Técnicas de Simulación

08. Calidad de imagen

Raúl Luna del Valle

Contents

Antes de empezar a trabajar, puedes usar este código para cargar las dependencias de R que necesitaremos

1 Introducción

Cuando Netflix empezó a emitir contenido para millones de personas, descubrió que las métricas clásicas que usaban para medir la calidad del vídeo decían que todo se veía perfecto… pero los usuarios no opinaban lo mismo. Había vídeos borrosos, texturas que desaparecían y escenas oscuras que parecían bloques de sombras, mientras que las métricas seguían marcando "calidad excelente".

Cuenta la gente del equipo que hubo un momento decisivo: en una reunión alguien dijo "estamos optimizando para las métricas, no para las personas". Y tenían razón. Las métricas antiguas decían que un vídeo era excelente… mientras que a simple vista se veía fatal. Así nació VMAF, un algoritmo entrenado usando miles de comparaciones hechas por evaluadores humanos. Durante meses, empleados voluntarios pasaban tiempo viendo vídeos casi iguales y diciendo cuál se veía mejor. Muchos bromeaban con que después de eso ya no podían ver una serie sin fijarse en cada defecto de la imagen.

Con el tiempo, VMAF se volvió tan fiable que Netflix decidió liberarlo públicamente. Muchos se sorprendieron, pero la lógica era simple: lo importante no era el algoritmo en sí, sino todo el sistema que Netflix había construido alrededor para ajustar la calidad del vídeo en tiempo real.

Hoy VMAF es una referencia mundial, usado incluso por competidores. Y todo empezó porque un algoritmo decía "perfecto" mientras la gente decía "esto no se ve bien".

En este tema no trabajaremos en la predicción de calidad de vídeos, sino de imágenes individuales. No obstante, la anterior anécdota sobre Netflix refleja a la perfección la necesidad de desarrollar algoritmos de predicción de calidad de imagen que reflejen fielmente los juicios que observadores humanos tendrían sobre la calidad de las mismas imágenes.

Y es que la calidad de una imagen no depende unívocamente de sus propiedades físicas, como la resolución, el contraste o el nivel de ruido, sino también de cómo el sistema visual humano interpreta esa información. En muchos contextos —desde el procesado digital hasta la compresión, la transmisión o el análisis automático de imágenes— resulta fundamental disponer de métodos que permitan predecir de forma fiable cómo de "buena" o "degradada" será percibida una imagen sin necesidad de realizar costosas evaluaciones con observadores humanos. Los algoritmos de predicción de calidad de imagen cumplen precisamente este propósito: estiman la calidad perceptual a partir de características objetivas, ya sea comparando la imagen con una referencia ideal (métodos "full reference") o evaluando la señal por sí misma cuando no existe tal referencia (métodos "no reference"). Estos algoritmos integran principios de visión humana, métricas estadísticas y modelos computacionales avanzados para aproximarse al juicio perceptivo.

2 Procedimiento de predicción de calidad de imagen "Full reference"

El procedimiento full reference para la predicción de calidad de imagen se basa en la idea de disponer siempre de una imagen original intacta que sirve como referencia, junto con versiones degradadas de esa misma imagen. Estas degradaciones pueden provenir de muchos tipos de distorsiones: compresión excesiva (artefactos tipo bloque), desenfoque, ruido añadido, pérdida de contraste, banding, distorsiones cromáticas, entre otras. Abajo tienes un ejemplo, a la izquierda, de imagen de referencia. Las imágenes a su derecha van añadiendo cada vez mayor nivel de distrosión.

Existen bases de datos de calidad de imagen en las que existe una vasta cantidad de ejemplares de imágenes de referencia así como versiones distorsionadas de las mismas. En estas bases, tanto las imágenes de referencia como sus versiones deterioradas han sido valoradas por observadores humanos, obteniéndose para cada una de ellas puntuaciones de calidad perceptual.

A partir de estas valoraciones, se calcula una medida llamada DMOS (Differential Mean Opinion Score), que representa la diferencia entre la calidad percibida de la imagen original y la calidad percibida de cada imagen degradada. Es decir, mide cuánto empeora una imagen respecto a su referencia. De ahí el nombre "full reference": siempre existe una imagen original impecable contra la que comparar.

