnames(cols) <- paste0(names(data)[cat_vars[i]], "_",
                             levels[[i]])
    cols
    })

  # Unir columnas codificadas con conjunto de datos original:
  if (!"CODIGO_PRUEBA" %in% names(data)) {
    data_encoded <- bind_cols(data %>%
                                select(-all_of(cat_vars)),
                                encoded_cols)
    } else {
      data_encoded <-  bind_cols(data %>%
                                   select(c(-all_of(cat_vars),
                                            -"CODIGO_PRUEBA")),
                                 encoded_cols)
      }
```

```
    # Reordenar columnas:
  data_encoded <- data_encoded %>%
  relocate(CTRL, .after = everything())

  return(list(data = data_encoded, levels = levels))
}
```

Función `factor_cat_var`.

Función para convertir las variables categóricas tipo string en factores y éstos a números enteros. Codifica las variables categóricas como enteros para luego pasarlas a una capa de embedding.

```
factor_cat_var <- function(data) {
  # Detección de variables categóricas:
  cat_vars <- which(sapply(data, is.character))

  # Conversión a factores y de factores a enteros:
  data[,cat_vars] <- lapply(data[,cat_vars], function(x) {
    levels_x <- unique(x)
    factor_x <- factor(x, levels = levels_x)
    as.integer(factor_x)
  })

  return(data)
}
```

Función `unique_lab_cases`.

Crea un índice de casos únicos de combinaciones de variables categóricas si están codificadas con one-hot.

```
unique_lab_cases <- function(df, categorical_vars) {
  unique_groups <- unique(df[categorical_vars])

  code <- apply(unique_groups, 1, function(row) {
    paste0(ifelse(row == 1, 1, 0), collapse = "")
  })
  unique_groups %<>% mutate(., code)
  coded_df <- df %>%
    merge(unique_groups, ., by = colnames(unique_groups)
          [-length(colnames(unique_groups))])
  coded_df <- coded_df[, c(colnames(df), "code")]

  return(coded_df)
}
```

Función `unique_lab_cases.2`.

Crea un índice de casos únicos de combinaciones de variables categóricas codificadas con factor_cat_var.

```r
unique_lab_cases.2 <- function(df, categorical_vars) {
  unique_groups <- unique(df[categorical_vars])

  code <- apply(unique_groups, 1, function(row) {
    paste(row, collapse = "")
  })
  unique_groups %<>% mutate(., code)
  coded_df <- df %>%
    merge(unique_groups, ., by = colnames(unique_groups)
          [-length(colnames(unique_groups))])
  coded_df <- coded_df[, c(colnames(df), "code")]

  return(coded_df)
}
```

Función `create_lstm_data.1`.

Diseñada para la creación de lotes de series temporales de longitud k (lookback). Basada en la función presentada en Chollet François, Deep learning with Python. Shelter Island, NY: Manning Publications Co; 2018. 335 p. Cap. 6.3.2

```r
create_lstm_data.1 <-
  function(data, lookback, delay, min_index, max_index,
           shuffle = FALSE, batch_size, step = 1,
           predseries) {

  if (is.null(max_index)) max_index <- nrow(data) - delay - 1
  i <- min_index + lookback
  gen <- function() {
    if (shuffle) {
      rows <- sample(c((min_index+lookback):max_index),
                     size = batch_size)
    } else {
      if (i + batch_size >= max_index)
        i <<- min_index + lookback
      rows <- c(i:min(i+batch_size, max_index))
      i <<- i + length(rows)
    }
    samples <- array(0, dim = c(length(rows),
                                lookback / step,
                                dim(data)[[-1]]-1))
    targets <- array(0, dim = c(length(rows)))
    for (j in 1:length(rows)) {
      indices <- (rows[[j]] - lookback + 1):(rows[[j]])
      samples[j,,] <- data[indices, -predseries]
      targets[[j]] <- data[rows[[j]] + delay, predseries]
    }
    list(samples, targets)
  }
  return(gen)
}
```

Carga y preprocesado de los datos

Lectura de los datos de medidas de muestras y características (features), eventos de QC y QC.

