

MÓDULO

HERRAMIENTAS BIG DATA

UNIDAD 5

TECNOLOGÍAS BIG DATA II: SPARK

ÍNDICE

TU RETO EN ESTA UNIDAD.....	3
1. INTRODUCCIÓN	5
1.1. MÁS ALLÁ DEL MODELO MAP-REDUCE	5
1.2. ¿QUÉ ES SPARK?.....	6
1.3. UN POCO DE HISTORIA	7
1.4. ENTORNO DE TRABAJO	8
1.4.1. LANZAR LA CONSOLA INTERACTIVA	9
2. COMPONENTES DE SPARK	11
2.1. SPARK CORE	12
2.1.1. ESTRUCTURAS DE DATOS EN MEMORIA: RDD	12
2.1.2. FLUJOS DE DATOS Y CÁLCULO: DAGS.....	14
2.2. PLATAFORMA DE EJECUCIÓN	18
2.3. COMPONENTES AVANZADOS.....	18
2.3.1. SPARK SQL, DATAFRAMES Y DATASETS.....	18
2.3.2. SPARK MLLIB	23
2.3.3. SPARK STREAMING	24
2.3.4. SPARK GRAPHX.....	24
3. ARQUITECTURA.....	25
3.1. ANATOMÍA DE UNA APLICACIÓN EN SPARK.....	25
4. PRIMEROS PASOS CON SPARK.....	27
4.1. TRABAJANDO EN MODO “STANDALONE”	27
4.1.1. EJECUTAR EL CÓDIGO DE LOS EJEMPLOS.....	28
4.1.2. CARGAR DATOS DE FICHERO.....	28
4.1.3. ACCIONES E INFORMACIÓN BÁSICA SOBRE LOS DATOS	32

4.1.4. MANIPULACION DE DATOS	35
4.1.5. UTILIZANDO MLLIB	44
4.2. APLICACIÓN SPARK	57
¿QUÉ HAS APRENDIDO?	63
AUTOCOMPROBACIÓN	65
SOLUCIONARIO	69
BIBLIOGRAFÍA	71

TU RETO EN ESTA UNIDAD

Cuando entras en un comercio *online* y te muestran algunos productos que puedan interesarte, o cuando te recomiendan canciones o videos, basándose en contenido o productos que ya has visitado, ¿te preguntas qué hay detrás?

Lo que hay son modelos de recomendación basados en *machine learning* aplicado a escala *big data*, cientos de millones de datos, y aprendiendo de forma ininterrumpida.

Apache Spark ofrece una plataforma de computación distribuida para desarrollar, probar y llevar a producción estos algoritmos a escala *big data* de forma sencilla y muy eficiente. Compañías como Amazon o TripAdvisor utilizan Spark en infinidad de análisis y modelos.

En esta unidad no vamos a llegar a cómo construir un modelo de recomendación como el de Amazon, pero sí vas a aprender cómo funciona Spark y cuáles son los puntos fuertes que lo han convertido en una de las plataformas líderes en *big data* y ciencia de datos.

1. INTRODUCCIÓN

1.1. MÁS ALLÁ DEL MODELO MAP-REDUCE

Como viste en la unidad anterior, Hadoop es una plataforma centrada en el procesamiento por lotes de grandes cantidades de información almacenada en ficheros. Su modelo de cálculo se basa fundamentalmente en dividir el trabajo en dos etapas: una etapa para aplicar un conjunto de operaciones sobre una porción de los datos originales (Map) y otra etapa en la que se recolectan y combinan los resultados parciales para obtener el resultado agregado final (Reduce).

El problema es que este esquema es poco eficiente para múltiples problemas y algoritmos habituales que trabajan con datos de forma iterativa. Entre ellos, los algoritmos de análisis estadístico y de *Machine Learning*.

El funcionamiento de este tipo de algoritmos se basa en evaluar de forma repetitiva funciones y parámetros sobre subconjuntos de los mismos datos. En Hadoop, cada iteración o evaluación con distintos parámetros requiere volver a leer los datos de disco (de HDFS). Además, Hadoop MapReduce no ofrece una implementación directa del proceso de iteración sobre los datos. Necesitamos lanzar explícitamente la secuencia completa de réplicas del proceso a iterar.

Estos mismos inconvenientes aparecen también cuando simplemente queremos explorar los datos de forma interactiva, algo necesario para el científico de datos. Hadoop está diseñado para trabajar lanzando lotes de cálculos, no para ser interactivo. Cualquier operación de filtrado, transformación o agregación requiere lanzar una nueva tarea que lea de disco y devuelva los resultados a disco. Este funcionamiento no es eficaz para el trabajo diario de un científico de datos.

En definitiva, Hadoop MapReduce es una herramienta muy potente para el objetivo y tipo de cálculos para el que fue diseñada, pero resulta limitado, lento y poco flexible para adaptarse a necesidades más avanzadas de manejo de datos. Ante estas limitaciones, surge la necesidad de desarrollar nuevas herramientas y plataformas.

1.2. ¿QUÉ ES SPARK?

Apache Spark es un proyecto de código abierto que proporciona una plataforma de computación distribuida flexible, que tiene como característica principal el uso masivo de memoria RAM para acelerar las tareas de procesamiento y cálculo con grandes cantidades de datos.

Además, permite definir de forma simple procesos de cálculo complejos, enlazando múltiples etapas o pasos de cálculo, según sea necesario.

A diferencia de Hadoop MapReduce (con su modelo de cálculo basado fundamentalmente en dos pasos, y con un flujo de datos basado en disco), las características de Spark permiten desarrollar algoritmos muy eficientes para tareas avanzadas que necesitan operar sobre los mismos datos de forma recurrente, como es el caso de los algoritmos de *Machine Learning* o de análisis estadístico.

Spark está desarrollado en el lenguaje Scala (basado en la tecnología de la máquina virtual de Java, JVM), pero ofrece también APIs y librerías para desarrollar y ejecutar programas usando los lenguajes Java, Python y R.

Otra característica fundamental de Apache Spark es precisamente el conjunto de librerías y componentes de alto nivel que incorpora para trabajar con datos, y que incluyen librerías de manejo y consulta de datos (tipo SQL), librerías de *Machine Learning*, de análisis de grafos, o de procesamiento de *streams* de datos.

Estas librerías junto con las características del propio motor de cálculo convierten a Spark en una de las plataformas más potentes, versátiles y fáciles de usar para científicos de datos a la hora de elaborar análisis y modelos, y construir nuevas aplicaciones basadas en ellos.

1.3. UN POCO DE HISTORIA

Apache Spark comenzó en 2009 como un proyecto de investigación en la Universidad de California en Berkeley, dentro del laboratorio AMPLab. El primer artículo sobre este proyecto aparece en 2010, firmado por Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker e Ion Stoica, del AMPLab (<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.pdf>).

Hasta ese momento, Apache Hadoop con su motor MapReduce constituía la plataforma dominante de computación paralela y distribuida en *clusters*, siendo el primer sistema de código abierto capaz de procesar grandes volúmenes de información de forma distribuida en *clusters* formados por miles de nodos.

De su investigación y experiencias de uso de MapReduce para problemas del mundo real pudieron extraer varias lecciones y un mejor entendimiento de los beneficios y desventajas de este nuevo modelo de programación, y qué características debería tener una nueva plataforma más generalista que superara las limitaciones existentes.

En concreto, llegaron a la conclusión de que el modelo MapReduce resultaba muy útil para procesos de cálculo *acíclicos*, es decir, compuestos por una secuencia de pasos o cálculos lineal, sin ciclos o iteraciones. Pero gran parte de los algoritmos que se usan para resolver la mayor parte de problemas no siguen un flujo lineal. En general los procesos de datos siguen flujos iterativos más o menos complejos. Por ejemplo, un algoritmo de aprendizaje estadístico o *machine learning* puede necesitar hacer 10 o 20 pasadas sobre el conjunto de datos de entrada.

En MapReduce, esto supondría que cada pasada debería ser configurada como un trabajo MapReduce distinto, lanzado en la plataforma Hadoop de forma separada y volviendo a cargar los datos independientemente del resto. Construir aplicaciones de cierta complejidad con MapReduce podía resultar arduo e ineficiente. El hecho de que, además, en cada paso de cálculo los datos de entrada y los resultados deban leerse y escribirse en disco no contribuye a la eficiencia de los procesos en este tipo algoritmos.

Para solventar este problema, el equipo de Spark diseñó primero una API basada en los principios de la programación funcional, que permitía escribir aplicaciones con múltiples pasos de cálculo y flujos complejos de forma sencilla. También desarrolló un nuevo motor de cálculo distribuido que podía ejecutar estos procesos de forma muy eficiente, manejando y compartiendo datos en memoria entre los pasos de cálculo.

La primera versión de Spark solo permitía trabajar con procesos por lotes (tipo *batch*). Pero pronto surgió la necesidad de dar cabida a otros tipos de usos más interactivos, como los análisis propios de la ciencia de datos o la ejecución de consultas y filtrados sobre los datos. Se decidió dotar a la plataforma de un intérprete interactivo y también de un motor que pudiera ejecutar consultas de tipo similar a SQL, de forma que facilitara el trabajo a analistas y científicos de datos.

Tras unas pocas versiones iniciales, se comenzaron a añadir librerías sobre el núcleo del proyecto para dar cobertura a las necesidades que se iban identificando. Así, arrancaron varios grupos en AMPLab encargados de desarrollar librerías como MLlib (la librería de *machine learning* de Spark), Spark Streaming (para procesar flujos de datos en tiempo real) o GraphX (para procesar grafos de manera más eficiente).

En 2013, Spark ya se había convertido en una plataforma de uso extendido, con un centenar de personas de más de 30 organizaciones externas al AMPLab contribuyendo a su desarrollo. AMPLab cedió el proyecto Spark a la Apache Software Foundation para que lo albergara dentro de su estructura de proyectos de código abierto. El equipo original de AMPLab que desarrolló la primera versión de Spark también creó una nueva compañía, Databricks, para seguir contribuyendo a ampliar y robustecer el proyecto. La versión 1.0 de Apache Spark se publicó en 2014, y la versión 2.0 apareció en 2016.

A día de hoy Spark sigue ganando usuarios y popularidad. Existen gran cantidad de proyectos en el “*ecosistema*” de Spark para ampliar las capacidades del sistema y cubrir nuevas necesidades. Compañías como Uber o Netflix colaboran y utilizan Spark como motor para el procesamiento de sus enormes flujos de datos y sus herramientas de *machine learning*.

1.4. ENTORNO DE TRABAJO

En esta unidad veremos varios ejemplos con código sobre Spark. En lugar de que tengas que descargar, instalar y configurar el software de la plataforma Spark en tu ordenador, utilizaremos la misma máquina virtual que ya usamos en la unidad anterior sobre Hadoop. La imagen de máquina virtual ya incluye todo el software listo para usar. Simplemente tendrás que arrancarla y seguir las instrucciones que iremos viendo en cada ejemplo.



Vídeo

En el campus podrás encontrar vídeos de apoyo con la explicación paso a paso de como abrir una sesión Spark en nuestro entorno de trabajo.

Spark está desarrollado en el lenguaje Scala y corre sobre la máquina virtual de Java (JVM). Por tanto, Scala es el lenguaje que soporta de forma nativa para desarrollar aplicaciones. Sin embargo, Spark también permite ser usado y programar con los lenguajes Python, R, Java o SQL. En nuestros ejemplos utilizaremos principalmente Python.

1.4.1. LANZAR LA CONSOLA INTERACTIVA

Para probar los códigos de ejemplo que veremos en esta unidad, o para hacer tus propias pruebas y desarrollar algunos ejercicios, vamos a utilizar la consola interactiva de Spark para el lenguaje Python, llamada PySpark.

Para ello, una vez que hayas arrancado nuestra máquina virtual y entrado en sesión, necesitarás abrir el terminal de comandos (MATE).

Para algún ejemplo tendremos que leer ficheros de HDFS, así que antes de lanzar la consola de PySpark, tendremos que arrancar los servicios de Hadoop utilizando los scripts que ya vimos en la unidad anterior.

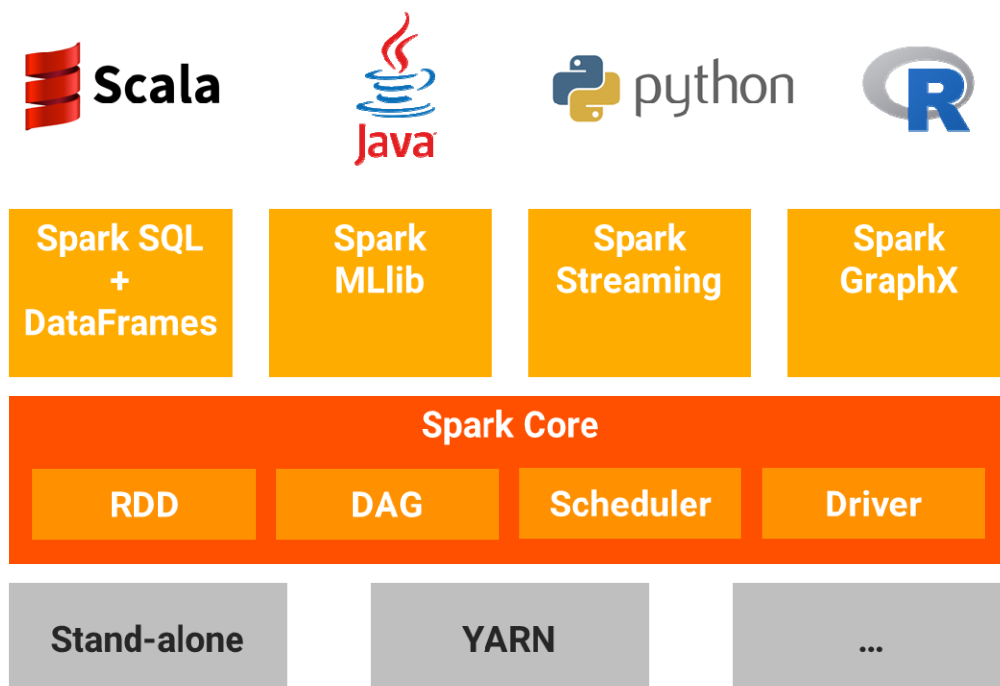
```
$ start-dfs.sh  
$ start-yarn.sh
```

Una vez arrancados, ya podemos lanzar la consola de PySpark. Simplemente introduce el siguiente comando en el terminal:

```
$ pyspark
```


2. COMPONENTES DE SPARK

El siguiente diagrama te muestra cuáles son los componentes principales de Apache Spark.