El siguiente paso consiste en aplicar un algoritmo de predicción de calidad de imagen que, del mismo modo, compara cada imagen degradada con su referencia y produce su propio valor de degradación (su propio "DMOS" estimado). Finalmente, se analiza hasta qué punto los DMOS generados por el algoritmo coinciden con los DMOS derivados de observadores humanos. Cuanto mayor sea la correlación entre ambos, mejor es la capacidad del algoritmo para predecir la calidad tal y como la percibe una persona real. Más adelante, haremos nosotros mismos este procedimiento y veremos en qué consiste todo esto.

3 Un algoritmo clásico: PSNR

El PSNR (Peak Signal-to-Noise Ratio) es una de las métricas más clásicas para evaluar la calidad de una imagen degradada comparándola con su versión original. Funciona de forma muy sencilla: toma ambas imágenes y mide, píxel a píxel, cuánto difieren entre sí. Primero calcula el error cuadrático medio entre ellas —una medida de cuánta "distorsión" se ha introducido— y con ese valor obtiene el PSNR, que básicamente indica cuánta señal útil queda frente al "ruido" provocado por la degradación. Un PSNR alto significa que la imagen degradada se parece mucho a la original, mientras que un valor bajo indica que ha sufrido cambios importantes.

El PSNR se define como:

$$\text{PSNR} = 10 \log_{10} \left(\frac{MAX_I^2}{\text{MSE}} \right)$$

donde

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (I_{\text{ref}}(i) - I_{\text{dist}}(i))^2$$

y MAX_I es el valor máximo posible de un píxel (en nuestro caso, $MAX_I 2 = 255$).

3.1 Tarea 1

Construye una función llamada "psnr" que tome dos matrices: "ref" (la imagen de referencia) y "dist" (la imagen distorsionada). Haz que la función calcule PSNR a partir de esas matrices. Carga las imágenes "I03.bmp" (la imagen de referencia) y "i03_01_5.bmp" una versión distorsionada de la misma, y calcula PSNR con la función que has hecho y muestra su valor. Aunque no es estrictamente necesario, cuando vayas a probar PSNR sobre tus imágenes, convierte estas antes a escala de grises usando la función "grayscale()". Puede que para ello necesites el paquete "imager".

```
if (!require(imager)) install.packages("imager", quiet = TRUE)
library(imager)
```

Nota: Verás que en la fórmula de PSNR, MSE se sitúa en el denominador, por lo que esto dará problemas cuando MSE=0. Programa tu función de tal manera que si MSE tuviera un valor de 0, se asigne en su lugar un valor de MSE=0.0001.

El principal inconveniente de PSNR es que, aunque es muy fácil y rápido de calcular, no se ajusta bien a la percepción humana: puede otorgar buena puntuación a imágenes que, a simple vista, se ven claramente deterioradas. Por eso se sigue usando como referencia básica, pero no como métrica fiable de calidad perceptual. En este tema intentaremos hacer nuestras propias predicciones de calidad de imagen con un algoritmo inspirado en la percepción humana.

4 Ejemplo de base de datos de calidad de imagen: TID2013

La base de datos TID2013 (Tampere Image Database 2013) es un conjunto ampliamente usado en investigación para evaluar la calidad perceptual de imágenes. Fue creada para probar y comparar algoritmos que intentan medir la calidad tal como la perciben los humanos. La colección incluye 25 imágenes de referencia sin distorsiones y 3000 imágenes distorsionadas generadas aplicando 24 tipos distintos de distorsiones en cinco niveles de severidad. Cada una de estas imágenes tiene un valor MOS (Mean Opinion Score) obtenido

mediante experimentos con observadores humanos, lo que permite usar esta base como un estándar confiable para validar modelos de calidad de imagen.

