Taller Procesamiento en BigData con Spark + R

Autor: Manuel J. Parra

Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) .

Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/).

University of Granada.

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/).
University of Granada.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Herramientas tradicionales de análisis de datos:

- Unix scripts
- Pandas
- SPSS
- SAS
- R
- ...
- Excel ... ;)

Todas ellas operan sobre una única máquina, no conectadas o conectadas, con recursos limitados, etc.

El problema de los grandes conjuntos de datos. El problema sobre *BigData*.

- Los datos crecen más rápido que la velocidad de computación.
- Fuentes de datos creciendo: datos de la web, datos científicos, datos móviles, datos de sistemas, ...
- Almacenamiento muy barato: El tamaño se dobla cada 18 meses.
- Cuellos de botella en la CPU y almacenamiento.

Ejemplos

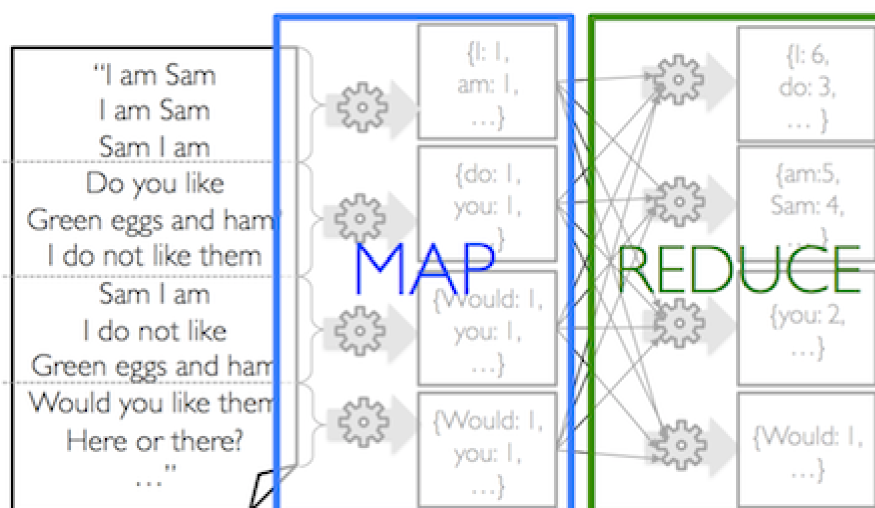
- Logs diarios de FaceBook: **60 TB**
- 1000 genomas: **200 TB**
- CERN: **25 PB anuales**
- Indice web de Google: **10+ PB**
- Coste de 1 TB de discoduro: **~35 ... 50 €**
- Tiempo de lectura de 1 TB: 3 horas !!!! (100 MB/s)

Un problema y una solución para almacenar y analizar grandes conjuntos de datos

Una máquina sola no puede procesar o incluso almacenar todos los datos

- **Sólo una solución: distribuir los datos sobre clusters**

Una de las soluciones iniciales: MAP REDUCE (https://www.ibm.com/analytics/us/en/technology/hadoop/mapreduce/ (https://www.ibm.com/analytics/us/en/technology/hadoop/mapreduce/))



MapReduce es el corazón de Hadoop.

Este paradigma de programación permite una escalabilidad masiva a través de cientos o miles de servidores en un clúster con Hadoop. El concepto de MapReduce es bastante simple de entender.

Limitaciones de Hadoop y MapReduce:

- Latencia para el acceso a datos.
- Cantidades grandes de ficheros pequeños.
- Escribe una vez, lee varias.
- No se puede acceder con los comandos tradicionales de Linux (ls, cat, vim...).

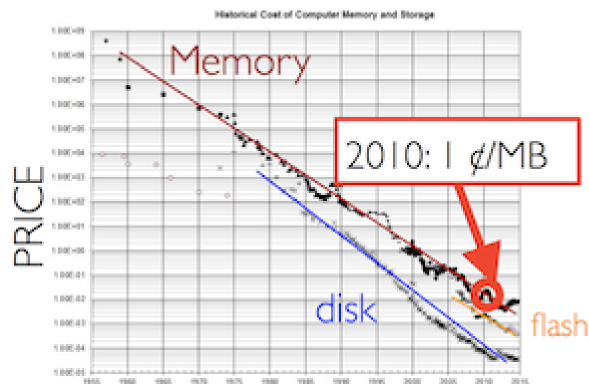
En cuanto a MapReduce:

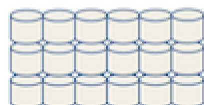
- Es muy difícil de depurar: Al procesarse el programa en los nodos donde se encuentran los bloques de datos.
- **No todos los algoritmos se pueden escribir** con el paradigma MapReduce. Por ejemplo, el famoso algoritmo de grafos *Dijkstra*, al necesitar un procesamiento secuencial, no se puede escribir en MapReduce.

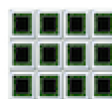
La motivación de Apache Spark

El uso de MAP REDUCE para trabajos complejos, consultas interactivas y procesamiento en tiempo real, involucra **enormes cantidades de entrada** y salida de disco.

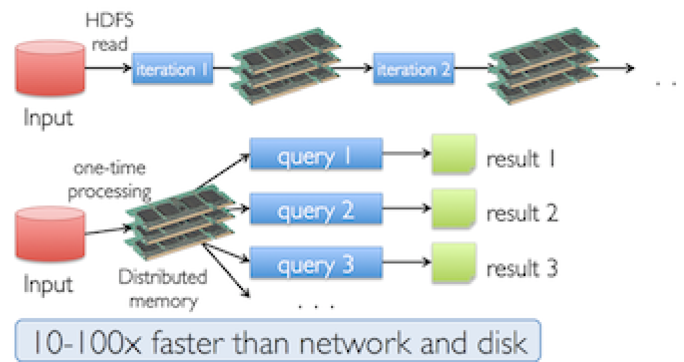
El trabajo con Disco es muy lento !







Almacenar más datos en memoria. Usar más memoria en lugar de disco.



Resilient Distributed Datasets (RDDs)

- Colecciones de objetos particionados y repartidos a lo largo del cluster, almacenado en memoría o disco.
- Manipulados a través de deversas transformaciones y acciones (map, filter, join, count, collect, save, ...)
- Son automáticamente reconstruidos en caso de fallo de la máquina

Hadoop (Map Reduce) y Spark

Caracteristica	Hadoop	Spark
Almacenamiento	Disco	En memoria o disco
Operaciones	Map Reduce	Map Reduce Join, Sample
Modelo de ejecución	Por lotes	Por lotes Interactivo Streaming
Lenguaje	Java	Scala, Java, R y Python

Procesamiento en Memoria, puede marcar gran diferencia

Las ventajas principales de Spark:

1. Una plataforma de código abierto con una comunidad activa
2. Una plataforma muy rápida
3. Una plataforma unificada para gestionar datos
4. Consola interactiva
5. Una gran API para trabajar con los datos
6. Programable desde Java, Python, R y Scala
7. Procesamiento de Grafos
8. Procesamiento de Datos en Streaming

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/).
University of Granada.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

R+ Spark + Big Data

La tendencia más reciente en el análisis de grandes conjuntos de datos indica la necesidad de un análisis *INTERACTIVO* de grandes conjuntos de datos.

Debido a ello se han desarrollado herramientas académicas y comerciales para trabajar de este modo con los datos.

Está claro que cada vez más el uso de herramientas como R para el análisis de datos, ya que permiten una mejora sustancial y más avanzada para el estudio de los datos. R, como sabemos, es ya hoy en día bastante popular y provee de soporte para procesamiento de datos estructurados usando dataframes y lo mejor de todo es que incluye en CRAN un número muy elevado de paquetes para análisis estadístico y visualización.

- R es la herramienta `de facto` para el análisis de datos
- R soporta procesamiento de datos estrcuturados `dataframes`
- R tiene miles de paquetes para todo tipo de tareas: analíticas, visualización, minería de datos, ...

R

El proyecto R, o el lenguaje R es un lenguaje de programación, un entorno de desarrollo interactivo, y un conjunto de bibliotecas de computación estadistica.

R es un lenguaje interpretado y provee de soporte para ejecución condicional, bucles, etc. R también incluye caracteristicas para computación nmérica, con tipos de datos vectores, matrices, etc. Otra característica fundamental es el uso de dataframes: estructuras de datos en tablas donde cada columna está formada por elementos con su tipo de dato. Estos dataframes pueden ser manipulados para filtrado, agregación, etc.

Entonces, ¿cuál es el problema? ¿Por qué no se usa para Datos Masivos?

El análisis de datos usando R está limitado por **la cantidad de memoria disponible** en una sola máquina y además R trabaja en un sólo hilo/proceso (*parallel*), por lo que **es poco práctico** usar R para grandes conjuntos de datos.

¿Qué es Spark?

Algunas de estas limitaciones se han abordado, a través de un mejor soporte de Entrada/Salida, la integración con Hadoop y el diseño de aplicaciones R distribuidas que se pueden integrar con los motores de Bases de Datos más extendidos

Por tanto, analizamos cómo podemos escalar los programas R al tiempo que facilitamos su uso y despliegue a través de varias cargas de trabajo.

Hay una serie de beneficios para el diseño de un frontend R que está integrado con Spark:

- Bibliotecas de soporte: Spark, contiene bilbiotecas para trabajar con fuentes de datos y SQL, Machine Learning Distrbuido, análisis de grafos, etc.
- Fuentes de datos: Desde bases de datos, HDFS, HBase, Cassandra, JSON, CSV, Parquet, ...
- Rendimiento: Planificación de tareas, generación de código, gestión de memoria, ...

Apache Spark es un motor de proposito general para procesamiento de conjuntos de datos masivos. El proyecto inicialmete introdujo el RDD, Resilient Distributed Datasets, una API para computación tolerante a fallos en un entorno de cluster.

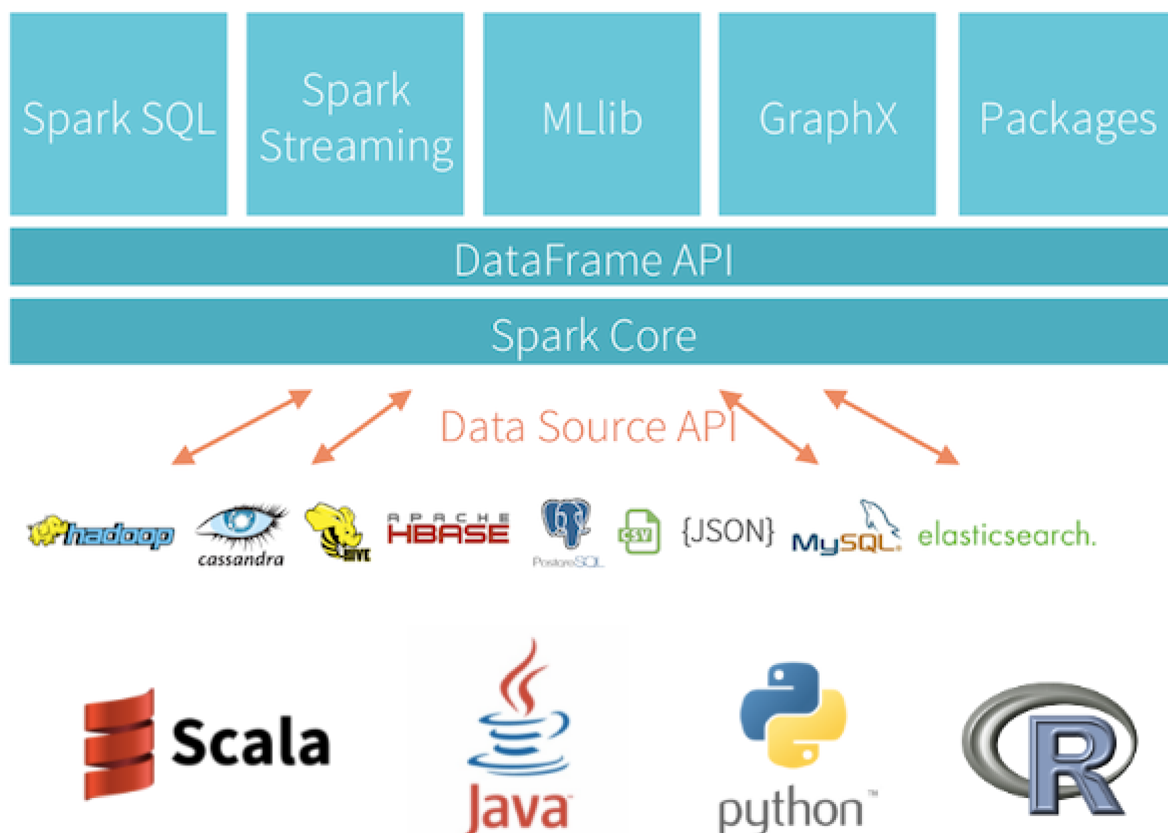
Más recientemte muchas más APIs de alto nivel están siendo desarrolladas en Spark. Estás son las siguientes:

- Machine Learning Library (MLLib) una biblioteca para machine learning a gran escala.
- GraphX, una biblioteca para procesamiento de grafos a gran escala
- SparkSQL, una API para consultas analíticas en SQL

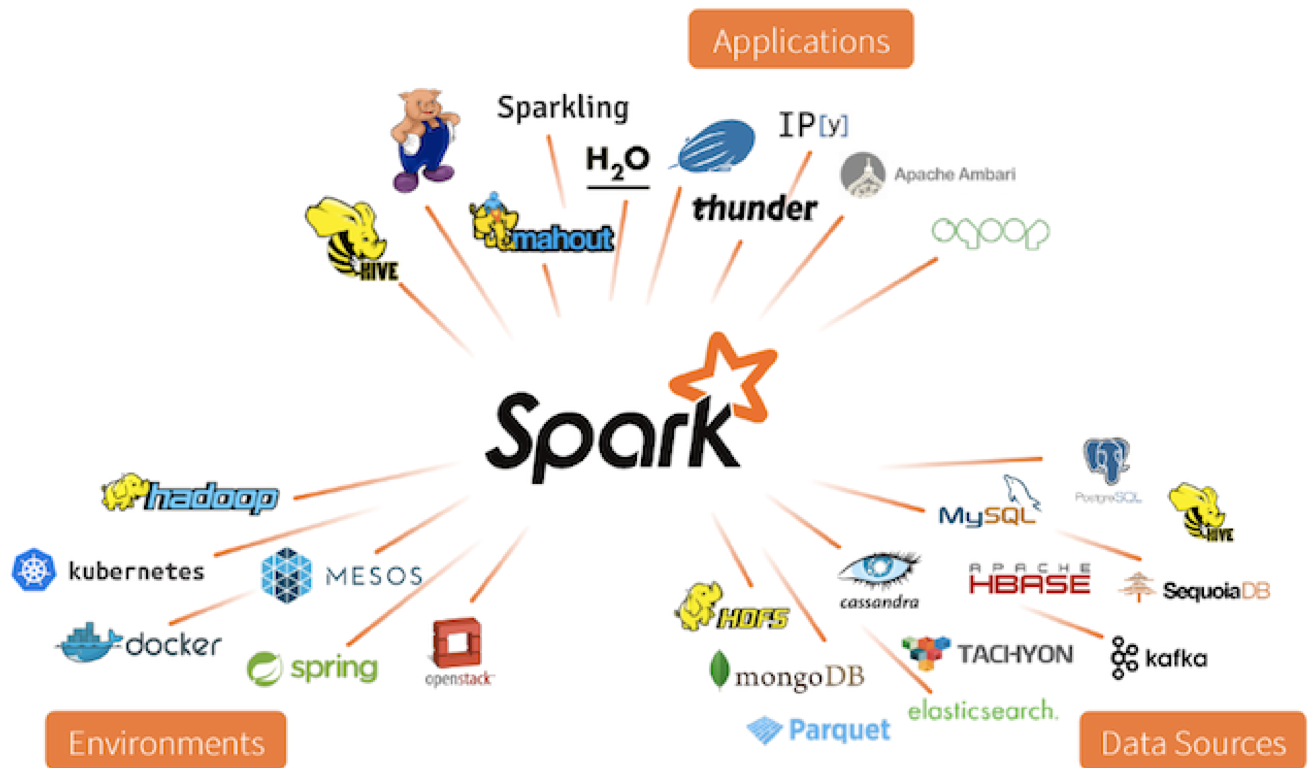
Las bibliotecas están integradas en el CORE de Spark, por lo que Spark, puede permitir flujos de trabajos complejos, donde consultas SQL pueden ser usadas para preprocesar posteriormente los datos y analizarlos de forma avanzada con la parte de Machine Learning

¿Por qué Spark? ¿Qué ventajas tiene?

Desde el inicio de Hadoop hace más de 10 años, poco a poco han aparecido nuevos problemas que ya no se podían resolver usando el paradigma de MapReduce. Esto ha echo que hayan aparecido nuevos sistemas para afrontar estos problemas, apareciendo :



Una característica importante es que muchas de las aplicaciones ya existentes se han hecho compatibles con Spark y que estén surgiendo nuevas enfocadas en trabajar con Spark en áreas especificas de procesamiento de datos masivos.



