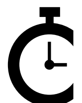
Análisis de datos masivos

Técnicas fundamentales

PID_00250690

Francesc Julbe

Tiempo mínimo de dedicación recomendado: 5 horas



Universitat Oberta de Catalunya

Índice

Introducción 5

Objetivos 6

1. Procesamiento de datos masivos 7
1.1. La problemática del procesado secuencial y el volumen de datos 7
1.2. ¿Qué es un algoritmo? 8
1.2.1. Algoritmos secuenciales 10
1.2.2. Algoritmos paralelos 11
1.2.3. Algoritmos serie paralelos 12
1.2.4. Otros algoritmos 13
1.3. Eficiencia en la implementación de algoritmos 14
1.3.1. Notación *Big O* 14
1.3.2. Computación paralela 16
1.3.3. Ejemplos de algoritmos y paralelización 19

2. Apache Hadoop y MapReduce 22
2.1. Paradigma MapReduce 22
2.1.1. Abstracción 22
2.1.2. Implementación MapReduce en Hadoop 23
2.1.3. Limitaciones de Hadoop 25
2.2. Ejemplo de aplicación de MapReduce 26
2.3. Apache Mahout 28
2.4. Conclusiones 29

3. Apache Spark 30
3.1. Resilient distributed dataset (RDD) 30
3.2. Modelo de ejecución Spark 33
3.2.1. Funcionamiento 34
3.3. Apache Spark MLlib 35
3.3.1. ML Pipelines 35

4. TensorFlow 37
4.1. Redes neuronales y *deep learning* 37
4.2. TensorFlow 40

5. Aprendizaje autónomo 41
5.1. Clasificación 42
5.2. Regresión 44
5.3. Agrupamiento 45

5.4. Reducción de dimensionalidad 47
5.5. Filtrado colaborativo/sistemas de recomendación............. 47

Resumen 49

Glosario 50

Bibliografía 51

Introducción

Las herramientas y tecnologías para el análisis de datos masivos (*big data*) se encuentran en un momento de alta ebullición. Continuamente aparecen y desaparecen herramientas para dar apoyo a nuevas necesidades, a la vez que las existentes van incorporando nuevas funcionalidades. Por eso, el objetivo principal de este módulo no es enumerar las tecnologías y herramientas disponibles, sino proporcionar las bases teóricas necesarias para entender este complejo escenario y ser capaz de analizar de forma autónoma las distintas alternativas existentes.

Aun así, sí que veremos con cierto grado de detalle la implementación, el funcionamiento y las características de algunas de las herramientas más importantes y estables dentro de este complejo escenario, como pueden ser las principales herramientas del ecosistema de Apache Hadoop o las librerías de Apache Spark.

Iniciaremos este módulo revisando los conceptos básicos de complejidad relacionados con la algoritmia. Veremos cómo se caracterizan los algoritmos secuenciales, paralelos y los serie paralelos. También repasaremos la notación *BigO* (*BigO notation*), que nos permite caracterizar la escalabilidad de los algoritmos frente al volumen de datos de entrada.

A continuación, veremos en detalle uno de los principales modelos de programación en entornos de datos masivos, el modelo MapReduce, que se implementa en los sistemas Apache Hadoop. Veremos sus características básicas, así como sus principales ventajas e inconvenientes.

En el apartado siguiente veremos la aproximación al análisis de datos utilizada en el otro gran entorno de trabajo (*framework*) de datos masivos, Apache Spark. Veremos sus principales características, su funcionamiento y sus diferencias frente al modelo MapReduce.

Finalizaremos con una revisión de los distintos métodos de aprendizaje automático, haciendo énfasis en los algoritmos que son soportados por las herramientas de datos masivos, y concretamente haremos referencia a las implementaciones que incorporan los entornos Apache Hadoop y Apache Spark.