2.1. SPARK CORE

El núcleo de Spark, denominado Spark Core, incluye todos los elementos y librerías básicas para el funcionamiento de la plataforma.

Incluye la implementación de procesos centrales de la plataforma, como el planificador y el distribuidor de tareas, reponsables de programar y coordinar la ejecución de los procesos Spark con el sistema de gestión del *cluster*.

Además, proporciona dos piezas clave que hacen diferente a Spark:

- Una abstracción para manejar datos en memoria de forma distribuida: los RDDs
- Un modelo de cálculo para ejecutar flujos de procesos sobre los datos de manera flexible y eficiente: los DAGs

2.1.1. ESTRUCTURAS DE DATOS EN MEMORIA: RDD

Una de las claves en el diseño de Spark es el uso generalizado de estructuras de datos en memoria. Mientras que en el modelo MapReduce de Hadoop los procesos leen y escriben los datos de disco, el equipo de desarrollo de Spark decidió desde el inicio que debía haber un mecanismo para distribuir y mantener los datos a procesar en memoria transparente para el usuario final de la plataforma. Esta decisión suponía añadir un nivel de complejidad superior en el diseño y desarrollo del sistema respecto a Hadoop. Pero permitía superar las limitaciones de MapReduce en dos aspectos.

Para empezar, trabajar con datos directamente en memoria disparaba el rendimiento de cálculo. Distintos tests y pruebas de rendimiento muestran que el tiempo de ejecución de diferentes tareas de cálculo en Spark usando datos en memoria es hasta 100 veces inferior al tiempo requerido usando MapReduce.

La otra ventaja de utilizar datos en memoria es que permite iterar y repetir flujos de cálculo sobre los mismos conjuntos de datos de manera muy eficiente y a la vez sencilla de implementar. Mientras que con MapReduce cada réplica del proceso suponía originalmente definir y lanzar una nueva tarea, con el engorro que ello implica añadido a la ineficiencia de volver a cargar los mismos datos una y otra vez, Spark nos permite volver sobre los mismos datos ya cargados en memoria para realizar nuevos cálculos (un poco más adelante veremos cómo funciona el modelo de cálculo de Spark).

Naturalmente, mantener gran cantidad de datos distribuidos en un *cluster* (con hasta cientos o miles de nodos) y con cierto grado de acceso compartido en memoria no es un problema simple.

La base que permite dar soporte a estas funcionalidades son una estructura de datos llamada RDD (*Resilient Distributed Dataset*). Sobre este mecanismo se han ido desarrollando abstracciones de datos más complejas y con funcionalidades de más alto nivel, como los Datasets, los Dataframes, o las tablas tipo SQL, que facilitan el trabajo con datos de los usuarios finales de la plataforma.

Los *Resilient Distributed Datasets* (traducido libremente como “conjunto de datos distribuidos tolerante a fallos”) o RDDs son la estructura de datos principal de Apache Spark y componen la piedra angular de la plataforma.

Un RDD es una abstracción que representa una colección de datos particionados y repartidos por múltiples nodos de un *cluster*, con recuperación frente a fallos gracias a su capacidad de recuperar información de otros nodos o de recalcular los datos perdidos. Las propiedades básicas son:

- **Particionamiento:** la colección de datos (formada por valores simples, secuencias o tipos más complejos) se divide en varias particiones o bloques.
- **Distribuido:** las particiones son repartidas entre los distintos nodos del cluster.
- **Robustez o tolerancia a fallos:** ante problemas para acceder a los datos de una o más particiones por fallos en nodos o en la red, Spark es capaz de regenerar los datos perdidos o corrompidos de un RDD, recalculando los pasos necesarios a partir de datos previos (esta propiedad está estrechamente ligada al modelo de cálculo de Spark que veremos más adelante).

Además de estas características, los RDDs poseen otros atributos adicionales:

- **Datos en memoria:** los RDDs están diseñados para que mantengan la mayor cantidad de datos en memoria durante el mayor tiempo posible. Obviamente, la cantidad de memoria RAM es limitada, y en algún momento otro proceso de cálculo puede requerir el acceso a memoria. El sistema de gestión de los RDDs se encarga de transferir datos entre memoria y disco automáticamente cuando sea necesario, optimizando la utilización de los recursos sin necesidad de intervención del usuario.

- **Inmutabilidad:** los RDDs son estructuras de datos inmutables. Esto significa que su contenido no puede ser modificado una vez que se han creado. Es decir, son datos de solo lectura. ¿Significa eso que no se puede hacer nada con los datos? ¡No, claro que no! Lo que ocurre simplemente es que cada vez que se realiza una operación que modifica los datos, se genera automáticamente un nuevo RDD resultado (también inmutable a su vez).
- **Evaluación retrasada o *perezosa*:** los datos contenidos en un RDD no se "*materializan*" o se hacen disponibles hasta que una operación posterior intenta acceder efectivamente a los valores o resultados. Un ejemplo: si realizamos una operación sobre un RDD (sumar 1 a todos sus valores), obtendremos un nuevo RDD resultado. Pero en realidad, este RDD no contendrá inicialmente los valores actualizados. En su lugar, almacena una referencia al RDD de partida y a la operación o *transformación* a aplicar. Hasta que no pidamos acceder a uno o más valores del resultado, la operación y los valores finales no se materializarán. ¿Y por qué hacer esto? Bueno, esta forma de operar permite no consumir memoria ni CPU hasta que los valores *realmente* hacen falta.
- **Datos en *cache*:** es posible forzar a que la plataforma mantenga *todos* los datos en memoria (la opción por defecto, mientras no haya escasez de recursos) u obligar a darles persistencia en disco.

Todas estas características son completamente transparentes para el usuario. Es decir, el programador no tiene que hacer nada especial para activar o aprovechar estas propiedades de forma explícita: no tiene que dividir los datos o forzar como repartirlos entre los nodos. Puede manejar los datos en un RDD como si fuera cualquier otro tipo de colección de datos en un lenguaje de programación de alto nivel.

2.1.2. FLUJOS DE DATOS Y CÁLCULO: DAGs


Como has visto, las operaciones sobre RDDs no modifican los datos directamente. En su lugar, se genera lo que podemos llamar un RDD *hijo* que representa el resultado de la operación aplicada sobre el RDD *padre*. Los RDDs son inmutables.

De hecho, la operación tampoco se lleva a cabo realmente hasta que no es necesario acceder a los resultados finales.

De esta forma, si encadenamos varias operaciones, obtenemos una secuencia de pasos de cálculo y RDDs enlazados, representando los resultados de cada paso.

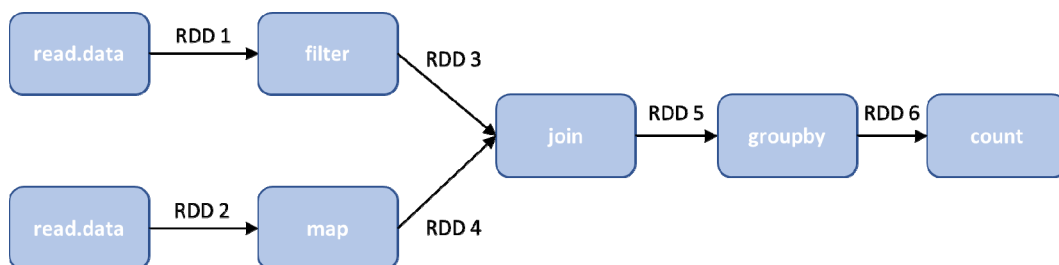
Podemos ver esta cadena de procesos y datos como un grafo dirigido, donde los vértices representan los diferentes procesos sobre los datos y los enlaces indican la secuencia de ejecución.

De hecho, Spark trata un programa como un tipo especial de grafo, un grafo dirigido sin ciclos, o DAG (del inglés *directed acyclic graph*).



+ Info

Un grafo dirigido es aquel en el que solo podemos viajar de un nodo a otro en una dirección. Un grafo acíclico es aquel en el que partiendo de un nodo no podemos volver al mismo nodo, no existe ningún camino que empiece y acabe en el mismo punto.



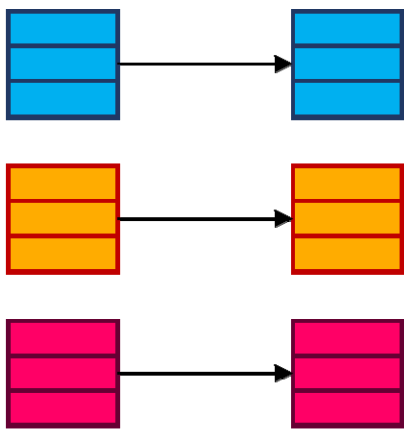
¿Y cuáles son las ventajas de representar este flujo de cálculos y datos en forma de grafo?

Para empezar, permite representar flujos o secuencias de procesos más complejas y flexibles que, por ejemplo, el esquema MapReduce.

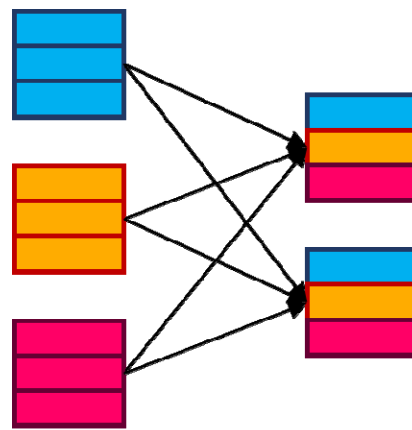
Además, si se produce cualquier error durante el proceso, Spark puede reconstruir los RDDs a partir del grafo y los RDDs previos, repitiendo los pasos necesarios. Esto proporciona robustez y tolerancia a fallos.

La otra gran ventaja tiene que ver con el rendimiento. Spark es capaz de analizar este grafo de procesos y optimizar la forma de repartir y ejecutar los cálculos en el *cluster*. Recuerda que estamos en un sistema distribuido, y que los datos están repartidos entre varios nodos para procesarlos en paralelo. Si imaginas un RDD como una tabla, habrá porciones distintas del RDD repartidas entre los nodos del cluster. Spark trata de realizar el máximo número de operaciones en memoria sobre la porción de datos que tiene cada nodo, minimizando el intercambio de datos entre nodos (que es muchísimo más costoso).

Para ello, Spark distingue clasifica las operaciones de transformación de los datos en dos tipos: transformaciones *estrechas* y transformaciones *amplias*.



Transformación Estrecha



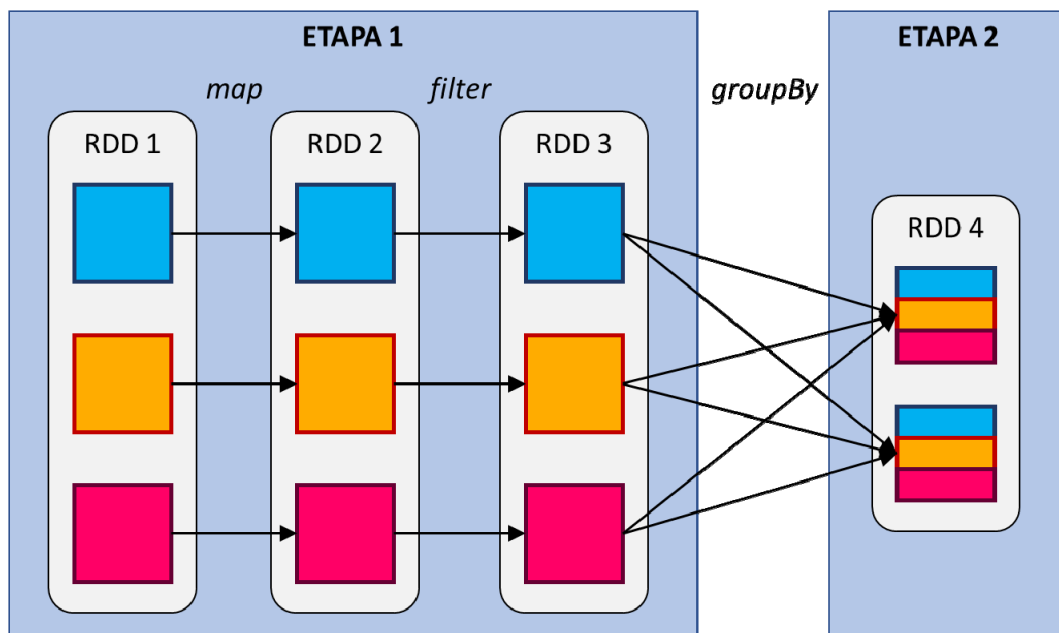
Transformación Amplia

Una transformación estrecha es una operación en la que, para calcular la porción del resultado en cada nodo, solamente se necesita la porción de datos de entrada de ese nodo. Ejemplos de estas transformaciones serían las operaciones tipo mapeo (*map*), filtrado (*filter*) o muestreo (*sample*).

Una transformación amplia es una operación en la que los datos de entrada necesarios para calcular la porción de resultados de un nodo pueden provenir de cualquier otro nodo del sistema. Estas transformaciones incluyen operaciones como la agregación por una clave (*groupByKey*), el cruce (*join*) o la intersección.

Las transformaciones estrechas pueden realizarse en memoria en cada nodo. Las transformaciones amplias necesitan intercambiar datos por red o salvarlos en disco temporalmente para completar los cálculos.

Spark utiliza esta información para dividir los procesos en bloques o etapas que contienen tantas operaciones en memoria (estrechas) como sea posible, para optimizar el rendimiento. Las transformaciones amplias marcan los límites entre etapas completamente paralelizables en memoria.



Recapitulando, las transformaciones crean nuevos RDDs a partir de RDDs previos. Spark organiza la secuencia de transformaciones en un grafo tipo DAG para controlar y optimizar los cálculos, según el tipo de operación.

La estructura de grafos ofrece una ventaja más. Y es que no nos hace falta ejecutar la secuencia de operaciones inmediatamente. Podemos retrasar la ejecución de estos cálculos hasta el momento en que sea realmente necesario acceder a los valores finales. Es lo que llamamos *evaluación perezosa* y permite a Spark posponer la ejecución. El grafo DAG y los RDDs proporcionan toda la información para completar los cálculos cuando llegue el momento.

¿Pero cuándo es realmente necesario ejecutar los cálculos? La respuesta es cuando lleguemos a una operación que no genere un nuevo RDD como resultado. A este tipo de operaciones las llamamos acciones. Ejemplos de acción serían escribir los resultados finales en disco o base de datos, o contar el número de elementos del resultado, o mostrar los N primeros resultados.