Spark es:

- Rápido
- Escalable
- Interactivo

En cuando a R tenemos que sus características principales son:

- Estadísticas
- Muchos paquetes para todo
- Gráficos

Si únimos todo (Spark+R):

Una ventaja adicional es el RDD

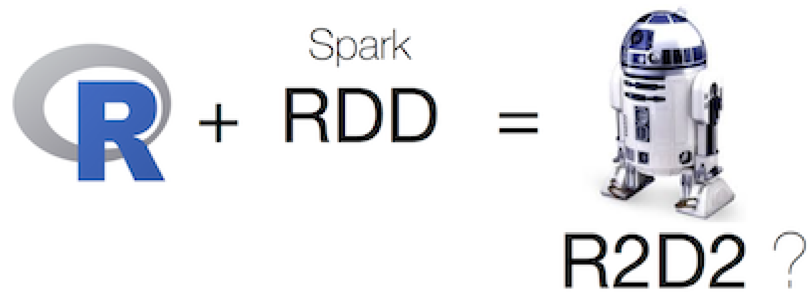
Incluye tranformaciones de datos con:

- map
- filter
- groupBy
- ...

Acciones sobre los datos:

- count
- collect
- save
- ...

En R entonces utilizamos SparkDataFrames (RDDs manejables dentro del entorno de R):

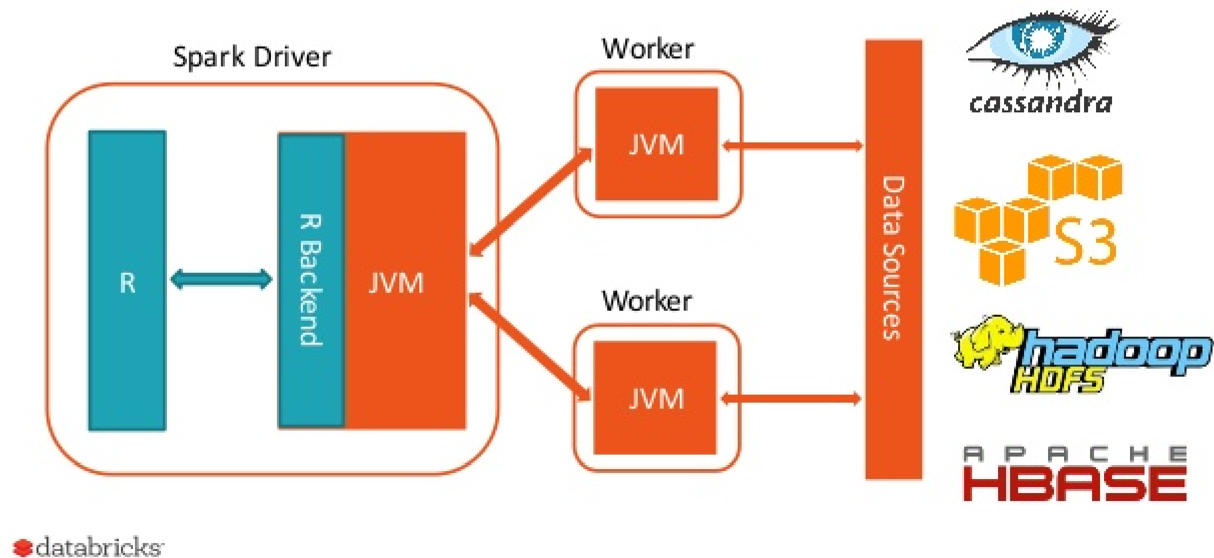


Análisis escalable interactivo con `SparkR` + `sparklyr`

SparkR y sparklyr están construidos como un paquetes en R, luego no es necesario hacer cosas especiales para que funcione.

El componente principal de SparkR es un DDF, Distributed Data Frame, que permite procesamiento de datos estructurados utilizando la sintaxis familiar para los usuarios acostumbrados a R.

Para mejorar el rendimiento, sobre grandes conjuntos de datos, SparkR provee de "evaluación perezosa" para operaciones sobre dataframes y utiliza además un optimizador de consultas relacionales para optimiar la ejecución.



In [ ]:

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/).
University of Granada.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Comenzando con el Análisis Interactivo en BigData con Spark

Spark soporta Python, Scala, R y Java.

Podemos elegir escribir una aplicación completa `standalone` o un utilizar cualquiera de los interpretes interactivos que ofrece Spark.

Spark pone a tu disposición tres herramientas fundamentales para el análisis interactivo de datos:

R

Desde la Máquina Virtual puedes escribir el siguiente comando y pulsar intro (para salir CRTL+D o escribe `exit`)

```
sparkR
```

Esto habilita un shell de `R` con una sesión en Spark.

> En el taller usaremos este entorno pero desde los frontends: Jupyter y RStudio !

Python

Desde la Máquina Virtual puedes escribir el siguiente comando y pulsar intro (para salir CRTL+D o escribe `exit`)

```
pyspark
```

Esto habilita un shell de `python` con una sesión en Spark.

Scala

Desde la Máquina Virtual puedes escribir el siguiente comando y pulsar intro (para salir CRTL+D o escribe `exit`)

```
spark-shell
```

Esto habilita un shell de `Scala` con una sesión en Spark.

Ejercicio práctico:

Prueba cada uno de las herramientas de analisis interactivo y comprueba están disponibles. Para salir de los interfaces interactivos, utilizad CTRL+D o bien escribe exit.

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/). **University of Granada**.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

DataSets del Taller de SparkR

Para todos los ejemplos y ejercicios del taller se ha habilitado una directorio en la Máquina Virtual con multiples conjuntos de datos de tipo variado (bioinformatica, logistica-transporte, salud).

Los que vamos a utilizar serán los siguientes:

Conjunto de datos: BNGheart

Directorio en la Máquina Virtual:

```
/SparkR/datasets/BNGhearth.dat
```

Este conjunto de datos es una base de datos de enfermedades del corazón y contiene los siguientes atributos:

- age
- sex
- chest pain type (4 values)
- resting blood pressure
- serum cholestoral in mg/dl
- fasting blood sugar > 120 mg/dl
- resting electrocardiographic results (values 0,1,2)
- maximum heart rate achieved
- exercise induced angina
- oldpeak = ST depression induced by exercise relative to rest
- the slope of the peak exercise ST segment
- number of major vessels (0-3) colored by flourosopy
- thal: 3 = normal; 6 = fixed defect; 7 = reversable defect

Y la variable de clase: Ausencia o presencia of problemas cardíacos

Conjunto de datos: ECBDL14

Directorio en la Máquina Virtual:

```
/SparkR/datasets/databig/ECBDL14_10tst.data
```

Este conjunto de datos proviene del campo de predicción de estructuras de proteínas y se generó originalmente para crear un predictor para la predicción de contacto residuo-residuo de CASP9. El conjunto de datos original tiene:

- 32 millones de instancias,
- 631 atributos,
- 2 clases,
- 98% de ejemplos negativos

y ocupa, aproximadamente 56 GB de espacio en disco. Los detalles de la generación de conjuntos de datos y una estrategia de aprendizaje utilizados para entrenar un método para este problema usando computación evolutiva están disponibles en http://bioinformatics.oxfordjournals.org/content/28/19/2441 (http://bioinformatics.oxfordjournals.org/content/28/19/2441).

Training set (3723MB): TrainingSet.arff.gz (http://cruncher.ncl.ac.uk/bdcomp/TrainingSet.arff.gz) Test set (347MB): TestSet.arff.gz (http://cruncher.ncl.ac.uk/bdcomp/TestSet.arff.gz)

Más información: http://cruncher.ncl.ac.uk/bdcomp/ (http://cruncher.ncl.ac.uk/bdcomp/)

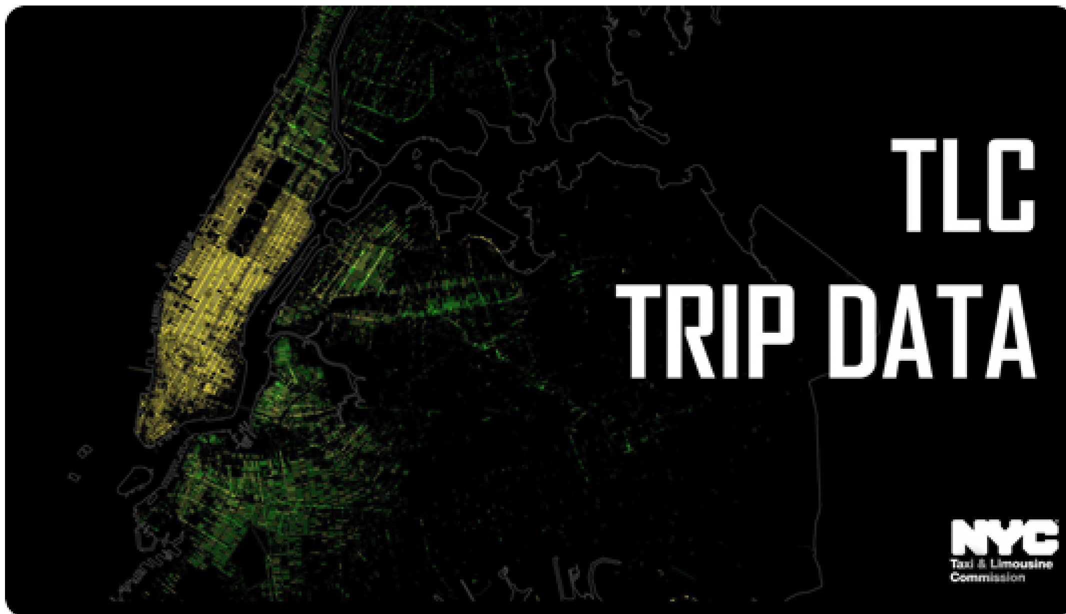
Conjunto de datos: NYC YellowCab

Directorio en la Máquina Virtual:

`/SparkR/datasets/yellow_tripdata_2016-02_small1.csv`

El dataset que vamos a usar para el procesamiento de dato masivos, corresponde con un conjunto de datos de los registros de viaje en TAXI, donde se capturan las fechas y horas de recogida y devolución de pasajeros, lugares de recogida y entrega (coordenadas), distancias de viaje, tarifas detalladas, tipos de tarifas, tipos de pago y conteos de pasajeros que van en el taxi. El dataset tiene MUCHAS posibilidades de procesamiento y también extracción de conocimiento. Estos conjuntos de datos adjuntos fueron recopilados y proporcionados por la Comisión de Taxisde Nueva York (TLC) http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml (http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml).

Características del conjunto de datos original:



El conjunto de datos NYCTaxiTrips en total tiene sobre 267GB, que pueden ser manejados sin problema por SparkR (en un cluster real, no sobre una máquina virtual sencilla como la del taller).

En total contiene 1100 millones de registros y se añaden más cada mes.

Más información de como se gestionan 1100 millones de instancias en la siguiente web y se soluciona este problema problema real: http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/ (http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/)

Más datasets masivos de NYCTaxiTrips en: http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml (http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml)

Adicionalmente también están disponibles en `/SparkR/datasets/` :

- `yellow_tripdata_2016-01.csv`
- `yellow_tripdata_2016-02_small1.csv`
- `yellow_tripdata_2016-02_small2.csv`
- `yellow_tripdata_2016-02_small3.csv`

¿Como cargar estos conjuntos de datos en SparkR?

La forma más sencilla de cargar los conjuntos de datos es usar las funciones de SparkR para leer conjuntos de datos en formato CSV.

Desde SparkR

In [ ]:

```
#Cargamos los paths
.libPaths(c(file.path(Sys.getenv("SPARK_HOME"),"R/lib/"),.libPaths()))

#Cargamos la libreria
library(SparkR)

#Abrimos la sesion con Spark
sparkR.session(appName="EntornoInicio",
               master = "local[*]",
               sparkConfig = list(spark.driver.memory = "1g"))

# Cargamos por ejemplo el dataset de NYC
df_nyctrips <- read.df("/root/TallerSparkR/datasets/yellow_tripdata_2016-02_smal
l1.csv",
                           "csv",
                           header = "true",
                           inferSchema = "true")
```

Desde sparklyr (no ejecutar aún)

In [ ]:

```
# library(sparklyr)
## Usamos la biblioteca para el manejo de los datos.
#library(dplyr)

## Abrimos la conexión. Importante indicar la versión de Spark que tenemos insta
lada. En nuestro caso tenemos la 2.0.1
#sc <- spark_connect(master = "local", version = "2.0.1")


## Lectura de un fichero de datos CSV

#tttm <- spark_read_csv(sc,
#                        name="tttm",
#                        path="/SparkR/datasets/BNGhearthBIG.dat",
#                        delimiter = ",",
#                        header=TRUE,
#                        overwrite = TRUE)
```

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/).
University of Granada.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Entorno de trabajo del taller

Para este taller en la Universidad de Navarra, se ha instalado en cada PC individual una Máquina Virtual con VirtualBox, desde la cual se podrá trabajar con todas las herramientas para el procesamiento de datos masivos.

De modo que sólo tendrás que iniciar la Máquina Virtual y seguir estos pasos para habilitar todos los servicios:

- Iniciar la Máquina Virtual con el Taller (Spark_R_UPN)-
- Introducir los datos de acceso:
  - Usuario (login): `root`
  - Clave (password): ``sparkR```
- Escribir el siguiente comando y pulsar intro:
  - `iniciar_taller`
- Al ejecutar el comando anterior, solicitará la clave de nuevo, por lo que escribimos: `sparkR`
- Esperamos a que termine de ejecutarse el comando y comprobamos que podemos acceder a las siguientes URLs:
  - **Jupyter**. Se accede desde: http://192.168.99.10:8888 (http://192.168.99.10:8888)
  - **RStudio**. Se accede desde: http://192.168.99.10:8787 (http://192.168.99.10:8787) El acceso es identificado, luego aquí usa como usuario: `test` y clave `test`

¿Qué contiene la máquina virtual?

- Hadoop
- Spark versión 2.01.
- Spark + Python versión 2.7
- Spark + R versión 3.3.1
- Spark + Scala versión 2.16
- Jupyter
- RStudio
- Zeppelin 0.62

Si ya has terminado el taller y quieres seguir trabajando desde casa, revisa la documentación para instalar todo el entorno de trabajo en tu PC de casa.

Spark All-in-one

La máquina virtual para el taller completo contiene todas las herramientas necesitas para el desarrollo del trabajo, por lo tanto es la opción más sencilla para poder empezar manos a la obra con el contenido, en el caso de que no estés ya en el taller.

Requisitos necesarios para trabajar con la Máquina Virtual:

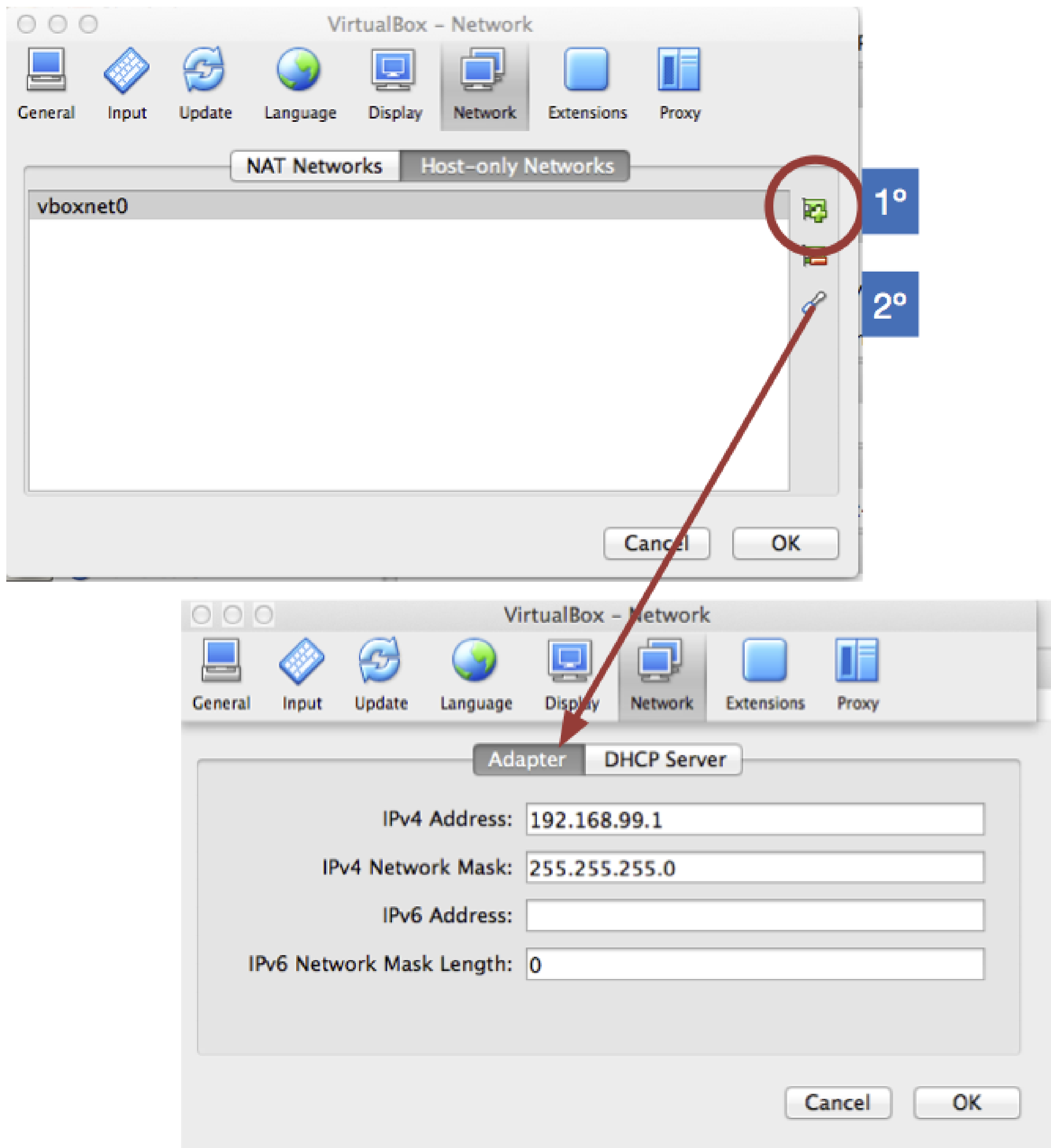
- Tener instalado VIRTUALBOX, disponible en: https://www.virtualbox.org/wiki/Downloads (https://www.virtualbox.org/wiki/Downloads)
- Disponer de al menos 2GB de RAM para la Máquina Virtual
- El PC debe ser de 64bits y contar con al menos 4GB de RAM (2GB para la MVirtual y otros 2GB para el PC)
- Compatible con Windows, Mac OSX y Linux

Descarga la máquina virtual del taller: https://drive.google.com/file/d/0ByPBMv-S_GMEakRCVVRTejZKVm8/view?usp=sharing (https://drive.google.com/file/d/0ByPBMv-S_GMEakRCVVRTejZKVm8/view?usp=sharing) (aprox: 4 GB)

Haz doble clic el fichero `Spark_UPN.ova` y se instalará la nueva Máquina Virtual en tu sistema.