Objetivos

En los materiales didácticos de este módulo encontraremos las herramientas indispensables para asimilar los objetivos siguientes:

1. Comprender cuáles son los diferentes modelos de procesamiento distribuido utilizados en datos masivos y en qué escenario es útil cada uno de ellos.
2. Descubrir los principales entornos de trabajo existentes para el procesado distribuido, como son Apache Hadoop (y su ecosistema) y Apache Spark.
3. Conocer las diversas herramientas y servicios existentes para el procesado distribuido de datos.

1. Procesamiento de datos masivos

En este apartado discutiremos la problemática relacionada con el procesamiento de grandes volúmenes de datos, y las posibles soluciones que desde un punto de vista algorítmico se pueden desarrollar.

1.1. La problemática del procesado secuencial y el volumen de datos

Las aproximaciones habituales a la computación y a la realización de cálculos complejos en modo secuencial (no paralelizables), o lo que en lenguaje de computación se traduce en el uso de un solo procesador, está convirtiéndose en un mecanismo arcaico para la resolución de problemas complejos por varias razones:

- El volumen de datos que se deben analizar crece a un ritmo muy elevado en la que llamamos la «era del *big data*». Así, los problemas que tradicionalmente podían solucionarse con un código secuencial bien implementado y datos almacenados localmente en un equipo, ahora son computacionalmente muy costosos e ineficientes, lo que se traduce en problemas irresolubles.

- Actualmente los equipos poseen procesadores multinúcleo capaces de realizar cálculos en paralelo. Deberíamos ser capaces de aprovechar su potencial mejorando las herramientas de procesado para que puedan utilizar toda la capacidad de cálculo disponible.

- Además, la capacidad de cálculo puede ser incrementada más allá de la que proveen los equipos individuales. Las nuevas tecnologías permiten combinar redes de computadores, lo que se conoce como escalabilidad horizontal. La escalabilidad horizontal incrementa la capacidad de cálculo de forma exponencial, así como la capacidad de almacenaje, aumentado varios órdenes de magnitud la capacidad de cálculo del sistema para resolver problemas que con una aproximación secuencial serían absolutamente irresolubles, tanto por consumo de recursos disponibles como por consumo de tiempo.

Escalabilidad horizontal

Un sistema escala horizontalmente si al agregar más nodos al mismo el rendimiento de este mejora.

Procesadores multinúcleo

Los procesadores multinúcleo son aquellos procesadores que combinan uno o más microprocesadores en una sola unidad integrada o paquete.

Sin embargo, aun teniendo en cuenta que los recursos disponibles para la resolución de problemas complejos son mucho mejores y eficientes, este factor no soluciona por sí solo el problema. Hay una gran cantidad de factores que tener

en cuenta para que un algoritmo pueda ser óptimo una vez se esté ejecutando en un entorno paralelo.

Aprovechar la mejora en los recursos de computación disponibles para ejecutar algoritmos de forma paralela requiere de buenas implementaciones de dichos algoritmos. Hay que tener en cuenta que si un programa diseñado para ser ejecutado en un entorno monoprocesador no se ejecuta de forma rápida en dicho entorno, será todavía más lento en un entorno multiprocesador. Y es que una aplicación que no ha sido diseñada para ser ejecutada de forma paralela no podrá paralelizar su ejecución aunque el entorno se lo permita (por ejemplo, en un entorno con múltiples procesadores).

Así, una infraestructura que va a ejecutar un algoritmo de forma paralela y distribuida debe tener en cuenta el número de procesadores que se están utilizando, así como el uso que hacen dichos procesadores de la memoria disponible, la velocidad de la red de ordenadores que forman el clúster, etc. Desde un punto de vista de implementación, un algoritmo puede mostrar una dependencia regular o irregular entre sus variables, recursividad, sincronismo, etc. En cualquier caso, es posible acelerar la ejecución del algoritmo siempre que algunas subtareas puedan ejecutarse de forma simultánea dentro del flujo de trabajo (*workflow*) completo de procesos que componen el algoritmo. Y es que cuando pensamos en el desarrollo de un algoritmo tendemos a entender la secuencia de tareas de un modo secuencial, dado que en la mayoría de ocasiones un algoritmo es una secuencia de tareas, las cuales dependen de forma secuencial unas de otras, lo que queda traducido en un código no paralelo cuando dicha aproximación se traslada a la implementación en software.