Como se comentaba, TID2013 abarca una variedad muy amplia de distorsiones, como por ejemplo: ruido gaussiano que introduce variaciones aleatorias en los píxeles, desenfoque gaussiano que suaviza los detalles finos, artefactos de compresión JPEG y JPEG2000 que generan bloques o pérdidas de textura, ruido impulsivo que produce píxeles aislados extremadamente brillantes u oscuros, errores de transmisión que degradan la estructura de la imagen, cambios de contraste o saturación que alteran la apariencia global y aberraciones cromáticas que afectan la coherencia del color. Cada una de estas distorsiones se aplica en 5 niveles de severidad. Por ello, dado un total de 25 imágenes de refencia, 24 tipos de distorsiones diferentes y 5 niveles de distorsión, existen un total de 25x24x5=3000 imágenes distorsionadas. Gracias a esta diversidad y a la disponibilidad de evaluaciones subjetivas precisas, TID2013 es una herramienta fundamental para entrenar, probar y comparar métodos en visión por computador, especialmente aquellos que buscan estimar o mejorar la calidad de una imagen.

Antes comentábamos el procedimiento full reference para que mediante algoritmos de predicción de calidad de imagen podamos arrojar métricas de calidad. Pues bien, la base de imágenes TID2013 está pensada para poner a prueba algortimos full reference de calidad de imagen. Es posible obtener valores DMOS a partir de las valoraciones de calidad de imagen por parte de observadores humanos sobre las imágenes de referencia y sus versiones distorsionadas. Así, la base de datos TID2013 pone a nuestra disposición 3000 valores DMOS, fruto de comparar la calidad percibida de cada imagen distorsionada con su respectiva imagen de referencia. Este mismo procedimiento se puede ejecutar, esta vez empleando un algoritmo de predicción de calidad de imagen como PSNR sobre las propias imágenes, que TID2013 también pone a nuestra disposición. Es así como obtenemos valores DMOS estimados que correlacionaremos con los 3000 valores DMOS empíricos.

Dicho esto, hagamos pruebas de predicicón de calidad de imagen sobre TID2013. Puedes descargar la base de datos en este link: https://www.ponomarenko.info/tid2013.htm

Deberás obtener una carpeta llamada “tid2013” que situarás en el mismo directorio donde esté el script R que deberás ejecutar próximamente. Tienes el script a continuación. Lo que este hace es ir cargando las imágenes de referencia, y sus respectivas versiones distorsionadas. A partir de una cierta imagen de referencia y una de sus versiones distorsionadas, la idea es que calcule el valor de PSNR. No obstante, la función PSNR no ha sido definida en el código. Tendrás que usar la que tú mismo/a definiste en la anterior tarea y situarla donde dice: “# — Definir la función PSNR (Peak Signal to Noise Ratio) —”

No olvides especificar el directorio de trabajo donde se sitúa TID2013. En mi caso es:

```
setwd("E:\UCM\Docencia\Asignaturas\Tecnicas_Simulacion\TSimulacion_cambiosRLV\102.Calidad_imagen\Calida
```

pero en el tuyo será otro diferente.

Lo que el código que puedes ver a continuación hace es graficar un scatterplot con los valores DMOS empíricos y los estimados, y calcula su correlación de Pearson.