Antes de iniciar la Máquina Virtual desde VirtualBox hay que configurar lo siguientes parámetros dentro de VirtualBox:

- Ir al Menu de Preferencias del VirtualBox -> Network (RED) -> Host-only Network (Adaptador sólo Host)
- Pulsar en añadir:



Utiliza la IP: `192.168.99.10` y Mascara de RED `255.255.255.0`

Una vez configurados los parámetros, iniciar la nueva Máquina Virtual (pulsando sobre su icono y "comenzar"); al arrancar nos pedirá el LOGIN y USUARIO, para ello hay que usar:

```
usuario: root
clave: sparkR
```

Hecho esto veremos la siguiente pantalla de información de los servicios que disponemos:

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/). **University of Granada**.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Inicio del entorno de trabajo con SparkR

Lo primero que hacemos para trabajar con el entorno es añadir la bilioteca de **SparkR** al **LibPath** de `R` y cargamos la biblioteca de **SparkR**.

In [1]:

```
#Fijamos la ruta donde está instalado Spark
Sys.setenv("SPARK_HOME"='/usr/local/spark/')
.libPaths(c(file.path(Sys.getenv("SPARK_HOME"),"R/lib/"),.libPaths()))
```

Comprobamos que está bien indicado el path y está tomando la bilbioteca adecuada:

In [ ]:

```
Sys.getenv("SPARK_HOME")
```

Cargamos la biblioteca de SparkR

In [ ]:

```
# Usamos a opción de warn=-1 si queremos que no muestre warning y sólo
# muestre errores (más limpias las salidas)
# options(warn=-1)
library(SparkR)
```

Comenzamos una sesión con Spark.

Para ello creamos una sesión en Spark, indicándole el modo de inicio de la sesión que queremos: *modo local* `local`, *todos los núcleos disponibles* `[*]` y la *cantidad de memoria RAM a utilizar* 2GB.

In [ ]:

```
# Iniciamos la sesión con Spark.
# Parametros mínimos: appName => Nombre del trabajo en Spark
#                     master  => local[*] Usará tantas hebras como cores disponi
bles.
#                     sparkConfig => opciones de ejecución de Spark, por ejemplo
 limitación de memoria a 2GB.
sparkR.session(appName="EntornoInicio", master = "local[*]", sparkConfig =
list(spark.driver.memory = "2g"))
```

Otros modos posibles de inicialización de Spark:

In [ ]:

```
# sparkR.session(appName="NombreAplicacion", master = "local", sparkConfig = lis
t(spark.driver.memory = "4g"))
# sparkR.session(appName="AplicacionMLData", master = "local[20]", sparkConfig =
 list(spark.driver.memory = "2g"))
# sparkR.session(appName="AplicacionMLData", master = "spark://host:port", spark
Config = list(spark.driver.memory = "200g"))
```

Más información sobre los métodos de inicialización de Spark en https://spark.apache.org/docs/latest/sparkr.html#starting-up-from-rstudio (https://spark.apache.org/docs/latest/sparkr.html#starting-up-from-rstudio) y para sparklyr https://spark.apache.org/docs/latest/sparkr.html#starting-up-from-rstudio (https://spark.apache.org/docs/latest/sparkr.html#starting-up-from-rstudio)

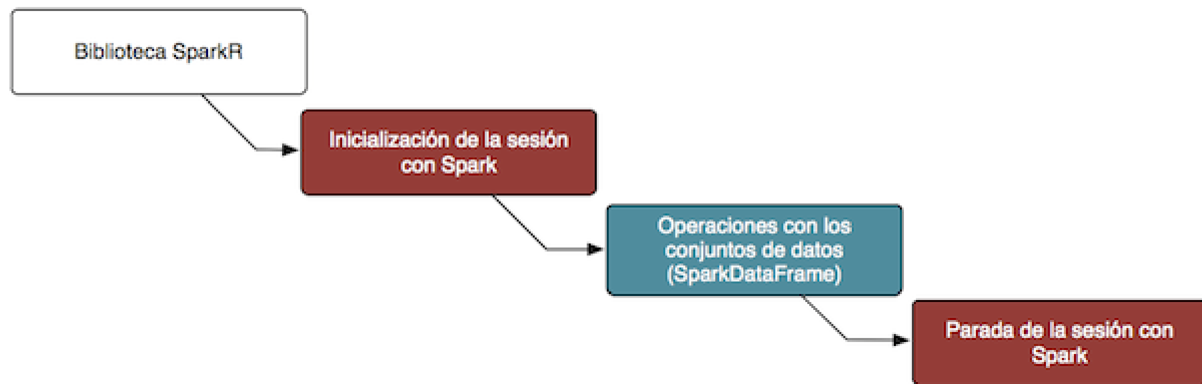
Una vez que hayamos terminado el script, **siempre al fina**l, es necesario siempre cerrar la sesión con Spark para liberar recursos.

El problema que tiene no usar este comando es que si no se liberan recursos y los dataframes están en memería y además tenemos limitada la memoria, entonces Spark no funcionará correctamente al acumuluar y no despejar lo que hemos usado.

In [ ]:

```
# Cerramos la sesión con Spark y liberamos recursos.
sparkR.session.stop()
```

Por tanto el flujo de trabajo con SparkR debe ser el siguiente:



Algunas diferencias entre SparkR 1.5 la versión de SparkR 2.0

- Se ha cambiado la función `table` por `tableToDF`.
- La clase DataFram se ha sustituido por SparkDataFrame para evitar conflictos.
- Ahora se usa sparkR.session() para arrancar la sesión con SparkR, en lugar de sparkR.init().
- No se necesita un sparkContext para las funciones para crearDataFrame, read.json, cacheTables, read.df, ...
- La función `registerTempTable` se ha reemplazado por `createOrReplaceTempView` para trabajar con SparkSQL.
- La función `dropTempTable` se ha cambiado por `dropTempView`.

Actualmente la versión más actual es las 2.1.0

- join no longer performs Cartesian Product by default, use crossJoin instead.

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/). **University of Granada**.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Primer ejemplo y toma de contacto con SparkR

Como siempre para todos nuestros `scripts` con **SparkR**, cargamos la biblioteca, y creamos una nueva sesión de SparkR.

In [ ]:

```
#Fijamos la ruta donde está instalado Spark
Sys.setenv("SPARK_HOME"='/usr/local/spark/')

.libPaths(c(file.path(Sys.getenv("SPARK_HOME"),"R/lib/"),.libPaths()))
# Añadimos la bibioteca
library(SparkR)
# Abrimos la conexión
sparkR.session(appName="Primeros_Pasos", master = "local[*]", sparkConfig =
list(spark.driver.memory = "1g"))
```

Con **SparkR** podemos crear un `DataFrame` de Spark desde un `data.frame` habitual usado en R.

Un `DataFrame` es una colección distribuida de datos organizada en columnas.

Los `dataframes` son conceptualmente equivalentes a bases de datos relacionales o a `data.frames` en R o Python, pero con una ventaja: son mucho más eficientes para el trabajo con grandes volumenes de datos. Los `DataFrames`pueden ser creados desde una amplio surtido de fuentes muy diferentes. Es decir, casi cualquier cosa puede ser un `dataframe`, por ejemplo ficheros estructurados, tablas en `HIVE`, bases de datos externas o RDDs.

Los RDDs son la principal abstracción de datos en Spark. Un RDD es una colección resilente y distribuida de registros. Esta es una de las claves de Spark y es uno de los componentes fundamentales del `core` de Spark.

In [ ]:

```
# Vamos a usar un dataset sencillo integrado en R
# El dataset contiene el tiempo de espera entre erupciones y duración
# de la erupción de un geiser de Yellowstone
class(faithful)

# Convertimos un dataframe de R en un DataFrame de Spark, que llamaremos SparkDa
taFrame
df_faithful <- createDataFrame(faithful)

# Vemos el tipo de dataset nuevo
class(df_faithful)

# Visualizamos de forma rápida el contenido
head(df_faithful)

# Usamos la función printSchema de SparkR para 'deducir' el esquema de datos (la
 estructura)
printSchema(df_faithful)
```

Un `SparkDataFrame` puede ser registrado como una vista temporal en `SparkSQL` y que permite ejecutar sentencias SQL sobre los datos. La funcionalidad de SQL permite a las aplicaciones y flujos de trabajo ejecutar consultas SQL de forma programatica, devolviendo el resultado también como SparkDataFrame.

Esto es importante, ya que todas las transformaciones a los conjuntos de datos que están en formato SparkDataFrames, siguen siendo SparkDataFrames, lo que hace que toda su manipulación corra por parte de Spark con todas las ventajas que eso tiene:

- Volumen masivo de datos
- Almacenamiento distribuido
- Resilencia

Spark es perezo

In [ ]:

```
df_iris <- createDataFrame(iris)
```

Ejecutamos los siguientes trozos de código en R:

In [ ]:

```
p <- proc.time()
df_filtrado <- filter(df_iris,df_iris$Species=="setosa")
proc.time()-p
```

In [ ]:

```
p <- proc.time()
count(df_filtrado)
proc.time()-p
```

Al ejecutar una función de tipo acción (en esta caso `count`, `print`, `head`, etc.) lanza todo los procesos necesarios para conseguir realizar la acción pedida. En este caso sería hacer el filtro y luego contar. Esto se puede ver en Spark UI (la veremos más adelante).

In [ ]:

```
p <- proc.time()
count(df_filtrado)
proc.time()-p
```

Hace exactamente lo mismo, primer filtra y luego cuenta, ya que por defecto no hace persistente el DataFrame intermedio aunque lo estemos definiendo como `df_filtrado`.

Si queremos hacer persistente un DataFrame intermedio podemos con la función de cacheado.

Operaciones sencillas con SparkR sobre SparkDataFrames

En estos ejemplos vamos a tratar de ver una parte muy muy simple sobre la manipulación de los datos en el formato que entiende SparkR.

In [ ]:

```
# Contamos los elementos a partir de un filtro normal
count(filter(df_faithful,"eruptions>3.0"))

# Convertimos a vista temporal de datos en SparkSQL y le damos el nombre faithfu
l a la 'tabla'
createOrReplaceTempView(df_faithful,"faithful")

# Usamos SparkSQL para hacer consultas a los datos.
eruptions_sql <- sql("SELECT eruptions FROM faithful WHERE eruptions >= 3.0")

# Contamos el resultado
count(eruptions_sql)

# Mostramos un resumen
head(eruptions_sql)
```

Operaciones sobre los conjuntos de datos

La guía de referencia de funciones de la API de Spark con R se puede ver en:

```
https://spark.apache.org/docs/2.0.0-preview/api/R/
```

Veamos las más importantes y sus diferencias con las de R equivalentes.

Recordamos que siempre para trabajar con SparkR, tenemos que terminar la sesión de Spark.

Interfaces de gestión de trabajos de Spark

Spark pone a disposición del administrador un sistema web que permite controlar el estado del cluster en cuando a jobs, workers, operaciones, ...

Gestión de nodos Workers:

- En http://192.168.99.10:8080 (http://192.168.99.10:8080)

Gestión de Jobs:

- En http://192.168.99.10:4040/ (http://192.168.99.10:4040/)

In [ ]:

```
sparkR.session.stop()
```

Con esta sentencia se cierra el contexto abierto en SparkR y se liberan todos los recursos.

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/). **University of Granada**.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Lectura y escritura de datos desde SparkR

Como siempre, para todos nuestros `scripts` con **SparkR**, cargamos la biblioteca, y creamos una nueva sesión de SparkR.

In [ ]:

```
#Fijamos la ruta donde está instalado Spark
Sys.setenv("SPARK_HOME"='/usr/local/spark/')

.libPaths(c(file.path(Sys.getenv("SPARK_HOME"),"R/lib/"),.libPaths()))
library(SparkR)
sparkR.session(appName="EntornoInicio", master = "local[*]", sparkConfig =
list(spark.driver.memory = "1g"))
```

Hoy en día el trabajo con BigData parece que siempre está asociado al ecosistema HADOOP. Hace unos años esto significaba que si también eras un buen programador en JAVA, trabajar en tal entorno, incluso un simple programa para hacer un WORDCOUNT, implicaba varias docenas de líneas de código.

Pero hace 3-4 años la cuestión ha cambiado gracias a Apache Spark con su API de estilo funcional.

Está escrito en SCALA pero también puede ser utilizado desde Python, JAVA y como estais viendo por este Taller: también en R

Fuentes de datos

Dentro de una sesión de Spark, las aplicaciones pueden crear `SparkDataFrames` desde variadas fuentes de datos, como por ejemplo: un fichero local (`data.frame`), desde HDFS (`hdfs:///`), desde tablas en `HIVE` o desde otras múltiples fuentes de datos (AmazonS3, etc).

Concretamente las principales fuentes u orígenes de datos desde las que **cargar datos** son los siguientes:

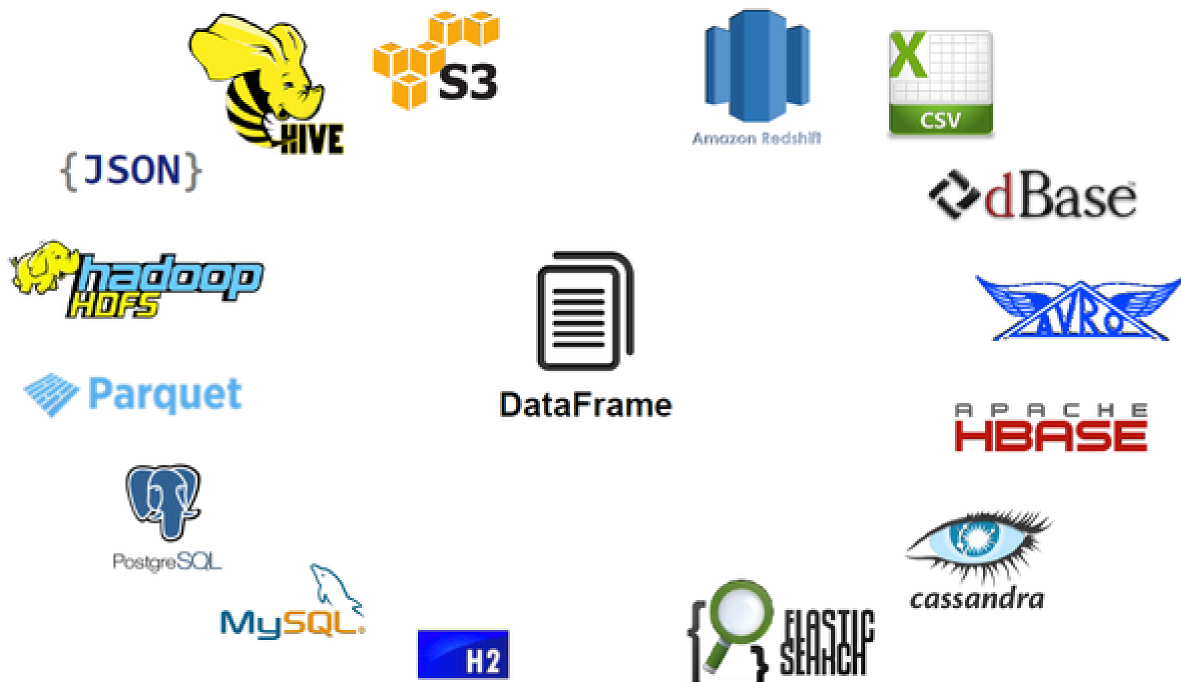
- Ficheros locales
- Ficheros en sistemas distribuidos de almacenamiento Hadoop HDFS
- Sistemas de almacenamiento de datos tipo HIVE
- Desde bases de datos relacionales a través de JDBC
- ...

Tipos de fuentes de datos

Una cosa son las **fuentes de datos** y otras cosa son los **tipo de fuentes de datos.** El tipo de fuente de datos puede ser visto como el formato de los datos.

Los conjuntos de datos pueden están almacenados en diferentes formatos, los más utilizados para SparkR (y Spark):

- Ficheros planos y CSV
- **Ficheros JSON**
- Ficheros de tipo `avro` (row-based)
- Ficheros de tipo `parquet` (column-based)
- Ficheros de tipo `orc` (column-based)



Repositorio de Datasets para todo el taller

Todos los conjuntos de datos que vamos a tratar para el Taller se encuentran disponibles en el directorio `datasets`. Para consultar, modificar y añadir datasets, ficheros, etc, puedes hacerlo usando el gestor de fichero de `Jupyter`.