Para ejecutar estas acciones *es necesario* tener los resultados finales ya calculados y disponibles. Una operación de este tipo *obliga* a Spark a ejecutar todas las operaciones previas del grafo para completar la acción.

Fíjate que la consecuencia de ejecutar una acción es que *todos* los RDDs del grafo se *materializan* con sus correspondientes valores ya calculados.

2.2. PLATAFORMA DE EJECUCIÓN

Spark está diseñado para poder escalar su funcionamiento desde una sola máquina a *clusters* formados por miles de nodos, de forma sencilla y eficiente. Para conseguirlo sin sacrificar la flexibilidad, Spark puede correr apoyándose en una gran variedad de plataformas de ejecución o gestores de *clusters*, como YARN (que ya vimos en la unidad anterior) o Apache Mesos (otro gestor de recursos de *cluster*).

Spark incluye también un gestor de plataforma de ejecución llamado *Standalone Scheduler* (planificador independiente), pensado para poder ejecutar Spark en una sola máquina de forma directa, sin las complicaciones de tener que configurar un gestor de plataforma adicional.

2.3. COMPONENTES AVANZADOS

2.3.1. SPARK SQL, DATAFRAMES Y DATASETS

Como ya hemos comentado, la estructura de datos básica en Spark son los RDDs, y constituyen uno de los pilares de la plataforma.

Sin embargo, las evoluciones en cada nueva versión de Spark han traído nuevas formas de manejar los datos, más sencillas y potentes. Recuerda que uno de los objetivos de Spark es simplificar para el programador las tareas de manejo de datos.

Una de las estructuras de datos que se incorporaron fueron los DataFrames. Ya conoces este tipo de estructura, la viste en R y también en Python, a través de la librería Pandas.

Un DataFrame es una tabla de datos, organizados en filas y columnas, donde las columnas tienen un nombre y todos los valores de una columna son del mismo tipo. Es algo análogo a una tabla en una base de datos; también es parecido a una hoja de cálculo.

La diferencia fundamental de los DataFrame en Spark respecto a los DataFrame en R o Python, o respecto a las tablas de bases de datos o las hojas de cálculo, es que en Spark el contenido de un DataFrame puede estar repartido entre varias máquinas o nodos del *cluster*. En realidad, un DataFrame almacena internamente los datos usando un RDD, así que todas las propiedades de los RDD están presentes en los DataFrame.

Los DataFrame en Spark no son simplemente una estructura de datos. También llevan consigo una API, todo un conjunto de operaciones y funcionalidades que hacen más intuitivo trabajar con los datos. Ésta es la principal ventaja respecto a usar RDDs. Veamos un ejemplo. Tenemos un fichero CSV con datos de películas, incluyendo presupuesto y taquilla (estos mismos datos los utilizamos en la unidad 10):

```
"mov-
ie_title";"title_year";"country";"language";"duration";"director_name";"budget";
"gross"
"Avatar";2009;"USA";"English";178;"James Cameron";2.37e+08;760505847
"Pirates of the Caribbean: At World's End";2007;"USA";"English";169;"Gore
Verbinski";3e+08;309404152
"Spectre";2015;"UK";"English";148;"Sam Mendes";2.45e+08;200074175
"The Dark Knight Rises";2012;"USA";"English";164;"Christopher No-
lan";2.5e+08;448130642
"John Carter";2012;"USA";"English";132;"Andrew Stanton";263700000;73058679
"Spider-Man 3";2007;"USA";"English";156;"Sam Raimi";2.58e+08;336530303
"Tangled";2010;"USA";"English";100;"Nathan Greno";2.6e+08;200807262
"Avengers: Age of Ultron";2015;"USA";"English";141;"Joss
Whedon";2.5e+08;458991599
"Harry Potter and the Half-Blood Prince";2009;"UK";"English";153;"David
Yates";2.5e+08;301956980
```

Imagina que queremos ver cuáles son los países cuyas películas han ingresado más por taquilla. Dejamos por ahora la parte de cómo cargar los datos, y suponemos que ya están en una variable. Si lo quisiéramos hacer directamente con una variable tipo RDD, el código sería algo así

```
1 movies_rdd.\
2     map(lambda r: (r['country'], float(r['gross']))).\
3     groupByKey().\
4     mapValues(sum).\
5     sortBy(lambda x: -x[1]).\
6     take(10)
```

```
[('USA', 178608312584.0),
 ('UK', 11808199558.0),
 ('Germany', 2462603234.0),
 ('France', 1978421823.0),
 ('Australia', 1688648202.0),
 ('Canada', 1615108801.0),
 ('New Zealand', 1298160522.0),
 ('Japan', 570412925.0),
 ('China', 268717774.0),
 ('Spain', 229749472.0)]
```

Si trabajamos con una variable tipo DataFrame, el código sería más o menos así:

```
1 movies_df.\
2     select(col('country'), col('gross')).\
3     groupBy('country').\
4     agg(
5         F.sum('gross').alias('sum_gross')
6     ).\
7     orderBy(F.desc('sum_gross')).\
8     show(10)
```

country	sum_gross
USA	178608312584
UK	11808199558
Germany	2462603234
France	1978421823
Australia	1688648202
Canada	1615108801
New Zealand	1298160522
Japan	570412925
China	268717774
Spain	229749472

Fíjate en los dos fragmentos de código, independientemente de la salida devuelta. A primera vista, ¿qué versión te parece más legible, más sencillo intuir lo que hace en cada paso?

Para completar el cálculo en la primera versión con RDDs, primero tenemos que generar una tupla (clave, valor) para cada registro de los datos de entrada. En este caso, generamos tuplas usando `country` como clave y `gross` (la recaudación) como valor (línea 2). En el siguiente paso agrupamos valores por su clave, el país en nuestro caso (línea 3). A continuación, aplicamos una función a todos los valores para una misma clave o grupo. En este caso, sumar todos los valores (recaudación de las películas) de cada grupo (país). Esto lo hacemos con `mapValues` (línea 4), que nos devuelve tuplas (clave, suma de valores). Tras esto, ordenamos los resultados agregados (línea 5). Los ordenamos por el segundo elemento de las tuplas (`x[1]`), que en este caso es la suma de las recaudaciones de un país. Como queremos ordenarlo de mayor a menor, le ponemos un signo negativo (por defecto ordena de menor a mayor). Finalmente, nos quedamos con los 10 primeros registros (línea 6).

A estas alturas, ya estás familiarizado con las funciones tipo `map`, `apply`, `groupby`, `reduce`... Estas operaciones típicas de programación funcional las hemos visto con distintas versiones en R, en Python, y también las vimos en Hadoop MapReduce.

Sin embargo, no es evidente a primera vista lo que hacen. Si no tienes experiencia, puede resultar complicado de descifrar.

El segundo fragmento usando un `DataFrame`, aunque haya elementos que no sepas qué significan, al menos parece más fácil de leer, y se puede intuir lo que ocurre en cada paso.

En la línea 2 seleccionamos las columnas `country` y `gross` (la recaudación). En la línea 3 decimos que vamos a agrupar por el valor de `country`. En la línea 4, queremos realizar agregaciones sobre los valores de cada grupo. En este caso, la única agregación que calculamos (línea 5) es la suma del campo `gross`. Al resultado de la suma le ponemos de nombre (alias) `sum_gross`. En la línea 6 ordenamos los resultados de forma descendente por el valor de `sum_gross`. Finalmente, mostramos los 10 primeros resultados. ¿No te parece más claro así?

Como decíamos antes, una de las principales ventajas de usar DataFrames es que incluyen operaciones de *más alto nivel*. Al trabajar con DataFrames indicamos qué operaciones de alto nivel queremos hacer sobre los datos. Con los RDDs tenemos que pensar y programar también cómo hacer esos cálculos.

Pero aún hay más. Hemos dicho que un DataFrame puede verse como algo análogo a una tabla en una base de datos. Para los desarrolladores y usuarios de Spark este concepto era tan claro y válido que construyeron toda una librería para trabajar con los DataFrames usando consultas, como harías en una base de datos tradicional. Esta librería es Spark SQL, y es uno de los componentes de serie de Spark.

Spark SQL te permite registrar cualquier DataFrame como una tabla o vista y ejecutar consultas y cálculos sobre ellas usando el lenguaje SQL. Además, no hay ninguna diferencia en velocidad o tiempo de cálculo entre usar consultas SQL o trabajar directamente con el DataFrame. Spark SQL se encarga de traducir las consultas SQL a la misma secuencia de operaciones que se obtienen con un DataFrame.

Veamos cómo sería el código para realizar el mismo cálculo que hemos visto con un RDD y con un DataFrame, usando ahora Spark SQL:

```
1 movies_df.createOrReplaceTempView("movies_view")
2 spark.sql(
3     "SELECT country, sum(gross) \
4     FROM movies_view \
5     GROUP BY country \
6     ORDER BY sum(gross) DESC \
7     LIMIT 10"
8 ).show()
```

country	sum_gross
USA	178608312584
UK	11808199558
Germany	2462603234
France	1978421823
Australia	1688648202
Canada	1615108801
New Zealand	1298160522
Japan	570412925
China	268717774
Spain	229749472

¿Puede ser más sencillo? Solamente hemos tenido que definir una vista SQL (`movies_view`) sobre los datos de `DataFrame` (línea 1), y después ya podemos ejecutar una consulta SQL normal, como en cualquier base de datos.

Por último, Spark ofrece otra forma más de trabajar con datos estructurados: los `Datasets`. Comparados con un `DataFrame`, podemos ver los `Datasets` como una colección de registros (filas en el `DataFrame`), donde cada registro es en realidad un objeto de una clase definida por el usuario. Trabajar con clases definidas por el programador permite incluir funcionalidades específicas cuando usamos los datos como objetos, pero manteniendo la posibilidad de usarlos como un `DataFrame` cuando lo necesitemos.

No obstante, los `Datasets` sólo están disponibles en la API de Spark para los lenguajes Scala y Java, pero no para Python ni R, ya que estos últimos son lenguajes con tipado dinámico, y los `Datasets` necesitan tener un control de tipos y clases estricto (es decir, que el tipo de una variable deba declararse antes de usarla, y que no pueda modificarse durante su utilización).

En el resto de los ejemplos que veremos durante la unidad usaremos fundamentalmente las API de `DataFrame` y SQL.

2.3.2. SPARK MLlib

Otro de los puntos fuertes de Spark reside en las facilidades que ofrece para ejecutar tareas de *machine learning* sobre datos a gran escala gracias a su librería de algoritmos `MLlib`. Esta librería incluye todos los componentes necesarios para el preprocesado y preparación de los datos, definición de modelos, entrenamiento, validación y ejecución de los modelos, a escala Big Data.

La librería `MLlib` ofrece implementaciones de todo tipo de algoritmos de aprendizaje estadístico, desde algoritmos de regresión o clasificación, a algoritmos de *clustering*, recomendación o *deep learning*.

2.3.3. SPARK STREAMING

Otro de los componentes que aportan funcionalidades avanzadas a la plataforma es Spark Streaming. Este módulo añade un sistema de procesamiento de flujos de entrada de datos (*streams*) al vuelo, de forma continua. Es capaz de recibir datos de múltiples fuentes en tiempo real, ofreciendo tolerancia a fallos y baja latencia para procesar nuevos datos conforme llegan, respondiendo de forma eficiente incluso ante grandes avenidas de datos. Algunas aplicaciones típicas incluyen el análisis de *logs* para monitorización de servidores, análisis en tiempo real de *feeds* o de mensajes en redes sociales (p.ej. *Twitter*) o procesar señales o telemidas enviadas por sensores de distintos dispositivos conectados a Internet (el llamado Internet de las Cosas o IoT).

Spark Streaming permite trabajar con estos flujos de datos en tiempo real igual que con cualquier otra estructura estática, como un *DataFrame*. Es posible realizar el mismo tipo de operaciones sobre estos flujos o canales de datos, incluso aplicar los algoritmos de Machine Learning. En realidad, los datos se procesan internamente de forma incremental, en pequeños bloques que facilitan secuenciar los cálculos, reduciendo la latencia.

2.3.4. SPARK GRAPHX

GraphX es otro componente de alto nivel incluido en Spark, orientado al cálculo con grafos de gran tamaño. El análisis y cálculo con grafos tiene una importancia crucial hoy en día, ya que es la base y motor para el funcionamiento de motores de búsqueda y de las grandes aplicaciones de redes sociales. Una red social es simplemente un grafo, donde los nodos serían usuarios, contenidos, aplicaciones, etc. y los enlaces serían las interacciones entre ellos (un “me gusta”, una compra, el envío de un mensaje...). Todos los motores de redes sociales utilizan algoritmos de grafos para calcular cosas como los usuarios más influyentes, el contenido más relevante, recomendaciones, etc.

Spark GraphX ofrece una completa librería de operaciones y algoritmos avanzados con grafos mediante una serie de funciones de alto nivel, fáciles de utilizar. Internamente, GraphX aprovecha todo el potencial del cálculo distribuido de Spark y sus RDDs, ocultando al usuario los detalles de programación más complejos.