La variable “images_to_process” aglutina las imágenes de la base de datos, para las cuales existen imagen de referencia y versión distorsionada. Existen un total de 25, pero en el código solo se expresan las 11 primeras . La variable “distortion_types” contiene los diferentes tipos de distorsiones que se aplican a cada una de las 25 imágenes. Existen un total de 24 posibles distorsiones, pero el código solo tiene en cuenta las 4 primeras. La variable “distortion_levels” contiene los niveles de distorsióm, 5 en total, que se aplican en cada tipo de distorsión. Tal y como está definido, el código tarda en ejecutarse. Si deseas acortar su tiempo de ejecución para hacer pruebas, te sugiero reducir el número de elementos en “images_to_process” y “distortion_types”. Si quisieras, también podrías añadir los elementos que faltan en cada caso, pero ello conllevará un muy largo tiempo de ejecución.

```
rm(list = ls())

# --- Dependencias ---
```

```r
if (!require(imager)) install.packages("imager", quiet = TRUE)
library(imager)

# --- Parámetros ---
images_to_process <- c("I01", "I02", "I03", "I04", "I05", "I06", "I07", "I08",
                       "I09", "I10", "I11")
# reference images
distortion_types <- c("01", "02", "03", "04", "05", "06", "07")
# distortion types
distortion_levels <- 1:5
# levels 1 to 5

# --- Especificar directorio de trabajo ---
setwd("E:/UCM/Docencia/Asignaturas/Tecnicas_Simulacion/TSimulacion_cambiosRLV/102.Calidad_imagen/Calida

# --- Cargar archivo con MOS (Mean Opinion Scores) ---
mos_data <- read.table("mos_with_names.txt", stringsAsFactors = FALSE)
colnames(mos_data) <- c("MOS", "FileName")




# --- Definir la función PSNR (Peak Signal to Noise Ratio) ---------------------


#-------------------------------------------------------------------------------




# --- Inicializar dataframe de resultados ---
results <- data.frame(RefImage = character(),
                      DistortedImage = character(),
                      DistType = character(),
                      DistLevel = integer(),
                      MOS = numeric(),
                      DMOS = numeric(),
                      PSNR = numeric(),
                      stringsAsFactors = FALSE)

# --- Procesar cada imagen de referencia ---
for(ref_img_prefix in images_to_process) {

  # --- Conseguir imagen de referencia (En la base de datos, prefijos con letra mayúscula) ---
  ref_row <- mos_data[grepl(paste0("^", ref_img_prefix, "_"), mos_data$FileName), ]
  ref_row <- ref_row[grepl("_1\\.bmp$", ref_row$FileName), ]  # first distortion level
  if(nrow(ref_row) != 1)
    stop(paste("Reference image not found or multiple matches for", ref_img_prefix))

  ref_path <- file.path("distorted_images", ref_row$FileName)
  ref_img <- load.image(ref_path)
  ref_img_gray <- grayscale(ref_img)
```

```
  mos_ref <- ref_row$MOS  # MOS de la imagen de refenrecia

  # --- Loop sobre los tipos de distorsión y niveles ---
  for(d_type in distortion_types) {
    for(level in distortion_levels) {

      # Las imágenes distorsionadas comienzan con 'i' minúscula
      pattern <- paste0("^i", substr(ref_img_prefix, 2, 3), "_", d_type, "_", level, "\\.bmp$")
      row_idx <- grep(pattern, mos_data$FileName, ignore.case = TRUE)

      if(length(row_idx) == 0) next  # skip si no se encuantra la imagen
      row_idx <- row_idx[1]          # si hay varias, coger el primer match

      distorted_row <- mos_data[row_idx, ]
      distorted_path <- file.path("distorted_images", distorted_row$FileName)
      dist_img <- load.image(distorted_path)
      dist_img_gray <- grayscale(dist_img)

      # Calcular PSNR
      psnr_value <- psnr(ref_img_gray, dist_img_gray)

      # Calcular DMOS (MOS de la imagen de referencia - MOS de la imagen distorsionada)
      dmos_value <- mos_ref - distorted_row$MOS

      # Almacenar resultados
      results <- rbind(results,
                       data.frame(RefImage = ref_img_prefix,
                                  DistortedImage = distorted_row$FileName,
                                  DistType = d_type,
                                  DistLevel = level,
                                  MOS = distorted_row$MOS,
                                  DMOS = dmos_value,
                                  PSNR = psnr_value,
                                  stringsAsFactors = FALSE))
    }
  }

  # --- Incluir la imagen de referencia, cuyo DMOS = 0 ---
  ref_psnr <- psnr(ref_img_gray, ref_img_gray)
  results <- rbind(results,
                   data.frame(RefImage = ref_img_prefix,
                              DistortedImage = ref_row$FileName,
                              DistType = "ref",
                              DistLevel = 1,
                              MOS = mos_ref,
                              DMOS = 0,
                              PSNR = ref_psnr,
                              stringsAsFactors = FALSE))
}

# --- Scatter plot PSNR vs DMOS ---
plot(results$DMOS, results$PSNR,
     xlab = "DMOS (MOS referencia - MOS distorsionada)",
```

```
     ylab = "PSNR",
     main = "PSNR vs DMOS",
     pch = 19, col = "blue")

# --- Correlación de Pearson ---
correlation <- cor(results$DMOS, results$PSNR, method = "pearson")
title(sub = paste("Correlación de Pearson:", round(correlation, 3)))
print(paste("Correlación de Perason: PSNR vs DMOS:", round(correlation, 3)))

# --- Visualizar resultados ---
head(results)
```

