Descifrado Mediante Algoritmo de Metropolis

Pablo Martín de Benito

2024-02-13

El objetivo que desarrollaremos en las siguiente páginas consiste en descifrar un mensaje encriptado a través del algoritmo de Metrópolis-Hasting con el cálculo de unos determinados scores.

El algoritmo de Metrópolis permite generar réplicas de distribuciones de probabilidad en muchas situaciones de interés.

Antes que nada, cabe destacar que la resolución del problema en cuanto al codigo se refiere ha sido escrito en inglés con el objetivo de lograr un enfoque más general, no obstante, toda explicación sobre este ha sido implementada en castellano.

DATOS PREVIOS NECESARIOS

Para el buen desarrollo de la práctica, vamos a utilizar los siguientes ficheros de datos que nos ayudarán al cálculo de scores.

```
bigrams <- read.csv("bigramas_espanol.txt", sep="")
words_length <- read.csv("long_palabras_espanol.txt", sep="")
trigrams <- read.csv("trigramas_espanol.txt", sep="")
colnames(trigrams) <- c("trigrama","frecuencia")
message <- paste(readLines("mensaje_cifrado.txt"), collapse="\n")

bigrams[,2] <- log(bigrams[,2],10)
bigrams <- na.exclude(bigrams)
trigrams[,2] <- log(trigrams[,2],10)
trigrams <- na.exclude(trigrams)
words_length[,2] <- log(words_length[,2],10)
```

- ***bigrams:*** Fichero de datos con gran cantidad de bigramas y su frecuencia en el lenguaje castellano.

```
##    bigrama frecuencia
## 1:      DE   6.409269
## 2:      ES   6.364248
## 3:      EN   6.356648
## 4:      EL   6.303755
## 5:      LA   6.254949
## 6:      OS   6.253985
```

- ***words_length:*** Fichero de datos con las frecuencias de las longitudes de las palabras que el lenguaje castellano.

```
##    longitud frecuencia
## 1:        1   6.640394
## 2:        2   7.435409
## 3:        3   7.137136
## 4:        4   6.836033
## 5:        5   6.907822
## 6:        6   6.863543
```

- ***trigrams:*** Fichero de datos con las frecuencias de gran cantidad de trigramas en el lenguaje castellano.

```
##    trigrama frecuencia
## 1:      DEL   5.874319
## 2:      QUE   5.868879
## 3:      ENT   5.826253
## 4:      ION   5.748396
## 5:      ELA   5.738016
## 6:      CON   5.729844
```

Al tratar con frecuencias extremadamente grandes, el cálculo posterior de los scores resultaría con números demasiado grandes, por esa razón se han transformado los datos con log10.

A continuación se presenta el mensaje objetivo a descifrar:

```
"HTSVBPX QTIVBHRXIXBSX XLNTBVBGVBGREX HVSBSXBPXIQVYRXIHTBSXBLTILRXILRVBCBSXB
XGRJRTIBXQHXBSX XLNTBRILGACXBGVBGREX HVSBSXBLVYERV BSXB XGRJRTIBTBSXBL
XXILRVBVQRBLTYTBGVBGREX HVSBSXBYVIRWXQHV BQAB XGRJRTIBTBQABL
XXILRVBRISRZRSAVGBCBLTGXLHRZVYXIHXBHVIHTBXIBPAEGRLTBLTYTBXIBP RZVSTBPT
BGVBXIQXMVIDVBGVBP VLHRLVBXGBLAGHTBCBGVBTEQX ZVILRVBHTSTBRISRZRSATBHRXIXBSX
XLNTBVBGVBGREX HVSBSXBTPRIRTIBCBSXBXKP XQRTIBXQHXBSX XLNTBRILGACXBXGBSXBITBQX
BYTGXQHVSTBVBLVAQVBSXBQAQBTPRIRTIXQBXGBSXBRIZXQHRJV BCB XLRER BRIWT
YVLRTIXQBCBTPRIRTIXQBCBXGBSXBSRWAISR GVQBQRIBGRYRHVLRTIBSXBW TIHX VQBPT BLAVGUARX
BYXSRTBSXBXKP XQRTI"
```

PREPARACIÓN DEL MENSAJE

Antes de calcular los scores necesarios, debemos preparar el mensaje de manera correcta pues el cálculo de scores se realiza con todas las palabras por separado para posteriormente realizar la suma de todos ellos.

Para ello vamos a utilizar la función que aparece a continuación ***decrypt***, que se encarga de traducir el mensaje dara una combinación de letras concreta.

Llevar a cabo dicha traducción es sencillo a través de fichero de funciones proporcionado en ***Funciones-Desencriptado*** en la carpeta correspondiente.