3. ARQUITECTURA

3.1. ANATOMÍA DE UNA APLICACIÓN EN SPARK

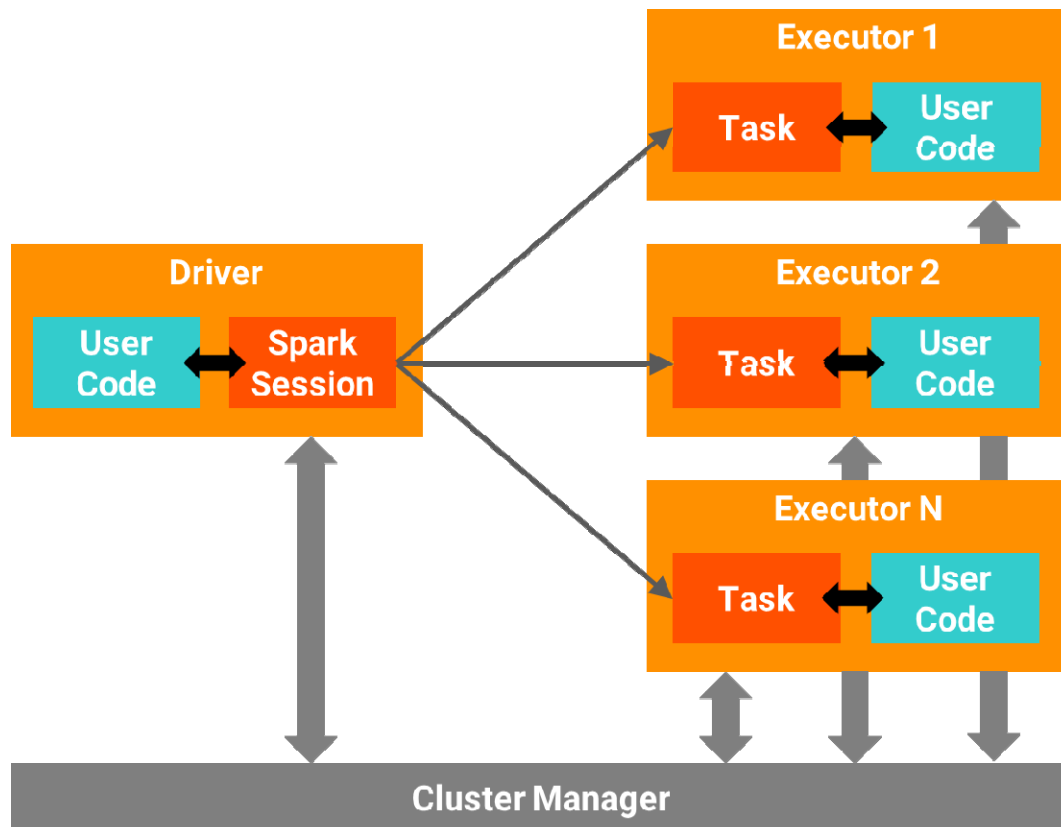
Una aplicación en Spark consiste en un proceso principal controlador (o *driver*) y una serie de procesos trabajadores o ejecutores.

El proceso controlador o *driver* es el encargado de arrancar la ejecución de la función principal (main) de nuestra aplicación en el *cluster* de Spark. El *driver* es responsable de mantener la aplicación durante su ciclo de vida. Eso conlleva varias tareas. En primer lugar, registrar toda la información relativa a la aplicación (nombre, código, librerías utilizadas, argumentos y parámetros de configuración y ejecución, estado de los procesos...). También gestiona la interacción y repuesta de la aplicación ante el usuario, o con otros programas y aplicaciones. Por último, es responsable de planificar y repartir las tareas que necesita realizar la aplicación entre los nodos ejecutores disponibles en el *cluster* en cada momento.

Los procesos ejecutores son los encargados de ejecutar realmente las diferentes tareas o cálculos de la aplicación, conforme les son asignados por el *driver*. Además, cada ejecutor informa al *driver* de su estado regularmente, indicando cuándo ha terminado un bloque de trabajo para poder recibir nuevas tareas.

Pensemos ahora en el programador que está desarrollando una aplicación en Spark. ¿Cómo accede nuestra aplicación a los objetos y servicios de la plataforma? La respuesta es a través de un objeto llamado **SparkSession**.

La **SparkSession** es el punto de acceso de nuestra aplicación a todas las funcionalidades de la plataforma Spark. Toda aplicación necesita establecer una sesión para que pueda ejecutarse en la plataforma.



Cuando estamos trabajando en una sesión interactiva (por ejemplo con PySpark), se creará automáticamente una sesión para trabajar (al fin y al cabo, en este caso la aplicación interactiva es *nuestro controlador* para ejecutar comandos sobre Spark).

Cuando desarrollemos una aplicación para ser sometida y ejecutada directamente sobre el *cluster* de Spark, tendremos que incluir al inicio el código para instanciar una nueva sesión. Más adelante te mostraremos cómo.

```
$ pyspark
```

[illegible]

```
Using Python version 3.6.0 (default, Dec 23 2016 12:22:10)
SparkSession available as 'spark'.
>>>
```

Una de las cosas que ocurren cuando arrancamos PySpark es que este crea una **SparkSession** de forma automática por nosotros. Este objeto está disponible a través de la variable `spark`. Recuerda que la **SparkSession** es nuestro punto de acceso a todas las funcionalidades de la plataforma Spark. En nuestros ejemplos con la consola interactiva utilizaremos varias veces la variable `spark`.

4.1.1. EJECUTAR EL CÓDIGO DE LOS EJEMPLOS

A lo largo de los próximos apartados te mostraremos varios ejemplos de código. Para probar su funcionamiento, puedes teclearlos directamente en la consola de PySpark.

No obstante, para facilitarte un poco el trabajo (algunos ejemplos son complejos y es fácil equivocarse al transcribirlos), puedes seleccionar y copiar el código directamente de este documento y después pegarlo en la consola de PySpark. Recuerda que para pegar algo copiado fuera de la máquina virtual en la consola tienes que pulsar `Ctrl+Mayus+V`.

Otra opción es usar el material que ya está incluido en la propia máquina virtual. Dentro de la "Carpeta personal bigdata" que verás en el escritorio de la máquina virtual, encontrarás una subcarpeta con los contenidos de esta unidad. Navega por las carpetas `bigdata` → `Curso` → `U13_Spark` → `src`

En esta última carpeta `src` encontrarás varios ficheros `.py` con los fragmentos de código que verás en los siguientes ejemplos.

Si haces doble click sobre un fichero, se abrirá en un editor de texto básico llamado Pluma. Puedes seleccionar y copiar (`Ctrl+C`) el fragmento de código que quieras y después ir a la consola de PySpark y pegarlo (`Ctrl+V`).

4.1.2. CARGAR DATOS DE FICHERO

Vamos a empezar cargando un juego de datos sobre los que trabajar en nuestros ejemplos.

Utilizaremos una colección de datos sobre propiedades y calidad de vinos. Se trata de un conjunto de datos que incluye medidas de varias propiedades químicas y organolépticas de distintos vinos tintos y blancos, junto con una valoración de 0 (pésimo) a 10 (excelente) sobre su calidad, calculada en base a valoraciones de enólogos. Estos datos están disponibles de forma pública para investigación a través del repositorio Irvine Machine Learning de la Universidad de California, creado por Paulo Cortez, Antonio Cerdeira, Fernando Almeida, Telmo Matos y Jose Reis (<https://archive.ics.uci.edu/ml/datasets/wine+quality>).

Estos datos se encuentran ya preparados en la máquina virtual, en el fichero **winequality.csv** dentro del directorio **/home/bigdata/Curso/U13_Spark/datos**

Puedes ver el contenido antes ejecutando en un terminal MATE:

```
$ head /home/bigdata/Curso/U13_Spark/datos/winequality.csv
fixed acidity;volatile acidity;citric acid;residual sugar;chlorides;free sulfur
dioxide;total sulfur dioxide;density;pH;sulphates;alcohol;quality;type
7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5;red
7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5;red
7.8;0.76;0.04;2.3;0.092;15;54;0.997;3.26;0.65;9.8;5;red
11.2;0.28;0.56;1.9;0.075;17;60;0.998;3.16;0.58;9.8;6;red
7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5;red
7.4;0.66;0;1.8;0.075;13;40;0.9978;3.51;0.56;9.4;5;red
7.9;0.6;0.06;1.6;0.069;15;59;0.9964;3.3;0.46;9.4;5;red
7.3;0.65;0;1.2;0.065;15;21;0.9946;3.39;0.47;10;7;red
7.8;0.58;0.02;2;0.073;9;18;0.9968;3.36;0.57;9.5;7;red
```

Vamos a cargar los datos en Spark, leyendo directamente el fichero del directorio en nuestra máquina virtual

```
1 local_file =
2 r'file:///home/bigdata/Curso/U13_Spark/datos/winequality.csv'
3
4 wineq = spark.read.csv(path=local_file, header=True, sep=';',
5 inferSchema=True)
6
```

Spark incluye varias funciones para leer datos de distintas fuentes y formatos. Aquí hemos usado la función para leer datos de ficheros tipo CSV.

A través de la variable de sesión **spark** accedemos al objeto **read**, que proporciona la interfaz de lectura de datos, con las distintas funciones disponibles. **spark.read** es en realidad un objeto de tipo **DataFrameReader**.

Comprobemos que tenemos los datos cargados

1	wineq.show(5)									
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+										
fixed acidity volatile acidity citric acid ... density										
pH sulphates alcohol quality type										
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+										
7.4 0.7 0.0 0.9978 3.51 0.56 9.4 5 red										
7.8 0.88 0.0 0.9968 3.2 0.68 9.8 5 red										
7.8 0.76 0.04 ... 0.997 3.26 0.65 9.8 5 red										
11.2 0.28 0.56 0.998 3.16 0.58 9.8 6 red										
7.4 0.7 0.0 0.9978 3.51 0.56 9.4 5 red										
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+										
only showing top 5 rows										

En la consola verás más columnas de las que mostramos aquí, hemos omitido algunas por espacio.

El método `spark.reader.csv()` permite especificar varias opciones. Algunas de las principales son:

- **path:** lista de rutas completas de ficheros a leer.
- **sep:** carácter separador de columnas (',' por defecto).
- **quote:** carácter usado para valores entrecomillados ("" por defecto).
- **header:** utilizar los valores de la primera línea como nombres de las columnas.
- **inferSchema:** dejar que Spark infiera el tipo de cada columna automáticamente.
- **nullValue:** cadena de texto usada en el CSV para representar valores nulos (por defecto la cadena vacía "").



Atención

Fíjate de nuevo en la variable `local_file` del ejemplo, donde definimos la ruta hasta el fichero con los datos. Para indicar que la ubicación corresponde a un fichero en un directorio local de nuestra máquina usamos el prefijo `'file://'`. Después ponemos la ruta completa hasta el fichero. En este caso, `/home/bigdata/Curso/U13_Spark/datos/winequality.csv`

Como te decíamos, tenemos varias funciones para leer datos de distintas fuentes y formatos. La tabla siguiente muestra algunos de los métodos disponibles:

- **spark.read.jdbc():** permite utilizar una conexión JDBC para conectarse y leer de una tabla en una base de datos.
- **spark.read.json():** permite cargar datos de uno o varios ficheros en formato JSON.
- **spark.read.parquet():** permite cargar datos de ficheros en formato Parquet (un formato basado en almacenamiento de datos comprimidos por columnas, muy utilizado en soluciones Big Data).
- **spark.read.text():** permite leer líneas de un fichero de texto; cada línea se convierte en una fila del resultado devuelto..

Por cierto, hemos usado el método **show()** sobre el objeto con los datos cargados.

```
1 wineq.show(5)
```

Este método imprime con un formato adecuado el contenido cargado. Podemos indicar el número de filas que queremos mostrar. Si no lo especificamos, mostrará las 20 primeras por defecto.

¿Pero qué clase de objeto es **wineq**?

```
1 type(wineq)
```

```
<class 'pyspark.sql.dataframe.DataFrame'>
```

Efectivamente, es un **DataFrame**. Todos los métodos para leer datos que hemos enumerado devuelven directamente un **DataFrame** (al fin y al cabo, son métodos de una clase que se llama **DataFrameReader**, ¿qué iban a devolver si no?)

Leer de HDFS

¿Y si tenemos nuestros datos en HDFS, en el *cluster*?

Afortunadamente esto no nos va a suponer mucha complicación.

Si arrancaste los servicios de Hadoop HDFS y YARN, puedes comprobar que tenemos una copia de nuestro fichero CSV en el sistema HDFS

```
$ hdfs dfs -ls /user/bigdata/U13/datos
```

```
Found 1 items
-rw-r--r--  1 bigdata bigdata    390514 2018-06-12 11:43
/user/bigdata/U13/datos/winequality.csv
```

Solamente tenemos que cambiar la ruta del fichero. Con el prefijo `'hdfs://'` le indicamos a Spark el *protocolo de conexión* a usar. En este caso, en lugar del sistema de ficheros local, debe conectarse al sistema de ficheros HDFS

```
1 hdfs_file = r'hdfs:///user/bigdata/U13/datos/winequality.csv'
2
3 wineqhd = spark.read.csv(path=hdfs_file, header=True, sep=';',
4                           inferSchema=True)
5 wineq.show(5)
```

fixed acidity	volatile acidity	citric acid	...	density	pH	sulphates	alcohol	quality	type
7.4	0.7	0.0		0.9978	3.51	0.56	9.4	5	red
7.8	0.88	0.0		0.9968	3.2	0.68	9.8	5	red
7.8	0.76	0.04	...	0.997	3.26	0.65	9.8	5	red
11.2	0.28	0.56		0.998	3.16	0.58	9.8	6	red
7.4	0.7	0.0		0.9978	3.51	0.56	9.4	5	red

only showing top 5 rows

4.1.3. ACCIONES E INFORMACIÓN BÁSICA SOBRE LOS DATOS

Veamos ahora algunos comandos básicos para obtener información de nuestros datos.

Para empezar, probemos a introducir simplemente el nombre de nuestra variable DataFrame

```
1 wineq
DataFrame[fixed acidity: double, volatile acidity: double, citric acid: double,
residual sugar: double, chlorides: double, free sulfur dioxide: double, total
sulfur dioxide: double, density: double, pH: double, sulphates: double, alcohol:
double, quality: int, type: string]
```

Nos muestra la lista de columnas que componen el DataFrame, junto con su tipo (`double`, `int`, `string`...)

Pero podemos ver mejor el esquema o estructura de columnas y tipos usando el método `printSchema()`

```
1 wineq.printSchema()
root
|-- fixed acidity: double (nullable = true)
|-- volatile acidity: double (nullable = true)
|-- citric acid: double (nullable = true)
|-- residual sugar: double (nullable = true)
|-- chlorides: double (nullable = true)
|-- free sulfur dioxide: double (nullable = true)
|-- total sulfur dioxide: double (nullable = true)
|-- density: double (nullable = true)
|-- pH: double (nullable = true)
|-- sulphates: double (nullable = true)
|-- alcohol: double (nullable = true)
|-- quality: integer (nullable = true)
|-- type: string (nullable = true)
```

Ahora podemos ver mejor la jerarquía de columnas que componen el DataFrame, con el nombre y tipo de dato de cada una de ellas.