4.1 Tarea 2

Ejecuta el código anterior, situando la función psnr que has hecho en la tarea anterior. Ejecuta el código y muestra el scatterplot resultante con la correlación de Pearson (el propio código ya hece esto). Tu único trabajo es incluir la función psnr y especificar correctamente el directorio de trabajo donde se sitúa TID2013). Ejecuta el código incluyendo 11 imágenes, los 7 primeros tipos de distorsión, y todos los niveles de distorsión:

images_to_process <- c("I01", "I02", "I03", "I04", "I05", "I06", "I07", "I08", "I09", "I10", "I11") distortion_types <- c("01", "02", "03", "04", "05", "06", "07") distortion_levels <- 1:5

Continuamos. Habrás visto que utilizando PSNR, la correlación que obtienes entre los valores del algoritmo con los DMOS empíricos se sitúa en torno a 0.5 (no importa si la correlación es negativa, nos importa el valor absoluto de esta correlación).Como se explicaba antes, PSNR no es ni mucho menos el mejor algoritmo de predicicón de calidad de imagen. Entre otras cosas, no reproduce las sutilezas de la percepción humana. Seguramente, si usamos un algoritmo que de alguna manera procese las imágenes en base a algún principio de la percepción humana, obtengamos mejores resultados.

La idea es traducir una imagen de referencia y una versión distorsionada de la misma a lo que llamamos un "espacio perceptivo". Es decir, procesamos estas imágenes mediante un proceso inspirado en el funcionamiento del sistema visual, lo cual devuelve estas mismas imágenes, pero habiendo sido alteradas de alguna manera. Decimos que estas imágenes se sitúan ahora en el "espacio perceptivo".

Una vez hecho lo anterior, debemos calcular un DMOS estimado. La lógica es la misma de siempre, comparar la imagen de referencia (ahora en el espacio perceptivo) con su versión distorsionada (también en el espacio perceptivo). Para esta tarea podemos usar el RMSE (Root mean squared error) entre ambas imágenes. Matemáticamente, definimos RMSE como:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

En esencia, esto se hace restando sobre la imagen de referencia en el espacio perceptivo, y_i, caracterizada por una matriz, la matriz de la imagen distorsionada, $haty_i$, también en el espacio perceptivo. Esto devuelve una nueva matriz de las mismas dimensiones que las anteriores. Seguidamente, cada uno de los elementos de la matriz (o píxeles de la imagen resultante de la resta) es elevado al cuadrado, lo cual devueleve una nueva matriz de las mismas dimensiones. Acto seguido, se calcula la maedia de todos los elementos de la matriz resultante. Esto devuelve un único valor. Finalmente, se calcula la raíz cuadrada de dicho valor. Este es nuestro valor RMSE entre la imagen de referencia y su versión distorionada.

R tiene su propia función "rmse()", la cual está incluida en el paquete "Metrics".

```
if (!require(Metrics)) install.packages("Metrics", quiet = TRUE)
library(Metrics)
```

Bastará con que hagas “rmse(imagen_referencia_espacio_perceptivo,imagen_de_distorsionada_espacio_perceptivo)”. Pero antes necesitamos ver cómo podemos traducir las imágenes a un espacio perceptivo. Posiblemente hayas pensado el realizar algún filtrado como los que veíamos al estudiar la convolución y la transformada de Fourier; filtrados basados en el procesamiento de neuronas de la corteza visual sintonizadas en frecuencia espacial y orientación. No irías mal encaminado/a, ya que esto supone una traducción a un espacio perceptivo. Pero te sugiero algo más sencillo aún:

Una imagen está compuesta por píxeles. Si traducimos esos píxeles a escala de grises, cada pixel está expresando un valor de luminancia. ¿Recuerda cómo la Ley de Stevens traducía valores de luminancia a brillo percibido? Te sugiero que, utilizando la ley de Stevens, conviertas cada pixel de una imagen (previamente convertida a escala de grises) a un valor de brillo percibido. Esta será la traducción que hagamos de las imágenes a un espacio perceptivo.