```
  # Leer los archivos Excel de Lotes y crear un data frame  feautures
  features <- unique(rbind(read_lot(), read_lotX()))
  features <- features[order(features[[1]]), ]
  length(which(is.na(features$TIEMPO_MUESTRA)))
  features <- features[!is.na(features$TIEMPO_MUESTRA),]

  head(features)
  summary(features)

  #Leer los archivos de eventos de QC y crear el data frame events:
  events <- unique(read_event())
  length(which(is.na(events$TIEMPO_EVENTO)))

  head(events)
  summary(events)

  # Leer los archivos de resultados de QC y crear el data frame
  # qc_results, fusión de fecha y hora en una sola columna y separación
  # de datos de media y desviación estándar en columnas individuales y
  # como valores numéricos:
  qc_results <- read_qc()

  qc_results <- qc_results %>% merge_date()

  length(which(is.na(qc_results$TIEMPO_QC)))

  qc_results <- qc_results[!is.na(qc_results$TIEMPO_QC),]

  qc_results <- figure_ext(qc_results, 3)



  head(qc_results)
  summary(qc_results)
```

Unión de datos de features, eventos de QC y QC.

```
# Unir los tres dataframes features, qc_results y events en uno
# Selección de pruebas opcional.
feat_qc <- merge_feat_qc(features, qc_results)

featqc_event <- merge_featqc_event(feat_qc, events,)
```

Creación de los conjuntos de targets por selección según criterios específicos.

```
# Creación de los targets mediante filtrado de casos que cumplan las
# condiciones necesarias para considerar una serie fuera de control,
# asignando valor 1 a la variable CTRL para cada caso en que se cumplan
# estos criterios.

# Criterio 1: Para cada unidad temporal de TIEMPO_QC, ANALIZADOR,
# CODIGO_PRUEBA y LOTE_REACTIVO:
# EVENTO==CAL AND
# Al menos 1 flag != NA

targets_1 <- featqc_event %>%
  filter(EVENTO == "CAL") %>%
  mutate(DIA = as.Date(TIEMPO_QC)) %>%
  group_by(DIA, ANALIZADOR, CODIGO_PRUEBA,
           LOTE_REACTIVO) %>%
  filter(any(!is.na(Flags))) %>%
  select(-DIA) %>%
  mutate(CTRL = 1)

# Criterio 2: Para cada unidad temporal de TIEMPO_QC, ANALIZADOR,
# CODIGO_PRUEBA y LOTE_REACTIVO:
# Al menos 1 flag !NA AND
# Observaciones !NA

targets_2 <- featqc_event %>%
  mutate(DIA = as.Date(TIEMPO_QC)) %>%
  group_by(DIA, ANALIZADOR, CODIGO_PRUEBA, LOTE_REACTIVO) %>%
  filter(any(!is.na(Flags)) &
           any(!is.na(Observaciones))) %>%
  select(-DIA) %>%
  mutate(CTRL = 1)

# Criterio 3: Para cada unidad temporal de TIEMPO_QC, ANALIZADOR,
# CODIGO_PRUEBA y LOTE_REACTIVO:
# >=2 flags 12s, 12.5s o 13s dentro del día

targets_3 <- featqc_event %>%
  mutate(DIA = as.Date(TIEMPO_QC)) %>%
  group_by(DIA, ANALIZADOR, CODIGO_PRUEBA,
           LOTE_REACTIVO, Control) %>%
  filter(sum(str_detect(Flags, "\\b(12s|12.5s|13s)\\b")) >= 2) %>%
  ungroup() %>%
  mutate(CTRL = 1)
```

Fusión de targets.

```
# Unir los 3 targets anteriores en uno solo y con el conjunto de datos
# featqc_event. Se añade el valor 0 a la variable CTRL en los casos no
# presentes en targets, que serán los que se consideran controlados.
```

```
targets <- rbind(targets_1, targets_2, targets_3)

data <- merge(featqc_event,
              targets[, c("TIEMPO_QC", "ANALIZADOR",
                          "CODIGO_PRUEBA", "LOTE_REACTIVO",
                          "CTRL")],
              by = c("TIEMPO_QC", "ANALIZADOR",
                     "CODIGO_PRUEBA", "LOTE_REACTIVO"),
              all.x=T) %>%
  unique() %>%
  arrange(match(row.names(.), row.names(featqc_event))) %>%
  mutate(CTRL = replace(CTRL, is.na(CTRL), 0)) %>%
  select(colnames(featqc_event), CTRL)
```

Creación de un diccionario de correspondencia entre códigos y nombres de pruebas. Eliminación de nombres de prueba con caracteres especiales

```
# Pruebas incluídas en los datos:
  iqtests <- unique(data.frame(data$NOMBRE_PRUEBA.x,
                               data$CODIGO_PRUEBA))

  # Corrección de caracteres incorrectos en NOMBRE_PRUEBA:
  iqtests$data.NOMBRE_PRUEBA.x <-
    iconv(iqtests$data.NOMBRE_PRUEBA.x, to = "UTF-8")
  iqtests <- iqtests[complete.cases(iqtests),]
  names(iqtests) <- c("NOMBRE_PRUEBA", "CODIGO_PRUEBA")

  # Sustitución en data de los nombres de prueba incorrectos:
  data$NOMBRE_PRUEBA.x <-
    iqtests$NOMBRE_PRUEBA[match(data$CODIGO_PRUEBA,
                                iqtests$CODIGO_PRUEBA)]
```

Selección de características de interés y transformación wavelet de los datos de resultados de medidas.