¿Y cuántos registros o filas tiene nuestro DataFrame? Podemos verlo con `count()`

```
1 wineq.count()
6497
```

También podemos pedir un resumen con las principales estadísticas de cada columna usando el método `describe()`

```
1 wineq.describe().show()
+-----+-----+-----+-----+-----+-----+
|summary|fixed acidity|volatile acidity|...|quality|type|
+-----+-----+-----+-----+-----+-----+
|count|6497|6497| |6497|6497|
|mean|7.215307064799134|0.33966599969217015| |5.818377712790519| null|
|stddev|1.296433757799806|0.16463647408467907| ... |0.8732552715311255| null|
|min|3.8|0.08| |3|red|
|max|15.9|1.58| |9|white|
+-----+-----+-----+-----+-----+-----+
```

Si solo nos interesa el resumen de alguna columna, podemos pasar como argumento a **describe()** una lista con los nombres de las columnas.

Fíjate que hemos puesto un **show()** tras el **describe()**.

```
1 wineq.describe(['alcohol','quality'])
DataFrame[summary: string, alcohol: string, quality: string]
```

describe() devuelve un nuevo objeto DataFrame con las estadísticas resumen de las columnas indicadas. Si queremos imprimir el contenido, tenemos que usar **show()**, como con cualquier otro DataFrame.

Vamos a seleccionar los 10 vinos con mejor puntuación de la muestra y veamos cuánto alcohol tienen y si son blancos o tintos.

```
1 import pyspark.sql.functions as F
2
3 wineq.\
4 select('type','alcohol','quality').\
5 sort(F.desc('quality')).\
6 limit(10).\
7 show()
```

```
+-----+-----+-----+
| type|alcohol|quality|
+-----+-----+-----+
|white|  12.5|      9|
|white|  10.4|      9|
|white|  12.7|      9|
|white|  12.4|      9|
|white|  12.9|      9|
|  red|  12.8|      8|
|  red|  12.6|      8|
|  red|  12.7|      8|
|  red|  12.9|      8|
|  red|   9.8|      8|
+-----+-----+-----+
```

En este código empezamos importando el módulo **pyspark.sql.functions** con el alias **F**. Más adelante te explicaremos qué ofrece este módulo.

En la línea 4 utilizamos el método `select()` para seleccionar las columnas que nos interesa usar (el resto no nos hacen falta ahora). En la línea 5 usamos el método `sort()` para ordenar los resultados del DataFrame. Por defecto, `sort()` ordena los valores de las columnas indicadas en orden ascendente. Pero aquí nosotros usamos la función `F.desc()` (del módulo que hemos importado) para indicar que queremos ordenar de forma descendente, de mayor a menor. Por último, el método `limit()` nos permite quedarnos solo con los 10 primeros resultados.

* ¡Ojo!! No es lo mismo `limit()` que `show()`. El resultado de `limit()` es un DataFrame tomando solo el número de filas indicadas. Mientras que lo único que hace `show()` es imprimir como texto el contenido, mostrando solo el número de filas que le digamos.

4.1.4. MANIPULACION DE DATOS

Pasemos ahora a hacer algunas operaciones más complejas, manipulando nuestros datos.

Antes hemos visto cómo obtener el conteo de filas de nuestro DataFrame. En la columna `type` tenemos si el vino analizado es blanco (*white*) o tinto (*red*). ¿Cuántos vinos de cada tipo se han analizado?

```
1 wineq.groupBy('type').count().show()
```

```
+-----+-----+
| type|count|
+-----+-----+
|white| 4898|
|  red| 1599|
+-----+-----+
```

Para agrupar datos según el valor de una o más columnas utilizamos el método `groupBy()`. El resultado de `groupBy()` es un objeto de tipo `GroupedData`.

```
1 wineq.groupBy('type')
```

```
<pyspark.sql.group.GroupedData object at 0xb405568c>
```

Sobre el resultado de un **groupBy()** podemos calcular directamente distintos valores

avg() / mean()	Valor promedio de una columna
count()	Conteo de filas
max()	Valor máximo de una columna
min()	Valor mínimo de una columna
sum()	Suma de valores de una columna

Por ejemplo, para ver la puntuación promedio de los vinos blancos y tintos:

```
1 wineq.groupBy('type').avg('quality').show()
```

```
+-----+-----+
| type|    avg(quality)|
+-----+-----+
|white| 5.87790935075541|
| red|5.6360225140712945|
+-----+-----+
```

Si queremos ver distintos valores, podemos utilizar el método **agg()** incluyendo la lista de agregados a calcular.

```
1 import pyspark.sql.functions as F
2
3 wineq.\
4 groupBy('type').\
5 agg(
6     F.avg('alcohol').alias('avg alcohol'),
7     F.max('alcohol').alias('max alcohol')
8 ).show()
9
```

```
+-----+-----+-----+
| type|    avg alcohol|max alcohol|
+-----+-----+-----+
|white|10.514267047770149|      14.2|
| red|10.422983114446502|      14.9|
+-----+-----+-----+
```

Aquí hemos calculado la graduación alcohólica promedio y máxima de los vinos analizados, según el tipo de vino.

Como ves en las líneas 7 y 8 hemos indicado las dos estadísticas que queremos calcular. Las funciones **avg()** y **max()** forman parte del módulo **pyspark.sql.functions**, que hemos importado como **F** en la línea 1.



+ Info

Este módulo incluye multitud de funciones útiles de cálculo y manipulación de datos. En los siguientes ejemplos veremos alguna función más. Si quieres ver un listado completo con su descripción, echa un vistazo a:

<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions>

En las líneas 7 y 8 también hemos usado el método **alias()**. Este método sirve para cambiar el nombre de una columna o para dar nombre a una expresión que nos genera una nueva columna. En este caso, al calcular el valor promedio y máximo estamos generando dos columnas dentro de **agg()**. Con **alias()** les damos nombre a estas columnas (si no, se les asignará un nombre por defecto).

Otra forma de crear nuevas columnas a partir de columnas existentes es usando los métodos **withColumn()** o **withColumnRenamed()**.

```
1 from pyspark.sql.functions import col
2
3 wineq.\
4 withColumn(
5     'avg_acidity',
6     (col('fixed acidity') + col('volatile acidity'))/2
7 ).\
8 withColumnRenamed('free sulfur dioxide', 'free_S02').\
9 select('type', 'avg_acidity', 'free_S02', 'quality').\
10 where(
11     (col('free_S02').isNotNull()) &
12     (col('free_S02') > 100)
13 ).\
14 show(5)
```

```
+-----+-----+-----+-----+
| type|avg_acidity|free_S02|quality|
+-----+-----+-----+-----+
|white|      3.885|   131.0|      5|
|white|      3.545|   122.5|      4|
|white|      3.475|   118.5|      3|
|white|      3.795|   146.5|      3|
|white|      3.865|   128.0|      5|
+-----+-----+-----+-----+
only showing top 5 rows
```

Para empezar, importamos la función `col()` del módulo `pyspark.sql.functions`. Esta función devuelve la columna con el nombre indicado del DataFrame. Esto es necesario cuando queremos construir expresiones que combinan varias columnas. Ya habíamos cargado el módulo completo con el alias `F`. Sin embargo, el uso de `col()` es tan habitual que merece la pena importarla directamente.

En la líneas 4-7 utilizamos `withColumn()` para crear una nueva columna llamada `'avg_acidity'` cuyos valores serán la media de las columnas `'fixed acidity'` y `'volatile acidity'`. Después, en la línea 8 renombramos la columna `'free sulfur dioxide'` con un nombre más corto y manejable como `'free_S02'`.

Tras preparar las columnas, hacemos un `select()` para usar solo aquellas que nos interesan.

Pero esta vez aplicamos también un filtro para quedarnos únicamente con aquellas filas que cumplan cierta condición. Esto lo conseguimos usando `where()` en la línea 10. Dentro de `where()` podemos combinar tantas condiciones como necesitemos. En este caso, comprobamos que `'free_S02'` no sea nulo y que supere las 100 unidades. En lugar de `where()` también puedes utilizar el método `filter()`, son alias uno de otro.

SQL

Al revisar los componentes de Spark ya viste que Spark SQL nos permite acceder a un DataFrame como si fuera una tabla o vista de una base de datos tradicional, y ejecutar consultas usando el lenguaje SQL.

Sigamos con algún ejemplo más.

```

1 wineq.createOrReplaceTempView('wineq_view')
2
3 spark.sql("SELECT * FROM wineq_view").show(5)

```

fixed acidity	volatile acidity	citric acid	...	density	pH	sulphates	alcohol	quality	type
7.4	0.7	0.0		0.9978	3.51	0.56	9.4	5	red
7.8	0.88	0.0		0.9968	3.2	0.68	9.8	5	red
7.8	0.76	0.04	...	0.997	3.26	0.65	9.8	5	red
11.2	0.28	0.56		0.998	3.16	0.58	9.8	6	red
7.4	0.7	0.0		0.9978	3.51	0.56	9.4	5	red

only showing top 5 rows

Lo primero que tenemos que hacer para poder usar la interfaz SQL es registrar el DataFrame como una vista. Aquí lo hacemos usando `createOrReplaceTempView()` en la línea 1 para registrar una vista temporal a la que llamamos `wineq_view`. El nombre que asignemos a la vista será el que tendremos que usar después en nuestras consultas SQL.

Existen varios métodos para definir una vista. Esencialmente se diferencian en qué hacer si ya existe una vista con el mismo nombre (reemplazarla o dar un error), y en si la vista es accesible solo para la sesión donde se crea o si es accesible para todas las sesiones activas (por ejemplo, una aplicación puede mantener varias sesiones para distintos usuarios simultáneos).

createTempView()	<p>Crea una vista a partir de un DataFrame</p> <p>La vista solo es accesible en la sesión donde se ha creado</p> <p>Si ya existe una vista con el mismo nombre, da un error</p>
createOrReplaceTempView()	<p>Crea una vista a partir de un DataFrame</p> <p>La vista solo es accesible en la sesión donde se ha creado</p> <p>Si ya existe una vista con el mismo nombre, la reemplaza</p>
createGlobalTempView()	<p>Crea una vista a partir de un DataFrame</p> <p>La vista es accesible por todas las sesiones activas mientras la aplicación Spark no termine</p> <p>Si ya existe una vista con el mismo nombre, da un error</p>
createOrReplaceGlobalTempView()	<p>Crea una vista a partir de un DataFrame</p> <p>La vista es accesible por todas las sesiones activas mientras la aplicación Spark no termine</p> <p>Si ya existe una vista con el mismo nombre, la reemplaza</p>

Terminando con el ejemplo, en la línea 3 ves como ejecutar una consulta SQL básica para seleccionar todos los datos mediante el método `spark.sql()`. Fíjate en lo que comentábamos, cómo usamos el nombre de la vista dentro de la consulta SQL.

Podemos utilizar las capacidades habituales de SQL en nuestras consultas

```

1 df = spark.sql(
2     "SELECT type, \
3         count(*) AS n, \
4         avg(alcohol) AS avg_alcohol, \
5         avg(`volatile acidity`) AS avg_vol_acid, \
6         avg(quality) AS avg_quality\
7     FROM wineq_view \
8     WHERE `free sulfur dioxide` > 60 \
9     GROUP BY type"
10 )
11
12 df

```

```
DataFrame[type: string, n: bigint, avg_alcohol: double, avg_vol_acid: double, avg_quality: double]
```

El resultado de una consulta SQL es también un `DataFrame`; podemos operar con él y combinarlo con otros `DataFrames` sin ningún tipo de diferencia.

```
1 df.show()
```

type	n	avg_alcohol	avg_vol_acid	avg_quality
white	346	9.838872832387274	0.27225433526011544	5.6502890173410405
red	4	9.9	0.5975	5.25



Importante

¿Te diste cuenta de un detalle? En los nombres de columnas que tienen espacios tuvimos que utilizar tildes invertidas (` `) para rodearlos y que Spark SQL lo entienda. Si no hubiéramos puesto las tildes invertidas, habríamos obtenido un error.

¡Pero cuidado! No te confundas y pongas unas comillas simples. Entonces Spark SQL lo entenderá como una cadena de texto, no como el nombre de una columna.

Funciones definidas por el usuario

Al manipular un DataFrame, también podemos aplicar funciones programadas por nosotros para hacer cálculos sobre columnas y generar nuevos valores.

A este tipo de funciones las denominamos *funciones definidas por el usuario* o UDFs (*User Defined Functions*).

En el siguiente ejemplo definimos una UDF que etiqueta los vinos según su calidad y la aplicamos a nuestro DataFrame.

```
1  from pyspark.sql.functions import udf
2  from pyspark.sql.types import StringType
3
4  def categorize(quality):
5      if quality >= 0 and quality < 3:
6          return '0 - VERY POOR'
7      elif quality < 5:
8          return '1 - POOR'
9      elif quality < 6.5:
10         return '2 - FAIR'
11     elif quality < 8:
12         return '3 - GOOD'
13     elif quality < 9.5:
14         return '4 - VERY GOOD'
15     elif qualite <= 10:
16         return '5 - EXCELLENT'
17     else:
18         return None
19
20  udf_categorize = udf(categorize, StringType())
21
22  wineq.\
23  withColumn(
24      "category",
25      udf_categorize("quality")
26  ).\
27  groupBy('type', 'category').\
28  count().\
29  orderBy('type', 'category').\
30  show()
```

type	category	count
red	1 - POOR	63
red	2 - FAIR	1319
red	3 - GOOD	199
red	4 - VERY GOOD	18
white	1 - POOR	183
white	2 - FAIR	3655
white	3 - GOOD	880
white	4 - VERY GOOD	180

Entre las líneas 4 y 18 tenemos nuestra función. Como ves, es una función de Python normal y corriente. Dependiendo de la calidad devuelve una cadena de texto con la categoría a la que pertenece.

Lo interesante viene a continuación. En la línea 20 utilizamos la función `udf()` del módulo `pyspark.sql.functions` para registrar nuestra función como una UDF dentro de Spark. Esto es imprescindible para que Spark sepa que nuestra función puede usarse con DataFrames. A `udf()` le pasamos el nombre de nuestra función (sin comillas) y el tipo de dato de la columna resultado. Para indicar el tipo debemos usar alguno de los definidos en el módulo `pyspark.sql.types`. En este caso es de tipo `StringType`. Otros tipos válidos comunes serían `BooleanType`, `DoubleType`, `IntegerType`; también tenemos `DateType` o `TimestampType` para fechas y valores temporales. Tienes el listado completo de tipos en:

<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.types>

Y una vez que hemos registrado nuestra función, ya podemos usarla con nuestros DataFrame. En el ejemplo, creamos una nueva columna aplicando la función (líneas 23-26) y después usamos esta columna para agrupar y contar ocurrencias.

Plan de ejecución

Si recuerdas del capítulo 3 de esta unidad, vimos que al trabajar con los datos (ya sea con RDDs o DataFrames), Spark distingue entre operaciones de transformación y acciones.

