TweetKell

Análisis de los datos privados de una cuenta de Twitter con Haskell

Universidad de Sevilla
Ingeniería Informática Tecnologías Informáticas
Programación Declarativa - Tercer curso

Juan Arteaga Carmona
Herrera, Sevilla, España
JuanArteaga@andalu30.me

4 de septiembre de 2019

Índice

Índice de figuras

1. Introducción

A continuación se presenta la documentación realizada para la entrega final de la convocatoria de Septiembre del trabajo de curso de la asignatura Programación Declarativa del grado de Ingeniería Informática - Tecnologías Informáticas de la Universidad de Sevilla.

2. Expectativas

Al comienzo del trabajo se pretendía obtener una versión final que cumpliese con los requisitos expuestos por nuestro profesor, los cuales se pueden observar en el apartado 3.

3. Criterios de evaluación

A continuación se muestran los criterios de evaluación a los que se someterá este trabajo, de acuerdo con la información que se encontraba en el aula virtual de la asignatura a día 2 de septiembre de 2019.

- La fecha de entrega del trabajo es el día 8 de septiembre hasta media noche. La defensa del trabajo tendrá lugar la semana siguiente, del 9 al 13 (se anunciará por este canal las fechas).
- El trabajo debe ser un programa de temática "libre"(dentro de los ofertados en la convocatoria de febrero, o los que se haya hablado con los profesores previamente), totalmente funcional. Para la superación del trabajo, éste debe contener como mínimo:
  - Dos aplicaciones de cada concepto básico de programación funcional visto en cada tema del primer bloque de la asignatura (temas 2,4,5,6,7,8). Es decir: al menos usar 2 funciones básicas de haskell, definir 2 funciones recursivas, 2 funciones por patrones, 2 usos de guardas, 2 usos de listas por comprensión, 2 usos de orden superior, 2 usos de tipados, etc.
  - Creación de un módulo
  - Creación de un tipo de datos nuevo y uso de éste.
  - Uso de al menos dos de los conceptos avanzados y librerías vistos en los temas 3, 9, 11, 12, 13, 14 o 15 (deben ser de temas distintos).
- La entrega debe contener, en un archivo comprimido (zip, máximo 50 mb):
  - El código fuente Haskell (incluyendo un pequeño readme explicando el uso del programa).
  - Un documento (formato libre) explicando la estructura del código y cómo se ha aplicado cada uno de los conceptos.

4. Elección del tipo de trabajo

Se decidió que el trabajo fuese uno completamente diferente al que se propuso en la primera convocatoria [1] ya que, tal y como se explicó en las conclusiones de este, elaborar un juego con Haskell y CodeWord no parecía ser una tarea sencilla ni interesante.

Así pues, este trabajo se centrara en la temática de Data Science y análisis de datos, con especial énfasis en la recolección de los datos a través de la librería Aeson [2] y en la interactividad del programa como se verá en el apartado 7.2.

5. Haskell

Para este proyecto utilizaremos en lenguaje de programación Haskell [3]. Haskell es un lenguaje de programación puramente funcional creado por Paul Hodax en 1990 [4].

6. Datos de Twitter

Al utilizar esta aplicación podremos extraer y consultar los datos de una cuenta de Twitter, para esto es necesario que dispongamos de los datos en un formato que nuestro PC pueda entender.

Para conseguir los datos deberemos de hacer a Twitter una petición para que nos permita descargarlos. Esta petición se hace a traves de los ajustes de privacidad en nuestro perfil y, dependiendo del tamaño de nuestra cuenta, en unos minutos se nos permitirá acceder a ellos. [5]

Una vez descargados, tendremos que descomprimirlos y ya podremos utilizar la aplicación.

7. Implementación

7.1. Estructura del código

El código se encuentra estructurado en varios módulos que, además de ser un requisito para la entrega del trabajo, permiten una mayor comodidad y simplicidad a la hora de desarrollar la aplicación. A continuación se describen los módulos:

- `TweeKell.hs`

  Archivo principal del programa, contiene una función main que llama al menu principal del módulo MenuInteractivo.