4.2 Tarea 3

Haz una función llamada “stevens_map” que tome una imagen (previamente convertida a escala de grises) y traduzca cada uno de sus píxeles a un nuevo valor de brillo percibido. Muy posiblemente podrás reciclar esta función de la última tarea que realizaste en el tema de “Ajuste de funciones”. Carga la imagen “I03.bmp”, conviértela a escala de grises usando la función “grayscale()”, y traduce sus príxeles a brillo percibido usando la ley de Stevens con parámetros beta=0.33 y k=1.1. Seguidamente, muestra la imagen “I03.bmp” en escala de grises, así como su versión en el espacio perceptivo, dada su conversión mediante la Ley de Stevens.

Nota: tu función “stevens_map” tomará tres parámetros de entrada, tal que: stevens_map(imagen, beta, k)

4.3 Tarea 4

Usa la función “stevens_map” que has construido anteriormente e intégrala en el código que tienes más abajo, justo el el hueco indicado por: “# — Definir la función PSNR (Peak Signal to Noise Ratio) —” Si ejecutas el código introduciendo en él la función, estarás probando la efectividad de la función “stevens_map” como método de predicción de calidad de imagen. Recuerda cómo antes habías hecho lo mismo para PSNR. Muestra el scatterplot resultante con la correlación de Pearson (el propio código ya hece esto). Tu único trabajo es incluir la función stevens_map y especificar correctamente el directorio de trabajo donde se sitúa TID2013). Ejecuta el código incluyendo 11 imágenes, los 7 primeros tipos de distorsión, y todos los niveles de distorsión:

images_to_process <- c(“I01”, “I02”, “I03”, “I04”, “I05”, “I06”, “I07”, “I08”, “I09”, “I10”, “I11”) distortion_types <- c(“01”, “02”, “03”, “04”, “05”, “06”, “07”) distortion_levels <- 1:5

