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

2023-06-20

MODELO LSTM_onehot_QC

```
require(readr)
require(readxl)
require(purrr)
require(dplyr)
require(filesstrings)
require(stringr)
require(tidyr)
require(lubridate)
require(ggplot2)
#require(caret)
require(wavelets)
library(gridExtra)
require(tictoc)
require(reticulate)
require(abind)
require(tensorflow)
require(tfdatasets)
require(keras)
require(ModelMetrics)
require(tfruns)

require(R6)
```

```
# 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_qc`.

Leer los archivos xls de valores de QC para cada técnica y cada máquina, añadir una columna con la id del equipo, otra con el código CLC de la prueba y crear 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 `qc_level`.

Función para detectar el nivel de cada control de calidad basándose en el string que contiene su nombre (detecta 1, 2 ó 3 al final del string o bien las palabras "alto", "medio" o "bajo"). Una vez detectado el nivel genera un valor numérico 1, 2, ó 3.

```
qc_level <- function(df) {
  # Creamos una nueva columna NIVEL:
  df$NIVEL <- NA

  # Extraemos el último caracter del string:
  last_char <- substr(df$Control, nchar(df$Control), nchar(df$Control))

  # Condiciones para establecer el nivel basado en el nombre del control:
  df$NIVEL <- ifelse(df$CODIGO_PRUEBA == "CLC00544",
  # Corregir el caso específico donde de CLC00544,
  # donde CARDIAC BIORAD 3 corresponde al nivel 3, hs-TnI
  # alto al 2 y hs-TnI medio al 1:
                     ifelse(df$Control == "CARDIAC BIORAD 3", 3,
                            ifelse(df$Control == "hs-TnI alto", 2,
                                   ifelse(df$Control == "hs-TnI medio", 1, NA))),
                     ifelse(grepl("\\b(bajo|medio|alto)\\b", df$Control, ignore.case = TRUE),
                            ifelse(grepl("\\bmedio\\b", df$Control, ignore.case = TRUE), 1,
                                   ifelse(grepl("\\balto\\b", df$Control, ignore.case = TRUE), 2, 3)),
                            ifelse(last_char %in% c("1", "2", "3"), as.integer(last_char), NA))
  )

  # Se reordena el dataframe:
  df %<>% relocate("NIVEL", .after = 6)

  return(df)
}
```

Función `figure_ext`.

Función para extraer 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`.

Función para fusionar columnas Fecha y Hora en una nueva columna Fecha_hora y borrar 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 `read_event`.

Función para leer 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 merge_qc

Función para unir los dataframes features día D-1 y qc_results día D en uno solo mediante Left outer join.

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

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

  return(distinct(merged_df))

}
```

Función `wt` y `wavelet_tr`.

Funciones para aplicar la transformación wavelet a la variable QC_RESULT para cada grupo de técnica. La función `wt` calcula la transformación wavelet de tipo Maximal Overlap Discrete Wavelet Transformation (MODWT), que devuelve los coeficientes de wavelet y los coeficientes de escalado. La función `wavelet_tr` aplica la transformación a un conjunto de datos agrupado por una determinada variable.

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

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

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

  # 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 `normalize_train`.

Función de normalización de las variables continuas y eliminación de coeficientes wavelet con valores NA. Se definen dos funciones, una para el set de training y otra para los sets de test y validación. La diferencia es que la de train devuelve vectores de medias y de desviaciones estándar que luego se usan como argumentos en las otras. 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 <- "QC_RESULT"

  # 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`.

Función de normalización para los datos de validación y test, con eliminación de valores NA en los wavelets.

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

  # Identificar las columnas numéricas y seleccionarlas
  wavelet_cols <- grep("^[WV]", colnames(data), value = TRUE)
  result_col <- "QC_RESULT"
  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("QC_"), .after = 1) %>%
    relocate(starts_with("W"), .after = 2) %>%
    relocate(starts_with("V"), .before = ANALIZADOR) %>%
    mutate(across(wavelet_cols,
                          ~ifelse(is.na(.), -99999, .)))

  names(norm_data)[2]<-result_col

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

Función `code_cat_var`.