- `MenuInteractivo.hs`

  Módulo encargado de generar los menús interactivos del programa además de recibir las instrucciones del usuario y llamar a la lógica correcta.

- `TiposDatos.hs`

Módulo encargado de definir los tipos de datos que se utilizan para decodificar los archivos JSON y mostrar la información.

- `TiposDatos2.hs`

  Igual que TiposDatos, con la única diferencia de que este contiene las definiciones de los tipos de datos que producen alguna colisión en caso de que se encontrasen en TiposDatos.

- `ParsersArchivos.hs`

  Módulo encargado de parsear los archivos JSON de Twitter y de crear los objetos de los tipos que se definen en TiposDatos y TiposDatos2.

- `AnalisisSimples.hs`

  Módulo encargado de recibir el parseado realizado por ParsersArchivos y analizar y mostrar la información que el usuario ha solicitado para todos los tipos de análisis excepto para los que dependen de Tweets.

- `AnalisisTweets.hs`

  Igual que AnalisisSimples pero centrado en los que dependen del tipo Tweet.

7.2. Aplicación de los conceptos y ajuste a los criterios de evaluación

- **Aplicaciones de conceptos básicos de programación funcional.**

  - 2 funciones básicas de Haskell

    El uso de funciones básicas de Haskell se puede observar por todo el trabajo, por ejemplo, se puede observar en dos de las funciones mas simples y cortas:

```haskell
modPhoneFile :: String -> String
modPhoneFile string ="{" ++ (drop 50 $ reverse $ drop 4 $ reverse string)

modAccFile :: String -> String
modAccFile string = drop 45 $ reverse $ drop 3 $ reverse string
```

  - 2 funciones recursivas:

    A continuación podemos ver dos funciones recursivas de este trabajo, una de ellas se encarga de imprimir los textos de una lista de Tweets y otra se encarga de contar el número de apariciones de los distintos idiomas de los tweets.

```haskell
imprimeTextosTweets :: [String] -> IO()
imprimeTextosTweets [] = putStrLn "" --Base
imprimeTextosTweets (x:ls) = do  --Recursiva
  putStr "\n->"
  print x
  imprimeTextosTweets ls

cuentaIdiomas :: Foldable t => [t a] -> [Int] -> [Int]
cuentaIdiomas [] ac = ac
cuentaIdiomas (x:xs) ac = do
  let newac = ac ++ [(length x)]
  cuentaIdiomas xs newac
```

- 2 funciones por patrones

  Las dos funciones recursivas anteriores están, claramente, definidas por patrones. Sin embargo no son las únicas. Un ejemplo de otra función sería el siguiente:

```haskell
tipoAnalisis :: String -> [String]
tipoAnalisis "account.js" = ["1: Ver informacion", "2: Abrir en navegador"]
tipoAnalisis "profile.js" = ["1: Ver informacion","8: Ver icono de perfil"]
tipoAnalisis "verified.js" = ["1: Ver informacion"]
tipoAnalisis "phone-number.js" = ["1: Ver informacion"]
tipoAnalisis "tweet.js" = ["3: Ver tweets", "4: Ver Retweets", "5: Tweet mas retweeteado",
                           "6: Tweet con más MG","7: Idiomas mas utilizados"]
```

- 2 usos de guardas

  En cuanto a funciones con guardas, dos ejemplos podrían ser:

```haskell
verInformacion :: String -> String -> IO()
verInformacion ruta archivo
  | archivo == "account.js" = verInformacionAccount (ruta ++"/"++ archivo)
  | archivo == "profile.js" = verInformacionProfile (ruta ++"/"++ archivo)
  | archivo == "verified.js" = verInformacionVerified (ruta ++"/"++ archivo)
  | archivo == "phone-number.js" = verInformacionPhone (ruta ++"/"++ archivo)

abrirNavegador :: String -> String -> IO()
abrirNavegador ruta archivo
  | archivo == "account.js" = abrirNavegadorAccount (ruta ++"/"++ archivo)
  | archivo == "profile.js" = abrirNavegadorAccount (ruta ++"/"++ archivo)
```

- 2 usos de tipados

  En cuanto al uso de tipados, debido a que Haskell es un lenguaje fuertemente tipado se deben de usar sin excepción. En las funciones de ejemplo anteriores se puede ver como, además, se escribe el tipo de la función para que el lenguaje nisiquiera tenga que inferirlo.