Una transformación implica realizar cálculos o manipulaciones sobre unos datos de entrada para producir unos datos de salida. Las acciones son operaciones que no producen nuevos resultados, pero requieren que todos los cálculos previos estén completados para poder ejecutarse (p.ej. mostrar las primeras filas o guardar el DataFrame). Spark intenta retrasar todo lo posible la ejecución de las transformaciones.

Además, para optimizar más el flujo de cálculos, Spark distingue internamente entre transformaciones estrechas y amplias. Una transformación estrecha (p.ej. sumar dos columnas o filtrar algunas filas con una condición) se puede ejecutar en los datos de cada nodo de forma independiente. Una transformación amplia (p.ej. agrupar y agregar datos, u ordenar por una columna) puede requerir intercambiar datos entre los nodos.

Para saber cómo va a ejecutar Spark una secuencia de operaciones, podemos pedirle que nos *explique* su plan. Para ello los objetos DataFrame disponen también del método `explain()`.

Repitamos la operación de contar cuántos vinos hay de cada tipo en los datos.

```
1 wineq.groupBy('type').count().show()
```

```
+-----+-----+
| type|count|
+-----+-----+
|white| 4898|
|  red| 1599|
+-----+-----+
```

¿Cuál es el plan de ejecución de Spark para calcular esto?

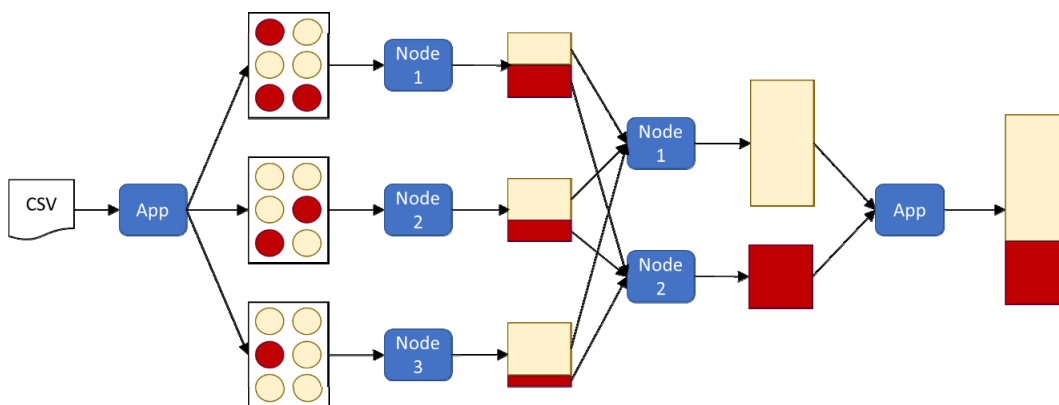
```
1 wineq.groupBy('type').count().explain()
```

```
== Physical Plan ==
*HashAggregate(keys=[type#67], functions=[count(1)])
+- Exchange hashpartitioning(type#67, 200)
   +- *HashAggregate(keys=[type#67], functions=[partial_count(1)])
      +- *FileScan csv [type#67] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/bigdata/Curso/U13_Spark/datos/winequality.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<type:string>
```

Empezando del nivel más interno hacia afuera, vemos que primero hay un *FileScan*, que significa que tiene que empezar procesando los datos de fichero.

La primera operación *HashAggregate* corresponde al **groupBy**, ejecutado en cada nodo por separado. Para entendernos, en cada nodo que tiene una porción de los datos Spark va a calcular cuántos vinos blancos y tintos hay. El siguiente paso es el *Exchange*. Esto es, intercambiar entre los nodos los datos de cada grupo (blanco o tinto), de manera que todos los conteos parciales de un mismo tipo de vino vayan a parar a un mismo nodo. Y por último, un nuevo *HashAggregate* para sumar todos los conteos parciales de cada nodo y obtener el número total de ocurrencias de cada tipo de vino.

Tal vez con la siguiente figura veas más clara la secuencia de pasos que ejecuta Spark para completar el ejemplo.



4.1.5. UTILIZANDO MLlib

Ahora que ya sabemos manejarnos con los datos, vamos a explorar un poco qué podemos hacer con la librería de aprendizaje estadístico y machine learning de Spark, MLlib.

MLlib incluye una gran variedad de algoritmos para construir modelos de previsión, clasificación, reconocimiento de patrones... En los siguientes apartados te mostramos dos ejemplos sencillos.

Regresión

Imagina que queremos poder dar una primera estimación de la calidad de un vino en base a algunas de las variables químicas y organolépticas que medimos, antes de pasar por el proceso de cata de los enólogos.

Esto es un caso típico de problema de regresión para estimar un valor no medido a partir de ciertas variables explicativas observadas.

Pero antes de diseñar nuestro modelo de regresión, necesitamos saber si existe relación entre las variables explicativas y el indicador de calidad que queremos tratar de estimar.

Para obtener una medida, calculamos la correlación entre cada variable y la calidad observada. Separamos entre vinos blancos y tintos para comprobar si existen diferencias entre ambos.

```

1  import pyspark.sql.functions as F
2  from pyspark.sql.functions import col
3
4  wineq.\
5  groupBy('type').\
6  agg(
7      F.round(F.corr('quality','fixed acidity'), 2).\
8          alias('corr_fix_acid'),
9      F.round(F.corr('quality','volatile acidity'), 2).\
10         alias('corr_vol_acid'),
11      F.round(F.corr('quality','citric acid'), 2).\
12         alias('corr_citr_acid'),
13      F.round(F.corr('quality','residual sugar'), 2).\
14         alias('corr_res_sugar'),
15      F.round(F.corr('quality','chlorides'), 2).\
16         alias('corr_chlor'),
17      F.round(F.corr('quality','free sulfur dioxide'), 2).\
18         alias('corr_free_S02'),
19      F.round(F.corr('quality','total sulfur dioxide'), 2).\
20         alias('corr_tot_S02'),
21      F.round(F.corr('quality','density'), 2).\
22         alias('corr_dens'),
23      F.round(F.corr('quality','pH'), 2).alias('corr_pH'),
24      F.round(F.corr('quality','sulphates'), 2).\
25         alias('corr_sulph'),
26      F.round(F.corr('quality','alcohol'), 2).\
27         alias('corr_alcohol')
28  ).show()
```

type	corr_fix_acid	corr_vol_acid	corr_citr_acid	corr_res_sugar	corr_chlor
white	-0.11	-0.19	-0.01	-0.1	-0.21
red	0.12	-0.39	0.23	0.01	-0.13

type	corr_free_S02	corr_tot_S02	corr_dens	corr_pH	corr_sulph	corr_alcohol
white	0.01	-0.17	-0.31	0.1	0.05	0.44
red	-0.05	-0.19	-0.17	-0.06	0.25	0.48

Podemos ver los coeficientes de correlación de cada variable con la calidad, dependiendo de si el tipo de vino es blanco o tinto.

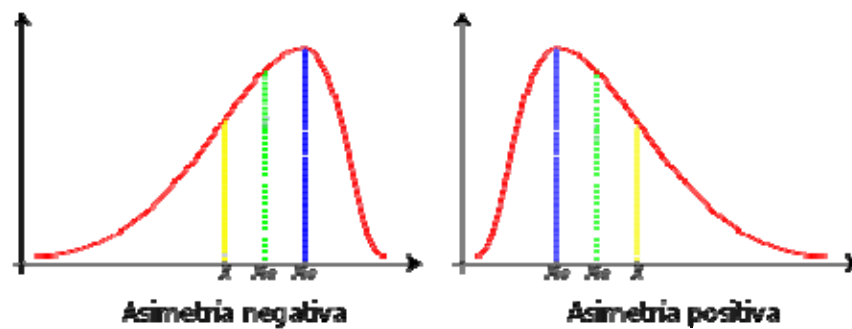


Ojo al dato

Recuerda que el coeficiente de correlación puede variar entre -1 y 1. Un coeficiente de correlación próximo a 1 nos indica que existe una fuerte correlación positiva (si la variable explicativa aumenta, también lo hace proporcionalmente la variable dependiente); Un valor igual a 1 nos dice que hay una correlación perfecta. Y al contrario, si el coeficiente de correlación está próximo a -1, nos indica que existe una fuerte correlación negativa (si la variable explicativa crece, la variable dependiente disminuye proporcionalmente). Si el coeficiente de correlación vale cero, no podemos establecer una relación lineal (proporcional) entre las variables (pero podría haber una relación no lineal).

Con los datos anteriores, parece que la calidad del vino muestra relación con el contenido alcohólico, la densidad o la acidez volátil (ácidos gaseosos que contribuyen al aroma y sabor del vino), entre otras variables.

Es un buen punto de partida. Sin embargo, sospechamos que algunas de estas variables pueden tener una distribución asimétrica. Esto significa que la variable registra más valores a un lado de su media que al otro.



De Fuente: Skewness_Statistics.svg: Original uploader was Godot at en.wikipediaderivative work: Tartaglia (talk) - Skewness_Statistics.svg, CC BY-SA 3.0:

<https://commons.wikimedia.org/w/index.php?curid=6410277>

La asimetría afecta a la relación de las variables y afectaría a nuestro modelo. Podemos calcular el coeficiente de asimetría (*skewness*) de cada variable.

```

1 wineq.\
2 groupBy('type').\
3 agg(
4     F.round(F.skewness('fixed acidity'), 2).\
5     alias('skw_fix_acid'),
6     F.round(F.skewness('volatile acidity'), 2).\
7     alias('skw_vol_acid'),
8     F.round(F.skewness('citric acid'), 2).\
9     alias('skw_citr_acid'),
10    F.round(F.skewness('residual sugar'), 2).\
11    alias('skw_res_sugar'),
12    F.round(F.skewness('chlorides'), 2).\
13    alias('skw_chlor'),
14    F.round(F.skewness('free sulfur dioxide'), 2).\
15    alias('skw_free_S02'),
16    F.round(F.skewness('total sulfur dioxide'), 2).\
17    alias('skw_tot_S02'),
18    F.round(F.skewness('density'), 2).\
19    alias('skw_dens'),
20    F.round(F.skewness('pH'), 2).\
21    alias('skw_pH'),
22    F.round(F.skewness('sulphates'), 2).\
23    alias('skw_sulph'),
24    F.round(F.skewness('alcohol'), 2).\
25    alias('skw_alcohol')
26 ).show()

```

type	skw_fix_acid	skw_vol_acid	skw_citr_acid	skw_res_sugar	skw_chlor
white	0.65	1.58	1.28	1.08	5.02
red	0.98	0.67	0.32	4.54	5.68

type	skw_free_S02	skw_tot_S02	skw_dens	skw_pH	skw_sulph	skw_alcohol
white	1.41	0.39	0.98	0.46	0.98	0.49
red	1.25	1.51	0.07	0.19	2.43	0.86

El coeficiente de asimetría puede ser positivo o negativo, dependiendo de si la cola de la distribución se estira hacia la derecha o hacia la izquierda. Si obtenemos un valor cero nos indica que no hay evidencia de asimetría. Normalmente se aplica el criterio

$skw(x) < -1$	$-1 < skw(x) \leq -0.5$	$-0.5 < skw(x) \leq 0.5$	$0.5 < skw(x) \leq 1$	$skw(x) > 1$
Asimetría negativa alta	Asimetría negativa moderada	Aproximadamente simétrica	Asimetría positiva moderada	Asimetría positiva alta

En nuestro caso, vemos que existen evidencias de asimetría moderada o alta para la mayoría de las variables.

En estos casos, una solución pasa por *transformar* las variables aplicando una función que *redistribuya* los resultados con una forma más simétrica. Como nuestras variables presentan unos coeficientes de asimetría positivos por presencia de algunos valores muy altos a la derecha de la distribución, vamos a aplicar una transformación logarítmica.

```

1 wineq_tr = (
2     wineq.\
3     select(
4         F.log1p('fixed acidity').alias('log_fix_acid'),
5         F.log1p('volatile acidity').alias('log_vol_acid'),
6         F.log1p('citric acid').alias('log_citr_acid'),
7         F.log1p('residual sugar').alias('log_res_sugar'),
8         F.log1p('chlorides').alias('log_chlor'),
9         F.log1p('free sulfur dioxide').alias('log_free_S02'),
10        F.log1p('total sulfur dioxide').alias('log_tot_S02'),
11        F.log1p('density').alias('log_dens'),

```

```

12         F.log1p('pH').alias('log_pH'),
13         F.log1p('sulphates').alias('log_sulph'),
14         F.log1p('alcohol').alias('log_alcohol'),
15         'quality',
16         'type'
17     )
18 )
19
20 wineq_tr.show(5)
21
22

```

log_fix_acid	log_vol_acid	log_citr_acid	log_res_sugar	log_chlor	log_free_S02
2.13	0.53	0.0	1.06	0.07	2.48
2.17	0.63	0.0	1.28	0.09	3.26
2.17	0.57	0.04	1.19	0.09	2.77
2.5	0.25	0.44	1.06	0.07	2.89
2.13	0.53	0.0	1.06	0.07	2.48

log_tot_S02	log_dens	log_pH	log_sulph	log_alcohol	quality	type
3.56	0.69	1.51	0.44	2.34	5	red
4.22	0.69	1.44	0.52	2.38	5	red
4.01	0.69	1.45	0.5	2.38	5	red
4.11	0.69	1.43	0.46	2.38	6	red
3.56	0.69	1.51	0.44	2.34	5	red

only showing top 5 rows

Como ves, hemos creado un nuevo DataFrame aplicando la función **F.log1p()** a las variables del DataFrame original, manteniendo la puntuación y el tipo de vino.