```
data_sel <- subset(data,
                   select = c(TIEMPO_MUESTRA, PACIENTE_SEXO,
                              PACIENTE_EDAD, RESULTADO, ANALIZADOR,
                              CODIGO_PRUEBA, NOMBRE_PRUEBA.x, CTRL)) %>%
  unique()

# Aplicar la transformación wavelet a la variable RESULTADOS para cada
# grupo de técnica:


# Aplicar la función wavelet_tr al dataframe data_sel
data_wavelet <- wavelet_tr(data_sel)
```

```r
# Escalograma procalcitonina (CLC00638):
result_pct <- data_sel %>%
  filter(CODIGO_PRUEBA == "CLC00638" &
           ANALIZADOR == "DXI800 num 1") %>%
  select(RESULTADO)

plot(result_pct$RESULTADO, xlab = "día", ylab = "PCT, ng/mL", main =
       "Resultados Procalcitonina a lo largo del tiempo de estudio")

plot.modwt(wt(result_pct$RESULTADO, "la20"))
```

Transformación de la variable edad en variable categórica por agrupación (binning) en rangos de edades:

```r
# Aplicar binning a la variable edad:

# Definir los límites para el binning de la variable edad
age_limits <- c(0, 18, 30, 60, Inf)

# Aplicar el binning a la variable edad :
data_sel_tr <- data_wavelet %>%
  mutate(EDAD_BIN = as.character(cut(PACIENTE_EDAD,
                        age_limits, labels = c("Niño", "Joven",
                                               "Adulto", "Anciano"))),
         .after = "PACIENTE_EDAD") %>%
  select(-"PACIENTE_EDAD") %>%
  replace_na(list(EDAD_BIN = "Desconocido"))
data_sel_tr <- as.data.frame(data_sel_tr)
head(data_sel_tr)
```

Guardado de datos en archivo csv.

```r
write.csv(data_sel_tr, row.names = F,
          file = file.path(resultsdir, "data_sel_tr"))
```

Carga de datos desde archivo csv.

```r
data_sel_tr <- read.csv(file.path(resultsdir, "data_sel_tr"),
                        sep = ",")

# Convertir fechas en formato POSIXct:
data_sel_tr[,1] %<>% as.POSIXct(tz = "Europe/Madrid")
```

Descriptiva de variables y gráficos de datos.

```
# Estructura de datos:
str(data_sel)

k1 <- kable(table(data$NOMBRE_PRUEBA.x, data$CTRL), format = "latex",
      escape = T,
      col.names = c("TEST", "CONTROLADO/NO CONTROLADO"))

sink(file.path(resultsdir, "tabla_targets.txt"))
cat(k1)
sink()


# Plots resultados por prueba divididos por valor de la variable de proceso controlado (CTRL):

ggplot(data , mapping = aes(x=TIEMPO_MUESTRA,
                                    y=RESULTADO,
                                    color = as.factor(CTRL))) +
         geom_point(size=0.25) +
         facet_wrap(~NOMBRE_PRUEBA.x, scales = "free", ncol = 4)+
         theme(strip.background = element_blank()) +
         theme(strip.text = element_text(size=8)) +
         scale_color_manual(values = c("blue", "red"),
                            name = "CTRL")

ggsave(file.path(figuresdir, "QC_state.jpeg"),
       width = 30, height = 20, units = "cm", dpi = 300,
       device = "jpeg")


for (i in seq_along(iqtests$CODIGO_PRUEBA)) {
  ggplot_qc_DxI_1 <- subset(qc_results,
                        CODIGO_PRUEBA == iqtests$CODIGO_PRUEBA[i])
  plot <- ggplot(ggplot_qc_DxI_1, mapping =
                   aes(x=TIEMPO_QC,
                       y=QC_RESULT,
                       color = Control)) +
          geom_point(size=0.5) +
          theme_bw() +
          labs(x="Date",y="Resultado QC") +
          scale_x_datetime(date_breaks = "1 month", date_labels =
                             "%b",
                   limits = as.POSIXct(c("2022-07-01 00:00:00",
                                         "2023-03-01 23:59:59"))) +
          ggtitle(paste(iqtests$NOMBRE_PRUEBA[i])) +
          theme(plot.title = element_text(size="10",
                                          face="bold",hjust = 0)) +
          theme(axis.title.x = element_text(size="10")) +
          theme(axis.title.y = element_text(size="10")) +
          theme(axis.text.y = element_blank()) +
          theme(axis.text.x = element_text(size=10)) +
          theme(axis.ticks.y = element_blank()) +
```

