

# PROYECTO FINAL: SIMILITUD COSENO

RECUPERACIÓN DE INFORMACIÓN  
BEATRIZ BELTRÁN MARTÍNEZ  
LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN  
PRIMAVERA 2022

GUSTAVO IVÁN MOLINA REBOLLEDO  
000000000

Benemérita Universidad Autónoma de Puebla

**ABSTRACT.** In this paper, we present a Haskell program that calculates the similarity between every pair of given documents. The similarity is calculated using the cosine similarity. The documents are given as text files which are parsed and stored in a data structure. Then the program calculates the number of occurrences of each word in the documents and the respective weight for each word. All of this is done in Haskell except the stemming process which is done through the use of a C library. We also discuss the performance of the program and the possible improvements that could be made.

3 de mayo de 2022

## I. INTRODUCCIÓN

Este trabajo retoma las ideas que se han presentado en los trabajos anteriores. Sin embargo, se está tomando en cuenta una nueva organización del código, de forma que se espera una mejora con respecto a las entregas previas. Parte de estas mejoras es gracias a que se tiene una mejor visión de lo que se está realizando.

Al igual que en los documentos ya presentados, este trabajo se encuentra codificado en el lenguaje de programación Haskell. Se han hecho uso de varias librerías para la implementación de las funciones, mismas que se encuentran disponibles en Hackage. Esta elección tiene un punto negativo que será explicado más adelante.

## II. DISEÑO DEL PROYECTO

El programa se encuentra dividido en varias secciones internas: *Parsing*, *Stage0*, *Stage1*, *Stage2*. Cada una de estas secciones se encargan de llevar a cabo trabajos específicos del procesamiento de los documentos. Además, se encuentra un módulo *main* que se encarga de la ejecución del programa.

### 2.1. *Parsing*

Para el *Parsing* asumimos que el documento se encuentra previamente almacenado en una estructura de datos de tipo *Text*. Ya que este proceso es de naturaleza impura, no se puede realizar en esta sección; *main* se encargará de ello.

Nuestro punto de entrada para esta sección es la función *parseDocument*. Esta función se encarga de extraer los datos del archivo de texto y de preprocesarlos para que puedan ser utilizados por el resto de las secciones.

El primer paso es hacer uso de la librería *Text.XML.Light* para leer el archivo XML en una estructura manejable. El problema de hacer esto directamente es que la librería toma en cuenta los saltos de línea al momento de leer las etiquetas. Por lo tanto, se debe realizar un filtro para eliminar todos aquellos objetos que no sean elementos con información relevante.

Una vez que tenemos esta división de datos, se procede a dividir los datos en dos partes: la información del documento y el texto de la noticia. Esto se retorna en una lista de tuplas que puede

ser accedida por los siguientes pasos.

Lo siguiente a hacer es limpiar y preprocesar el texto de la noticia. Para ello eliminamos todos los números de la noticia, los signos de puntuación, las urls y se convierte el texto a minúsculas. Posteriormente, el texto es dividido en palabras y se filtra de forma que no se incluyan *stopwords*. Por último, se ensambla el texto en un solo *Text* y se da como entrada al algoritmo de *stemming*.

El algoritmo de *stemming* que estamos usando en este trabajo es un proyecto antiguo, además de que se trata de una biblioteca que hace *ffi* a una biblioteca en C. Una de las complicaciones al usar este algoritmo es que no se puede hacer uso de la librería de forma directa, sino que hay que modificar el código de la biblioteca para que funcione en un compilador de Haskell más moderno.

El algoritmo usado, a pesar de que es una biblioteca de C «antigua», compila sin problemas en Darwin moderno.

## 2.2. Stage0 y representación interna

Lo más interesante (e importante) de esta sección es que se ha diseñado un tipo producto para la representación de los datos que nos permite tener un mejor control y acceso en las etapas siguientes. Esta estructura está definida como se muestra a continuación:

```
data Documents = Documents {  
  __docsN :: Int,  
  __docs :: M.Map Int [(Int, Int)],  
  __weights :: M.Map Int [(Int, Float)],  
  __words :: [T.Text]  
} deriving (Show, Generic)
```

disponibles (*\_\_docsN*) y la lista de las palabras que se usan a lo largo de todos los documentos (*\_\_words*). En cuanto a *\_\_docs*, se define como una representación de índice invertido, pero manejando un *Map* para tener orden en el acceso. Esta última guarda las apariciones de cada palabra en cada documento (si el documento la contiene).

Tener *\_\_docs* y *\_\_weights* puede parecer un poco ineficiente. Técnicamente lo es, pero en este

proyecto se requiere para poder mostrar los datos calculados por el programa. Además, en un principio requerimos de `__docs` para poder tener suficiente información para el calculo de `__weights`. Por lo que pensamos que su uso (en este caso específicamente) es justificado.

Además se definen una serie de *lenses* y morfismos para tener un acceso más fácil a los datos de la estructura. Esto es una mejora muy grande con respecto a las versiones anteriores que se han realizado. Simplifica grandemente la claridad del código y reduce la necesidad de declarar más funciones.