- **Creación de al menos un módulo**

  Un ejemplo de la creación de un módulo en este trabajo sería el módulo TiposDatos, que se encarga de definir parte de los distintos tipos de datos que se usan en el trabajo.

- **Creación de un tipo de datos y uso de este**

  Utilizando el mismo archivo que en el apartado anterior, podemos ver varias definiciones de tipos de datos, por ejemplo uno de los mas interesantes sería el tipo de datos Tweet:

```haskell
data Tweet = Tweet {
    favorite_count :: String,
    retweet_count :: String,
    created_at :: String,
    full_text :: String,
    lang :: String
} deriving (Generic, Show)
instance FromJSON Tweet
```

  En este tipo de datos recogemos 5 de entre todos los parámetros que nos encontramos en los archivos de Twitter.

- **Uso de dos conceptos avanzados**

  En cuanto a uso de conceptos avanzados, como este trabajo se centraba en Data Science y analisis de datos, uno de los mas importantes ha sido el procesamiento de archivos JSON con Aeson

  - Procesamiento JSON con Aeson:

    Para decodificar archivos JSON se ha usado la libreria Aeson [2], que nos ha permitido recojer los datos de los archivos sin problemas. Todo esto se ha realizado con las distintas funciones del archivo *ParsersArchivos.hs*. A continuación podemos ver un ejemplo de parseado de los Tweets.

```haskell
parseTweets :: String -> IO [Tweet]
parseTweets path = do
    preparaTweets path
    file <- B.readFile (path++".mod")
    let tweets = decode file :: Maybe [Tweet]
        case tweets of
            Nothing -> exitSuccess
            Just tweets -> return tweets
```

  - Programa interactivo:

    En cuanto al segundo concepto avanzado, la primera idea fue utilizar la documentación de la asignatura para intentar implementar una paralelización

sencilla. Sin embargo, finalmente se decidió implementar un programa interactivo, ya que sería una solución mas sencilla y rápida, además de que el tamaño de los datos de una cuenta de Twitter, especialmente de la parte que analizamos nosotros, no es tan grande como para necesitar paralelismo o clustering.

Esta interactividad se hace con las funciones del módulo *MenuInteractivo.hs*, por ejemplo, aqui podemos ver el menú principal de la aplicación:

```
menuPrincipal :: IO()
menuPrincipal = do
    clearScreen
    putStrLn "Hola, bienvenido al wizard de TweeKell"
    putStrLn "Menu principal:"
    putStrLn "Pulse enter para comenzar o escriba alguna opción"

    putStrLn "\ni: Información"
    putStrLn "q: Salir"

    option <- getLine
    case option of
        "i" -> menuInformacion
        "q" -> exitSuccess
        _ -> menu1
```

Como se puede observar, se espera la interacción del usuario y se responde correctamente dependiendo de lo elegido.

8. Uso de la aplicación

Para usar la aplicación se puede ejecutar el archivo compilado en cualquier sistema GNU/Linux o, si se prefiere, se puede utilizar el código fuente. Una vez ejecutado deberiamos de ver el menu principal de la aplicación tal y como muestra la figura 1.