Función para convertir las variables categóricas tipo string en vectores numéricos y codificarlas (one hot), así como normalizar los valores de las variables continuas, 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)
    }

  # Reconversión de NIVEL a string:
```

```
  data$NIVEL <- as.character(data$NIVEL)
    # 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)
      }

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

Función `unique_lab_cases`.

Función para crear un código de casos únicos de combinaciones de ANALIZADOR, CÓDIGO_PRUEBA y NIVEL 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 `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 codificadas con factor_cat_var.

```
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)
}
```

Carga y preprocesado de los datos.

Lectura de archivos

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(events$NOMBRE_PRUEBA,
                               events$CODIGO_PRUEBA))
```

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

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

  length(which(is.na(iqtests$NOMBRE_PRUEBA)))
```

```
# Unir los dos dataframes qc_results y events en uno solo.
qc_event <- merge_qc_events(qc_results, events)

# Completar nombres de pruebas ausentes:
qc_event$NOMBRE_PRUEBA <-
    iqtests$NOMBRE_PRUEBA[match(qc_event$CODIGO_PRUEBA,
                                iqtests$CODIGO_PRUEBA)]
length(which(is.na(qc_event$NOMBRE_PRUEBA)))

head(qc_event)
```

Selección de características y transformación wavelet.

```
# Selección de características de interés y recodificación de variables:

qc_data_sel <- qc_event %>% subset(.,
                      select = c(TIEMPO_QC, QC_RESULT, ANALIZADOR,
                                 CODIGO_PRUEBA, NOMBRE_PRUEBA, NIVEL))%>%
  unique(.)
qc_data_sel %<>% arrange(TIEMPO_QC)
```

```
# Aplicación de transformación wavelet:
qc_data_sel_wl <- qc_data_sel %>% wavelet_tr()
```

Guardado de archivo csv.

```
# Guardado de datos en archivo csv.

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

Carga de datos desde archivo csv.

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

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

Modelo ARMA aplicado a los datos temporales.

```r
# Crear un archivo PDF para guardar los gráficos
pdf(file.path(figuresdir, "graficos.pdf"))

# Configurar la distribución de los gráficos en la página
par(mfrow = c(2, 3))

# Generar los gráficos
  qc_data_sel_wl %>%
  group_by(NOMBRE_PRUEBA, NIVEL) %>%
  nest() %>%
  mutate(acf_plots = map(data, ~ {
    acf(.x$QC_RESULT, type = "correlation", main="")
    title(main = unique(NOMBRE_PRUEBA),
          sub = paste0("NIVEL:", unique(NIVEL)),
          outer = F)
  })) %>%
  pull(acf_plots) %>%
  grid.arrange(grobs = ., ncol = 10)

# Restaurar los parámetros gráficos a su valor original
par(mfrow = c(1, 1))

# Cerrar el archivo PDF
dev.off()
```

División de los sets de entrenamiento, validación y prueba. Normalización.

```r
# 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(qc_data_sel_wl[[1]], 0.5)))

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

time_point_50
time_point_75
```

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

  # Normalización del dataset y eliminación de coeficientes
  # wavelet con valor NA:
  val_n <- val %>% norm_test_val(means = train_n$means, stds = train_n$stdev)
```

```r
# Muestra de test:
test <- qc_data_sel_wl %>%
  subset(.[[1]] > time_point_75)
  # Normalización del dataset y eliminación de coeficientes
  # wavelet con valor NA:
test_n <- test %>% norm_test_val(means = train_n$means, stds = train_n$stdev)
```

Codificación de las variables categóricas con enteros:

```r
# Conversión de strings a factores y codificación con enteros:
train_code <- factor_cat_var(train_n$scaled_data)

val_code <- factor_cat_var(val_n)

test_code <- factor_cat_var(test_n)
```

Creación de series temporales de datos de longitud determinada.

Las redes neuronales recurrentes requiren que los datos entren en forma de tensores con dimensión (n,k,p), siendo n el número de muestras, k la longitude una serie temporal y p el número de características o variables del modelo.