```
          theme(axis.ticks.x = element_blank()) +
          theme(panel.border = element_blank()) +
          theme(plot.margin = margin(t = 1, r = 1, b = 1, l = 1,
                                     unit = "cm")) +
          scale_color_manual(values = c("red", "blue", "green",
                                        "black", "orange", "cyan",
                                        "yellow"),
                             name = "Control")

  ggsave(plot,
         file=paste(iqtests$NOMBRE_PRUEBA[i], "qc.png", sep='_'),
         path = file.path(figuresdir, "Controles"),
         width=15, height = 10, units=c("cm"))
}

#Plot frecuencia de resultados obtenidos por prueba:

ggplot(data, aes(x=NOMBRE_PRUEBA.x)) +
  geom_bar(fill="blue") +
  labs(title="Frecuencia de medidas por NOMBRE_PRUEBA",
       x="NOMBRE_PRUEBA", y="frecuencia") +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5))

ggsave("frequency.png", width = 30, height = 20, units = "cm", dpi = 400,
       path = file.path(figuresdir), device = "png"
       )

# Plot distribución casos}

plot_ctrl_dist <- function(ds, clc){
  # Crea una lista vacía para almacenar los plots:
  plot_list <- list()

  count <- 0 # Contador de plots

  # Genera plots de frecuencia de casos distribuido para cada
  # CODIGO_PRUEBA suministrado en la variable clc y separados
  # según valor de variable CTRL:
for (i in seq(1, length(clc), by = 4)){

  g <- ds[ds$CODIGO_PRUEBA %in% clc[i:(i+3)], ] %>%
  ggplot(., aes(x = as.Date(TIEMPO_MUESTRA))) +
  geom_bar() +
  labs(x = "TIEMPO", y = "FRECUENCIA") +
  scale_x_date(date_labels = "%b %Y", date_breaks = "1 month") +
  coord_cartesian(ylim = c(0, 100)) +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  facet_grid(CTRL~NOMBRE_PRUEBA.x)

  count <- count +1

  plot_list[[count]] <- g
```

```
}
# Guarda archivos png con los plots en el directorio de resultados:
for (i in seq_along(plot_list)) {
  filename <- paste0("Distribución_CTRL_",
                     deparse(substitute(ds)), "_", i, ".png")
  ggsave(filename, plot = plot_list[[i]], device = "png",
         width = 30, height = 20, units = "cm", dpi = 300,
         path = file.path(figuresdir)
         )
}

}

plot_ctrl_dist(data_sel_tr, iqtests$CODIGO_PRUEBA)
```

División de los datos en sets de entrenamiento, validación y test.

```
# Se divide el dataset en 50% de datos para entrenamiento, 25% para test
# y 25% para validación. Para evitar fraccionar las unidades temporales
# diarias usamos percentiles de la secuencia temporal en días:

time_point_50 <- as.POSIXct(as.Date(
  quantile(data_sel_tr[,1], 0.5)))

time_point_75 <- as.POSIXct(as.Date(
  quantile(data_sel_tr[,1], 0.75)))

time_point_50
time_point_75
```

```
# Muestra de entrenamiento:
train <- data_sel_tr %>%
  subset(.[[1]] <= time_point_50)

table(train$NOMBRE_PRUEBA.x, train$CTRL)
```

```
# Muestra de validación:
val <- data_sel_tr %>%
  subset(
    .[[1]] > time_point_50 &
    .[[1]] <= time_point_75)

table(val$NOMBRE_PRUEBA.x, val$CTRL)
```

```
# Muestra de test:
test <- data_sel_tr %>%
  subset(.[[1]] > time_point_75)

table(test$NOMBRE_PRUEBA.x, test$CTRL)
length(unique(train))
```

Normalización de los datasets.

```
train_n <- train %>% normalize_train()

val_n <- val %>% norm_test_val(means = train_n$means,
                               stds = train_n$stdev)

test_n <- test %>% norm_test_val(means = train_n$means,
                               stds = train_n$stdev)
```

Downsampling de casos con CTRL=0 en el training set.

Se emplea la función `sub_sample` diseñada para eliminar la proporción deseada de casos pertenecientes a la clase CTRL = 0 por muestreo estratificado aleatorio.