Clúster

En computación, un clúster es un conjunto de ordenadores conectados entre si formando una red de computación distribuida.

En este apartado vamos a explorar los diferentes tipos de algoritmos, algunos de ellos muy utilizados en el mundo de los datos masivos, y también vamos a entender la naturaleza del paralelismo para cada uno de ellos.

1.2. ¿Qué es un algoritmo?

Aunque pueda parecer una pregunta obvia, conviene concretar qué es un algoritmo para poder entender cuáles son sus diferentes modalidades y qué soluciones existen para poder acelerar su ejecución en entornos de datos masivos, esto es, para poder ejecutarlos de forma paralela y distribuida.

> Un algoritmo es la secuencia de procesos que deben realizarse para resolver un problema con un número finito de pasos. Algunos de estos procesos pueden tener relaciones entre ellos y otros pueden ser totalmente independientes. La comprensión de este hecho es crucial para entender cómo podemos paralelizar nuestro algoritmo para, posteriormente, implementarlo y ejecutarlo sobre un gran volumen de datos.

Así, los componentes de un algoritmo son:

1) tareas,

2) dependencias entre las tareas (las salidas (*outputs*) de una tarea son las entradas (*inputs*) de la siguiente),

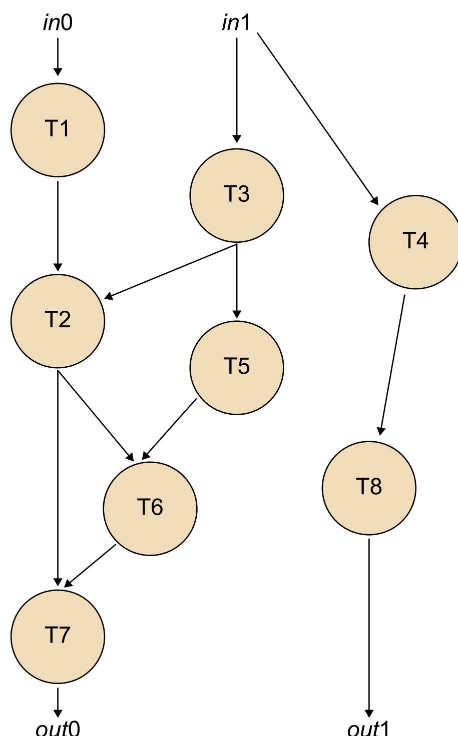
3) entradas necesarias para el algoritmo,

4) resultado o salidas que generará el algoritmo.

Estas dependencias pueden describirse mediante un grafo dirigido (en inglés, *Direct Graph*, DG) donde el término *dirigido* indica que hay un orden en la secuencia de dependencias y, por lo tanto, que es necesaria la salida de cierta tarea para iniciar otras tareas dependientes.

Un DG es un grafo en el cual los nodos representan tareas y los arcos (o aristas) son las dependencias entre tareas, tal como se han definido anteriormente. La figura 1 muestra un ejemplo de un algoritmo representado por su DG.

Figura 1. Representación DG de un algoritmo con dos entradas *in0* e *in1*, dos salidas *out0* y *out1*, diversas tareas *Ti* y sus dependencias



Un algoritmo también puede representarse mediante la llamada **matriz de adyacencia**, muy utilizada en teoría de grafos.