A continuación, siguiendo los menus introduciremos la ruta al archivo que queremos analizar y comprobaremos que lo hemos introducido correctamente tal y como se ve en la figura 2

Tras esto, podremos seleccionar el tipo de análisis que queremos realizarle al archivo, un ejemplo que podemos ver en la figura 3 serían los tipos de analisis sobre el archivo *Tweet.js*

Finalmente, tras seleccionar el tipo de análisis que queremos ver nos encontraremos con una pantalla que nos presentará los resultados. Un ejemplo seria la pantalla de la figura 4

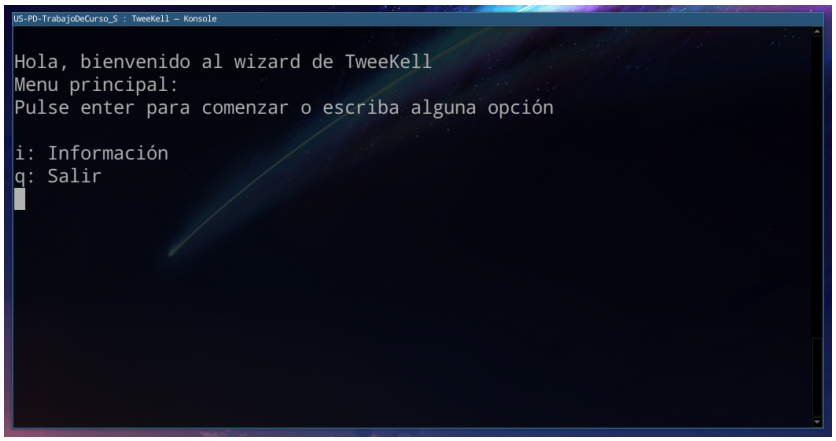

Figura 1: Menú principal de la aplicación

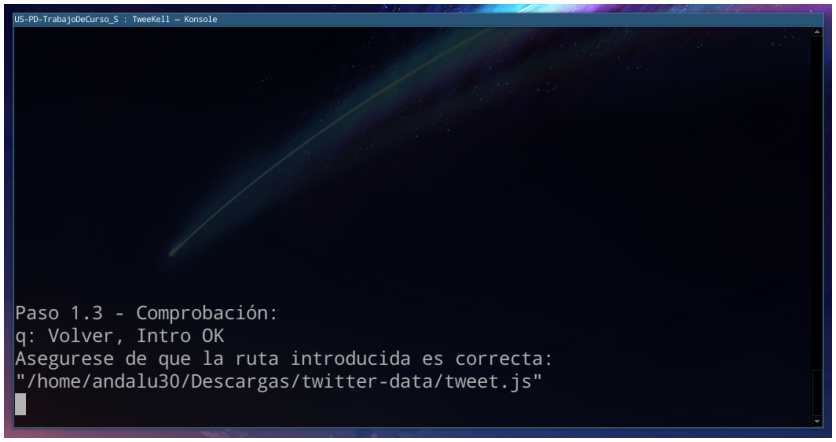

Figura 2: Pantalla de confirmación de la ruta del archivo a analizar

8.1. Utilizando el archivo compilado

El archivo compilado *TweeKell* se debe de poder ejecutar en cualquier sistema GNU/Linux sin problemas[1] mediante la ejecución del comando *./Tweekell*. Tras esto podremos ver el menu principal tal y como se ve en la figura 1

8.2. Utilizando el código fuente

Si se prefiere utilizar el código fuente para ejecutar la aplicación se podrá, por ejemplo, utilizar ghci. En este caso, ejecute *ghci* en una terminal, cargue el archivo *TweeKell.hs* con *:l Tweekell.hs* y ejecute *main*, debería de ver el menu principal.

[1]Testeado en varios sistemas corriendo Fedora Workstation, Manjaro Linux, Ubuntu Server y Arch Linux