En cuanto a la generación, en esta etapa sólo se inicializa la estructura con el número de documentos y la lista de palabras en orden. Además se establecen los mapas con listas vacías.

### 2.3. Stage1

En esta sección no hay necesidad de una explicación muy extensa, ya que se trata de una etapa que se encarga de llenar la estructura con el número de apariciones de cada palabra en cada documento. Simplemente suministramos el texto de cada documento y la estructura de datos que se encuentra en la etapa anterior. El resultado de esta etapa es una estructura de datos con la información actualizada.

### 2.4. Stage2

Aquí nos encargamos de calcular el peso de cada palabra en cada documento. En este caso basta con proporcionar sólo la estructura de datos de la etapa anterior, ya que contiene la información necesaria para calcular cada uno de los pesos.

#### 2.4.1. TF

Definimos la función *tf* que calcula la frecuencia de una palabra en un documento de la siguiente forma:

```
idf :: Int -> Documents -> Float
idf word doc = let
  n = fromIntegral $ doc ^. _docsN
  d = fromIntegral $ length $ doc ^. _docs &
```

```
M.lookup word & fromMaybe []
in logBase 2 $ n / d
```

#### 2.4..2. IDF

Definimos la función *idf* que calcula la frecuencia de una palabra en todos los documentos de la siguiente forma:

```
idf :: Int -> Documents -> Float
idf word doc = let
  n = fromIntegral $ doc ^. _docsN
  d = fromIntegral $ length $ doc ^. _docs &
    M.lookup word & fromMaybe []
  in logBase 2 $ n / d
```

#### 2.4..3. TF-IDF

Finalmente, definimos la función *tfidf* que calcula el pesado del termino de la siguiente forma:

```
tfidf :: Int -> Int -> Documents -> Float
tfidf word doc docs =
  tf word doc docs * idf word docs
```

#### 2.4..4. Peso

Simplemente basta con proveer de los indice necesarios (y de la estructura de datos de la etapa anterior) para calcular el peso de cada palabra. Esto se puede obtener con facilidad usando los lentes. Además de que la estructura de la función *tfidf* es más similar a su definición teórica.

Al acabar el procesor para todos los indices, tenemos una estructura actualizada con los valores para *\_\_weights* correspondientes a cada palabra.



## PROYECTO FINAL: SIMILITUD COSENO





## PROYECTO FINAL: SIMILITUD COSENO

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]





[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

#### IV. TABLA DEL PESO DE DOCUMENTOS Y PALABRAS



# PROYECTO FINAL: SIMILITUD COSENO

31





## PROYECTO FINAL: SIMILITUD COSENO

33



# PROYECTO FINAL: SIMILITUD COSENO

35



# PROYECTO FINAL: SIMILITUD COSENO

37



# PROYECTO FINAL: SIMILITUD COSENO

39





## PROYECTO FINAL: SIMILITUD COSENO

41



## PROYECTO FINAL: SIMILITUD COSENO

43



# PROYECTO FINAL: SIMILITUD COSENO

45



# PROYECTO FINAL: SIMILITUD COSENO

47





## PROYECTO FINAL: SIMILITUD COSENO

49



## PROYECTO FINAL: SIMILITUD COSENO

51

## V. TABLA DE LA SIMILITUD COSENO

[illegible]

## VI. EJECUCIÓN DEL PROGRAMA

El programa se ejecuta desde la línea de comandos haciendo uso de *ghci*, o bien, compilando el programa con *ghc*. El resultado de la compilación nos da un programa que es capaz de generar cada uno de los pasos y de escribir los datos en un archivo  $\text{\LaTeX}$  compilable.

Para más información, consulte el archivo *README.md*

## VII. CONCLUSIONES

Algo importante que se debe tener en cuenta es que el proceso de indexación es muy lento por la forma en la que se realiza el cálculo de los textos. Por lo que es recomendable que en una implementación más eficiente, se haga uso de estructuras que permitan el acceso de datos más rápido.

Haskell es un lenguaje de programación inmutable, por lo que no se puede modificar el contenido de una estructura de datos sin crear una nueva. Este es un problema ya que nos obliga a crear una nueva estructura para cada cambio que se realice.

Obviamente estos cambios afectan considerablemente al rendimiento de la aplicación. Sin embargo, siempre hay alternativas que podrían resolver este problema, como el uso de monadas, IO, estructuras ligadas a tipos lineales, etc.

Pero todas estas alternativas son muy complicadas y no son parte del alcance de este proyecto.

Otro punto a tomar en cuenta es que si bien el proceso de similitud debería ser muy rápido, esto no es lo que se ha observado. En realidad el cálculo de la similitud es bastante lento, siendo la parte más lenta de toda la aplicación.

Esto se debe probablemente a que hay que generar una gran cantidad de vectores. Un total de  $101 * 101$  resultados son generados, lo que es bastante grande y puede ser complicado de calcular.

Estos tiempos son aún más lentos cuando se ejecuta desde el comando *ghci*. Por lo que es recomendable que se compile el programa con *ghc* para que se ejecute más rápido la generación de los documentos.