La matriz de adyacencia $M_{N \times N}$ es una matriz de $N \times N$ elementos, donde N es el número de tareas. Tal y como se muestra a continuación, el valor 1 indica si

existe una dependencia entre las tareas correspondientes (podrían ser valores distintos de 1, lo que otorgaría pesos diferentes a dichas dependencias).

$$M_{N\times N} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

En esta matriz podemos identificar las entradas fácilmente, ya que sus columnas contienen todos los valores iguales a 0. De forma similar, las salidas se pueden identificar a partir de las filas asociadas, que presentan todos los valores iguales a 0. En resto de nodos intermedios presentan valores distintos de 0 en sus filas y columnas.

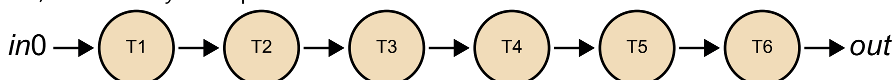
A partir del tipo de dependencia entre los procesos, podemos clasificar los algoritmos en secuenciales, paralelos o serie paralelos.

1.2.1. Algoritmos secuenciales

Es el algoritmo más simple y puede entenderse como aquel en el cual cada tarea debe finalizar antes de empezar la siguiente, que depende del resultado de la anterior.

Un ejemplo sería un cálculo en serie, en la cual cada elemento es el anterior más una cantidad dada. El cálculo de la serie de Fibonacci, donde cada elemento es la suma de los dos anteriores, puede ejemplificar este caso. La figura 2 muestra el DG de un algoritmo secuencial.

Figura 2. Representación DG de un algoritmo secuencial con una entrada *in0*, una salida *out0*, las tareas *Ti* y sus dependencias



Un ejemplo de implementación de la serie de Fibonacci en lenguaje de programación Java se incluye a continuación:

```
public static void main(String[] args) {
  int limit = Integer.parseInt(args[0]);
  int fibSum = 0;
  int counter = 0;
  int[] serie  = new int[limit];

  while(counter < limit) {
    serie = getMeSum(counter, serie);
    counter++;
}

public static int[] getMeSum(int index, int[] serie) {
  if(index==0) {
    serie[0] = 0;
  } else if(index==1) {
    serie[1] = 1;
  } else {
    serie[index]  = serie[index-1] + serie[index-2];
  }
  return serie;
}
```

Solo imponiendo la condición para los dos primeros números pueden calcularse todos los valores de la serie hasta un límite especificado. En este caso se resuelve de forma elegante un algoritmo netamente secuencial con el uso de patrones recursivos, muy habituales en el desarrollo de algoritmos secuenciales.

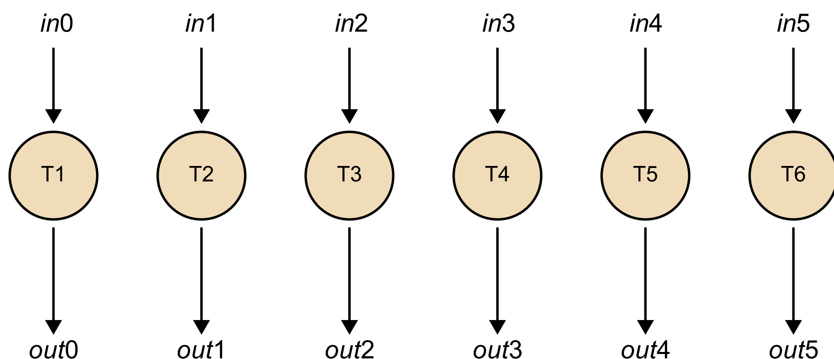
1.2.2. Algoritmos paralelos

A diferencia de los anteriores, en los algoritmos paralelos las tareas de procesado son todas completamente independientes las unas de las otras, y en ningún momento muestran una correlación o dependencia.

A modo de ejemplo, un algoritmo puramente paralelo sería aquel que realiza un procesamiento para datos segmentados sin realizar ningún tipo de agregación al finalizar los procesos, tal como la suma de elementos de un tipo A y la suma de elementos de un tipo B, dando como resultado las dos sumas.