A continuación se muestra el código sobre el que tienes que introducir la función “stevens_map”

```
rm(list = ls())

# --- Dependencias ---
if (!require(imager)) install.packages("imager", quiet = TRUE)
library(imager)

if (!require(Metrics)) install.packages("Metrics", quiet = TRUE)
library(Metrics)


# --- Parámetros ---
images_to_process <- c("I01", "I02", "I03", "I04", "I05", "I06", "I07", "I08",
                       "I09", "I10", "I11")  # reference images
```

```r
distortion_types <- c("01", "02", "03", "04", "05", "06", "07") # distortion types
distortion_levels <- 1:5                                        # levels 1 to 5

# --- Especificar directorio de trabajo ---
setwd("E:/UCM/Docencia/Asignaturas/Tecnicas_Simulacion/TSimulacion_cambiosRLV/102.Calidad_imagen/Calida

# --- Cargar archivo con MOS (Mean Opinion Scores) ---
mos_data <- read.table("mos_with_names.txt", stringsAsFactors = FALSE)
colnames(mos_data) <- c("MOS", "FileName")



# Función que mapea la luminancia con el brillo percibido de acuerdo con la
#función de Stevens

# --- Definir la función stevens_mao, que implemente la ley de Stevens ---------







# ------------------------------------------------------------------------------


# --- Inicializar dataframe de resultados ---
results <- data.frame(RefImage = character(),
                      DistortedImage = character(),
                      DistType = character(),
                      DistLevel = integer(),
                      MOS = numeric(),
                      DMOS = numeric(),
                      PSNR = numeric(),
                      stringsAsFactors = FALSE)

# --- Procesar cada imagen de referencia ---
for(ref_img_prefix in images_to_process) {

  # --- Conseguir imagen de referencia (En la base de datos, prefijos con letra mayúscula) ---
  ref_row <- mos_data[grepl(paste0("^", ref_img_prefix, "_"), mos_data$FileName), ]
  ref_row <- ref_row[grepl("_1\\.bmp$", ref_row$FileName), ]  # first distortion level
  if(nrow(ref_row) != 1)
    stop(paste("Reference image not found or multiple matches for", ref_img_prefix))

  ref_path <- file.path("distorted_images", ref_row$FileName)
  ref_img <- load.image(ref_path)
  ref_img_gray <- grayscale(ref_img)
  mos_ref <- ref_row$MOS  # MOS de la imagen de refenrecia

  # --- Loop sobre los tipos de distorsión y niveles ---
  for(d_type in distortion_types) {
    for(level in distortion_levels) {

      # Las imágenes distorsionadas comienzan con 'i' minúscula
```

```
      pattern <- paste0("^i", substr(ref_img_prefix, 2, 3), "_", d_type, "_", level, "\\.bmp$")
      row_idx <- grep(pattern, mos_data$FileName, ignore.case = TRUE)

      if(length(row_idx) == 0) next  # skip si no se encuantra la imagen
      row_idx <- row_idx[1]          # si hay varias, coger el primer match

      distorted_row <- mos_data[row_idx, ]
      distorted_path <- file.path("distorted_images", distorted_row$FileName)
      dist_img <- load.image(distorted_path)
      dist_img_gray <- grayscale(dist_img)

      # Calcular RMSE (entre imágenes mapeadas mediante Stevens)
      rmse_value <- rmse(stevens_map(ref_img_gray, beta = 0.33, k = 1.1),
                         stevens_map(dist_img_gray, beta = 0.33, k = 1.1))

      # Calcular DMOS (MOS de la imagen de referencia - MOS de la imagen distorsionada)
      dmos_value <- mos_ref - distorted_row$MOS

      # Almacenar resultados
      results <- rbind(results,
                       data.frame(RefImage = ref_img_prefix,
                                  DistortedImage = distorted_row$FileName,
                                  DistType = d_type,
                                  DistLevel = level,
                                  MOS = distorted_row$MOS,
                                  DMOS = dmos_value,
                                  RMSE = rmse_value,
                                  stringsAsFactors = FALSE))
    }
  }


  # --- Incluir la imagen de referencia, cuyo DMOS = 0 ---
  ref_rmse <- rmse(stevens_map(ref_img_gray, beta = 0.33, k = 1.1),
                   stevens_map(ref_img_gray, beta = 0.33, k = 1.1))
  results <- rbind(results,
                   data.frame(RefImage = ref_img_prefix,
                              DistortedImage = ref_row$FileName,
                              DistType = "ref",
                              DistLevel = 1,
                              MOS = mos_ref,
                              DMOS = 0,
                              RMSE = ref_rmse,
                              stringsAsFactors = FALSE))
}

# --- Scatter plot PSNR vs DMOS ---
plot(results$DMOS, results$RMSE,
     xlab = "DMOS (MOS referencia - MOS distorsionada)",
     ylab = "Steven's",
     main = "Steven's vs DMOS",
     pch = 19, col = "blue")
```

```r
# --- Pearson correlation ---
correlation <- cor(results$DMOS, results$RMSE, method = "pearson")
title(sub = paste("Correlación de Pearson:", round(correlation, 3)))
print(paste("Correlación de Perason: PSNR vs DMOS:", round(correlation, 3)))

# --- Visualizar resultados ---
head(results)
```

¡Enhorabuena! Has puesto a prueba un algoritmo de predicción de calidad de imagen basado en nuestro conocimiento sobre cómo el ser humano percibe el brillo. Este nueva método, que sí tiene en cuenta la percepción humana, consigue mejores resultados que PSNR. Piensa ahora en los parámetros que has usado para definir la función potencial de Stevens, beta = 0.33 y k = 1.1. Estos valores salen de literatura previa que ajustaba la función potencial de Stevens a datos empíricos, obteniéndose que los valores que lograban un mejor ajuste eran aquellos. Por tanto, los valores de beta = 0.33 y k = 1.1 son aquellos que desde un punto de vista empírico, mejor reproducen la percepción del brillo en humanos. Y al fin y al cabo, son humanas aquellas personas que han realizado los juicios de calidad de imagen el la base de datos TID2013. Muy posiblemente, otros valores distintos de los parámetros "beta" y "k" no lograrán una buena predicción de calidad de imagen, dado que estos no se ajustarían tan bien a la percepción humana. Prueba a mantener el valor de k = 1.1, pero esta vez usa beta = 0.5. ¿Empeora la correlación entre los DMOS empíricos y los estimados por el algoritmo?