Conjuntos de datos disponibles aquí (./Datasets del Taller de SparkR.ipynb).

Trabajo con ficheros en formato CSV

Vamos a revisar todas las funcionalidades que ofrece SparkR para el manejo de CSV

Lectura

Comprobamos que hay en el directorio donde tenemos los datasets

In [ ]:

```
list.files("/SparkR/datasets/",full.names = T,recursive = T)
```

Leemos un fichero concreto de datos en formato `CSV` del directorio `datasets/csv`. El fichero de ejemplo sólo tiene 1000 registros:

Para la lectura de datos con SparkR usamos la función `read.df( )`

In [ ]:

```
# Sólo indicamos un fichero concreto .... No hay problema Spark es muy listo !
 ;)
df <- read.df("/SparkR/datasets/csv/buy_costumers_amazon01.csv", "csv", header =
 "true", inferSchema = "true")
printSchema(df)
count(df)
head(df)

df1 <- filter(df,df$amount>1000.0)

write.df(df1, path = "/tmp/df_full.csv", source = "csv", mode = "overwrite")
```

Si los datasets se encuentran almacenados en HDFS, S3, etc.:

In [ ]:

```
# Si el fichero está en HDFS o S3Amazon

#df <- read.df("hdfs://users/datasets/csv/buy_costumers_amazon01.csv", "csv", he
ader = "true", inferSchema = "true")

#df <- read.df("s3://SparkR/datasets/csv/buy_costumers_amazon01.csv", "csv", hea
der = "true", inferSchema = "true")
```

In [ ]:

```
# Sólo indicamos un fichero concreto .... No hay problema Spark es muy listo !
 ;)
df <- read.df("/SparkR/datasets/csv/buy_costumers_amazon01.csv", "csv", header =
 "true", inferSchema = "true")
print("Estructura sin parsear:")
printSchema(df)

# Creamos un esquema para definir cual será la estructura del fichero a leer.
schema_amazon <- structType(structField("id", "integer"),
                     structField("first_name", "string"),
                     structField("last_name", "string"),
                     structField("buy_hours", "string"),
                     structField("amount", "double"),
                     structField("credit_card", "string"))

df <- read.df("/SparkR/datasets/buy_costumers_amazon01.csv", "csv", header = "tr
ue", schema=schema_amazon)
print("Estructura parseada:")
printSchema(df)
head(df)
```

Si queremos leer todos los ficheros de un directorio sin entrar en los subdirectorios:

In [ ]:

```
# Esto leería todos los ficheros de la carpeta pero no entraría a cada subdirect
orio... Spark no eres muy listo !
df <- read.df("/SparkR/datasets/csv/", "csv", header = "true", inferSchema = "tr
ue", schema=schema_amazon)
count(df)
```

Comprobamos que estructura de ficheros y directorios tenemos. Observamos que en `datasets/csv/` existen subdirectorios, por lo que hay que usar comodines: *

In [ ]:

```
list.files("/SparkR/datasets/csv",full.names = T,recursive = T)
```

Ejecutamos la siguiente instrucción para poder leer todo lo que cuelgue del directorio donde están los CSV.

In [ ]:

```
# Leemos todos los ficheros CSV que haya en el directorio y todo en las subcarpe
tas... Spark que bien, no?
df_full <- read.df("/SparkR/datasets/csv/*/", "csv", header = "true", inferSchem
a = "true")
```

Verificamos que ahora tenemos todos los datos cargados desde todos los ficheros `CSV` de la estructura de directorios:

In [ ]:

```
count(df_full)
```

Escritura

Una vez que hemos realizado transformaciones con los datos del SparkDataFrame, podemos dejarlo en memoria o bien pasarlo a DISCO (local) o HDFS (distribuido).

La API de fuentes de datos puede también ser usada para guardar y almacenar `SparkDataFrames` en múltiples formatos. Por ejemplo podemos almacenar el `SparkDataDrame` desde/hacia otros formatos como `CSV`, `PARQUET` usando la función `write.df`.

Esto da mucha versatilidad, ya que independiente del tipo de fuente, podemos almacenarlo y leerlo desde cualquiera otra fuente. Como no podía ser de otra forma.

In [ ]:

```
# Escritura desde CSV a CSV:
write.df(df_full, path = "datasets/results/df_full.csv", source = "csv", mode =
"overwrite")

# Escritura desde CSV a PARQUET
write.df(df_full, path = "datasets/results/df_full.parquet", source = "parquet",
 mode = "overwrite")
```

En `mode` podemos usar 'append', 'overwrite', 'error', 'ignore'.

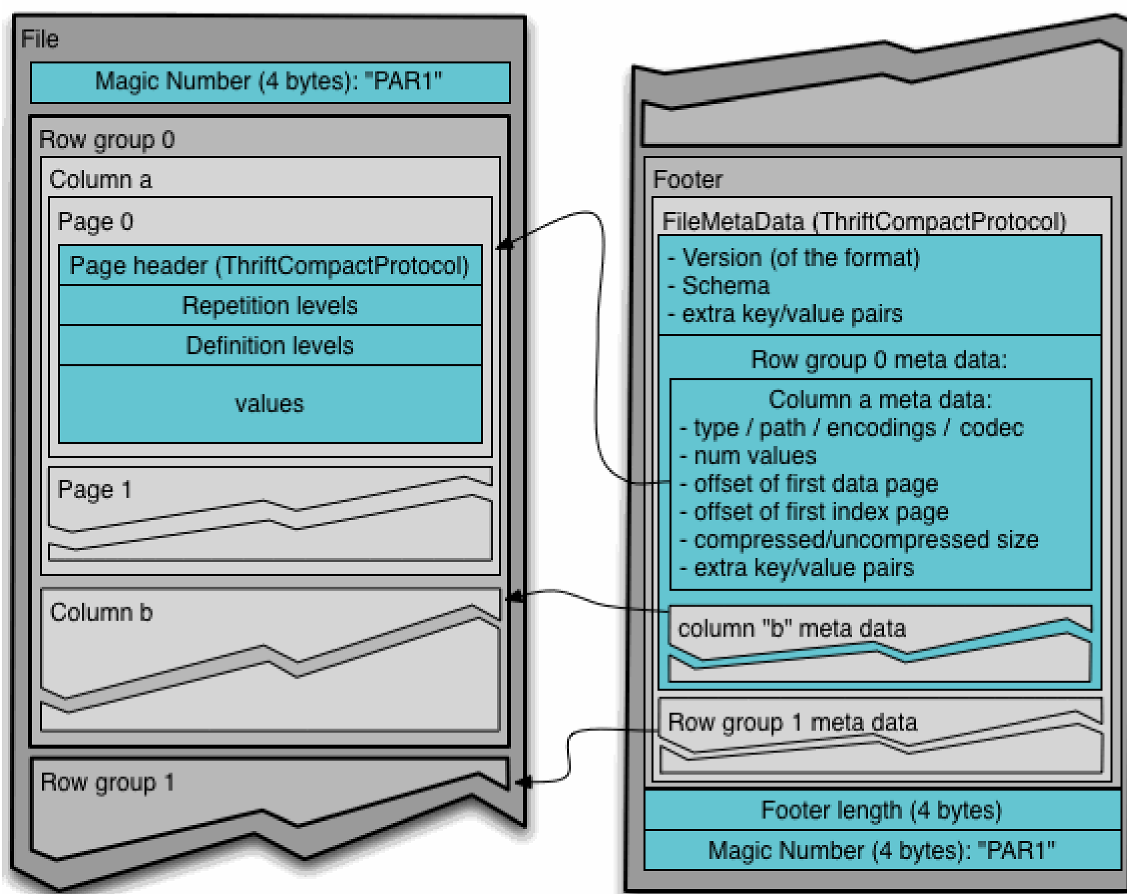
PARQUET

Parquet es un formato de **almacenamiento en columnas** disponible para cualquier proyecto dentro del ecosistema de Hadoop, enfocado en la mejora del procesamiento de datos, modelado de datos y programación.

Parquet está diseñado para soportar esquemas de compresión y codificación muy eficientes. Múltiples proyectos han demostrado el impacto en el rendimiento de aplicar el correcto sistema de compresión y codificación a los datos. Parquet permite que los esquemas de compresión se especifiquen a nivel de columna.

Es un formato bien estructurado para ser usado **para problemas de BigData**.

La estructura del fichero se **segmenta en N columnas partidas en M grupos de filas**:



Leemos el dataset en formato Parquet, luego el resultado de la lectura es un SparkDataFrame, compatible con el trabajo en SparkR

Lectura de los datos

Al igual que con los otros formatos, se pueden exportar a cualquier otro.

In [ ]:

```
# Leemos un dataset que contiene los datos en formato Parquet
df_parquet <- read.df("/SparkR/datasets/parquet/", "parquet")
```

In [ ]:

```
# Vemos la estructura del fichero y sus atributos
printSchema(df_parquet)
```

In [ ]:

```
# Vemos un resumen de los datos del fichero ...
head(df_parquet)
```

In [ ]:

```
# Hacemos un pequeño cambio en el nombre de las columnas del SparkDataFrame.
colnames(df_parquet) <- c("user_id","cat","R1","R2","R3")
```

Vemos de nuevo el cambio de las columnas:

In [ ]:

```
head(df_parquet)
```

In [ ]:

```
# Contamos los registros del dataset ... es pequeño, no es BigData...
count(df_parquet)
```

Aplicamos unas transformaciones sencillas al SparkDataFrame, copiando la tabla en una Vista Temporal para poder trabajar con ella en SQL.

In [ ]:

```
# Creamos una vista SparkDataFrame con el nombre "tmp_parquet".
# Este nombre tmp_parquet es el nombre que se usará ahora.
createOrReplaceTempView(df_parquet,"tmp_parquet")
```

Una vista temporal, permite trabajar con una copia temporal de los datos.

Contamos el número de registros:

In [ ]:

```
# Usamos SparkSQL para hacer consultas a los datos.
count_rows <- sql("SELECT user_id,count(user_id) as registers FROM tmp_parquet g
roup by user_id")
# Cuidado como obtener las cosas en SparkR: ---> Nooooooooo !!!! ;)
# print(collect(count_rows))
```

Compara el tiempo la opción anterior y la siguiente

In [ ]:

```
head(count_rows)
```

Si usamos una vista temporal, está estará disponible durante toda la sesión a no ser que se elimine la vista temporal con `unpersist(....)`

Probamos con otro ejemplo, para saber las categorías que hay:

In [ ]:

```
# createOrReplaceTempView(df_parquet,"tmp_parquet") --> No volvermos a cargarla!
# Usamos SparkSQL para hacer consultas a los datos.
categories <- sql("SELECT cat FROM tmp_parquet group by cat")
```

In [ ]:

```
head(categories)
```

¿Cómo se calcularía el número de elementos de cada categoría?

In [ ]:

```
# createOrReplaceTempView(df_parquet,"tmp_parquet")   --> No volvemos a cargarla
 !
# Usamos SparkSQL para hacer consultas a los datos.
categories_list <- sql("SELECT cat,count(user_id) as num_users FROM tmp_parquet
 group by cat")
```

In [ ]:

```
head(categories_list)
```

¿Cuántos usuarios distintos hay y qué suma total tienen por usuario?

In [ ]:

```
# createOrReplaceTempView(df_parquet,"tmp_parquet") --> No volvemos a cargarla!
# Usamos SparkSQL para hacer consultas a los datos.
table_summary <- sql("SELECT user_id,SUM(R1) as sum_index FROM tmp_parquet group
 by user_id")
```

In [ ]:

```
count(table_summary)
head(table_summary)
```

In [ ]:

```
# Libreamos los recursos de la vista
unpersist(table_summary)
```

Escritura de los datos

Al igual que con los otros formatos, se pueden exportar a cualquier otro.

In [ ]:

```
# Escritura del fichero de formato parquet a formato parquet
write.df(finals, path = "/SparkR/datasets/results/finals.parquet", source = "par
quet", mode = "overwrite")

# Escritura del fichero de formato csv a formato a CSV
write.df(finals, path = "/SparkR/datasets/results/finals.csv", source = "csv", m
ode = "overwrite")
```

JSON

JSON, acrónimo de JavaScript Object Notation, es un formato de texto ligero para el intercambio de datos.

JSON es un subconjunto de la notación literal de objetos de JavaScript aunque hoy, debido a su amplia adopción como alternativa a XML, se considera un formato de lenguaje independiente.

Es un formato actualmente muy utilizado ya que se ha impuesto como modelo para la entrada y salida de datos desde multiples y variados servicios web. Por ejemplo por citar varios:

- Twitter. https://dev.twitter.com/rest/public (https://dev.twitter.com/rest/public)
- Google APIs
- FaceBook API
- ...

Veamos en http://localhost:25980/tree/datasets (http://localhost:25980/tree/datasets) como son los ficheros JSON por dentro.

Es muy utilizado este formato para servicios web de información, donde lo que prima es la sencillez y versatilidad.

Lectura de datos

La sintaxis es la misma, pero varía el identificador del tipo de fuente, en este caso JSON.

In [ ]:

```
costumers <- read.df("/SparkR/datasets/json/buy_costumers_amazon.json", "json")
```

Revisamos la información del SparkDataFrame y su estructura.

In [ ]:

```
# Un resumen
head(costumers)

# El numero de registros
count(costumers)

# La estructura
printSchema(costumers)
```

¿Por qué no se lee correctamente el JSON? Revisamos el dataset http://localhost:25980/tree/datasets (http://localhost:25980/tree/datasets) y arreglamos el error.

La lectura de multiples ficheros es similar lo que ocurre con CSV, donde podemos indica que hay más archivos que queremos leer desde el directorio.

Lectura desde varios ficheros JSON:

In [ ]:

```
costumers_double <- read.json(c("/SparkR/datasets/json/buy_costumers_amazon.json", "/SparkR/datasets/json/buy_costumers_amazon.json"))
```

¿Cuántos registros hay ahora?

In [ ]:

```
count(costumers_double)
```

Escritura de datos a formato JSON

Al igual que para los otros formatos se usa el mismo esquema para guardar datasets a disco.

In [ ]:

```
write.df(costumers_double, path = "datasets/results/costumers.json", source = "json", mode = "overwrite")
```

Ejercicio práctico:

Lee el dataset `/SparkR/datasets/BNGhearth.dat` que está en formato CSV y guardalo directamente en formato `PARQUET```. Verifica que el nuevodataset`` está en el formato correcto.

Como siempre cerramos la sesion de Spark

In [ ]:

```
sparkR.session.stop()
```

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/). **University of Granada**.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Operaciones sobre SparkDataFrames

Como siempre para todos nuestros `scripts` con **SparkR**, cargamos la biblioteca, y creamos una nueva sesión de SparkR.

En este caso:

> Cuidado con la cantidad de MEMORIA que usamos para esta sección !

In [ ]:

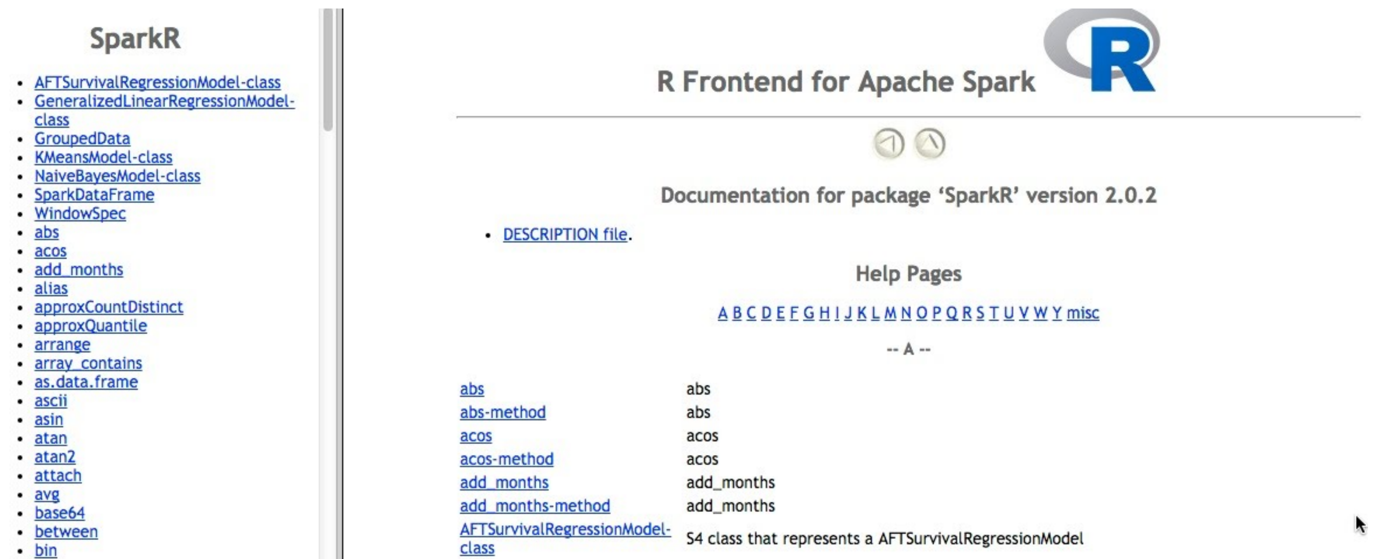
```
#Fijamos la ruta donde está instalado Spark
Sys.setenv("SPARK_HOME"='/usr/local/spark/')

.libPaths(c(file.path(Sys.getenv("SPARK_HOME"),"R/lib/"),.libPaths()))
library(SparkR)
sparkR.session(appName="EntornoInicio", master = "local[*]", sparkConfig =
list(spark.driver.memory = "1g"))
```

Los `SparkDataFrames` soportan un alto número de funciones para hacer un procesado de datos estructurado.