En la figura 3 vemos cómo cada tarea, que podría ser la suma de elementos segmentados $\{A,B,\ldots,F\}$, se realiza de forma independiente de las otras tareas.

Figura 3. Representación DG de un algoritmo paralelo con una entrada *in0*, una salida *out0* y diversas tareas *Ti*

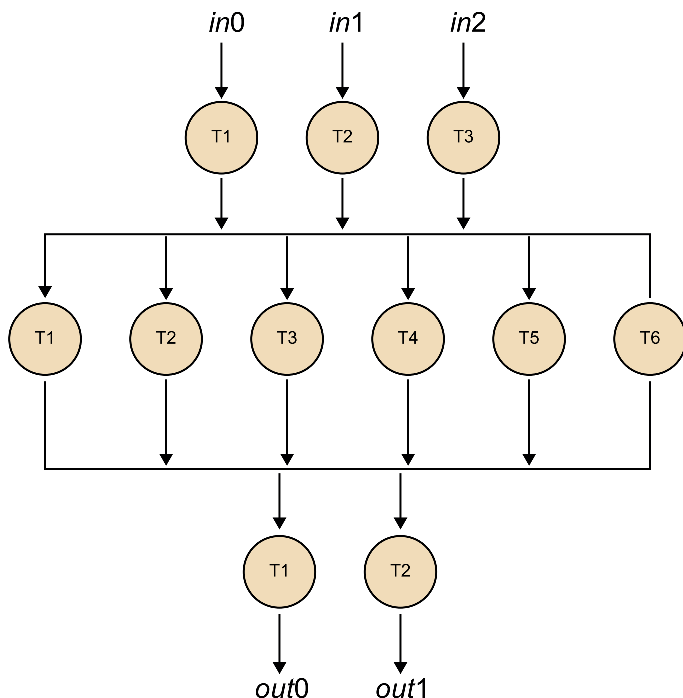


1.2.3. Algoritmos serie paralelos

A partir de los dos tipos de algoritmos ya descritos, podemos introducir un tercer tipo, que se compone de tareas ejecutadas de forma paralela y tareas ejecutadas de forma secuencial.

Así, ciertas tareas se descomponen en subtareas independientes entre sí y, tras su finalización, se agrupan los resultados de forma síncrona. Si es necesario, se vuelve a dividir la ejecución en nuevas subtareas de ejecución en paralelo, tal y como muestra la figura 4.

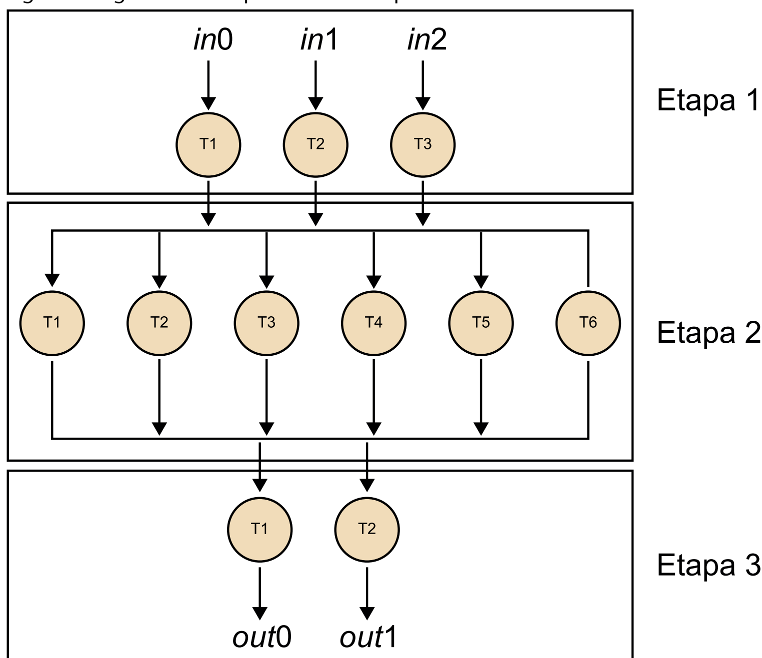
Figura 4. Algoritmo con conjuntos de diferentes tareas ejecutadas en paralelo en diferentes ciclos secuenciales