Comprobemos cómo quedan las correlaciones ahora con estas variables transformadas.

```

1 wineq_tr.\
2 groupBy('type').\
3 agg(
4     F.round(F.corr('quality','log_fix_acid'), 2).\
5     alias('corr_log_fix_acid'),
6     F.round(F.corr('quality','log_vol_acid'), 2).\
7     alias('corr_log_vol_acid'),
8     F.round(F.corr('quality','log_citr_acid'), 2).\
9     alias('corr_log_citr_acid'),
10    F.round(F.corr('quality','log_res_sugar'), 2).\
11    alias('corr_log_res_sugar'),
12    F.round(F.corr('quality','log_chlor'), 2).\

```

```

13     alias('corr_log_chlor'),
14     F.round(F.corr('quality','log_free_S02'), 2).\
15     alias('corr_log_free_S02'),
16     F.round(F.corr('quality','log_tot_S02'), 2).\
17     alias('corr_log_tot_S02'),
18     F.round(F.corr('quality','log_dens'), 2).\
19     alias('corr_log_dens'),
20     F.round(F.corr('quality','log_pH'), 2).\
21     alias('corr_log_pH'),
22     F.round(F.corr('quality','log_sulph'), 2).\
23     alias('corr_log_sulph'),
24     F.round(F.corr('quality','log_alcohol'), 2).\
25     alias('corr_log_alcohol')
26 ).show()

```

```

+-----+-----+-----+-----+
| type|corr_log_fix_acid|corr_log_vol_acid|corr_log_citr_acid|
+-----+-----+-----+-----+
|white|          -0.11|          -0.2|           0.01|
| red|           0.12|          -0.39|           0.22|
+-----+-----+-----+-----+

+-----+-----+-----+-----+-----+
| type|corr_log_res_sugar|corr_log_chlor|corr_log_free_S02|corr_log_tot_S02|
+-----+-----+-----+-----+-----+
|white|          -0.08|          -0.22|           0.09|          -0.12|
| red|           0.02|          -0.13|          -0.05|          -0.17|
+-----+-----+-----+-----+-----+

+-----+-----+-----+-----+-----+
| type|corr_log_dens|corr_log_pH|corr_log_sulph|corr_log_alcohol|
+-----+-----+-----+-----+-----+
|white|          -0.31|           0.1|           0.05|           0.43|
| red|          -0.18|          -0.06|           0.28|           0.48|
+-----+-----+-----+-----+-----+

```

Ahora ya podemos diseñar nuestro modelo. Empezaremos creando un modelo específico para los vinos tintos. Te presentamos el código paso a paso.

```

1  from pyspark.ml.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3
4  assembler = VectorAssembler(
5      inputCols=['log_vol_acid','log_citr_acid','log_chlor',
6                'log_tot_S02','log_dens','log_sulph',
7                'log_alcohol'],
8      outputCol="features")
9
10 wineq_trf = assembler.transform(wineq_tr)
11
12 wineq_trf_red = (
13     wineq_trf.\

```

```

14     where(col('type') == 'red').\
15     select('features','quality')
16 )
17
18 wineq_trf_red.show(5)

```

```

+-----+-----+
|          features|quality|
+-----+-----+
|[2.34180580614732...|      5|
|[2.37954613413017...|      5|
|[2.37954613413017...|      5|
|[2.37954613413017...|      6|
|[2.34180580614732...|      5|
+-----+-----+
only showing top 5 rows

```

Lo primero que tenemos que hacer es preparar nuestros datos para poderlos usar con los algoritmos de machine learning de la librería MLlib. La práctica totalidad de algoritmos de esta librería esperan que los datos de entrada tengan la misma forma. Una columna para la variable a predecir y otra columna donde las variables explicativas se *empaquetan* en vectores.

Para preparar los datos en este formato, utilizamos un **VectorAssembler**. En las líneas 4-7 ves cómo configuramos este *ensamblador* de vectores. Le decimos que debe buscar determinadas columnas de entrada (**inputCols**) y para cada fila generar un vector compuesto por los valores de dichas columnas. Estos vectores los añadirá en una columna de salida (**outputCol**).

Una vez que hemos configurado el **VectorAssembler**, simplemente hay que utilizar su método **transform()** para que construya la nueva columna de vectores a partir del DataFrame de entrada (línea 9).

A continuación nos quedamos únicamente con las columnas **'quality'** y **'features'** (los vectores de variables explicativas) correspondientes a los vinos tintos (líneas 11-15). Ya tenemos los datos preparados.

A continuación, vamos a dividir los datos en un conjunto de entrenamiento y otro de test, y procederemos a crear el modelo y ajustarlo.

```
1 from pyspark.ml.regression import LinearRegression
2 import numpy as np
3
4 wineq_train, wineq_test = wineq_trf_red.randomSplit(
5                                     [0.7, 0.3])
6
7 lr = LinearRegression(featuresCol = 'features',
8                       labelCol = 'quality')
9
10 lr_model = lr.fit(wineq_train)
```

En la primera línea cargamos la clase que implementa los algoritmos de regresión lineal del módulo `pyspark.ml.regression`. En las líneas 4-5 utilizamos el método `randomSplit()` del `DataFrame` para dividir los datos aleatoriamente en dos conjuntos: uno con el 70% de las observaciones (`wineq_train`) y otro con el 30% restante (`wineq_test`).

En las líneas 7-8 construimos el modelo de regresión lineal con la clase `LinearRegression`. Tenemos que indicar cuál es la columna que tiene el vector de variables explicativas (`featuresCol`) y cuál es la columna objetivo (`labelCol`).

Tras construir el modelo, lo ajustamos usando el conjunto de entrenamiento (línea 10). La variable `lr_model` contiene el modelo ajustado, que podremos usar para predecir.

Vamos a mostrar algunos resultados del modelo ajustado.

```
1 def print_model_results(model, feature_names):
2     coefs = np.append(0, model.coefficients.values)
3     p_values = model.summary.pValues
4     t_values = model.summary.tValues
5
6     print("Variable\tCoefficient\tp-Value.\tt-Value.")
7     for model_res in zip(feature_names, coefs,
8                           p_values, t_values):
9         print("%s\t%f\t%f\t%f" % model_res)
10
11     print("RMSE : %f" % model.summary.rootMeanSquaredError)
12     print("r2 : %f" % model.summary.r2)
```

```

13
14
15 print_model_results(lr_model,
16                       ['Intercept', 'log_vol_acid',
17                       'log_citr_acid', 'log_chlor',
18                       'log_tot_S02', 'log_dens',
19                       'log_sulph', 'log_alcohol'])

```

Variable	Coefficient	p-Value.	t-Value.
Intercept	0.000000	0.000000	-6.744534
log_vol_acid	-1.568092	0.930227	0.087579
log_citr_acid	0.016869	0.001018	-3.294341
log_chlor	-1.927375	0.075106	-1.781493
log_tot_S02	-0.053022	0.630350	-0.481368
log_dens	-13.621182	0.000000	6.758416
log_sulph	1.623495	0.000000	11.157312
log_alcohol	3.277196	0.714645	0.365719

RMSE : 0.649295

r2 : 0.334448

Hemos preparado una pequeña función auxiliar (**print_model_results**) para facilitar la tarea de imprimir los principales indicadores. Mostramos los coeficientes ajustados para cada variable explicativa, la raíz del error cuadrático medio (RMSE) y el coeficiente de correlación (r2).

Vamos a estimar ahora la calidad sobre el conjunto de datos de test.

```

1 lr_predict = lr_model.transform(wineq_test)
2
3 lr_predict.\
4 select("prediction", "quality", "features").\
5 show(5)

```

prediction	quality	features
6.55803836829238	6	[0.14842000511827...
6.485034681007593	6	[0.16551443847757...
6.222930463309186	6	[0.16551443847757...
6.222930463309186	6	[0.16551443847757...
5.728738165909047	5	[0.18232155679395...

only showing top 5 rows

Para ejecutar una predicción, utilizamos el método **transform()** del modelo ajustado (línea 1). Tenemos que pasarle el DataFrame con los datos de entrada (**wineq_test**).

El resultado de ejecutar la predicción es un nuevo DataFrame, que incluye una columna "**prediction**" con el nuevo valor estimado para la calidad.

Para terminar, vamos a evaluar la corrección de los resultados obtenidos con el conjunto de test.

```
1 from pyspark.ml.evaluation import RegressionEvaluator
2
3 lr_evaluator = RegressionEvaluator(
4     predictionCol = "prediction",
5     labelCol = "quality",
6     metricName = "rmse")
7
8 print("RMSE on test data = %g" %
9     lr_evaluator.evaluate(lr_predict))
```

```
RMSE on test data = 0.654682
```

Spark incluye varios métodos de evaluación y análisis de resultados de los modelos de *machine learning* en el módulo `pyspark.ml.evaluation`. En este caso, usamos la clase **RegressionEvaluator** para evaluar los resultados de la regresión (línea 1). Debemos indicarle cuál es la columna con el valor estimado por el modelo (**predictionCol**), la columna con el valor real (**labelCol**) y qué métrica queremos calcular para evaluar los resultados (**metricName**). En este caso, queremos calcular la raíz del error cuadrático medio ("rmse").

Para evaluar dicho indicador, simplemente usamos el método **evaluate()** (línea 9), pasándole el DataFrame con los resultados de la predicción.

Clasificación

Otra tarea típica de *machine learning* es construir un modelo de clasificación que nos ayude a detectar si una entidad es de tal o cual tipo en base a algunas características medibles. La librería MLlib de Spark nos proporciona también varios algoritmos para desarrollar estas tareas de clasificación.

Siguiendo con los datos de vino, vamos a intentar construir un clasificador que sepa distinguir si un vino es blanco o tinto solamente a partir de las variables químicas y organolépticas que medimos.

Si recuerdas de la unidad 8, la regresión logística es una de las técnicas más extendidas para predecir una variable categórica. Naturalmente, MLlib también incluye este algoritmo.


```

1  from pyspark.ml.classification import LogisticRegression
2  from pyspark.ml.feature import StringIndexer
3
4  assembler = VectorAssembler(
5      inputCols=['fixed acidity', 'volatile acidity',
6                 'citric acid','chlorides',
7                 'total sulfur dioxide','density',
8                 'sulphates', 'alcohol'],
9      outputCol="features")
10
11  wineqf = assembler.transform(wineq)
12
13  indexer = StringIndexer(inputCol="type",
14                          outputCol="typeIndex")
15
16  wineqf_ind = indexer.fit(wineqf).transform(wineqf)
17
18  wineqf_ind = (
19      wineqf_ind.\
20      select('features','typeIndex','type')
21  )
22
23  wineq_train, wineq_test = wineqf_ind.randomSplit([0.7, 0.3])
24
25  lr = LogisticRegression(featuresCol = 'features',
26                          labelCol = 'typeIndex',
27                          fitIntercept = True)
28
29  lr_model = lr.fit(wineq_train)

```

Lo primero que hacemos es importar la clase **LogisticRegression** del módulo **pyspark.ml.classification**. Igual que en el ejemplo anterior, tenemos que empaquetar las variables explicativas en vectores (líneas 4-11). Recuerda que esto será necesario para la mayoría de modelos.

En las líneas 13-16 hacemos algo que es particular para los modelos de clasificación. Queremos predecir el tipo de vino (blanco o tinto). Pero la variable `type` es de tipo cadena. Sin embargo, el modelo de clasificación necesita que las categorías estén representadas de forma numérica. El **StringIndexer** de la línea 13 nos permite asignar a cada categoría (cadena 'white' o 'red') un índice numérico. El resultado se almacena en una nueva columna '**typeIndex**'. Para entrenar el modelo, nos quedamos únicamente con las columnas con la cadena del tipo blanco/tinto original (para tenerlo de referencia), el índice de tipo (que necesita el clasificador) y con los vectores de variables explicativas ('**features**') (líneas 18-20). En línea 23 dividimos los datos en los conjuntos de entrenamiento y test.

Finalmente, creamos el objeto con la definición del modelo (líneas 25-27) y lo ajustamos con los datos de entrenamiento.

Vamos a ver información sobre el modelo ajustado.

```
1 print("Coefficients: " + str(lr_model.coefficients))
2 print("Intercepts: " + str(lr_model.intercept))
3 print("areaUnderROC: " + str(lr_model.summary.areaUnderROC))
```

```
Coefficients: [0.870560477041,10.6628576526,-3.25572563584,32.4294964189,-
0.0626662927366,-8.04179792612,12.0732545723,-0.442177677557]
Intercepts: -1.1235136098553278
areaUnderROC: 0.9937223145071984
```

Mostramos los valores de los coeficientes para cada variable explicativa y el intercept. También mostramos el indicador del área bajo la curva ROC.

Vamos a aplicar el modelo al conjunto de test.

```
1 lr_predict = lr_model.transform(wineq_test)
2
3 lr_predict.\
4 select("prediction", "typeIndex", "type", "features").\
5 show(5)
```

```
+-----+-----+-----+-----+
|prediction|typeIndex| type|          features|
+-----+-----+-----+-----+
|      0.0|      0.0|white|[3.8,0.31,0.02,0....|
|      0.0|      0.0|white|[4.2,0.17,0.36,0....|
|      0.0|      0.0|white|[4.2,0.215,0.23,0...|
|      0.0|      0.0|white|[4.4,0.32,0.39,0....|
|      0.0|      0.0|white|[4.5,0.19,0.21,0....|
+-----+-----+-----+-----+
only showing top 5 rows
```

Como ves, usamos también el método **predict()**, igual que en la regresión lineal. Si te fijas, vemos que al tipo vino blanco (white) le corresponde el índice (**typeIndex**) 0.

Para comprobar cómo de bien clasifica nuestro modelo los vinos blancos y tintos, calculamos la tabla de contingencia de casos dependiendo del tipo real (filas) y el tipo predicho por el modelo (columnas).

```
1 lr_predict.\
2 groupBy('typeIndex').\
3 pivot('prediction', values=[0.0,1.0]).\
4 count().\
5 show()
```

```
+-----+-----+
|typeIndex| 0.0|1.0|
+-----+-----+
|      0.0|1467| 24|
|      1.0| 19|462|
+-----+-----+
```

Parece que nuestro clasificador funciona muy bien. Podemos obtener el indicador de área bajo la curva ROC usando otra clase del módulo `pyspark.ml.evaluation`, específica para modelos de clasificación binarios (con dos clases).

```
1 from pyspark.ml.evaluation import
2   BinaryClassificationEvaluator
3
4 lr_evaluator = BinaryClassificationEvaluator(
5     rawPredictionCol = "prediction",
6     labelCol = "typeIndex",
7     metricName = "areaUnderROC")
8
9 print("Area under ROC = %g" %
10      lr_evaluator.evaluate(lr_predict))
```

```
Area under ROC = 0.972201
```

4.2. APLICACIÓN SPARK

Hasta ahora hemos estado trabajando con la consola de PySpark para introducir y probar nuestro código de forma interactiva.

Sin embargo, cuando ya tengamos nuestros programas para Spark probados y depurados, nos gustaría poder ejecutarlos directamente en el *cluster* de Spark, igual que ejecutamos un script de Python.