Vamos a poner en práctica las más utilizadas.

La lista completa de operaciones que se pueden aplicar se puede ver desde API de SparkR en [https://spark.apache.org/docs/latest/api/R/index.html (https://spark.apache.org/docs/latest/api/R/index.html)](https://spark.apache.org/docs/latest/api/R/index.html)



Operaciones con SparkDataFrames

In [ ]:

```
# Cargamos una versión reducida de los datos en CSV
df_nyctrips <- read.df("/SparkR/datasets/yellow_tripdata_2016-02_small1.csv", "c
sv", header = "true", inferSchema = "true")
```

In [ ]:

```
# Probamos de nuevo sin INFERIR SCHEMA

# Cargamos una versión reducida de los datos en CSV
df_nyctrips <- read.df("/SparkR/datasets/yellow_tripdata_2016-02_small1.csv", "c
sv", header = "true", inferSchema = "false")
```

¿Cuál de las dos sentencias anteriores ha tardado más?

Estudiamos de manera superficial el dataset

In [ ]:

```
# Comprobamos los campos del dataset
printSchema(df_nyctrips)

# Comprobamos como son los datos:
head(df_nyctrips)

# Contamos el total del registros:
count(df_nyctrips)
```

Selección de instancias y columnas

Para la selección de columnas y filas, usamos `select` y `filter`.

Todas las operaciones se pueden combinar para producir un nuevo dataset o `SparkDataFrame`. **Son equivalentes a usar SPARKSQL** .

Estas operaciones son esenciales si queremos transformar el dataset en otra versión preprocesada del mismo.

In [ ]:

```
# Seleccionamos sólo la columna longitud, por el id de la columna
# Por ID de columna
head(select(df_nyctrips,df_nyctrips$pickup_longitude))
```

In [ ]:

```
# Seleccionamos sólo la columna longitud, por el nombre de la columna.
# Por nombre de columna del dataset
head(select(df_nyctrips,"pickup_longitude"))
```

Para aplicar filtros de para las filas usamos `filter` que admite expresiones con operadores condicionales:

```
< = > ! & | ...
```

In [ ]:

```
# Aplicamos un filtro para ver los viajes aquellos viajes de taxi de más de 10 m
illas.
head(filter(df_nyctrips, df_nyctrips$trip_distance > 10 & df_nyctrips$total_amou
nt> 20 ))
```

Ejercicio práctico:

Selecciona todos los viajes que se hacen desde las 10 de la noche a las 6 de la mañana que tienen más de 3 pasajeros

---

In [ ]:

```
# Aplicamos un filtro para ver los viajes aquellos viajes de taxi de más de 10 m
illas y el importe mayor de $ 20
head(filter(df_nyctrips, df_nyctrips$trip_distance > 10 & df_nyctrips$total_amou
nt> 20 ))
```

Para agrupar datos se usa `agg`.

In [ ]:

```
# Aplicamos un filtro para ver el viaje más caro en Taxi que se ha hecho:
head( agg(df_nyctrips ,max = max(df_nyctrips$total_amount)))
```

In [ ]:

```
# Aplicamos un filtro para ver el viaje menos caro en Taxi que se ha hecho:
head(agg(df_nyctrips, min = min(df_nyctrips$total_amount)))
```

Ejercicio práctico:

- Calcula cual es el viaje más largo que se ha hecho en kilometros (1 milla aprox 1.6 kilómetros).

Uso de Agrupamiento y Agregación

Los SparkDataFrames soportan funciones de agregado despues de agrupar:

- `groupBy`
- `summarize`

Por ejemplo podemos utilizar lo siguiente:

In [ ]:

```
# Agrupamos por Vendedor y mostramos el número de viajes.
head(summarize(groupBy(df_nyctrips, df_nyctrips$VendorID), count =
n(df_nyctrips$VendorID)))
```

In [ ]:

```
# Agrupamos por Vendedor y mostramos el número de viajes.
head(summarize(groupBy(df_nyctrips, df_nyctrips$VendorID), max =
max(df_nyctrips$total_amount)))
```

In [ ]:

```
# Agrupamos y ordenamos

numsum <- summarize(groupBy(df_nyctrips, df_nyctrips$VendorID), num = n(df_nyctr
ips$VendorID))
head(arrange(numsum,asc(numsum$num)))
```

In [ ]:

```
# Agrupamos por numero de pasajeros y mostramos el numero de viajes
trips_passenger <- summarize(groupBy(df_nyctrips, df_nyctrips$passenger_count),
count = n(df_nyctrips$passenger_count))
```

In [ ]:

```
# Cuidado con el COLLECT !
trips_df <- head(collect(trips_passenger))
```

Ejercicio práctico:

¿Qué ocurre si hacemos `collect` de un SparkDataFrame?

In [ ]:

```
head(trips_df)
```

Operaciones con columnas

Otras operaciones en R, corresponden con la manipulación o transformación de valores en los registros de un dataset. En este caso la manipulación es muy sencilla:

In [ ]:

```
# Convertimos la columna de millas a kilómetros, igual que en R.
df_nyctrips$trip_distance <- df_nyctrips$trip_distance*1.6
```

In [ ]:

```
head(df_nyctrips)
```

Añadir columnas

In [ ]:

```
# Usamos mutate para añadir columnas que operan con elementos de las demás colum
nas.

# mutate(sql_nyc,  uniform = rand(10),  normal  = randn(27))

head(mutate(df_nyctrips,  uniform = rand(10),  normal  = randn(27)))
head(mutate(df_nyctrips,  uniform =df_nyctrips$total_amount*1.1355,  normal  = r
andn(27)))
```

Pregunta

Añáde una columna que sea el tiempo del viaje. Pista `INT(unix....())`.

In [ ]:

```
# Otro modo de hacerlo es:

head(withColumn(df_nyctrips,"uniform",rand(20)))
```

dapply -- dapplayCollect

Aplicar una función a un conjunto datos masivo con `dapply` y `dapplyCollect`

dapply

Aplica una función a cada partición de un `SparkDataFrame`. La función que será aplicada para cada partición y debería tener sólo un parámetro. La salida de la función deberá ser igualmente un data.frame. Además hay que especificar el `schema` del formato de los datos del `SparkDataFrame` resultante y deberá corresponder con tipo de datos del valor devuelto.

In [ ]:

```
# Hacemos una copia del SparkDataFrame para usarla en una vista temporal en SQL
createOrReplaceTempView(df_nyctrips,"slqdf_filtered_nyc")

# Hacemos una selección de los registros, donde calculamos el tiempo del viaje d
e cada viaje
sql_nyc <- sql("select VendorID,INT(unix_timestamp(tpep_dropoff_datetime)- unix_
timestamp(tpep_pickup_datetime)) AS trip_time,passenger_count,trip_distance,tota
l_amount from slqdf_filtered_nyc")

# Mostramos un trozo de SparkDataFrame
head(sql_nyc)

schema(sql_nyc)

# Indicamos el Schema, que debe coincidir con lo que queremos
schema <- structType(
    structField("VendorID", "integer"),
    structField("trip_time", "integer"),
    structField("passenger_count", "integer"),
    structField("trip_distance", "double"),
    structField("total_amount", "double"),
    structField("total_amount_euro", "double")
)

# Creamos la función que hará los cambios.
new_sql_nyc <- dapply(
    sql_nyc,
    function(x) {
        x <- cbind(x, x$total_amount*1.1355)
    },
    schema)

# Vemos el cambio
head(new_sql_nyc)
```

gapply -- gapplyCollect

Aplica una función a cada uno de los grupos de un `SparkDataFrame`. La función será aplicada a cada grupo del `SparkDataFrame` y debería tener sólo dos parámetros: agrupamiento por llave y data.frame al que corresponde esa llave. La salida de la función debería ser un data.frame.

In [ ]:

```
# Esquema del SparkDataFrame
schema <- structType(
    structField("VendorID", "integer"),
    structField("trip_time", "integer"),
    structField("passenger_count", "integer"),
    structField("trip_distance", "double"),
    structField("total_amount", "double"),
    structField("max_amount", "double")
)

# Aplicamos la función gapply. Calculamos el máximo de cada Vendedor.
result <- gapply(
    sql_nyc,
    "VendorID",
    function(key, x) {
        y <- data.frame(key, max(x$total_amount))
    },
    schema)

# Mostramos el resultado.
head(result[order(result$trip_distance, decreasing = TRUE), ])
```

In [ ]:

```
head(sql_nyc)

# Ahora probamos el gapplycollect:
# Como el gapply, aplica una funcion a cada partición y luego hace un collect de
l resultado en un data.frame en R.
result <- gapplyCollect(

    sql_nyc,
    "VendorID",
    function(key, x) {
        y <- data.frame(key, max(x$trip_distance))
        colnames(y) <- c("VendorID", "max_trip_distance")
        y
    })

# Vemos el resultado.
head(result[order(result$trip_distance, decreasing = TRUE), ])
```

Operando con SparkSQL sobre cojuntos masivos de datos.

Todas las funciones de manejo de datos que se han usado con SparkR, pueden hacerse de una forma sencilla e intuitiva con SparkSQL

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/). **University of Granada**.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Operaciones sobre SparkDataFrames: `pipes` y `magrittr`

Como siempre para todos nuestros `scripts` con **SparkR**, cargamos la biblioteca, y creamos una nueva sesión de SparkR.

En este caso:

> Cuidado con la cantidad de MEMORIA que usamos para esta sección !

In [ ]:

```
#Fijamos la ruta donde está instalado Spark
Sys.setenv("SPARK_HOME"='/usr/local/spark/')

.libPaths(c(file.path(Sys.getenv("SPARK_HOME"),"R/lib/"),.libPaths()))
library(SparkR)
sparkR.session(appName="EntornoInicio", master = "local[*]", sparkConfig =
list(spark.driver.memory = "1g"))
```

Los `SparkDataFrames` soportan un alto número de funciones para hacer un procesado de datos estructurado.

Vamos a poner en práctica las más utilizadas.

La lista completa de operaciones que se pueden aplicar se puede ver desde API de SparkR en [https://spark.apache.org/docs/latest/api/R/index.html (https://spark.apache.org/docs/latest/api/R/index.html)](https://spark.apache.org/docs/latest/api/R/index.html)

Operaciones con SparkDataFrames

In [ ]:

```
# Primero cargamos los datos
df_nyctrips <- read.df("/SparkR/datasets/yellow_tripdata_2016-02_small1.csv", "c
sv", header = "true", inferSchema = "true")
```

In [ ]:

```
# Hacemos una copia del SparkDataFrame para usarla en una vista temporal en SQL
createOrReplaceTempView(df_nyctrips,"slqdf_filtered_nyc")

# Hacemos una selección de los registros, donde calculamos el tiempo del viaje d
e cada viaje
sql_nyc <- sql("select VendorID,INT(unix_timestamp(tpep_dropoff_datetime)- unix_
timestamp(tpep_pickup_datetime)) AS trip_time,passenger_count,trip_distance,tota
l_amount from slqdf_filtered_nyc")

head(sql_nyc)
```

Uso de magrittr para el trabajo con los datos

El paquete `magrittr` permite:

- mejorar el tiempo de desarrollo y
- mejorar enormemente la legibilidad y mantenibilidad del código.

Para usarlo hay que importar la biblioteca `magrittr` dentro del proyecto y apartir de ese momentos podemos utilizar el operador

```
%>%
```

para concaternar operaciones y poder trabajar con flujos de datos y pipelines.

Provee de un operador que sirve para hacer `pipes` con el cual se puede `encauzar` un valor hacia adelante dentro de una expresión o llamada a función.

Veamos todas las operaciones que hemos realizado sobre los datos y su equivalente con `pipes`.

In [ ]:

```
# Usamos magrittr
library(magrittr)

# results <- sql("select VendorID, MAX(trip_distance) from slqdf_filtered_nyc GR
OUP BY VendorID ")
#summarize(groupBy(df_nyctrips, df_nyctrips$passenger_count), count = n(df_nyctr
ips$passenger_count))

df_nyctrips %>%
        groupBy( df_nyctrips$passenger_count) %>%
        summarize(count = n(df_nyctrips$passenger_count)) %>%
        head()
```

In [ ]:

```
df_nyctrips %>%
        groupBy( df_nyctrips$passenger_count) %>%
        summarize( avg_total_amount=avg(df_nyctrips$total_amount) ,avg_trip_dist
ance=avg(df_nyctrips$trip_distance)) %>%
        head()
```

Ejercicio práctico:

¿Cómo sería la sentencia del bloque superior en SparkSQL?

In [ ]:

```
df_nyctrips %>%
         groupBy( df_nyctrips$passenger_count) %>%
        summarize(min = min(df_nyctrips$trip_distance),max = max(df_nyctrips$tri
p_distance)) %>%
        head()
```

In [ ]:

```
df_nyctrips %>%
         groupBy( df_nyctrips$passenger_count, hour(df_nyctrips$tpep_pickup_date
time)) %>%
        summarize(total_pickup = n(df_nyctrips$tpep_pickup_datetime)) %>%
        head()
```

In [ ]:

```
count(df_nyctrips)

num_regs <- as.integer(count(df_nyctrips))

# Mostramos el número de registros
print(num_regs)
```

Pregunta:

Haz un flujo de funciones con `magrittr` donde selecciones las columnas `total_amount` y `passenger_count`, y filtres cuando `total_amount` sea mayor de `50.0`

In [ ]:

```
# El quivalente al codigo del bloque anterior

sql_nyc %>% count()

num_regs <- sql_nyc %>% count() %>% as.integer()

num_regs %>% print()
```

Pregunta:

¿Cuál es la mejor opción para trabajar: `pipes`, `SparkSQL` o funciones?

In [ ]:

```

```

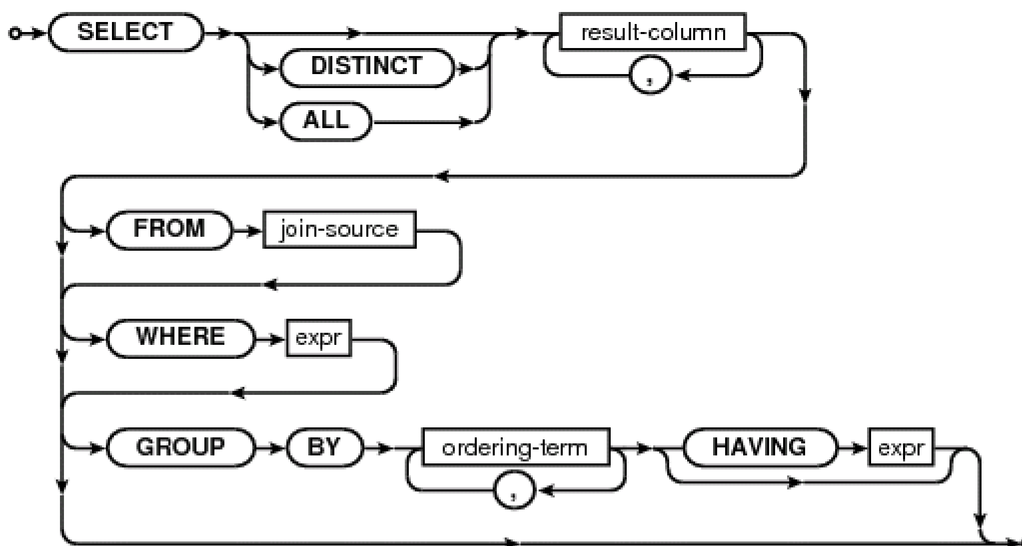
Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/).
University of Granada.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Procesando datos con SparkSQL

SQL (por sus siglas en inglés Structured Query Language; en español lenguaje de consulta estructurada) es un lenguaje específico del dominio que da acceso a un sistema de gestión de bases de datos relacionales que permite especificar diversos tipos de operaciones en ellos. Una de sus características es el manejo del álgebra y el cálculo relacional que permiten efectuar consultas con el fin de recuperar, de forma sencilla, información de bases de datos, así como hacer cambios en ellas. (+info: https://es.wikipedia.org/wiki/SQL (https://es.wikipedia.org/wiki/SQL))



Forma básica:

```
SELECT [ALL | DISTINCT ]
             <nombre_campo>
            FROM <nombre_tabla>
            [WHERE <condición> [{ AND|OR <condición>}]]
            [GROUP BY <nombre_campo> [{,<nombre_campo >}]]
            [HAVING <condición>[{ AND|OR <condición>}]]
            [ORDER BY <nombre_campo>|<indice_campo> [ASC | DESC]
                [{,<nombre_campo>|<indice_campo> [ASC | DESC ]}]]
```

Como siempre para todos nuestros `scripts` con **SparkR**, cargamos la biblioteca, y creamos una nueva sesión de SparkR.

En este caso:

Cuidado con la cantidad de MEMORIA que usamos para esta sección !

In [ ]:

```
#Fijamos la ruta donde está instalado Spark
Sys.setenv("SPARK_HOME"='/usr/local/spark/')

.libPaths(c(file.path(Sys.getenv("SPARK_HOME"),"R/lib/"),.libPaths()))
library(SparkR)
sparkR.session(appName="EntornoInicio", master = "local[*]", sparkConfig =
list(spark.driver.memory = "1g"))
```

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/). **University of Granada**.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Aplicando técnicas de Machine Learning en SparkR

Como siempre para todos nuestros scripts con SparkR, cargamos la biblioteca, y creamos una nueva sesión de SparkR.