Podemos añadir un nivel de abstracción a este tipo de algoritmo, en el que varias tareas que se ejecutan en paralelo componen una etapa (*stage*) cuyo

resultado es la entrada para otro conjunto de tareas que se ejecutarán en paralelo, formando otra etapa. Así, las tareas individuales se ejecutan de forma paralela, pero las etapas se ejecutan de forma secuencial, lo que acaba generando un llamado *pipeline*. La figura 5 muestra un posible esquema de esta estructura.

Figura 5. Algoritmo serie paralelo con etapas secuenciales



Más adelante en el módulo volveremos a analizar este tipo de algoritmos, ya que es la metodología de procesado de Apache Spark.

Un ejemplo de dichos algoritmos seria el paradigma MapReduce. Las tareas Map son independientes entre sí y su resultado se agrupa en un conjunto (típicamente menor) de tareas Reduce, siguiendo la metodología «divide y vencerás» (*divide & conquer*).

1.2.4. Otros algoritmos

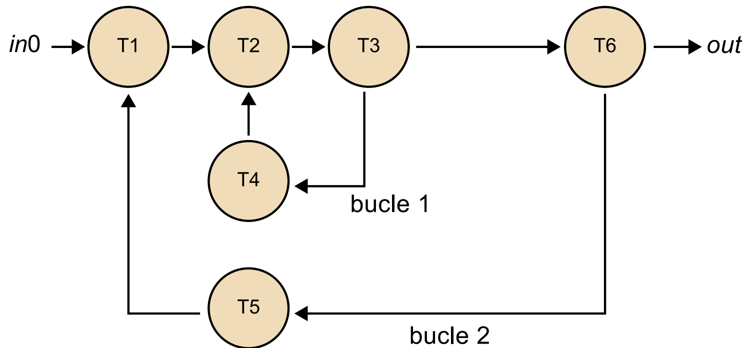
Hay algoritmos que no pueden ser clasificados en ninguna de las categorías anteriores, ya que el DG de dichos algoritmos no sigue ningún patrón. En este caso, podemos distinguir dos tipos de algoritmos:

- DAG (*directed acyclic graph*): en el DAG no hay ciclos y el procesado muestra una «dirección»; sin embargo, no hay un sincronismo claro entre tareas. La figura 1 es un ejemplo de DAG. Se trata de un tipo de algoritmo importante por ser el grafo que representa la metodología de ejecución de tareas de Apache Spark, que se estudiará más adelante.

- DCG (*directed cyclic graph*): son aquellos algoritmos que contienen ciclos en su procesado. Es habitual encontrar implementaciones en sistemas de

procesado de señal, en el campo de las telecomunicaciones o de la compresión de datos, donde hay etapas de predicción y corrección. La figura 6 muestra un ejemplo de esta estructura.

Figura 6. Representación de un algoritmo con dependencias cíclicas



1.3. Eficiencia en la implementación de algoritmos

El uso de entornos de computación distribuidos requiere de una implementación en software adecuada de los algoritmos para que pueda ser ejecutada de forma eficiente. Ejecutar un algoritmo en un entorno distribuido requiere de un análisis previo del diseño de dicho algoritmo para comprender el entorno en el que se va a ejecutar.

Las tareas que componen dicho algoritmo deben descomponerse en subtareas más pequeñas para poder ejecutarse en paralelo y, en muchas ocasiones, la ejecución de dichas subtareas debe ser síncrona, permitiendo la agrupación de los resultados finales de cada tarea.