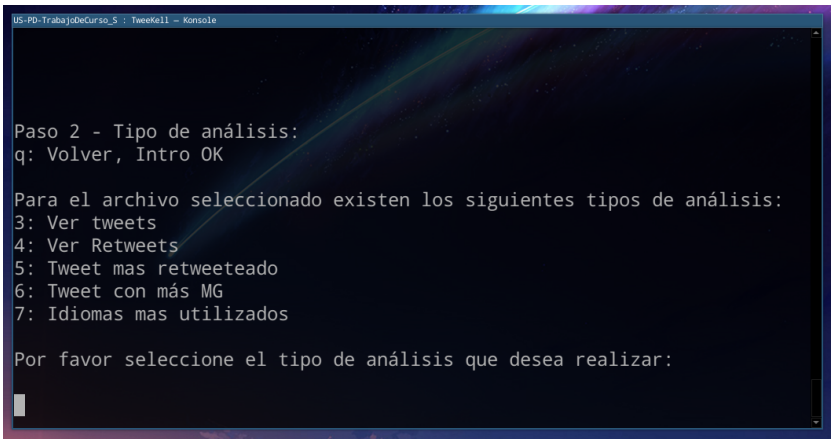

Figura 3: Distintos tipos de análisis que se pueden realizar sobre el archivo *Tweet.js*

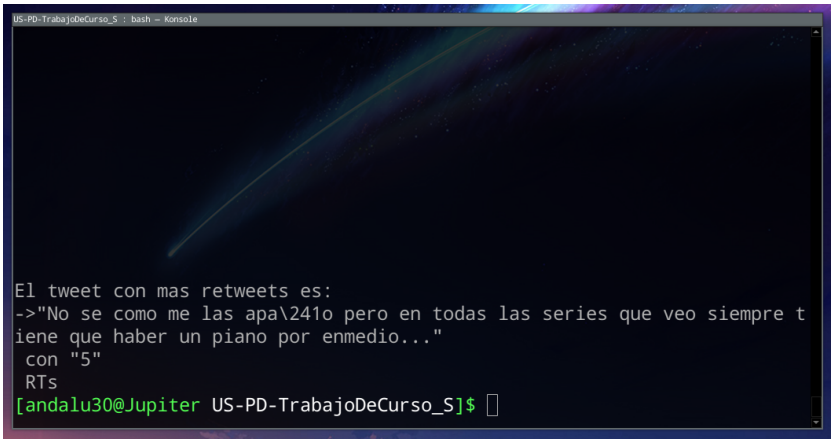

Figura 4: Pantalla de resultados del análisis

9. Posibles mejoras

9.1. Analizar más información

Una de las posibles mejoras a este trabajo, sería la de analizar más información de la que se analiza actualmente. De entre los 45 archivos que Twitter nos ofrece, apenas analizamos 5 de ellos. Los archivos más interesantes de analizar serian posiblemente *like.js*, *ad-engagement.js* y *ad-impressions.js*.

9.2. Paralelización y computación distribuida

Otra posible mejora al trabajo sería la implementación de paralelismo en los análisis para que de esta forma podamos aprovechar la gran potencia de los PCs actuales.

Otra forma de escalar nuestra aplicación para que soporte grandes cantidades de

datos sería la de utilizar computación distribuida. Un ejemplo de esto sería Apache Spark [6], que nos permitiría crear clusters para poder analizar una gran cantidad de información.

Referencias

[1] Juan Arteaga Carmona *Recreando un juego interactivo con Haskell y CodeWorld-API* https://github.com/Andalu30/US-PD-TrabajoDeCursogithub.com/Andalu30/US-PD-TrabajoDeCurso - Repositorio privado en GitHub

[2] Bryan O'Sullivan. *Aeson: A fast Haskell JSON library* https://github.com/bos/aesongithub.com/bos/aeson - Repositorio en GitHub

[3] Paul Hudak, *Haskell* https://www.haskell.org/ Web del lenguaje.

[4] Hudak, Paul; Hughes, John; Peyton Jones, Simon; Wadler, Philip (2007), *A History of Haskell: Being Lazy with Class* ISBN 978-1-59593-766-7.

[5] Twitter *Cómo acceder a tus datos de Twitter* https://help.twitter.com/es/managing-your-account/accessing-your-twitter-dataEnlace a la web de asistencia de Twitter

[6] The Apache Software Foundation *Apache Spark, Lightning-fast unified analytics engine* https://spark.apache.org Enlace a la web