In [ ]:

```
#Fijamos la ruta donde está instalado Spark
Sys.setenv("SPARK_HOME"='/usr/local/spark/')

.libPaths(c(file.path(Sys.getenv("SPARK_HOME"),"R/lib/"),.libPaths()))

# Biblioteca SparkR
library(SparkR)

# Abrimos la sesion con SparkR
sparkR.session(appName="EntornoInicio", master = "local[*]", sparkConfig =
list(spark.driver.memory = "1g"))
```

La biblioteca de `SparkR` actualmente soporta los siguientes algoritmos de aprendizaje automático :

- modelo lineal generalizado,
- modelo de regresión de supervivencia con tiempo de fallo acelerado (AFT),
- modelo Bayes Naive y
- modelo KMeans.

SparkR utiliza MLlib para entrenar el modelo. Por tanto se puede analizar el resumen del modelo ajustado, predecir y hacer predicciones sobre nuevos datos y escribir/leer el modelo para guardar / cargar los modelos ajustados.

Además de ello, al igual que ocurre cuando usamos cualquier funcion en R, SparkR soporta el uso de fómulas, lo cual mejora bastante la adopción de SparkR para análisis de datos másivos. SparkR soporta un subconjunto de los operadores de fórmula R disponibles para el ajuste del modelo, incluyendo '~', '.', ':', '+' y '-'.

Funciones enmascaradas

Dado que parte de SparkR está modelado en el paquete dplyr, ciertas funciones de SparkR comparten los mismos nombres con los de dplyr. Dependiendo del orden de carga de los dos paquetes, algunas funciones del paquete cargado primero son enmascaradas por las del paquete cargado después.

`cov in package:stats`

`filter in package:stats`

`sample in package:base`

Por tanto hay siempre que usar el paquete que queramos usar al final de la importación de las bibliotecas para que se haga efectiva la función que queremos para SparkR.

Algoritmos

El paquete SparkR soporta las siguientes funcionalidades de Machine Learning y Data mining

Generalized Linear Model

`Spark.glm()` o `glm()` se ajusta a un modelo lineal generalizado contra un Spark DataFrame. Actualmente se apoyan las familias "gaussianas", "binomiales", "poisson" y "gamma".

Usamos la función de R

In [ ]:

```
gaussianDF <- iris
gaussianTestDF <- iris
gaussianGLM <- glm(data = gaussianDF, Sepal.Length ~ Sepal.Width + Species, fami
ly = "gaussian")

summary(gaussianGLM)
```

Usamos la función para glm de SparkR

In [ ]:

```
irisDF <- suppressWarnings(createDataFrame(iris))
# Fit a generalized linear model of family "gaussian" with spark.glm
gaussianDF <- irisDF
gaussianTestDF <- irisDF
gaussianGLM <- spark.glm(gaussianDF, Sepal_Length ~ Sepal_Width + Species, famil
y = "gaussian")

# Model summary
summary(gaussianGLM)
```

¿Qué diferencias hay entre las dos?

Calculamos el modelo

In [ ]:

```
# Calculamos la predicción
gaussianPredictions <- predict(gaussianGLM, gaussianTestDF)
# Mostramos las predicciones
showDF(gaussianPredictions)

# Usamos la función de R de glm con la familia gaussian
gaussianGLM2 <- glm(Sepal_Length ~ Sepal_Width + Species, gaussianDF, family =
"gaussian")
summary(gaussianGLM2)

# Ahora usamos la función de glm de spark para la familia binomial.
binomialDF <- filter(irisDF, irisDF$Species != "setosa")
binomialTestDF <- binomialDF
binomialGLM <- spark.glm(binomialDF, Species ~ Sepal_Length + Sepal_Width, famil
y = "binomial")

# Imprimimos el modelo
summary(binomialGLM)

# Obtenemos la predicción
binomialPredictions <- predict(binomialGLM, binomialTestDF)
showDF(binomialPredictions)
```

Accelerated Failure Time (AFT) Survival Regression Model

`spark.survreg()` ajusta a un modelo de regresión AFT (accelerated failure time) sobre un SparkDataFrame. Para esta función no se permite el uso del operador . en la formula.

In [ ]:

```
# Use the ovarian dataset available in R survival package
library(survival)

ovarianDF <- suppressWarnings(createDataFrame(ovarian))

head(ovarianDF)

aftDF <- ovarianDF
aftTestDF <- ovarianDF

#Aplicamos la función
aftModel <- spark.survreg(aftDF, Surv(futime, fustat) ~ ecog_ps + rx)

# Model summary
summary(aftModel)

# Prediction
aftPredictions <- predict(aftModel, aftTestDF)
showDF(aftPredictions)
```

K-Means

`Spark.kmeans()` se ajusta a un modelo de clustering de k-means contra un SparkDataFrame, de forma similar a `kmeans()` de R.

In [ ]:

```
library(ggplot2)
ggplot(iris, aes(Petal.Length, Petal.Width, color = Species)) + geom_point()

set.seed(20)
irisCluster <- kmeans(iris[, 3:4], 3, nstart = 20)
irisCluster
```

In [ ]:

```
# Ajustamos un modelo k-medias.

irisDF <- suppressWarnings(createDataFrame(iris))
kmeansDF <- irisDF
kmeansTestDF <- irisDF
kmeansModel <- spark.kmeans(kmeansDF, ~ Sepal_Length + Sepal_Width + Petal_Lengt
h + Petal_Width,
                            k = 3)

# Vemos el resumen
summary(kmeansModel)

# Vemos los resultados del ajuste
showDF(fitted(kmeansModel))

# y vemos la predicción
kmeansPredictions <- predict(kmeansModel, kmeansTestDF)
showDF(kmeansPredictions)

# Mostramos la información de los grupos.
table(kmeansModel$cluster, iris$Species)
```

Naive Bayes

In [ ]:

```
### Aplicamos NAIVE.BAYES

titanic <- as.data.frame(Titanic)
titanicDF <- createDataFrame(titanic[titanic$Freq > 0, -5])
nbDF <- titanicDF
nbTestDF <- titanicDF
nbModel <- spark.naiveBayes(nbDF, Survived ~ Class + Sex + Age)

# Resumen del modelo
summary(nbModel)

# Predicción
nbPredictions <- predict(nbModel, nbTestDF)
showDF(nbPredictions)
```

Creación de Conjuntos de entrenamiento y prueba

Existen varias formas de hacer los conjuntos de prueba y test. Se pueden usar las funciones de muestreo (sample) que trabajan sobre los SparkDataFrames.

In [ ]:

```
train_df <- sample(df_training, withReplacement=FALSE, fraction=0.85, seed=42)
test_df <- except(df_training, train_df)

count(train_df)
count(test_df)
```

Los `SparkDataFrames` y SparkSQL soportan un alto número de funciones para hacer un procesado de datos estructurado.

Vamos a poner en práctica las más utilizadas. La lista completa de operaciones que se pueden aplicar se puede ver desde API de SparkR en https://spark.apache.org/docs/latest/api/R/index.html (https://spark.apache.org/docs/latest/api/R/index.html)

Operando con SparkSQL sobre cojuntos masivos de datos.

Una utilidad importante de Spark SQL es ejecutar consultas SQL.

Spark SQL también se puede utilizar para leer datos de una instalación de HIVE existente. Al ejecutar SQL desde otro lenguaje de programación, los resultados se devolverán como Dataset / DataFrame. También puede interactuar con la interfaz SQL utilizando la línea de comandos o sobre JDBC / ODBC.

Todas las funciones de manejo de datos que se han usado con SparkR, pueden hacerse de una forma sencilla e intuitiva con SparkSQL

In [ ]:

```
df_nyctrips <- read.df("/SparkR/datasets/yellow_tripdata_2016-02_small1.csv", "csv", header = "true", inferSchema = "true")
```

¿Cómo se crea un vista de un SparkDataFrame?

In [ ]:

```
# df_nyctrips es nuestro DataFrameSpark de SQL y el nombre que le pondremos a la
 'vista' es slqdf_filtered_nyc
# y será usado para trabajar desde SPARKSQL.
createOrReplaceTempView(df_nyctrips,"slqdf_filtered_nyc")
```

Para aplicar una consulta a una vista de un SparkDataFrame usamos la función SQL `sql` e indicamos como nombres de las tablas, las vistas que hemos creado de SparkDataFrames disponibles.

In [ ]:

```
# Hacemos una consulta para los 3 primeros registros del dataset.
results <- sql("select * from slqdf_filtered_nyc LIMIT 3 ")
```

In [ ]:

```
# Vemos el resultado.
head(results)
```

Buscamos el total de kilómetros recorridos por cada vendedor:

In [ ]:

```
results <- sql("select VendorID, SUM(trip_distance) from slqdf_filtered_nyc GROU
P BY VendorID ")

# Vemos el resultado
head(results)
```

In [ ]:

```
results <- sql("select VendorID, SUM(trip_distance) as Dist from slqdf_filtered_
nyc GROUP BY VendorID ")

# Vemos el resultado
head(results)
```

Ejercicio práctico:

Crea una SparkDataDrame que se llame `sql_nyc` y que tenga una nueva columna que calcule el tiempo de cada viaje en segundos; llama a esa nueva columna `trip_time` y contenga los campos: `VendorID`, `passenger_count`,`trip_distance`,``total_amount```

Pista: INT(unix_timestamp(tpep_dropoff_datetime)- unix_timestamp(tpep_pickup_datetime))

Necesitamos ese SparkDataFrame para poder seguir con los ejemplos siguientes

---

Calculamos el tiempo en segundos consumido en los viajes de cada Vendedor.

In [ ]:

```
results <- sql("select VendorID, SUM(trip_time) from sql_nyc GROUP BY VendorID
")

# Vemos los resultados
head(results)
```

Calculamos el tiempo en minutos

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/).
University of Granada.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Biblioteca `sparklyr`

Este paquete admite la conexión a clústeres con Apache Spark locales y remotos. Proporciona un backend compatible con "dplyr" y proporciona una interfaz para los algoritmos de aprendizaje de máquina incorporados en Spark.

Permite

- Conectar a Spark desde R. El paquete sparklyr proporciona un backend dplyr .
- Filtrar y agrupar los datasets de Spark para el análisis y visualización.
- Utilizar la biblioteca de aprendizaje distribuida de Sparks de R.
- Crear extensiones que llamen a la API Spark completa y proporcione interfaces con los paquetes Spark.

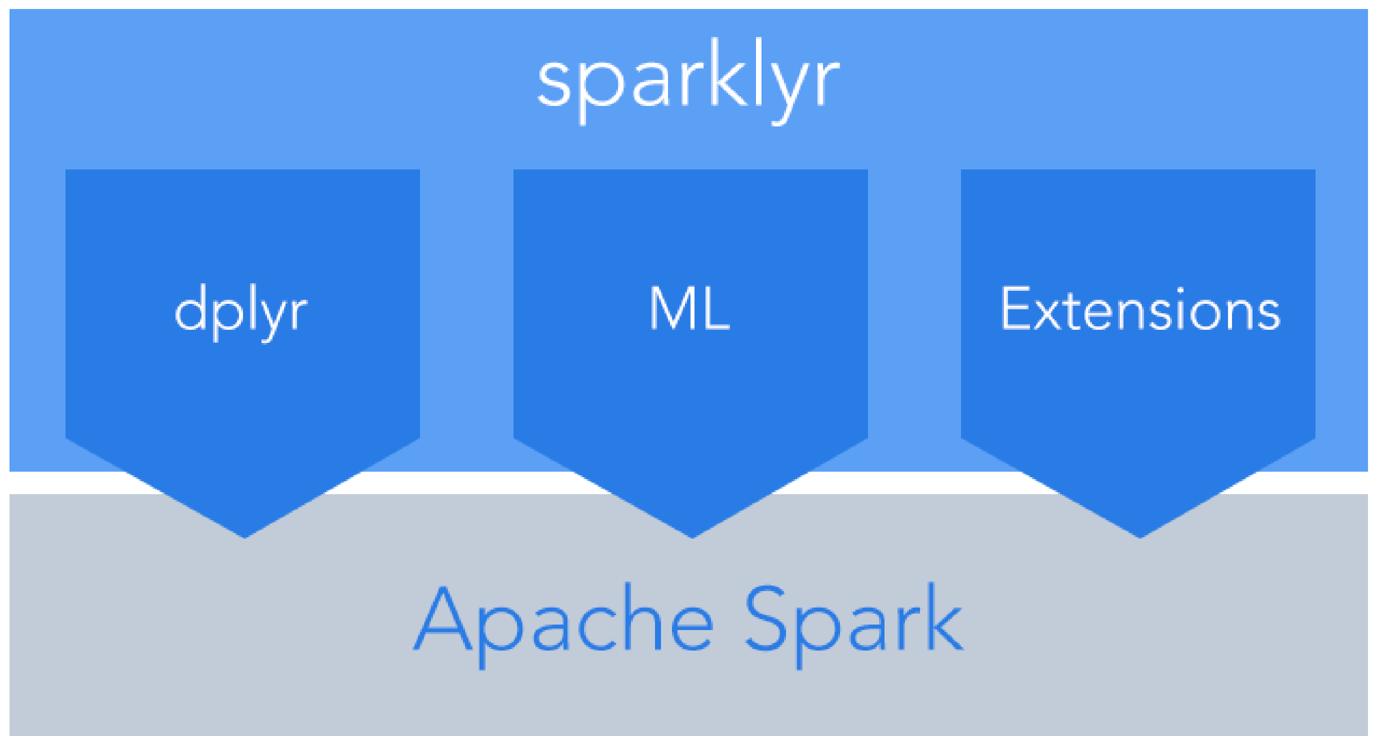
Además junto con la interfaz `dplyr` de `sparklyr`, puede crear y afinar fácilmente los flujos de trabajo de ML en Spark, orquestados dentro de R.

Sparklyr proporciona tres familias de funciones que puede utilizar con el aprendizaje de máquina Spark:

- Algoritmos de aprendizaje automático para el análisis de datos (funciones `ml_*`)
- Transformadores de características para manipular características individuales (funciones `ft_*`)
- Funciones para manipular Spark DataFrames (funciones `sdf_*`)

Disponible:

https://github.com/rstudio/sparklyr (https://github.com/rstudio/sparklyr)



El paquete `sparklyr` proporciona una interfaz `dplyr` a **Spark DataFrames**, así como una interfaz R para los méodos de ML de Spark.

Spark es un sistema de computación de clusters de uso general, y existen muchas otras interfaces R que podrían ser implementadas (por ejemplo, interfaces con pipelines de ML, interfaces con paquetes de Spark de terceros, etc.).

El flujo de trabajo para el análisis de datos con sparklyr podría estar compuesto de las siguientes etapas:

- Realizar consultas SQL a través de la interfaz sparklyr dplyr,
- Utilizar la familia de funciones sdf *y ft* para generar nuevas columnas o particionar su conjunto de datos,
- Elegir un algoritmo de aprendizaje automático apropiado de la familia de funciones ml_ * para modelar los datos,
- Inspeccionar la calidad del ajuste de su modelo y usarlo para hacer predicciones con nuevos datos,
- Recopilar los resultados para la visualización y análisis posterior en R

Inicialización del entorno con `sparklyr`

Es necesario reiniciar Spark para poder trabajar con esta sesión de sparklyr
Para ello, ve a la Máquina Virtual para el proceso de Spark, para luego volver a cargarlo

In [ ]:

```
# Usamos la libreria sparklyr y dplyr.
# Ajustar el nivel de visualización de errores !
options(warn=0)

# Incluimos la bilbioteca de sparklyr
library(sparklyr)
# Usamos la biblioteca para el manejo de los datos.
library(dplyr)

# Abrimos la conexión. Importante indicar la versión de Spark que tenemos instal
ada. En nuestro caso tenemos la 2.0.1
sc <- spark_connect(master = "local", version = "2.0.1")
```

Características

La biblioteca sparklyr tiene asociado un paquete que hace de complemento ideal para la manipulación de datos masivos. Este paquete es dplyr un paquete en R para trabajar con datos estructurados dentro y fuera de R. dplyr hace la manipulación de datos muy sencilla para los usuarios de R, además ofrece interfaces consistentes y con un buen rendimiento. La librería tiene las siguientes funcionalidades básicas:

- Seleccion, filtrado y agregación.
- Funciones para muestreo.
- Funciones de JOIN para Dataframes.
- Funciones Collect para transformar datos de Spark a R (importante!). ...

Lectura y escritura de datos con sparklyr

Para la **lectura** de datos tenemos las siguientes funciones:

- spark_read_csv: Lee un CSV y el resultado lo hace compatible con las funciones de dplyr.
- spark_read_json: Lee un fichero JSON y el resultado es compatible con la interfaz de dplyr.
- spark_read_parquet: Lee un fichero PARQUET.

Además del formato de los datos, Spark soporta la lectura de datos desde una variedad de fuentes de datos. Estos incluyen, almacenamiento en

- HDFS (hdfs:// protocol),
- Amazon S3 (s3n:// protocol), o
- ficheros locales disponibles en en los nodos (file:// protocol).

Para la **escritura** de DataFrames existen las mismas funcione según el tipo de fuente de datos:

- spark_write_csv: Escribe a CSV y recibe una fuente de datos compatible con dplyr.
- spark_write_json: Escribe a JSON.
- spark_write_parquet: Escribe a parquet desde cualquier fuente compatible con dplyr.