Por tanto, paralelizar algoritmos está fuertemente relacionado con una arquitectura que pueda ejecutarlos de forma eficiente. No es posible paralelizar la ejecución de un algoritmo secuencial porque el procesador no va a saber qué tareas puede distribuir entre sus núcleos.

1.3.1. Notación *Big O*

La notación *Big O* permite cuantificar la complejidad de un algoritmo a partir del volumen de datos de entrada, que generalmente se identifica con la letra n. Veamos algunos de los casos más relevantes:

- $O(1)$: es el caso en el que un algoritmo requiere un tiempo constante para ejecutarse, independientemente del volumen de datos por tratar. Suele ser bastante inusual. Un ejemplo seria acceder a un elemento de un vector.

- $O(n)$: en este caso, la complejidad del algoritmo se incrementa de forma lineal con el volumen de la entrada de datos. Es decir, el tiempo de cálculo

requerido aumenta de forma lineal respecto a la cantidad de datos que debe tratar. Por ejemplo, escoger el valor máximo de un vector de valores tendría esta complejidad, ya que se debe recorrer una vez todo el vector para poder determinar cuál es el valor máximo. Generalmente, cálculos que impliquen un bucle `for` que itere sobre los n elementos de entrada suele tener esta complejidad.

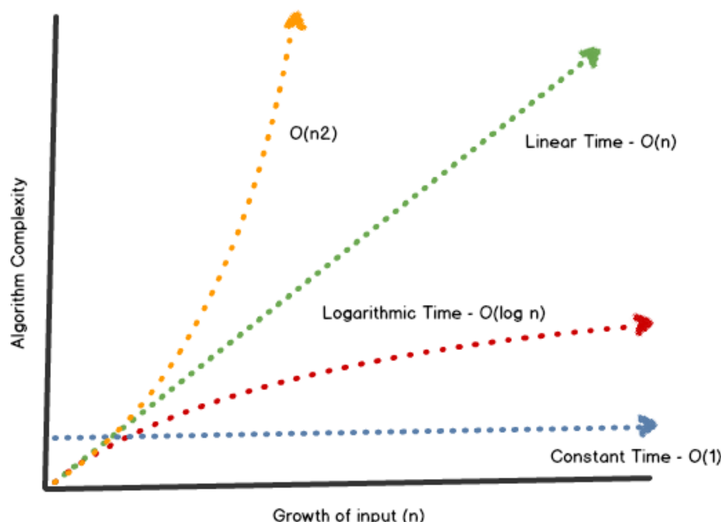
- $O(log(n))$: también conocida como «complejidad logarítmica». Este tipo de algoritmos son aquellos en los que el tiempo no se incrementa de forma lineal, sino que dicho incremento es menor a medida que aumenta el conjunto de datos. Generalmente, este tipo de complejidad se encuentra en algoritmos de búsqueda en árboles binarios, donde no se analizan todos los datos del árbol, solo los contenidos en ciertas ramas. De esta forma, la profundidad del árbol que deberemos analizar no crece de forma proporcional al volumen de la entradas.

- $O(n^2)$: este caso es conocido como «complejidad cuadrática». Indica que el tiempo de cálculo requerido crecerá de forma exponencial a partir del valor de n, es decir, del volumen de datos de entrada. Generalmente, cálculos que impliquen un doble bucle `for` que itere sobre los n elementos de entrada suele tener esta complejidad.

  Los algoritmos con este tipo de complejidad no suelen ser aplicables en datos masivos, ya que en estos casos el valor de n suele ser muy grande. Por lo tanto, no es computacionalmente posible resolver problemas con datos masivos empleando algoritmos de orden $O(n^2)$ o superior, como por ejemplo «cúbicos» $O(n^3)$.

La figura 7 muestra, de forma aproximada, cómo crece el tiempo de cálculo necesario según el volumen de datos de la entrada.