Contamos con las siguientes funciones:

- ***decodificador*** -> Transforma el mensaje según una permutación de símbolos del alfabeto.
- ***mensaje_a_numero*** -> Convierte una cadena (formada por letras mayúsculas y espacios) en la secuencia correspondiente de números entre 1 y 28 (A corresponde a 1,. . . , ' ' corresponde a 28.
- ***mensaje_a_letra*** -> Convierte una secuencia de números (entre 1 y 28) en la correspondiente cadena.

```
# Decrypt a message by a combination given.
decrypt<- function(message, combination){
  decrypted_message <- decodificador(mensaje_a_numero(message),combination)
  decrypted_message <- mensaje_a_letra(decrypted_message)

  return(decrypted_message)
}
```

Seguimos con la preparación del mensaje para el cálculo de scores.

Esta vez separamos el mensaje dado por espacios y los almacenamos en un vector.

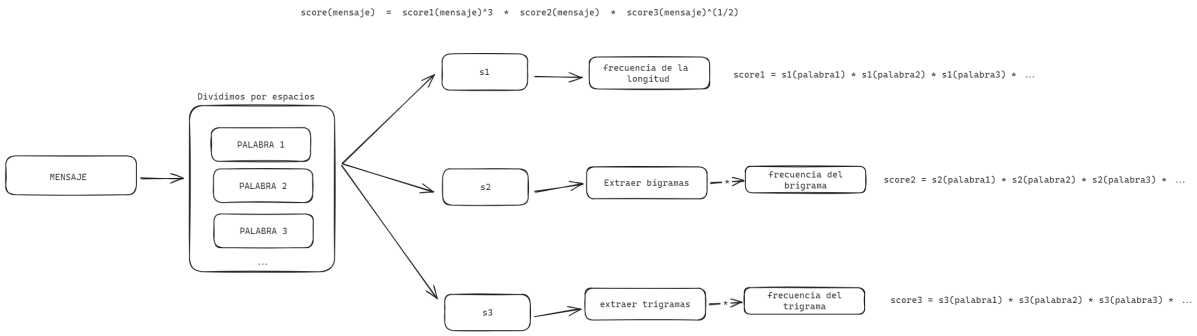
Utilizamos la sentencia ***unlist*** pues la función ***strsplit*** nos separa la cadena en una lista y sería un poco más complejo de tratar.

```
# Split a message by spaces.
split_message <- function(message){
  split_message <- strsplit(message,split=" ")
  split_message <- unlist(split_message)
  return(split_message)
}
```

CÁLCULO DE SCORES

A continuación, pasamos a la explicación del cálculo de scores.

Contamos con tres medidas que nos van a poder permitir cuantificar la lógica de una combinación de letras en el lenguaje español como indica el siguiente esquema.



El primer score (***Score1***) se refiere a la frecuencia de las longitudes de las palabras.

Contamos con un fichero que nos proporciona estos datos desde longitudes de 1 a 23.

El funcionamiento de dicha función es simple, extraemos la longitud de la palabra correspondiente y observamos si se encuentra en el archivo, si estuviera en el archivo, extraemos la frecuencia y se la sumamos al score1 total.

En el caso de que sea una longitud mayor que 23, le restamos 8 unidades, esto es porque es prácticamente imposible que una palabra tenga una longitud mayor que 23 por lo que le establecemos una penalización al score que prácticamente rechace la combinación.

```r
#### s1 Function #####
# Given a word, returns the frequency of that word's length.
# It has to be between 1 and 23, in any other case, the function will return 0.
s1 <- function(word){
  word_length <- nchar(word)
  if (word_length != 0 & word_length <=23){
    s1T <- words_length[word_length,2]
  }else{
    s1T <- -8
  }
  return(as.numeric(s1T))
}

#### Score 1 Function ####
# Given the split message, we sum all the s1 scores from every word of the message.
score1 <- function(split_message){
  score1T <- 0

  for (word in split_message){
    score1T <- score1T + s1(word)
  }
  return(score1T)
}
```

Para el segundo score, lo que haremos es sumar todas las frecuencias de los bigramas de una palabra, para ello rechazaremos las palabras que sean menores de dos caracteres.

No obstante, he implementado una condición que comprobará si, en el caso que sea una palabra de longitud 1, es una entre 'Y', 'A' o 'O'; sumaremos 8 unidades al score.

Esto lo he hecho porque existía casos donde la solución se atrapaba en las letras de longitud uno con una consonante o combinaciones extrañas que no se podían distinguir puesto que mis scores no lo contemplaban. Así, si es una de las letras más comunes del lenguaje, se sumará 8 afirmando que puede que esa letra sea esa dentro de la combinación que estemos.