In [ ]:

```
# Lectura de un fichero de datos CSV

tttm <- spark_read_csv(sc,
                       name="tttm",
                       path="/SparkR/datasets/BNGhearth.dat",
                       delimiter = ",",
                       header=TRUE,
                       overwrite = TRUE)
```

¿Ha cargado los datos rápido?

In [ ]:

```
count(tttm)
```

La escritura de datos es sencilla y simplemente requiere la función concreta para almacenar los datos. El valor del parámetros path puede ser de diferente origen de datos:

- HDFS (`path="hdfs://...."`)
- AmazonS3 (`path="s3://..."`)
- Local (`path="..."`)

In [ ]:

```
# Escritura de un fichero de datos CSV (en local)
spark_write_csv(tttm,
                path="/SparkR/datasets/results/BNGhearth_RESULT.csv",
                delimiter = ",",
                header=TRUE)

# Escritura de un fichero en S3 de Amazon
# spark_write_csv(tttm,
#                 path="s3://mybucket/BNGhearth_RESULT.csv",
#                 delimiter = ",",
#                 header=TRUE)


# Escritura de un fichero en HDFS
# spark_write_csv(tttm,
#                 path="hdfs://user/manuelparra/BNGhearth_RESULT.csv",
#                 delimiter = ",",
#                 header=TRUE)
```

Para los demás formatos, se usa la función correspondiente, teniendo en cuenta que la entrada de la función de escritura, siempre tiene que ser compatible con dplyr

In [ ]:

```
#spark_write_json(tttm, .....

#spark_write_parquet(tttm, .....
```

Filtrado, selección, agrupación.

Las funciones de manejo de datos son comandos `dplyr` para manipular datos

Cuando se conecta a un Spark DataFrame, `dplyr` traduce los comandos a las sentencias `Spark SQL`. Las fuentes de datos remotas utilizan exactamente los mismos cinco verbos que las fuentes de datos locales. Aquí están los cinco verbos con sus comandos `SQL` correspondientes:

- select ~ SELECT
- filter ~ WHERE
- arrange ~ ORDER
- summarise ~ aggregators: sum, min, sd, etc.
- mutate ~ operators: +, *, log, etc.

In [ ]:

```
# Veamos la cabecera del dataset

head(tttm)
```

In [ ]:

```
# Seleccionamos las columnas que queremos con select

select(tttm, age,sex,class)
```

In [ ]:

```
# Filtro los registros cuando sex y class son 1
res_filtered<-filter(tttm, sex==1 & class==1)

# Contamos los registros.
count(res_filtered)
```

¿Cómo es el dataset `/SparkR/datasets/BNGhearth.dat` : Balanceado o no Balanceado con respecto a la variable de clase `class` ?

In [ ]:

```
res_class1 <- filter(tttm, class==1)
count(res_class1)

res_class0 <- filter(tttm, class==0)
count(res_class0)
```

Otro modo de saber si es o no balanceado de una forma más directa:

In [ ]:

```
# Agrupamos el dataset por clase y luego contamos los registros.

# En SQL sería select count(class) ...group by class

# --> Importante, uso collect(...)
num_regs <- as.integer(collect(count(tttm)))

# Mostramos el número de registros
print(num_regs)

# Agrupamos por clase y contamos el numero de elementos de cada clase, ademñas
# añadimos una columna que calcula el porcentaje que supone cada clase del total
summarize( group_by(tttm,class), count = n(), percent= n()/num_regs *100.0)
```

¿Cómo es el dataset `/SparkR/datasets/databig/ECBDL14_10tst.data` : Balanceado o no Balanceado con respecto a la variable de clase `class` ?

In [ ]:

```
## Ejercicio
tttm2 <- spark_read_csv(sc,
                       name="tttm1",
                       path="/SparkR/datasets/databig/ECBDL14_10tst.data",
                       delimiter = ",",
                       header=TRUE,
                       overwrite = TRUE)
```

In [ ]:

```
## Ejercicio
num_regs <- as.integer(collect(count(tttm2)))

# Mostramos el número de registros
print(num_regs)

# Agrupamos por clase y contamos el numero de elementos de cada clase, ademñas
# añadimos una columna que calcula el porcentaje que supone cada clase del total
summarize( group_by(tttm2,class), count = n(), percent= n()/num_regs *100.0)
```

Preprocesado de dataframes

En `sparklyr` el interfaz de `dplyr` utiliza la SparkSQL para realizar los metodos de procesamiento de datos como agrupación, selección, mutacion, etc.

In [ ]:

```
library(magrittr)

# Atención collect (...)
num_regs <- as.integer(collect(count(tttm)))

# También podemos usar magittr para hacer los mismo de un modo más claro. El eje
mplo de arriba y este son lo mismo.
tttm %>%
    group_by(class) %>%
    summarize(count = n(), percent= n()/num_regs *100.0 )

tttm %>%
    group_by(class) %>%
    summarize(count = n(), maxf1=max(f1),minf1= min(f1))

# Imprimimos los primeros registros
head(tttm)
```

In [ ]:

```
# Añadimos una columna con el doble del valor de la columna f6
tttm <- tttm %>%
             mutate(chest2=chest*2.0)

# Selecciono la columna chest2 que hemos creado recientemente.
tttm %>% select(chest2)
```

In [ ]:

```
# Si del anterior ejemplo queremos construir un nuevo dataframe, usamos
# select para tomar las columnas que nos interesen.
tttm_new <- tttm %>%
               select (age,sex, chest )

head(tttm_new)
```

In [ ]:

```
# La función arrange permite aplicar funciones de ordenación, ... sobre datafram
es.
tttm %>%
    select (age,chest,bloodpressure ) %>%
    arrange (desc(age)) %>%
    filter (chest> 1.5)
```

¿Cómo sabemos cuantos objetos tenemos en el contexto de Spark?

In [ ]:

```
src_tbls(sc)
```

Partición de un SparkDataFrame

Permite la partición de un SparkDataFrame en varios grupos. Esta función es útil para dividir un DataFrame en, por ejemplo, conjuntos de datos de entrenamiento y prueba.

```
sdf_partition(x, ..., weights = NULL, seed = sample(.Machine$integer.max,
 1))
```

Los pesos de muestreo definen la probabilidad de que una determinada observación se asignará a una partición en particular, no el tamaño resultante de la partición. Esto implica que particionar un DataFrame con, por ejemplo, sdf_partition (x, training = 0.5, test = 0.5)

Nota: No está garantizado para producir particiones de entrenamiento y prueba de igual tamaño.

In [ ]:

```
partitions <- tttm %>%
                sdf_partition(training = 0.6, test = 0.4)

print(count(partitions$test))

print(count(partitions$training))
```

Realizar predicciones sobre los datos

Dado un modelo ml_model junto a un nuevo conjunto de datos, producir un nuevo Spark DataFrame con valores predichos codificados en la columna "predicción".

```
sdf_predict(object, newdata, ...)
```

Lectura de una columna

Lee una sola columna de un SparkDataFrame y devuelva el contenido de esa columna en R.

```
sdf_read_column (x, columna)
```

Devuelve el contenido en un objeto manipulable por R.

Registra un SparkDataFrame

Regista un SparkDataFrame asignándole un nombre para la table en el contexto de SparkSQL.

```
sdf_register(x, name = NULL)
```

Permite usar ese DataFrame desde una consulta en SQL con Spark.

Extracción de una muestra de un SparkDataFrame

Extrae un muestreo aleatorio de filas de un SparkDataFrame

```
sdf_sample(x, fraction = 1, replacement = TRUE, seed = NULL)
```

In [ ]:

```
# Indico la fracción de la muestra y la semilla para la extracción.

sdf_sample(tttm,fraction=0.2,seed=98765)
```

Ordenar un SparkDataFrame

Ordena por una o varias columnas, con cada columna ordenada en orden ascendente.

```
sdf_sort(x, columns)
```

In [ ]:

```
sdf_sort(tttm, c("age"))
```

Preparación del dataset para Machine Learning

Al tener un conjunto de datos grande y desbalanceado, podemos tomar varias alternativas para trabajar con el mismo:

- Hacer un sobremuestreo de la clase minoritaria
- Hacer un submuestreo de la clase mayoritaria

Vamos a trabajar con los algoritmos de ML utilizando submuestreo Random UnderSampling sencillo, para ello:

- Calculamos el número de instacias de la clase minoritaria.
- Hacemos un muestreo sólo de la clase mayoritaria para igualar en instancias la clase minoritaria.
- Fusionamos ambos muestreos

In [ ]:

```
# Contamos los registros de la clase minoritaria
regs_minor <- tttm2 %>%
            filter(class==1) %>%
            count %>%
            collect %>%
            as.integer


# Extraemos un sample con un numero similar de instancias que de la clase minori
taria
only_class_0 <- tttm2 %>%
        filter(class==0) %>%
        sdf_sample(regs_minor,
fraction=as.double(regs_minor/as.integer(collect(count(tttm)))))

# Extraemos las instancias de la clase minoritaria
only_class_1 <- tttm2 %>%
            filter(class==1)
```

Contamos los registros de ambos

In [ ]:

```
count(only_class_0)
# as.integer(collect(count(only_class_0)))
# only_class_0 %>% count %>% collect %>% as.integer
count(only_class_1)
# as.integer(collect(count(only_class_1)))
# only_class_1 %>% count %>% collect %>% as.integer
```

Unimos los dos dataframes con rbind

In [ ]:

```
# Unimos
ds_ml <- rbind(only_class_1,only_class_0, name="ds_ml")

# Calculamos el tamaño de cada clase.
ds_ml %>%
        filter(class==0) %>%
        count()

ds_ml %>%
        filter(class==1) %>%
        count()
```

Para el particionado usamos una función de sparklyr: sdf_partition

In [ ]:

```
# La función sdf_partition devuelve el dataframe separado en training y test.
# Para acceder a cada dataframe usamos ...$training , ...$test
partitions <- sdf_partition(ds_ml,training=0.80,test=0.20)
```

Contamos el número de registros de cada conjunto:

In [ ]:

```
count(partitions$test)
count(partitions$training)
```

Usando SPARKSQL para el tratamiento de datos

In [ ]:

```
library(DBI)
```

In [ ]:

```
resultssql <- dbGetQuery(sc,"select sex as sexo, age as edad from tttm LIMIT 10
 ")
print(resultssql)
```

In [ ]:

```
resultssql <- dbGetQuery(sc,"select count(class) as clases1 from tttm where clas
s=1")
print(resultssql)
```

In [ ]:

```
resultssql <- dbGetQuery(sc,"select sex, count(class) as clases1 from tttm group
 by sex")
print(resultssql)
```

In [ ]:

```
resultssql <- dbGetQuery(sc,"select age from tttm where bloodpressure>=130 LIMIT
 10")
print(resultssql)
```

In [ ]:

```
resultssql <- dbGetQuery(sc,"select age from tttm where bloodpressure>=130 order
 by age desc LIMIT 10")
print(resultssql)
```

In [ ]:

```
results <- sql("select VendorID, SUM(trip_time)/60.0 as min_trip from sql_nyc GR
OUP BY VendorID ")

# Vemos los resultados
head(results)
```

Buscamos la ganacia total cada vendedor:

In [ ]:

```
results <- sql("select VendorID, SUM(total_amount)*1.10373 as Total_Amount_Euro
 from slqdf_filtered_nyc GROUP BY VendorID ")

# Vemos el resultado
head(results)
```

Calculamos la media y la desviación típica del tiempo de recorrido y ganancia por numero de personas:

In [ ]:

```
results <- sql("select passenger_count, AVG(trip_time), AVG(total_amount) ,AVG(t
rip_distance)
                from sql_nyc
                GROUP BY passenger_count
                order by passenger_count ASC ")
head(results)
```

Coeficiente de correlación

In [ ]:

```
results <- sql("select corr(total_amount,trip_distance) as correlation_coef
                from
                slqdf_filtered_nyc")
# Ver resultados
head(results)
```

In [ ]:

```
results <- sql("select corr(total_amount,trip_time) as correlation_coef
                from
                sql_nyc")
head(results)
```

Pregunta:

¿Existe alguna correlación entre tiempo de viaje y distancia de viaje?

In [ ]:

```
results <- sql("select corr(trip_time,trip_distance) as correlation_coef
                from
                sql_nyc")
head(results)
```

Ejercicio práctico:

¿ Qué deducimos de estos coeficiente de corelación ?

Un ejemplo más completo:

In [ ]:

```
# Calculamos el número de viajes por hora del día y los dibujamos:
results <- sql("select hour(tpep_pickup_datetime)  as hourpick ,count(*) as numtrips from slqdf_filtered_nyc group by  hour(tpep_pickup_datetime) order by hourpick ASC")

dframeR <- collect(results)

library(ggplot2)
ggplot(data=dframeR, aes(x=hourpick, y=numtrips)) +
    geom_line() + scale_x_continuous(  breaks = c(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23))+
    geom_point()
```

Ejercicio práctico:

Compara la media del tiempo de viaje por hora del día e imprime el gráfico resultante.

Ahora utilizamos el conjunto de datos de BNGHeart para realizar un análisis exploratorio.

In [ ]:

```
heart_df <- read.df("/SparkR/datasets/BNGhearth.dat", "csv", header = "true", inferSchema = "true")

printSchema(heart_df)

head(heart_df)

count(heart_df)
```

In [ ]:

```
# Calculamos por sexos la media de bloodpressure, choresterol y heartrate
createOrReplaceTempView(heart_df,"heart")

res_heart <- sql("SELECT sex, avg(bloodpressure),avg(choresterol),avg(heartrate)
 from heart group by sex")

head(res_heart)
```

Ejercicio práctico:

¿Existe alguna correlación entre `age` y `heartrate`?

Ejercicio práctico:

¿Cuántos individuos hay por clase y por sexo?

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/). **University of Granada**.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Visualización interactiva de datos con SparkR

En esta parte final del taller, vamos a trabajar con una herramienta llamada APACHE ZEPPELIN.

> Este software aún es experimental.

Apache Zeppelin

Es un NOTEBOOK (similar a Jupyter, del que ya sois expertos), que permite el análisis interactivo de datos. Se pueden realizar documentos dirijidos a datos, interactivos y colaborativos, con SPARKSQL, SCALA, R, python y muchos más.

Es un NOTEBOOK Multiproposito y contiene:

- Data Ingestion
- Data Discovery
- Data Analytics
- Data Visualization & Collaboration

Usaremos la versión 0.62 de Apache Zeppelin.

Para inicial el entorno de trabajo con Apache ZEPPELIN accedemos a la Máquina Virtual:

Despues de ejecutar estos comandos se habilita un nuevo puerto en la Máquina Virtual que permite acceder a la siguiente URL:

http://192.168.99.10:9090 (http://192.168.99.10:9090)

Veremos está aplicación en la Nube:

Zeppelin Notebook Search your Notebooks anonymous

Welcome to Zeppelin!

Zeppelin is web-based notebook that enables interactive data analytics.
You can make beautiful data-driven, interactive, collaborative document with SQL, code and even more!

Notebook
Import note
Create new note
Filter
R Tutorial
Zeppelin Tutorial
Zeppelin Tutorial: Python - matplotlib basic

Help
Get started with Zeppelin documentation

Community
Please feel free to help us to improve Zeppelin,
Any contribution are welcome!

Mailing list
Issues tracking
Github

Para comenzar a trabajar, usamos `create new note`.

Le asignamos un nombre. Una vez dentro de la nueva nota usamos:

```
%spark.r
```

Con esto le indicamos que queremos crear un notebook para trabajar con Spark + R.

En Zeppelin, no es necesario abrir la conexión con Spark como se hacía con SparkR, ya que Zeppelin abre una conexión por defecto que se ha confingurado de antemano. Tampoco es necesario importar la biblioteca `sparkR`.

Igualmente podemos trabajar con `sparklry`, pero no tendremos las mejoras que ofrece sparkR para la visualización dinámica de datos.

```
df_nyctrips <- read.df("/SparkR/datasets/yellow_tripdata_2016-02_small3.csv", "csv", header = "true", inferSchema = "true")
createOrReplaceTempView(df_nyctrips,"slqdf_filtered_nyc")
```

Ahora usaremos una celda de tipo SQL

```
%sql
```

Aquí podemos escribir sentencias SQL para poder tratar datos y visualizarlos.

In [ ]:

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/).
University of Granada.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Machine Learning con ``sparklyr```

Biblioteca de Machine Learning Spark (MLlib) con la Interfaz sparklyr

Carácteristicas fundamentales:

La bilbioteca sparklyr proporciona enlaces a la biblioteca de ML distribuida de Spark. En particular, sparklyr le permite acceder a las rutinas de ML proporcionadas por el paquete spark.ml de Spark. Además junto con la interfaz dplyr de sparklyr, puede crear y afinar fácilmente los flujos de trabajo de ML en Spark, orquestados enteramente dentro de R. Sparklyr proporciona tres familias de funciones que puede utilizar con el aprendizaje de máquina Spark:

- Algoritmos de aprendizaje automático para el análisis de datos (funciones ml_*)
- Transformadores de características para manipular características individuales (funciones ft_*)
- Funciones para manipular Spark DataFrames (funciones sdf_*)

El flujo de trabajo para el análisis de datos con sparklyr podría estar compuesto de las siguientes etapas:

- Realizar consultas SQL a través de la interfaz sparklyr dplyr,
- Utilizar la familia de funciones sdf *y ft* para generar nuevas columnas o particionar su conjunto de datos,
- Elegir un algoritmo de aprendizaje automático apropiado de la familia de funciones ml_ * para modelar los datos,
- Inspeccionar la calidad del ajuste de su modelo y usarlo para hacer predicciones con nuevos datos,
- Recopilar los resultados para la visualización y análisis posterior en R

En esta sección vamos a trabajar con las herramientas que proporciona la biblioteca sparklyr para Machine Learning. Como vamos a ver, la funcionalidad es muy similar a la de la biblioteca SparkR, aunque cambia el nombre de las funciones y varios extras más.