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 crea la función `create_lstm_data.2`, análoga a `create_lstm_data.1` pero adaptada para leer como variable objetivo el siguiente valor al último presente en la serie temporal de longitud k. Se crean funciones generadoras basadas esta función 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).

```
create_lstm_data.2 <-
  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
  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]]))
    targets <- array(0, dim = c(length(rows)))

    for (j in 1:length(rows)) {
      indices <- (rows[[j]] - lookback + 1):(rows[[j]])
      samples[j,,] <- data[indices, ]
      targets[[j]] <- data[rows[[j]] + delay,predseries]
    }
    list(samples, targets)
  }
}
```

Función generadora para dataset de training `train_k()`

```
# Con wavelets:
# train_data <- as.matrix(train_code[,-c(1,19)])
# Sin wavelets:
train_data <- as.matrix(train_code[,-c(1, 3:16, 19)])

lookback <- 50     # Número de pasos de tiempo que se toman (k).
batch_size <- 50 # Tamaño de batch de datos de la función generadora
delay <- 1         # Número de pasos de tiempo hacia el futuro para target
min_index <- 1     # Posición de inicio de las lecuras de series.
predseries <- 1    # Posición en la que se encuentra la variable target.

train_k <-  create_lstm_data.2(
    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)
```

Función generadora para dataset de validación `val_k()`

```
# Con wavelets:
# val_data <- as.matrix(val_code[,-c(1, 19)])
# Sin wavelets:
val_data <- as.matrix(val_code[,-c(1, 3:16, 19)])


val_k <- create_lstm_data.2(
    data = val_data,
    lookback = lookback,
    delay = delay,
    batch_size = batch_size,
    min_index = min_index,
    max_index = NULL,
    predseries = predseries)

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

Función generadora para dataset de training `test_k()`.

```
# Con wavelets:
test_data <- as.matrix(test_code[,-c(1, 19)])
# Sin wavelets:
test_data <- as.matrix(test_code[,-c(1, 3:16, 19)])

test_k <-  create_lstm_data.2(
    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 test:
test_steps <- round((nrow(test_data)-lookback) / batch_size)
```

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

Devuelven los tensores mencionados.

```r
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],  # Sin wavelets
         #seq[[1]][, , 1:15, drop = FALSE], # Con wavelets
         seq[[1]][, , -1]),  # Sin wavelets
         #seq[[1]][, , 16:18]), # Con wavelets
    list(seq[[2]])
  )

  return(tensor_list)
}
```

```r
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],  # Sin wavelets
         #seq[[1]][, , 1:15, drop = FALSE], # Con wavelets
         seq[[1]][, , -1]),  # Sin wavelets
         #seq[[1]][, , 16:18]), # Con wavelets
    list(seq[[2]])
  )

  return(tensor_list)
}
```

```r
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],  # Sin wavelets
         #seq[[1]][, , 1:15, drop = FALSE], # Con wavelets
         seq[[1]][, , -1]),  # Sin wavelets
         #seq[[1]][, , 16:18]), # Con wavelets
    list(seq[[2]])
  )

  return(tensor_list)
}
```

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 (QC_RESULT)
static_features = 3  # Variables estáticas (ANALIZADOR, CODIGO_PRUEBA Y
                       #NIVEL)
lstm_units = 32      # Número de unidades en la capa LSTM
dense_u = 24 # Número de unidades de la capa densa.
vocabulary_size = 32 # Número de niveles codificados de las variables
                     # estáticas.
epochs <- 13
optimizer <- "adam"
loss <- "mae"
```