```
set.seed(123)

train_ss <- train_n$scaled_data %>% sub_sample(prop=0.5)

table(train_ss$NOMBRE_PRUEBA.x, train_ss$CTRL)

clc_dropout <- iqtests$CODIGO_PRUEBA[which(!(iqtests$CODIGO_PRUEBA %in%
               unique(train_ss$CODIGO_PRUEBA)))]
```

Exclusión del modelo general de las técnicas que no alcanzan representación de la categoría CTRL = 1 en todos los datasets.

```
# Filtrado por código de pruebas que en training no tienen representación
# en la categoría CTRL = 1, y creación de una lista.

clc_sel <- list()

clc_sel[[1]] <- train_ss %>%
  group_by(CODIGO_PRUEBA) %>%
  filter(all(CTRL != 1) | all(CTRL != 0))

# Unión de los elementos de la lista y extracción de los CODIGO_PRUEBA
# que las componen:
clc_sel %<>% do.call(rbind, .)
clc_other <- c(clc_dropout, unique(clc_sel$CODIGO_PRUEBA))
names_other <- iqtests$NOMBRE_PRUEBA[iqtests$CODIGO_PRUEBA %in%
                                       clc_other]

# Subsetting de los dataframes de train, validation y test eliminando las
# entradas correspondientes a las pruebas encontradas en el paso
# anterior:
train_sel <- train_ss %>% subset(., !(CODIGO_PRUEBA %in% clc_other))
val_sel <- val_n %>% subset(., !(CODIGO_PRUEBA %in% clc_other))
test_sel <- test_n %>% subset(., !(CODIGO_PRUEBA %in% clc_other))
```

```
# Distribución tras la eliminación de estas pruebas

table(train_sel$NOMBRE_PRUEBA.x, train_sel$CTRL)
table(val_sel$NOMBRE_PRUEBA.x, val_sel$CTRL)
table(test_sel$NOMBRE_PRUEBA.x, test_sel$CTRL)

cat("Técnicas que no pueden entrenarse con estos datasets: ", names_other)
```

Codificación de variables categóricas.

Para el uso de embeddings usaremos codificación con factor_cat_var

Con wavelets + RESULTADO.

```
train_code <- factor_cat_var(subset(train_sel,
                                    select = -NOMBRE_PRUEBA.x))

val_code <- factor_cat_var(subset(val_sel,
                                    select = -NOMBRE_PRUEBA.x))

test_code <- factor_cat_var(subset(test_sel,
                                    select = -NOMBRE_PRUEBA.x))

lapply(list(train_code$data, val_code$data, test_code$data), ncol)
```

Sin wavelets.

```
train_code <- factor_cat_var(subset(train_sel,
                                    select = -c(3:18, NOMBRE_PRUEBA.x)))

val_code <- factor_cat_var(subset(val_sel,
                                    select = -c(3:18, NOMBRE_PRUEBA.x)))

test_code <- factor_cat_var(subset(test_sel,
                                    select = -c(3:18, NOMBRE_PRUEBA.x)))
```

Creación de las funciones generadoras para las series temporales etiquetadas de los datasets de training, validación y prueba, y de sus correspondientes funciones envolventes o wrapper.

Se crean funciones generadoras basadas en create_lstm_data.1 que devuelven las series temporales tomando los argumentos necesarios. A su vez, éstas funciones se envuelven en una función wrapper que las llama de forma constante y previene los casos en los que pudieran fallar durante el entrenamiento por falta de datos en alguna serie.

Estas funciones envolventes están diseñadas con objeto de que puedan servir de entrada de datos a una red neuronal de entrada doble diferenciada para los datos dinámicos y los estáticos, además de generar también la variable objetivo. Para ello generan una lista con dos sublistas, la primera de las cuales contiene 2 tensores,

un tensor de orden 3 para los datos de series temporales de la variable dinámica y un tensor de orden 2 para los datos de las variables estáticas. Los datos de la variable objetivo salen en la segunda lista en forma de tensor de orden 0 (vector).

Dataset de training.

```
train_data <- as.matrix(train_code[,-1])
lookback <- 10
batch_size <- 256
delay <- 0
min_index <- 0
predseries <- ncol(train_data)

train_k <-  create_lstm_data.1(
    data = train_data,
    lookback = lookback,
    delay = delay,
    batch_size = batch_size,
    min_index = min_index,
    max_index = NULL,
    predseries = predseries)
```

```
# Cálculo del número de steps de training:
train_steps <- round((nrow(train_data)-lookback) / batch_size)
```

Dataset de validación.