Inicialización del entorno

Es necesario reiniciar Spark para poder trabajar con esta sesión de sparklyr

> Es necesario reiniciar Spark para poder trabajar con esta sesión de sparklyr

Ahora para conectar con Spark y abrir una sesión usaremos la siguiente sintaxis (simiar a la del paquete SparkR aunque con ligeras diferencias):

In [ ]:

```
# Usamos la libreria sparklyr y dplyr.

# Ajustar el nivel de visualización de errores !
options(warn=0)

# Incluimos la bilbioteca de sparklyr
library(sparklyr)
# Usamos la biblioteca para el manejo de los datos.
library(dplyr)

# Abrimos la conexión. Importante indicar la versión de Spark que tenemos instal
ada. En nuestro caso tenemos la 2.0.1
sc <- spark_connect(master = "local", version = "2.0.1")
```

Si queremos conectarnos a un entorno Spark completo dentro de un cluster usaríamos la siguiente información:

In [ ]:

```
# Abrimos la conexión ahora a un clúster
sc <- spark_connect(master = "spark://HOST:PORT", version = "2.0.1")

# Donde HOST y PORT deben ser indicados tal y como el adminstrador de clúster te
nga establecido.
```

Parada de la sesion con Spark

In [ ]:

```
# Paramos la sesion y el contexto de Spark, para liberar recursos
spark_disconnect(sc)
```

Algoritmos de ML para el análisis de datos

Con Spark + R y la biblioteca sparklyr se puede orquestar la ejecución de varios algoritmos de ML en un cluster con Spark.

Estas funciones de ML, conectan directamente con la API de Spark, por lo que están vinculadas a la librería MLLib de Spark.

La biblioteca tiene mayor número de funciones para la minería de datos que la propia de SparkR.

Contiene los siguientes métodos

- Alternating Least Squares (ALS) matrix factorization
- Decision Trees
- Generalized Linear Regression
- Gradient-Boosted Tree
- K-Means Clustering
- Latent Dirichlet Allocation
- Linear Regression
- Logistic Regression
- Multilayer Perceptron
- Naive-Bayes
- One vs Rest
- PCA (Principal Components Analysis)
- Random Forests
- Survival Regression

Veamos cada uno de ellos

Alternating Least Squares (ALS) matrix factorization

Realiza la factorización de matriz de mínimos cuadrados alternativos sobre un Spark DataFrame.

```
ml_als_factorization(x, rating.column = "rating", user.column = "user",
  item.column = "item", rank = 10L, regularization.parameter = 0.1,
  implicit.preferences = FALSE, alpha = 1, nonnegative = FALSE,
  iter.max = 10L, ml.options = ml_options(), ...)
```

Decision Trees

Realiza una regresión o clasificación usado árboles de decisión:

```
ml_decision_tree(x, response, features, max.bins = 32L, max.depth = 5L,
  type = c("auto", "regression", "classification"),
  ml.options = ml_options(), ...)
```

Generalized Linear Regression

Realiza regresión línearl generalizada sobre un SparkDataFrame

```
ml_generalized_linear_regression(x, response, features, intercept = TRUE,
  family = gaussian(link = "identity"), iter.max = 100L,
  ml.options = ml_options(), ...)
```

Gradient-Boosted Tree

Aplica el algoritmo GBT sobre un SparkDataFrame

```
ml_gradient_boosted_trees(x, response, features, max.bins = 32L,
  max.depth = 5L, type = c("auto", "regression", "classification"),
  ml.options = ml_options(), ...)
```

K-Means

Aplica el algortimo K-Means para clustering a un SparkDataFrame

```
ml_kmeans(x, centers, iter.max = 100, features = dplyr::tbl_vars(x),
  compute.cost = TRUE, tolerance = 1e-04, ml.options = ml_options(), ...)
```

Latent Dirichlet Allocation

Ajusta un modelo LDA a un SparkDataFrame

```
ml_lda(x, features = dplyr::tbl_vars(x), k = length(features),
  alpha = (50/k) + 1, beta = 0.1 + 1, ml.options = ml_options(), ...)
```

Linear Regression

Aplica una regresión lineal a los datos de un SparkDataFrame

```
ml_linear_regression(x, response, features, intercept = TRUE, alpha = 0,
  lambda = 0, iter.max = 100L, ml.options = ml_options(), ...)
```

Logistic regression

Aplica una regresión logística a los datos de un SparkDataFrame

```
ml_logistic_regression(x, response, features, intercept = TRUE, alpha = 0,
  lambda = 0, iter.max = 100L, ml.options = ml_options(), ...)
```

Multilayer Perceptron

Crea y entrena Percetron Multicapa para un SparkDataFrame:

```
ml_multilayer_perceptron(x, response, features, layers, iter.max = 100,
  seed = sample(.Machine$integer.max, 1), ml.options = ml_options(), ...)
```

Naive-Bayes

Aplica regresión o clasificación utilizando Naive-Bayes:

```
ml_naive_bayes(x, response, features, lambda = 0, ml.options = ml_options
(),
  ...)
```

One vs Rest

Aplica regresión o clasificación utilizando O vs R:

```
ml_one_vs_rest(x, classifier, response, features, ml.options = ml_options
(),
  ...)
```

PCA

Aplica PCA - Principal Components Analysis:

```
ml_pca(x, features = dplyr::tbl_vars(x), ml.options = ml_options(), ...)
```

RandomForest

Aplica Regresión o clasificación utilizando el algoritmo de RandomForest:

```
ml_random_forest(x, response, features, max.bins = 32L, max.depth = 5L,
  num.trees = 20L, type = c("auto", "regression", "classification"),
  ml.options = ml_options(), ...)
```

Utilidades y extensiones para Machine Learning

`sparklyr` ofrece adicionalmente una serie de funciones para interactuar con los modelos ajustados de Spark ML así como operaciones con los modelos

- `ml_binary_classification_eval`. Binary Classification Evaluator
- `ml_classification_eval`. Spark ML - Classification Evaluator
- `ml_tree_feature_importance Spark ML` - Feature Importance for Tree Models
- `ml_saveload` , `ml_load ml_save` - Save / Load a Spark ML Model Fit
- `ml_create_dummy_variables`. Create Dummy Variables
- `ml_model` Create an ML Model Object
- `ml_options` Options for Spark ML Routines
- `ml_prepare_dataframe`. Prepare a Spark DataFrame for Spark ML Routines
- `ml_prepare_response_features_intercept` , `ml_prepare_inputs` , `ml_prepare_features`. Pre-process the Inputs to a Spark ML Routine

Ejemplos y pruebas con ML de ``sparklyr```

Preparamos el dataset:

In [ ]:

```
# Cargamos el dataset a SparkDataFrame:
heart <- spark_read_csv(sc,
                        name="heart",
                        path="/SparkR/datasets/BNGhearth.dat",
                        delimiter = ",",
                        header=TRUE,
                        overwrite = TRUE)


# Creamos un test y un train:
partitions_heart <- sdf_partition(heart,training = 0.8, test = 0.2, seed = 1099)

# Contamos las particiones:
count(partitions_heart$test)
count(partitions_heart$training)

# Imprimimos un resumen para recordar las variables
```

In [ ]:

```
head(partitions_heart$test)
```

Regresión lineal

In [ ]:

```
# La función para la regresión lineal puede usarse con la sintaxis propia
# de la función y tambien con la forma tradicional de R: formulae

# Opción 1
#model <- partitions$training %>%
#    ml_linear_regression(response = "f1", features = c("f2","f3"))

# Opción 2
#model <- partitions$training %>%
#    ml_linear_regression(f1~f2+f3)

# Opción 3
model <- ml_linear_regression(partitions_heart$training,bloodpressure~age)
```

Vemos la calidad del ajuste

In [ ]:

```
# Vemos la calidad del ajuste ...
summary(model)
```

In [ ]:

```
# Utilizamos Predict

predicted <- predict(model, newdata = partitions_heart$test)
```

Regresión logística

In [ ]:

```
# Aplica la función de regresión logística
ml_log <- partitions_heart$training %>%
             ml_logistic_regression(response = "class", features =
c("age","sex","bloodpressure"))

# O también equivalente
# ml_log <- ml_logistic_regression(partitions_heart$training, response = "clas
s", features = c("age","sex","bloodpressure"))
```

In [ ]:

```
summary(ml_log)
```

K-Medias (K-Means)

K-means es un método de agrupamiento, que tiene como objetivo la partición de un conjunto de n observaciones en k grupos en el que cada observación pertenece al grupo cuyo valor medio es más cercano. Forma parte de las técnicas de clustering

In [ ]:

```
# Equivalente:
# model <- ml_kmeans(select(iris_tbl,Petal_Width, Petal_Length),centers = 3)

model <- partitions_heart$test %>%
          select(age, bloodpressure) %>%
          ml_kmeans(centers = 3)

print (model)

# Cuidado con collect !!!
partitions_heart$test %>%
  select(age, bloodpressure) %>%
  collect %>%
  ggplot(aes(age, bloodpressure)) +
    geom_point(data = model$centers, aes(age, bloodpressure), size = 60, alpha =
 0.1) +
    geom_point(aes(age, bloodpressure), size = 2, alpha = 0.5)
```

In [ ]:

```
predicted <- sdf_predict(model,partitions_heart) %>% collect

table(predicted$class,predicted$prediction)
```

RandomForest

Random forest también conocido en castellano como '"Selvas Aleatorias"' es una combinación de árboles predictores tal que cada árbol depende de los valores de un vector aleatorio probado independientemente y con la misma distribución para cada uno de estos. Es una modificación sustancial de bagging que construye una larga colección de árboles no correlacionados y luego los promedia. Se usa para regresión y para clasificación.

In [ ]:

```
ml_rf <-
ml_random_forest(partitions_heart$training,response="class",features=c("age","bl
oodpressure", "chest"),type="classification")
```

In [ ]:

```
print(ml_rf)

res_predict <- sdf_predict(ml_rf, partitions_heart$test)

res_predict_2 <- collect(ft_string_indexer(sdf_predict(ml_rf, partitions_heart$t
est),"class","class_idx"))
```

In [ ]:

```
table(res_predict_2$class_idx,res_predict_2$prediction)
```

In [ ]:

```
head(select(res_predict,age,bloodpressure,chest,class))
```

In [ ]:

```
feature_imp <- ml_tree_feature_importance(sc, ml_rf)
```

In [ ]:

```
features <- as.character(feature_imp[1:3, 2])
```

In [ ]:

```
features
```

PCA Análisis de componentes principales

En estadística, el análisis de componentes principales es una técnica utilizada para reducir la dimensionalidad de un conjunto de datos. Técnicamente, el PCA busca la proyección según la cual los datos queden mejor representados en términos de mínimos cuadrados. Esta convierte un conjunto de observaciones de variables posiblemente correlacionadas en un conjunto de valores de variables sin correlación lineal llamadas componentes principales. El PCA se emplea sobre todo en análisis exploratorio de datos y para construir modelos predictivos.

In [ ]:

```
# PCA
pca_model <- ml_pca(partitions_heart$training)

print(pca_model)
```

Comparación de modelos

In [ ]:

```
# Hacemos una lista para verificar que modelo nos da mejores resultados.
ml_models <- list(
  "Logistic" = ml_log,
#  "Decision Tree" = ml_dt,
  "Random Forest" = ml_rf
#  "Gradient Boosted Trees" = ml_gbt,
#  "Naive Bayes" = ml_nb,
#  "Neural Net" = ml_nn
)

# Create a function for scoring
score_test_data <- function(model, data=partitions$test){
  pred <- sdf_predict(model, data)
  select(pred, class, prediction)
}

# Score all the models
ml_score <- lapply(ml_models, score_test_data)
```

Cierre de la conexión

In [ ]:

```
# Obligatorio al terminar la session
spark_disconnect(sc)
```

Taller de procesamiento de BigData en Spark + R

Manuel Parra (manuelparra@decsai.ugr.es). Soft Computing and Intelligent Information Systems (http://sci2s.ugr.es/es) . Distributed Computational Intelligence and Time Series (http://sci2s.ugr.es/dicits/). **University of Granada**.

Departamento de Estadística e Investigación Operativa
Universidad Pública de Navarra

Utilizando `spark-submit` para enviar trabajos al clúster de Spark

El script `spark-submit` de Spark se utiliza para iniciar aplicaciones en un clúster sin tener que hacerlo de modo interactivo.

En el taller hemos realizado un trabajo interactivo con R. Es posible que sea necesario lanzar un experimento en el clúster sin tener que realizarlo de modo interactivo por las característica propias del desarrollo, como por ejemplo:

- Coste en tiempo, alto
- Análisis profundos
- Baterias de análsis
- etc.

Esto es importante, pues permite enviar trabajos a la plataforma Spark de una forma cómoda independientemente del lenguaje se se haya utilizado para hacer los experimentos y esperar el resultado cuando se haya realizado completamente el experimento (o ejecución) en el clúster.

De modo que se podrán enviar trabajos al clúster en:

- Scala
- Python
- R
- Java

La sintaxis es la siguiente:

```
spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

Las opciones más usadas son:

- **--class** : El punto de entrada de tu aplicación o experimento.
- **--master** : La URL del Master de tu cluster. Si es local usamos `local`
- **--deploy-mode** : Desplegar en los nodos `worker` o localmente (`local`) como cliente externo.
- **--conf** : Configuración genérica de Spark.
- **--num-executors** : Número de nodos que ejecutarán el trabajo
- **--num-cores** : Número de `cores` por nodo ejecutor
- **--executor-memory** : Memoria en `GB` por utilizada máxima por cada nodo ejecutor

Este comando se usa directamente desde el shell de la Máquina virtual

Ejemplo de aplicación en R para `spark-submit`

Este ejemplo recibe dos parámetros desde la línea de comandos, uno con la cantidad (`amount`) y otro con el fichero de salida de los datos (`resultpath`). Conecta con Spark, lee un dataset de ejemplo (`/SparkR/datasets/csv/buy_costumers_amazon01.csv`) y luego filtra ese dataset por la cantidad (`amount`). Finalmente almacena el resultado en la ruta indicada por el parametro de entrada `resultpath`.

Copia el siguiente código en un fichero en R con el nombre `miaplicacion.R`.

```
# Leo los agurmentos
args <- commandArgs(trailingOnly =TRUE)

# Tomo el parametyos amount para hacer alguna operación
amount <- args[1]
resultpath <- args[2]

# Fijamos la ruta donde está instalado Spark
Sys.setenv("SPARK_HOME"='/usr/local/spark/')
.libPaths(c(file.path(Sys.getenv("SPARK_HOME"),"R/lib/"),.libPaths()))

library(SparkR)

# Abro la sesion de SparkR
sparkR.session(appName="EntornoInicio", master = "local[*]", sparkConfig =
 list(spark.driver.memory = "1g"))

# Leo un dataset
df <- read.df("/SparkR/datasets/csv/buy_costumers_amazon01.csv", "csv", he
ader = "true", inferSchema = "true")

# Filtro el dataset por el parametro de entrada a la aplicacion
df1 <- filter(df,df$amount>amount)

#Escribimos el SparkDataFrame df1
write.df(df1, path = resultpath, source = "csv", mode = "overwrite")

sparkR.session.stop()
```

Ejecución para R en local

Para ejecutar una aplicación en R y enviarla como trabajo a Spark, hay que usar la siguiente sentencia:

```
spark-submit
        --deploy-mode <modo>
        --master <master>
        --num-executors <num>
        --executor-memory <mem g>
        <aplicacion.R> <parametro1> <parametro2> <parametro3> ...
```

In [ ]:

```
spark-submit --deploy-mode client  --master local  --num-executors 1  --
executor-memory 1g miaplicacion.R 2000.0 /tmp/resultados.csv
```

Ejecución para R usando Rscript tanto en modo local como en clúster

In [ ]:

```
nohup Rscript miaplicacion.R 2000.0 /tmp/resultados.csv &
```

En R en clúster

Para ejecutar una aplicación en R y enviarla como trabajo a Spark, hay que usar la siguiente sentencia:

In [ ]:

```
spark-submit --deploy-mode cluster  --master spark://HOST:PORT  --num-executors
5  --executor-cores 5  --executor-memory 4g  miaplicacion.R 2000.0 /tmp/resultad
os.csv
```

Ejercicio práctico:

Crea una aplicación con el nombre `seleccion.R` que reciba 4 parámetros:

- nombre de la columna: `colname`
- número de registros: `numrows`
- fichero de entrada: `source_file`
- fichero de salida: `destination_file`

Tome el fichero de entrada, extraiga (seleccione: select(...)) sólo la columna `colname` y sólo `numrows` filas del dataset. El dataset resultante (de una columna y numrows filas) debe ser almacenado en el fichero `destination_file` en CSV.

Prueba como fichero de entrada: `/SparkR/datasets/databig/ECBDL14_10tst.data` y como fichero de salida: `/tmp/salida.csv`

Ejecuta la aplicación utilizando `spark-submit`

Verifica que al terminar de ejecutar el trabajo, aparece el dataset en `/tmp/salida.csv`.