Función `conditional_lstm`.

```
# Función para crear el modelo:

conditional_lstm <- function(lstm_units, dense_u, vocabulary_size,
                             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 dps valores de estado oculto de la capa de
# embeddings con la información de las variables estáticas:
lstm_layer_1 <- layer_lstm(units = lstm_units,
                           name = "lstm_1",
                           dropout = 0.3,
                           recurrent_dropout = 0.3,
                           #kernel_regularizer =
                           #regularizer_l2(l=0.01),
                           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 = 0.5,
                           recurrent_dropout = 0.3,
                           #kernel_regularizer =
                           #regularizer_l2(l=0.01),
                           return_sequences = F)(lstm_layer_1)

dense1 <- layer_dense(units = dense_u,
                      activation = "relu")(lstm_layer_2)


# Capa de salida con una unidad de activación lineal para la predicción:
output_layer <- layer_dense(units = 1,
                            activation = "linear",
                            name = "output_layer")(dense1)

# Compilación:
```

```
  model <-
    keras_model(
      inputs  = list(dynamic_input_layer, static_input_layer),
      outputs = list(output_layer)
      ) %>%
    compile(
      optimizer = optimizer,
      loss      = loss,
      metrics = list("mae",
                     "mse",
                     rmse
                     )
    )

  summary(model)

  # Guardar pesos modelo naïve


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



  return(model)
}
```

Métrica rmse.

Se define una función para calcular la métrica error cudrático medio (rmse) usando el backend de Keras y así poder usarla durante el entrenamiento y validación.

```
K <- keras::backend()

rmse <- function(y_pred, y_true) {
  sq_diff <- K$square(y_pred - y_true)
  mean_sq_diff <- K$mean(sq_diff)
  rmse_value <- K$sqrt(mean_sq_diff)
  return(rmse_value)
}

attr(rmse, "py_function_name") <- "rmse"
```

Callbacks

```
# Callbacks
  file_name <- paste0("lstm_QC_", 100, ".keras")

  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, file_name),
                            monitor = "rmse",
                            mode = "min",
                            verbose=1,
                            save_weights_only = T,
                            save_best_only = T),
  callback_reduce_lr_on_plateau(monitor = "val_loss", factor = 0.1,
                                patience = 5, min_lr = 0.000001),
  callback_early_stopping(monitor = "val_loss", patience = 20,
                          verbose = 1)
  )
```

```
# tensorboard dev upload --logdir ./ --name "LSTM_QC_cv" --description "LSTM_QC crossvalidation"
```

Ajuste del modelo condicional LSTM.

```
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
        )
```

Optimización

Se usa el paquete `tfruns` que permite la optimización múltiple de hiperparámetros y la visualización de los informes. Se ha preparado un script con flags que marcan los valores de los hiperparámetros y el código de definición, compilación y entranamiento del modelo, que después se usa en la función tuning_run para evaluar las combinaciones de valores de hiperparámetros.

```
library(tfruns)
set.seed(123)
runs <- tuning_run("LSTM_QC_script.R",
                   runs_dir = file.path(resultsdir, "1_lb_units_regul"),
                   sample = 1,
                   flags = list(
                     lookback = c(50),
                     batch_size = c(256),
                     lstm_units= c(
                      #8),
                      32),
                      #64),
                     dense_u= c(24),
                     dropout1 = c(0.3, 0.5),
                     recurrent_dropout1 = c(0.3),
                     dropout2 = c(0.3, 0.5),
                     recurrent_dropout2 = c(0.3)
                     #L2regularizer1 = c(0.01, 0.001)

))
library(kableExtra)
b <- kable(runs)
kable_styling(b)
runs[order(runs$eval_, decreasing = TRUE), ]
compare_runs(c(file.path(
  resultsdir, "1_lb_units_regul/2023-06-19T12-37-14Z"),
  file.path(
  resultsdir, "1_lb_units_regul/2023-06-19T11-54-23Z")))
```

Validación cruzada Day Forward Chain

Consiste en considerar cada unidad de tiempo del dataset como un punto para dividirlo, de manera que todos los puntos previos forman una cadena consecutiva de puntos que se usa como set de entrenamiento. Los puntos posteriores a ese set se utilizan como conjunto de test. A su vez, el set de entrenamiento se divide en un subset de entrenamiento y otro de validación y se entrena con ellos. En una nueva iteración se aumentará el tamaño de la cadena, repitiendo el proceso. Para simplificar la elección de los puntos donde se inicia la cadena, se usarán percentiles de la serite temporal en días, y se crearán funciones generadoras que tomarán min y max_index de manera acorde, realizando los entrenamientos todo en una sola función.