```
val_data <- as.matrix(val_code[,-1])

val_k <-  create_lstm_data.1(
    data = val_data,
    lookback = lookback,
    delay = delay,
    batch_size = batch_size,
    min_index = min_index,
    max_index = NULL,
    predseries = ncol(val_data))
```

```
# Cálculo del número de steps de training:
val_steps <- round((nrow(val_data)-lookback) / batch_size)
```

Dataset de test.

```
test_data <- as.matrix(test_code[,-1])

test_k <-  create_lstm_data.1(
    data = test_data,
```

```
    lookback = lookback,
    delay = delay,
    batch_size = batch_size,
    min_index = min_index,
    max_index = NULL,
    predseries = predseries)


# Cálculo del número de steps de training:
test_steps <- round((nrow(test_data)-lookback) / batch_size)
```

Funciones wrapper `wrap_train()`, `wrap_val()` y `wrap_test()`.

Devuelven los tensores mencionados.

```
wrap_train <- function() {
  seq <- NULL

  while (is.null(seq)) {
    tryCatch(
      seq <- train_k(),
      error = function(e) {
        warning("Empty sequence error occurred. Trying the next one.")
        return(NULL)  # Devuelve NULL si ocurre un error en la función
                      # generadora y continúa el loop while.
      }
    )
  }

  tensor_list <- list(
    list(seq[[1]][, , 1, drop = FALSE],
         seq[[1]][, , -1]),
    list(seq[[2]])
  )

  return(tensor_list)
}
```

```
wrap_val <- function() {
  seq <- NULL

  while (is.null(seq)) {
    tryCatch(
      seq <- val_k(),
      error = function(e) {
        warning("Empty sequence error occurred. Trying the next one.")
        return(NULL)  # Devuelve NULL si ocurre un error en la función
                      # generadora y continúa el loop while.
      }
    )
  }
```

```
  tensor_list <- list(
    list(seq[[1]][, , 1, drop = FALSE],
         seq[[1]][, , -1]),
    list(seq[[2]])
  )

  return(tensor_list)
}
```

```
wrap_test <- function() {
  seq <- NULL

  while (is.null(seq)) {
    tryCatch(
      seq <- test_k(),
      error = function(e) {
        warning("Empty sequence error occurred. Trying the next one.")
        return(NULL)  # Devuelve NULL si ocurre un error en la función
                      # generadora y continúa el loop while.
      }
    )
  }

  tensor_list <- list(
    list(seq[[1]][, , 1, drop = FALSE],
         seq[[1]][, , -1]),
    list(seq[[2]])
  )

  return(tensor_list)
}
```

Definición del modelo de Machine Learning. Red Neuronal Recurrente LSTM (Long-short term memory) y entrenamiento.

Métricas a medida:

Se definen funciones para calcular las métricas precisión, recall y F1 score, que combina ambas, usando el backend de Keras.

```
# Funciones para calcular precisión, recall y F1 como métricas para los
# modelos:

  K <- backend()

  precision <- function(y_true, y_pred) {
    # Verdaderos positivos:
    true_positives <- sum(K$round(K$clip(y_true * y_pred, 0, 1)))

    # Posibles positivos:
    possible_positives <- sum(K$round(K$clip(y_true, 0, 1)))
```

```
    # Positivos predichos:
    predicted_positives <- sum(K$round(K$clip(y_pred, 0, 1)))

    # Precisión:
    precision <- true_positives / (predicted_positives + K$epsilon())

  return(precision)
  }

  recall <- function(y_true, y_pred) {
    # Verdaderos positivos:
    true_positives <- sum(K$round(K$clip(y_true * y_pred, 0, 1)))

    # Posibles positivos:
    possible_positives <- sum(K$round(K$clip(y_true, 0, 1)))

    # Recall:
    recall <- true_positives / (possible_positives + K$epsilon())

  return(recall)
}

  f1_score <- function(y_true, y_pred) {
    # Verdaderos positivos:
    true_positives <- sum(K$round(K$clip(y_true * y_pred, 0, 1)))

    # Posibles positivos:
    possible_positives <- sum(K$round(K$clip(y_true, 0, 1)))

    # Positivos predichos:
    predicted_positives <- sum(K$round(K$clip(y_pred, 0, 1)))

    # Precisión:
    precision <- true_positives / (predicted_positives + K$epsilon())

    # Recall:
    recall <- true_positives / (possible_positives + K$epsilon())

    # F1 score:
    f1_score <- 2 * precision * recall / (precision + recall +
                                          K$epsilon())
  return(f1_score)
}

# Asignación de los nombres de las funciones como atributos py_function_name:

attr(f1_score, "py_function_name") <- "f1_score"
attr(precision, "py_function_name") <- "precision"
attr(recall, "py_function_name") <- "recall"
```

RNN LSTM condicional

Se define una función para crear un modelo completo de red neuronal recurrente condicional, con entrada separada de las variables que cambian con el tiempo a la red LSTM y el resto a través de una capa de embedding.