Para hacer esto en Spark utilizamos el comando **spark-submit**. La forma más rápida de ver como funciona es de nuevo con un ejemplo. En nuestra máquina virtual, abre un terminal MATE y ejecuta el siguiente comando:

```
$ spark-submit --master local
./Curso/U13_Spark/src/ejemplo_5_2_submit.py
```

Si todo va bien, empezarás a ver un montón de mensajes de Spark por la consola del terminal conforme el sistema arranca y se configura, y empieza a ejecutar nuestro código.

El comando **spark-submit** envía nuestro programa o aplicación a la plataforma Spark, para que ésta se encargue de asignar los recursos necesarios para la aplicación, distribuir el código entre los nodos del *cluster* donde se vaya a ejecutar y proceda con la ejecución.

A **spark-submit** hay que indicarle el punto de entrada al *cluster* de Spark con la opción **--master**. En nuestro ejemplo básico, como no tenemos ningún *cluster*, lanzaremos una instancia de Spark local en nuestra propia máquina. Esto lo indicamos con **--master local**. Si tuviéramos acceso a un *cluster* en alguna URL o dirección IP, usaríamos dicha dirección.

El segundo argumento que pasamos a **spark-submit** es el fichero con el script en Python a ejecutar. Puedes ver el código accediendo al fichero incluido en el material complementario de la unidad (disponible en el Campus) o en la propia máquina virtual. Lo incluimos también aquí para que podamos revisarlo y comentarlo.

```
1 from pyspark.sql import SparkSession
2 import pyspark.sql.functions as F
3 from pyspark.sql.functions import col
4 from pyspark.ml.linalg import Vectors
5 from pyspark.ml.feature import VectorAssembler
6 from pyspark.ml.regression import LinearRegression
7 import numpy as np
8
9 # Crear la sesión de Spark
10 spark = SparkSession\
11     .builder\
12     .master("local")\
13     .appName("ejemplo-spark")\
14     .getOrCreate()
```

```

15
16
17 # Leer datos de fichero local
18 local_file =
19 r'file:///home/bigdata/Curso/U13_Spark/datos/winequality.csv'
20
21 wineq = spark.read.csv(path=local_file, header=True, sep=';',
22                         inferSchema=True)
23
24 # Transformamos las variables predictoras, pasando a logaritmos
25 wineq_tr = (
26     wineq.\
27     select(
28         F.log1p('fixed acidity').alias('log_fix_acid'),
29         F.log1p('volatile acidity').alias('log_vol_acid'),
30         F.log1p('citric acid').alias('log_citr_acid'),
31         F.log1p('residual sugar').alias('log_res_sugar'),
32         F.log1p('chlorides').alias('log_chlor'),
33         F.log1p('free sulfur dioxide').alias('log_free_SO2'),
34         F.log1p('total sulfur dioxide').alias('log_tot_SO2'),
35         F.log1p('density').alias('log_dens'),
36         F.log1p('pH').alias('log_pH'),
37         F.log1p('sulphates').alias('log_sulph'),
38         F.log1p('alcohol').alias('log_alcohol'),
39         'quality',
40         'type'
41     )
42 )
43
44 # Construimos los vectores de variables explicativas
45 assembler = VectorAssembler(
46     inputCols=['log_vol_acid', 'log_citr_acid', 'log_chlor',
47               'log_tot_SO2', 'log_dens', 'log_sulph',
48               'log_alcohol'],
49     outputCol="features")
50
51 wineq_trf = assembler.transform(wineq_tr)
52
53 # Nos quedamos solo con los datos de vinos tintos
54 wineq_trf_red = (
55     wineq_trf.\
56     where(col('type') == 'red').\
57     select('features', 'quality')
58 )
59
60 # Dividimos los datos aleatoriamente
61 # en conjuntos de entrenamiento y test
62 wineq_train, wineq_test = wineq_trf_red.randomSplit(
63     [0.7, 0.3])

```

```
64
65 # Construimos el modelo de regresion lineal
66 lr = LinearRegression(featuresCol = 'features',
67                        labelCol = 'quality')
68
69 # Ajustamos el modelo con el subconjunto de entrenamiento
70 lr_model = lr.fit(wineq_train)
71
72 # Predecir la calidad sobre el conjunto de test
73 lr_predict = lr_model.transform(wineq_test)
74
75 # Guardamos las predicciones en un fichero CSV local
76 lr_predict.\
77     select("prediction", "quality").\
78     coalesce(1).\
79
80 write.csv(r'file:///home/bigdata/Curso/U13_Spark/output/wqpred',
81          header=True, sep=';', mode = 'overwrite')
```

En general, el código de este script simplemente recopila todo lo que ya hemos hecho en ejemplos anteriores: leer los datos de fichero, transformarlos, construir los vectores de variables explicativas, dividir los datos en conjuntos de entrenamiento y test, construir un modelo de regresión lineal, ajustarlo y predecir los resultados sobre el conjunto de test.

Lo importante de este ejemplo está en las líneas 10 a 14. Aquí es donde obtenemos de la plataforma la **SparkSession** para nuestra aplicación. Recuerda que este objeto es nuestro punto de acceso a la plataforma Spark. Cuando trabajamos con la consola de PySpark, ésta se encarga de proporcionarnos la sesión automáticamente. Pero cuando creamos un script para lanzarlo sobre la plataforma, lo primero que tendremos que hacer será solicitar a Spark que nos asigne una sesión.

Una vez que tenemos la sesión, ya podemos seguir normalmente. Como puedes comprobar, el resto del código es idéntico al que ya has visto antes.

Al final del script escribimos los resultados de `lr_predict` a fichero en formato CSV (líneas 76-81). El método **coalesce(1)** indica a Spark que queremos reunir los resultados particionados en un único bloque o partición. Con **write.csv()** terminamos escribiendo estos resultados con formato CSV.

En el terminal MATE puedes comprobar que se ha creado un directorio **wqpred** con los resultados.

```
$ ls -alrt ./Curso/U13_Spark/output/wqpred
```

```
total 28
drwxrwxr-x 3 bigdata bigdata 4096 jun 27 10:02 ..
-rw-r--r-- 1 bigdata bigdata  84 jun 27 10:02 .part-00000-c03081c7-0210-4e55-
a37e-a8765e01f3c9.csv.crc
-rw-r--r-- 1 bigdata bigdata 9294 jun 27 10:02 part-00000-c03081c7-0210-4e55-
a37e-a8765e01f3c9.csv
-rw-r--r-- 1 bigdata bigdata    0 jun 27 10:02 _SUCCESS
drwxrwxr-x 2 bigdata bigdata 4096 jun 27 10:02 .
-rw-r--r-- 1 bigdata bigdata    8 jun 27 10:02 _SUCCESS.crc
```

Verás que hay un fichero CSV. El nombre variará de una ejecución a otra, pero comenzará por algo parecido a **part-00000**.

Puedes imprimir las primeras líneas del fichero usando el comando **head**:

```
$ head ./Curso/U13_Spark/output/wqpred/part-00000-c03081c7-0210-4e55-
a37e-a8765e01f3c9.csv
```

```
prediction;quality
6.607870531947825;6
6.484864302160463;6
6.661953283967504;5
5.919234317486635;5
6.475045811874275;6
6.130525903362955;5
6.250726855743103;6
6.432166686508001;7
6.586077684780939;6
```


¿QUÉ HAS APRENDIDO?

En esta unidad te hemos presentado la plataforma de computación distribuida Spark. Se trata de un sistema especialmente diseñado para facilitar al desarrollador y al científico de datos las tareas de manejo, transformación y análisis de grandes volúmenes de información de forma interactiva, mediante unas librerías muy fáciles de utilizar.

Has visto cuáles son los conceptos fundamentales sobre los que se basa Spark: el uso de estructuras de datos en memoria RDDs y la representación de los flujos o procesos de cálculo mediante grafos DAG.

También conoces ya las principales piezas de la plataforma: los DataFrames y la interfaz SQL, la librería de algoritmos de *machine learning* MLlib, o las librerías para procesar flujos de datos al vuelo (Spark Streaming) y trabajar con grandes grafos (Spark GraphX).

Hemos introducido con varios ejemplos las principales operaciones y funcionalidades para trabajar con datos en Spark, usando DataFrames directamente o bien mediante su interfaz SQL.

Finalmente, has aprendido las bases para empezar a construir y probar tus primeros modelos de aprendizaje estadístico usando MLlib, con ejemplos de regresión y de clasificación.

Hasta aquí hemos llegado con la parte correspondiente a las plataformas de computación Big Data, que ha incluido Hadoop y Spark. En la próxima unidad veremos un tipo especial de bases de datos especialmente adaptadas para los problemas de tipo Big Data: las bases de datos NoSQL. Te esperamos allí...

AUTOCOMPROBACIÓN

1. ¿Qué es Apache Spark?

- a) Es una plataforma de computación distribuida de código abierto, diseñada para tareas de análisis de datos y Machine Learning de forma eficiente, mediante el manejo de datos masivos en memoria.
- b) Es una plataforma de computación distribuida escrita en Java, diseñada para tareas de análisis de datos y Machine Learning de forma eficiente, mediante el manejo interactivo de datos masivos.
- c) Es una plataforma de computación distribuida de código abierto, diseñada para tareas de análisis de datos y Machine Learning de forma eficiente, mediante una extensión del modelo MapReduce.
- d) Es una plataforma de computación distribuida de código abierto, derivada de Hadoop para realizar tareas de análisis de datos interactivos y en tiempo real.

2. ¿Cuáles son los principales componentes de Apache Spark?

- a) Spark Core junto con Spark SQL, MLlib, Streaming y GraphX.
- b) Spark Core.
- c) Spark Core, YARN y las extensiones del nucleo.
- d) RDDs, DAGs, Scheduler y Drivers.

3. ¿Qué dos piezas clave de Spark Core diferencian a Apache Spark de otras plataformas como Hadoop?

- a) Los RDDs para manejar datos distribuidos en memoria y los DAGs para representar flujos de procesos de datos complejos.
- b) Los DAGs para manejar datos distribuidos en memoria y los RDDs para representar flujos de procesos de datos complejos.
- c) El planificador y el distribuidor de tareas en memoria sobre los nodos del cluster.
- d) El Scheduler y los Drivers para gestionar y coordinar la ejecución de procesos distribuidos en el cluster.

4. Los RDDs.

- a) Están diseñados para mantener la información siempre en memoria, una vez cargada desde disco.
- b) Están diseñados para mantener la mayor cantidad de datos en memoria durante el mayor tiempo posible.
- c) Están diseñados para transferir la información de un nodo a otro con más memoria libre si no hay recursos suficientes.
- d) Están diseñados para transferir datos entre memoria y disco automáticamente tras cada operación.

5. ¿Cómo contribuyen los RDDs y los DAGs a la tolerancia ante fallos en Spark?

- a) Spark puede recalcular un RDD a partir del grafo DAG y los RDDs con los pasos de cálculo y datos previos.
- b) Spark puede utilizar el grafo DAG para recuperar copias del RDD fallido de otros nodos.
- c) Spark utiliza el grafo DAG para volver a ejecutar todos los cálculos y regenerar todos los RDDs.
- d) Spark aprovecha que los RDDs inmutables mantienen copias en disco para recuperar los datos si un RDD falla en cualquier punto del grafo DAG.

6. ¿Qué es una transformación amplia en Spark?

- a) Una operación que para cada valor de entrada genera múltiples valores de salida.
- b) Una operación que para múltiples valores de entrada genera múltiples valores de salida.
- c) Una operación que puede necesitar datos de varios nodos del cluster para calcular la porción del resultado en un nodo.
- d) Una operación que a partir de la porción de datos de un nodo genera porciones del resultado en múltiples nodos.

7. ¿Qué circunstancia fuerza la ejecución de los cálculos en Spark?

- a) Todos los cálculos se evalúan de forma perezosa hasta que Spark determina que es necesario completarlos.
- b) Los cálculos se ejecutan cuando ya están disponibles todos los resultados de los pasos previos.
- c) La ejecución de una acción que no genere ningún tipo de valor, como grabar los datos en fichero.
- d) La ejecución de una acción que no genere un nuevo RDD resultado, como imprimir el número de resultados o grabarlos en fichero.

8. ¿Qué ventajas ofrece el uso de DataFrames en lugar de los RDDs en Spark?

- a) Poder utilizar las operaciones que ya conocemos de R o Pandas para DataFrames sobre los RDDs.
- b) Poder trabajar directamente con datos de DataFrames en R o Pandas.
- c) Trabajar más fácilmente con una estructura tabular de datos usando operaciones de más alto nivel que las de los RDDs, sin perder las propiedades de los RDDs.
- d) Las operaciones sobre DataFrames están optimizadas para ejecutarse mucho más rápido que sobre RDDs.

9. ¿Qué es más eficiente, trabajar directamente con DataFrames o usar Spark SQL para operar con ellos?

- a) Trabajar directamente con DataFrames.
- b) Usar Spark SQL para operar con los DataFrames.
- c) Las dos opciones son equivalentes a nivel de eficiencia o tiempo de cálculo.
- d) Depende de si tenemos una vista SQL registrada sobre el DataFrame o no.

10. ¿Apache Spark puede trabajar con datos de un cluster Hadoop?

- a) Sí, Spark puede leer y escribir ficheros de HDFS, si los servicios de HDFS y el cluster están accesibles desde las máquinas donde corre Spark.
- b) Sí, Spark puede leer y escribir ficheros de HDFS, siempre que estén en formato CSV.
- c) Sí, solo es necesario anteponer el prefijo 'hdfs://' a cualquier fichero para leerlo de HDFS.
- d) Sí, pero es necesario instalar y configurar una librería especial para trabajar con HDFS.

SOLUCIONARIO

1.	a	2.	a	3.	a	4.	b	5.	a
6.	c	7.	d	8.	c	9.	c	10.	a

BIBLIOGRAFÍA

- Página principal del proyecto:
<https://spark.apache.org/>
- Página principal de la documentación oficial de Apache Spark.
<https://spark.apache.org/docs/latest/>
- Documentación de la API de Spark para Python.
<https://spark.apache.org/docs/latest/api/python/index.html>
- Databricks. Apache Spark Reference Guide.
<https://docs.databricks.com/spark/latest/index.html>
- "Spark: The Definitive Guide". Bill Chambers, Matei Zaharia. O'Reilly Media, 2018.
- "Advanced Analytics with Spark: Patterns for Learning from Data at Scale". Sandy Ryza, Uri Laserson, Sean Owen, Josh Wills. O'Reilly Media, 2nd Edition, 2018.