Creación de funciones generadoras para el Day Forward Chain

Cada función tendrá asignado un rango temporal de datos según un esquema incremental en el una vez dividido el espacio muestral en percentiles, se asignan los primeros al dataset de training y el último al de validación, incrementándose el dataset de training en cada ciclo, y manteniéndose el tamaño del de validación.

```
# Muestra de validación:
forward_chain_cv <- function(s){
  # quantiles dependientes del número de divisiones que se pase (s):
  q <- seq(1/s, 1, 1/s)

  # Cálculo de las fechas correspondientes a cada cuantil
```

```r
time_points <-
  do.call(quantile,
          list(as.POSIXct(as.Date(qc_data_sel_wl[[1]])), q))

# Listas para contener las funciones generadoras
train_ks <- list()
val_ks <- list()

# Vectores para guardar el número de steps de cada cadena:
train_cvsteps <- c()
val_cvsteps <- c()

# Loop a lo largo de los percentiles de tiempo:
for(i in 1:(length(time_points)-1)){

# Workflow aplicado a cada forward chain:

  train_n <- qc_data_sel_wl %>%
    subset(.[[1]] < time_points[i]) %>%
    normalize_train(.)

  val_n <- qc_data_sel_wl %>%
    subset(.[[1]] > time_points[i] & .[[1]] <= time_points[i+1]) %>%
    norm_test_val(., means = train_n$means, stds = train_n$stdev)

  ######################################################################

  train_code <- factor_cat_var(train_n$scaled_data)
  val_code <- factor_cat_var(val_n)

  ######################################################################

  # Crear una lista de funciones generadoras para cada cadena:

  # Función generadora de train:
  train_k <- create_lstm_data.2(
      data = as.matrix(train_code[,-c(1, 3:16, 19)]),
      lookback = lookback,
      delay = delay,
      batch_size = batch_size,
      min_index = 1,
      max_index = NULL,
      predseries = predseries)

  train_cvsteps <- rbind(train_cvsteps,
                         round((nrow(train_data)-lookback) /
                                 batch_size))

  # Wrapper de train:
  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)
}

    ######################################################################

    # Generadora de val:
    val_k <- create_lstm_data.2(
        data = as.matrix(val_code[,-c(1, 3:16, 19)]),
        lookback = lookback,
        delay = delay,
        batch_size = batch_size,
        min_index = 1,
        max_index = NULL,
        predseries = predseries)

    val_cvsteps <- rbind(val_cvsteps,
                         round((nrow(val_data)-lookback) / batch_size))

    # Wrapper de validación:

    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]][, lookback, -1]),
```

```
    list(seq[[2]])
  )

  return(tensor_list)
}

    # Guardado de las funcione wrapper en las listas correspondientes:

    train_ks[[i]] <- wrap_train

    val_ks[[i]] <- wrap_val

    train_ks %>% setNames(paste0("train_", i, sep = ""))
  }

  chain <- list(train_ks, val_ks)
  names(chain) <- c("train", "val")
  chain$trainst <- train_cvsteps
  chain$valst <- val_cvsteps

  return(chain)

}


chain <- forward_chain_cv(s = 4)
```

Entrenamiento y validación con las funciones generadoras Day Forward Chain:

```
history <- list()

for (i in 1:length(chain$train)) {

  # Generamos un modelo naive sin entrenar:

  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)

  # Nombre del archivo donde se guardarán los weights:
  current_time <- format(Sys.time(), "%Y%m%d%H%M%S")
  file_name <- paste0("lstm_QC_cv",current_time, ".keras")

    # Entrenamos
  history[[i]] <- naive_model_final %>%
    fit(
      chain$train[[i]],
      steps_per_epoch = chain$trainst[[i]],
```

```
      epochs = epochs,
      validation_data = chain$val[[i]],
      validation_steps = chain$valst[[i]],
      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, file_name),
                            monitor= "rmse",
                            mode="min",
                            verbose=1,
                            save_best_only = TRUE,
                            save_weights_only = TRUE),
  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)
    )
  )

  plot(history[[i]])
}
```

Evaluación

Línea base de sentido común `evaluate_naive_method`.