```
# Parámetros para definir el modelo:

dynamic_features = 1 # Variables dinámicas (RESULTADO, Wavelets)
static_features = 4  # Variables estáticas (EDAD_BIN, PACIENTE_SEXO,
                     #  ANALIZADOR,CODIGO_PRUEBA)
lstm_units = 8      # Número de unidades en la capa LSTM
dense_u = 12 # Número de unidades de la capa densa.
vocabulary_size = length(unique(train_code$PACIENTE_SEXO)) +
  length(unique(train_code$EDAD_BIN))+
  length(unique(train_code$ANALIZADOR))+
  length(unique(train_code$CODIGO_PRUEBA))
# Número de niveles codificados de las variables estáticas.
epochs <- 15
optimizer <- "adam"
```

Función `conditional_lstm`.

```
# Función para crear el modelo:

conditional_lstm <- function(lstm_units, vocabulary_size, dense_u,
                             dynamic_features, static_features){
  # Modelo LSTM:

  # Capa de entrada de datos dinámicos:
  dynamic_input_layer <- layer_input(shape =
                                       c(lookback, dynamic_features),
                                     name = "dynamic_input_layer")
  #dynamic_input_layer <- layer_masking(mask_value =
                                         #-99999)(dynamic_input_layer)
  # Capa de entrada de datos estáticos:
  static_input_layer <- layer_input(shape = c(lookback, static_features),
                                    name = "static_input_layer")

   # Aplanado de la capa de datos estáticos antes del embedding:
  flatten_static_layer <- layer_flatten()(static_input_layer)

  # Capa  de embedding para los datos estáticos:
  embedding_layer <- layer_embedding(
    input_dim = vocabulary_size,
    output_dim = lstm_units)(flatten_static_layer)

  # Capa lambda para seleccionar los embeddings que modificarán los
  # estados ocultos de la capa LSTM:
  embedding_layer <- layer_lambda(f = \(x) x[,1,],
                          name = "lambda_embeddings")(embedding_layer)
```

```
  # Capa LSTM que recibe los valores de estado oculto de la capa lambda:

  lstm_layer_1 <- layer_lstm(units = lstm_units,
                             name = "lstm_1",
                             dropout = 0.3,
                             recurrent_dropout = 0.3,
                             stateful = T,
                             return_sequences = T)(dynamic_input_layer,
                                                   initial_state = list(
                                                     embedding_layer,
                                                     embedding_layer))
  lstm_layer_2 <- layer_lstm(units = lstm_units,
                             name = "lstm_2",
                             dropout = 0.3,
                             recurrent_dropout = 0.3,
                             return_sequences = F)(lstm_layer_1)
# Capa densa:
  dense_layer <- layer_dense(units = dense_u,
                             activation = "relu",
                             name = "dense_layer")(lstm_layer_2)

# Capa de salida con una unidad de activación lineal para la predicción:
  output_layer <- layer_dense(units = 1,
                              activation = "sigmoid",
                              name = "output_layer")(dense_layer)

  # Compilación:

  model <-
    keras_model(
      inputs  = list(dynamic_input_layer, static_input_layer),
      outputs = list(output_layer)
      ) %>%
    compile(
      optimizer = optimizer_adam(learning_rate = 0.01),
      loss      = loss_sigmoid_focal_crossentropy(alpha = 0.8,
                                                  gamma = 4,
                    reduction = tf$keras$losses$Reduction$AUTO),
      # binary_crossentropy
      list("binary_accuracy",
                  precision,
                  recall,
                  f1_score)
    )

  summary(model)

  # Guardar pesos modelo naïve


  save_model_weights_hdf5(model,
                          file.path(resultsdir, "naive_model.h5"),
                          overwrite = T)
```

```
  return(model)