Para todas las palabras, lo que haremos será ir recorriendo la palabra y cogiendo cada posible bigrama, obtener su frecuencia y sumarla a un total.

En el caso que tengamos un bigrama raro, que no esté en el fichero de bigramas, le restaremos 8 unidades al global.

```
#### s2 Function ####
# Given a word, returns the sum of the frequencys of all the possible bigrams the
# word has.
# Because of the frequency's file has been log-transformed, we are doing additions
# If a bigram not exists in the frequency's file, we will multiply by 10^-8, but
# because of the transformation, we substract 8 units.
s2 <- function(word){
  word_length <- nchar(word)
  s2T <- 0

  if (word_length < 2){
    if (word %in% c('Y','A','O')){
      return(8)
    }else return(0)
  }

  for (i in 1:(word_length-1)){
    bigram <- substr(word,i,i+1)
    if (bigram %in% bigrams$bigrama){
      s2T <- s2T + as.numeric(bigrams[bigrams$bigrama == bigram,2][1])
    }else{
      s2T <- s2T - 8
    }
  }
  return(s2T)
}

#### Score 2 Function ####
# Given the split message, we sum all the s2 scores from every word of the message
score2 <- function(split_message){
  score2T <- 0

  for (word in split_message){
    score2T <- score2T + s2(word)
  }
  return(score2T)
}
```

Haremos lo mismo que antes pero para los trigramas, es decir, rechazar las palabras con longitud menor que tres, recorrer cada palabra para obtener todos los posibles bigramas y sumar la frecuencia en caso de que este registrada en el fichero o restar 8 unidades en el caso de un trigrama raro.

```
#### s3 Function ####
# Given a word, returns the sum of the frequencys of all the possible trigrams the
# word has.
# Because of the frequency's file has been log-transformed, we are doing additions
# , instead of multiplications.
# If a bigram not exists in the frequency's file, we will multiply by 10^-8, but
# because of the transformation, we substract 8 units.
s3 <- function(word){
  word_length <- nchar(word)
  s3T <- 0

  if (word_length < 3){
    return(0)
  }

  for (i in 1:(word_length-2)){
    trigram <- substr(word,i,i+2)
    if (trigram %in% trigrams$trigrama){
      s3T <- s3T + as.numeric(trigrams[trigrams$trigrama == trigram,2][1])
    } else{
      s3T <- s3T - 8
    }
  }
  return(s3T)
}


#### Score 3 Function ####
# Given the split message, we sum all the s3 scores from every word of the message
score3 <- function(split_message){
  score3T <- 0

  for (word in split_message){
    score3T <- score3T + s3(word)
  }
  return(score3T)
}
```

Por último, obtenemos el score total, esto se consgue a través de la siguiente fórmula que ha sido proporcionada para la realización de la práctica.

$score(m) = 3score_1(m) + score_2(m) + (1/2)score_3(m)$

```
#### Score Function ####
# Given a message, this is split by spaces and calculated its specified score by a
# previous thought formula.
```

```
score <- function(message){
  split_message <- split_message(message)
  scoreT <- 3*score1(split_message) + score2(split_message) + (1/2)*score3(split_message)

  return(scoreT)
}
```

Algoritmo de Metrópolis-Hasting

Este algoritmo consiste en ir generando una CMTDH en donde se elige una matriz estocástica $Q = [q_{i,j}]_{i,j} \in E$ irreducible, persistente positiva, por lo que asumimos tambien $q_{i,i} = 0$.

Si $X_n = i$ entonces

- se genera un "canditadto" a un nuevo estado: j con probabilidad $q_{i,j}$.
- se acepta el candidato generado con probabilidad $\alpha_{i,j} = min(1, \pi_j q_{j,i}/\pi_i q_{i,j})$ con el caso de aceptación $X_{n+1} = j$ y en caso contrario $X_{n+1} = i$.

Así, en el código, lo que hacemos es crear un hashmap donde almacenaremos los scores.

Para este hashmap, utilizo una libreria llamada ***'r2r'*** que se encarga de implementar hashmaps de manera muy sencilla, así, lo inicializamos con todos los valores a cero (esto lo utilizaremos más adelante).

La primera combinación debemos escogerla de manera arbitraria entre todas las combinaciones posibles, por lo que la escogemos y calculamos su score inicial, ese será nuestro primer máximo score.

A partir de aqui comienza el algoritmo (5000 iteraciones más que suficiente, en principio), escogiendo el intercambio de dos letras entre las 28 posibles (contando el espacio).

Una vez escogido el cambio comprobamos que ese cambio no esté ya probado y almacenado en nuestra hashmap, si es así, básicamente pasará de iteración.

En el caso que no esté en nuestro hashmap, calcularemos su score correspondiente y lo almacenaremos en el hashmap y validaremos el cambio.