Función para calcular el MAE promedio para un modelo basado en predecir los targets del set de validación con las mismas muestras de validación, mediante la iteración de lotes de muestras. Sería equivalente a un modelo que predice el resultado QC a día D+1 con el resultado a día D, y constituye el punto de partida para mejorar (common sense baseline).

```
evaluate_naive_method <- function() {
  batch_mae <- c()
  batch_mse <- c()
  batch_rmse <- c()
  for (step in 1:val_steps) {
    c(samples, targets) %<-% test_k()
    preds <- samples[,dim(samples)[[2]],2]
    mae <- mean(abs(preds - targets))
    mse <- mean((preds - targets)^2)
    rmse <- sqrt(mse)
    batch_mae <- c(batch_mae, mae)
    batch_mse <- c(batch_mse, mse)
    batch_rmse <- c(batch_rmse, rmse)
  }
  naive_mse <- mean(batch_mse)
  naive_mae <- mean(batch_mae)
  naive_rmse <- mean(batch_rmse)
```

```
  cat("naive_mae", mae, "\n",
      "naive_mse", mse, "\n",
      "naive_rmse", rmse)
  naive <- list(naive_mae, naive_mse, naive_rmse)
  names(naive) <- c("mae", "mse", "rmse")
  return(naive)

}
```

Evaluación del modelo.

```
set.seed(123)
# Evaluación usando el dataset de test:

# Reconstrucción del modelo a partir de los pesos guardados:
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)

model <- load_model_weights_hdf5(naive_model_final, filepath =
                                   file.path(resultsdir,
                                      "lstm_QC_cv20230619193025.keras"))

evaluation_result <- evaluate(model, wrap_test, steps = test_steps)

evaluation_result

# Modelo naive:
naive <- evaluate_naive_method()
```

Evaluación de los resultados de la validación cruzada.

```
set.seed(123)

# Evaluación del modelo:
# Rutas hacia las carpetas de resultados de crossvalidation:
folders <- list.files(resultsdir, recursive = TRUE,
                        full.names = TRUE)

matching_folders <- grep(".*cv\\.keras*", folders, value = TRUE)

# Loop para obtener un informe de los modelos de cross validation:
cross_validation <- list()

for (folder in matching_folders) {
```

```
  # Reconstrucción del modelo a partir de los pesos guardados:
  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)

  model <- load_model_weights_hdf5(naive_model_final, filepath = folder)

  # Evaluación del modelo
  evaluation_result <- evaluate(model, wrap_test, steps = test_steps)

  # Guardar resultados de evaluación en la lista
  folder_name <- substr(folder, nchar(folder) - 4, nchar(folder))
  metrics_name <- paste0("metrics_", folder_name)
  cross_validation[[metrics_name]] <- evaluation_result

  # Comparación del modelo LSTM y el Naive a través del error absoluto medio
  mae <- evaluation_result[2]

  eval_model <- data.frame(
    CODIGO_PRUEBA = train_n$stdev[[2]],
    ANALIZADOR = train_n$stdev[[1]],
    SD = train_n$stdev[[3]],
    NAIVE_MAE_SD = naive$mae * train_n$stdev[[3]],
    MAE_SD = mae * train_n$stdev[[3]]
  )

  colnames(eval_model) <- c("CODIGO_PRUEBA", "ANALIZADOR", "SD", "NAIVE_MAE_SD", "MAE_SD")

  # Guardar resultados de comparación en la lista
  eval_model_name <- paste0("eval_model_", folder_name)
  cross_validation[[eval_model_name]] <- eval_model
}

  # Imprimir los resultados de evaluación:

  sink(file.path(resultsdir, paste0("lstm_QC", format(Sys.time(),
                                    "%Y%m%d_%H%M%S"), ".cv")))

  cat("\nCross Validation Results:\n")
  table_html <- kable(cross_validation, format = "html") %>%
  kable_styling(bootstrap_options = "striped", full_width = FALSE)

# Escribir la tabla en el archivo de salida
cat(capture.output(table_html))

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

Evaluación por categorías.

Se hace una selección con interfaz gráfica mediante la función `menu()`y se asigna una función generadora a esa combinación. Posteriormente se generan las predicciones y las métricas para dicha combinación, mostrando un gráfico de correlación.

Interfaz gráfica para la selección de combinaciones de prueba, analizador y nivel de QC.