}
```

Callbacks

```
# Callbacks

  callbacks <- list(
  callback_tensorboard(log_dir=file.path(resultsdir, "run_a")
                       # write_grads = T,
                       # histogram_freq = 1,
                       # update_freq = "batch",
                       # write_images = T
                       ),
  callback_model_checkpoint(file.path(resultsdir,
                                      "LSTM_samples_100"),
                            monitor = "f1_score",
                            mode = "max",
                            verbose=1,
                            save_best_only = T,
                            save_weights_only = T),
  callback_reduce_lr_on_plateau(monitor = "val_loss", factor = 0.1,
                                patience = 10, min_lr = 0.000001),
  callback_early_stopping(monitor = "val_loss", patience = 20,
                          verbose = 1)
  )
```

Training

tensorboard dev upload –logdir ./ –name “LSTM_samples” –description “LSTM_samples exp final”

```
set.seed(123)

# Crear arquitectura del modelo y guardarlo como modelo naive:
naive_model_final <- conditional_lstm(
  lstm_units = lstm_units,
  dense_u = dense_u,
  dynamic_features = dynamic_features,
  static_features = static_features,
  vocabulary_size = vocabulary_size)

 # Training:

  model_final <- naive_model_final %>%
    fit(wrap_train,
        epochs = epochs,
        steps_per_epoch = train_steps,
        validation_data = wrap_val,
```

```
        validation_steps = val_steps,
        callbacks = callbacks
        )
```

Evaluación del modelo LSTM

Carga del modelo:

Se definen distintas rutas para cargar el modelo en función de si se ha entrenado en RStudio o en Google Colab.

```
set.seed(123)

# Evaluación del modelo:
path <- file.path(resultsdir, "lstm_1.keras/f1_score/max/1/TRUE")
#path <- file.path(resultsdir, "GC_lstm_15.keras")

model <- load_model_hdf5(path,
                         #custom_objects =
                           list("precision" = precision,
                                "recall" = recall,
                                "f1_score" = f1_score),
                                 compile = T)
model %<>% compile(
  optimizer = "adam",
  #loss = "binary_crossentropy",
  loss = loss_sigmoid_focal_crossentropy(reduction = tf$keras$losses$Reduction$AUTO),
  metrics = list("binary_accuracy",
                 precision,
                 recall,
                 f1_score)
  )

# Cálculo de la evlaución del modelo a partir de la función generadora de # datasets de test:
evaluation_result <- evaluate(model, test_k, steps = test_steps)

# Obtención el nombre de las métricas:
metric_names <- names(evaluation_result)

# Crear un vector para almacenar los resultados
evaluation_results <- length(metric_names)

# Almacenar los resultados en el vector
for (i in 1:length(metric_names)) {
  evaluation_results[i] <- evaluation_result[[i]]
}
```

Línea base de sentido común

```r
# Función para calcular la exactitud mínima del modelo, basada en la
# predicción de todos los casos con el target más frecuente

evaluate_naive_method <- function(data_df, ctrl_colname) {
  # Cálculo de la moda de CTRL:
  ctrl_mode <- as.numeric(names(sort(table(
    data_df[[ctrl_colname]]),
    decreasing = TRUE)[1]))

  # Cálculo de accuracy:
  actual_ctrl <- data_df[[ctrl_colname]]
  predicted_ctrl <- rep(ctrl_mode, length(actual_ctrl))
  accuracy <- sum(actual_ctrl == predicted_ctrl) /
    length(actual_ctrl)

  # Cálculo de F1:
  f1 <- f1Score(actual_ctrl, predicted_ctrl)

  cat("accuracy =", accuracy, "F1score =", f1)
}

# Cálculo de accuracy basado en predecir con la moda:
accuracy_train <- evaluate_naive_method(train_n$scaled_data, "CTRL")
accuracy_val <- evaluate_naive_method(val_n, "CTRL")
accuracy_test <- evaluate_naive_method(test_n, "CTRL")

sprintf("Training naive accuracy: %.2f", accuracy_train)
sprintf("Validation naive accuracy: %.2f", accuracy_val)

# Comparación con rendimiento obtenido:


names(evaluation_results[3]) <- "f1_score"

# Imprimir los resultados de evaluación:

sink(file.path(resultsdir, "Conv_1.txt"))

for (i in 1:length(metric_names)) {
  metric <- metric_names[i]
  value <- evaluation_results[i]
  cat(sprintf("%s: %f\n", metric, value))
}

sprintf("Training naive accuracy: %.2f", accuracy_train)
sprintf("Validation naive accuracy: %.2f", accuracy_val)
sprintf("Test naive accuracy: %.2f", accuracy_val)

# Cerrar el archivo de salida
sink()
```

‘