La validación del cambio se lleva a cabo teniendo en cuenta $\alpha_{i,j} = min(1, \pi_j q_{j,i}/\pi_i q_{i,j})$ y que hemos utilizado logaritmos en base 10.

En el caso de haberlo hecho con neperianos cambiaríamos la condición a e en vez de 10.

```
decrypt_message <- function(message){
  # Initialize the scores hashmap.
  scores <- hashmap(default = 0)

  # The first combination used, has to be random
  maxCombination <- sample(1:28)
  message_init <- decrypt(message,maxCombination)

  maxScore <- score(message_init)
  scores[[toString(maxCombination)]] <- maxScore
```

```
  # Metropolis-Hastings Algorithm
  N <- 5000
  for (iteration in 1:N) {
    swaps <- sample(1:28,2)
    newCombination <- maxCombination
    newCombination[swaps[1]] <- maxCombination[swaps[2]]
    newCombination[swaps[2]] <- maxCombination[swaps[1]]

    # Check if the current combination is not in the hashmap.
    if (scores[[toString(newCombination)]] == 0){
      newMessage <- decrypt(message,newCombination)
      newScore <- score(newMessage)
      scores[[toString(newCombination)]] <- newScore
    } else{
      newScore <- scores[[toString(newCombination)]]
    }

    # Validate the change.
    if (runif(1) < 10^(newScore - maxScore)){
      maxScore <- newScore
      maxCombination <- newCombination
    }

    if ((iteration %%  100) == 0){
      cat("Iteracion: ",iteration, ", Score actual: ",newScore, ", Score Max: "
          ,maxScore,"\n",decrypt(message,maxCombination),"\n\n")
    }
  }
  decryptedMessage <- decrypt(message,maxCombination)
  return(decryptedMessage)
}
```

Para ejecutar el programa solo habrá que llamar a la función pasando como argumento el mensaje que queremos descifrar.

```
decrypt_message(message)
```

Tras ejecutar el programa, habiendo puesto 5000 iteraciones, en este caso con 4000 iteraciones obtenemos el mensaje competamente descifrado, el cual es el siguiente.

```
Iteracion:  4000 , Score Max:  5285.592

TODA PERSONA TIENE DERECHO A LA LIBERTAD DE PENSAMIENTO DE CONCIENCIA Y DE
```

```
RELIGION ESTE DERECHO INCLUYE LA LIBERTAD DE CAMBIAR DE RELIGION O DE CREENCIA ASI
COMO LA LIBERTAD DE MANIFESTAR SU RELIGION O SU CREENCIA INDIVIDUAL Y
COLECTIVAMENTE TANTO EN PUBLICO COMO EN PRIVADO POR LA ENSEÑANZA LA PRACTICA EL
CULTO Y LA OBSERVANCIA TODO INDIVIDUO TIENE DERECHO A LA LIBERTAD DE OPINION Y DE
EXPRESION ESTE DERECHO INCLUYE EL DE NO SER MOLESTADO A CAUSA DE SUS OPINIONES EL
DE INVESTIGAR Y RECIBIR INFORMACIONES Y OPINIONES Y EL DE DIFUNDIRLAS SIN
LIMITACION DE FRONTERAS POR CUALQUIER MEDIO DE EXPRESION
```

Como podemos observar en la salida, obtenemos el mensaje sin problemas con un score total de ***5285.592*** según los criterior previamente explicados.

CONCLUSIONES

Una vez obtenido el mensaje descifrado y habiendo ejecutado el programa en multitud de ocasiones podemos destacar los siguientes puntos:

- Al tener suficientes criterios de cálculo de scores pero no demasiados como para entrar en detalles de optimización y seguridad, la obtención del resultado correcto no siempre está garantizado pues nos damos cuenta que estamos utilizando combinaciones con cambios de dos letras completamente arbitrarias en cada iteración lo que hace que en una ejecución te encuentre el mensaje descifrado en 1000 iteraciones o pasen 5000 y todavía no lo haya encontrado.
- Existen casos particulares que hacen que cambie el mensaje y aunque el score varíe muy poco. Me refiero al caso en el que aparece la letra *ñ*, la palabra *enseñanza* del mensaje descifrado causa conflicto ya que el score es aceptado utilizando la propia letra *ñ* o utilizando la letra *j* tal que *ensejanza*. He ejecutado varias veces y acepta los dos scores en bucle por lo que la probabilidad de escoger al terminar las iteraciones la combinación correcta es la misma.

Seguro el codigo e implementación del algoritmo se puede optimizar en mayor medida incluso utilizando otras estructuras de datos, etc; pero la consecución del objetivo está cumplido con éxito sin mayores problemas.