```
# Función para seleccionar los índices en test_indx que corresponden a
# una combinación determinada de CODIGO_PRUEBA, NIVEL Y ANALIZADOR:

make_selection <- function() {

  # Opciones para elegir CLC, NIVEL y ANALIZADOR:
  iqtest_opt <- iqtests[match(unique(test_n$CODIGO_PRUEBA),
                              iqtests$CODIGO_PRUEBA),]
  level_opt <- as.character(unique(test_n$NIVEL))
  anal_opt <- unique(test_n$ANALIZADOR)

  # Introducción de prueba, nivel y analizador:
  clc <- iqtest_opt[menu(iqtest_opt[,1],
                      title = "Nombre de la prueba: ",
                      graphics = T),2]

  #level <- level_opt[menu(level_opt,
   #                       title = "Nivel de QC: ",
    #                      graphics = T)]
  #anal <- anal_opt[menu(anal_opt,
   #                     title = "Analizador: ",
    #                    graphics = T)]


  df <- test_n[test_n$CODIGO_PRUEBA == clc,]
                #test_n$ANALIZADOR == anal &
                #test_n$NIVEL == level, ]

  return(df)
}
```

Selección de variables y creación de función generadora selectiva.

```
test_n_i <- make_selection()
test_n_i %>% arrange(TIEMPO_QC)
test_code_i <- factor_cat_var(test_n_i)
test_data_i <- as.matrix(test_code_i[,-c(1, 3:16, 19)])

test_k_i <-  create_lstm_data.2(
    data = test_data_i,
```

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


# Carga del modelo.

# Reconstrucción del modelo a partir de los pesos guardados:
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)

model <- load_model_weights_hdf5(naive_model_final,
                                 filepath = file.path(resultsdir,
                                                      "LSTM_QC_100.keras"))

# Predicciones y métricas


# Obtener las series y sus predicciones utilizando la función generadora
# de tests por categorías y obtener la predicción del modelo:
seq <- test_k_i()
x <- list(seq[[1]][, , 1, drop = FALSE],
          seq[[1]][, , -1])
y_true <- seq[[2]]
y_pred <-  predict(model, x = x)


metrics <- list(
  MSE <- mse(actual = y_true, predicted = y_pred),
  MAE <- mae(actual = y_true, predicted = y_pred),
  Correlación <- cor(y_pred, y_true)
)
names(metrics) <- c("MSE", "MAE", "Coef. correlación")
metrics

# Crear el gráfico de dispersión entre y_pred e y_true
plot(y_true, y_pred, pch = 16, col = "blue", xlab = "y_true", ylab = "y_pred",
     main = "Gráfico de dispersión: y_true vs. y_pred")

# Calcular el coeficiente de correlación
correlacion <- cor(y_true, y_pred)

# Agregar el coeficiente de correlación al gráfico
texto_cor <- paste("Correlación:", round(correlacion, 2))
mtext(texto_cor, side = 3, line = -2.5)
```

```r
## Gráfico de reales y predichos vs tiempo.

num_prediccion <- seq_along(y_pred)

# Obtener los límites de los ejes y
y_min <- min(c(y_pred, y_true))
y_max <- max(c(y_pred, y_true))

# Graficar los valores predichos y reales con límites de ejes y ajustados
jpeg(" .jpg", quality = 90)

plot(num_prediccion,
     y_pred, type = "b", col = "blue", pch = 16,
     xlab = "Número de predicción",
     ylab = "Valor",
     ylim = c(y_min, y_max))

points(num_prediccion,
       y_true,
       type = "b",
       col = "red",
       pch = 16)

legend("topright", legend = c("Predichos", "Reales"),
       col = c("blue", "red"), pch = 16)

# Finalizar la salida del archivo JPEG
dev.off()
```