Figura 7. Representación de la complejidad de los algoritmos según la notación O grande



Fuente: https://dennis-xlc.gitbooks.io

Un algoritmo puede paralelizarse y de este modo mejorar su tiempo de ejecución, aprovechar mejor los recursos disponibles, etc. Sin embargo, la eficiencia en la ejecución de un algoritmo también depende en gran medida de su implementación. Un ejemplo claro y fácil de entender son los diferentes mecanismos para ordenar elementos en una colección de elementos desordenados.

Ejemplo: ordenación de valores de una colección

La implementación más habitual para ordenar un vector de *n* elementos de mayor a menor consiste en iterar dos veces el mismo vector, utilizando una copia del mismo, comparando cada valor con el siguiente. Si es mayor intercambian sus posiciones, si no se dejan tal y como están. Este es un algoritmo llamado «ordenamiento de burbuja» (*bubble sort*)*. Pese a que su implementación es muy sencilla, también es muy ineficiente, ya que debe recorrer el vector *n* veces. En el caso óptimo (ya ordenado), su complejidad es $O(n)$, pero en el peor de los casos es $O(n^2)$. El ordenamiento de burbuja, además, es más difícil de paralelizar debido a su implementación secuencial, ya que recorre el vector de forma ordenada para cada elemento, siendo doblemente ineficiente.

* http://bit.ly/1bfggqp

Hay otros mecanismos para ordenar vectores mucho más eficientes, como el *Quicksort***, probablemente uno de los más eficientes y rápidos. Este mecanismo se basa en el divide y vencerás. El mecanismo del algoritmo no es trivial y su explicación está fuera de los objetivos de este módulo; aun así, es interesante indicar que este algoritmo ofrece una complejidad del orden de $O(nlog(n))$. Además, al aproximar el problema dividiendo el vector en pequeños problemas es más fácilmente paralelizable.

** http://bit.ly/2cYSPFa

Otro algoritmo de ordenamiento muy popular es el «ordenamiento por mezcla» (*merge sort*)***, que divide el vector en dos partes de forma iterativa, hasta que solo quedan vectores de dos elementos, fácilmente ordenables. Este problema es también fácilmente paralelizable debido a que las operaciones en cada vector acaban siendo independientes entre ellas. Estos ejemplos de ordenación sirven para entender que un mismo algoritmo puede implementarse de formas distintas y, además, el mecanismo adecuado puede permitir la paralelización, lo que mejora su eficiencia final.

*** http://bit.ly/2tzAl4u

1.3.2. Computación paralela

En un entorno de computación paralela existen diversos factores relevantes que configuran su capacidad de paralelización.

Entornos multiprocesador

A nivel de un solo computador, algunos de los factores más importantes son:

1) Número de procesadores y **número de núcleos** por procesador.

2) Comunicación entre procesadores: buses de comunicaciones en origen, actualmente mediante el sistema de comunicaciones Network On-Chip (NoC).

Network-On-Chip (NoC)

Network-On-Chip (NoC) es un subsistema de comunicaciones en un circuito integrado.

3) Memoria: las aplicaciones pueden particionarse de modo que varios procesos compartan memoria (*threads* o hilos), o de modo que cada proceso gestione su memoria.

Los sistemas de memoria compartida ofrecen la posibilidad de que todos los procesadores compartan la memoria disponible. Por lo tanto, un cambio en la memoria es visible para todos ellos, compartiendo el mismo espacio de trabajo (por ejemplo, variables).

Existen arquitecturas de memoria distribuida entre procesadores, en la cual cada uno de ellos tiene su espacio de memoria reservado y exclusivo. Estas arquitecturas son escalables, pero más complejas desde un punto de vista de gestión de memoria.

4) Acceso a datos: cada proceso puede acceder a una partición de los datos o, por el contrario, puede haber concurrencia en el acceso a datos, lo que implica una capacidad de sincronización del algoritmo, incluyendo mecanismos de integridad de datos.

Entornos multinodo

En redes de múltiples computadores, algunos de los factores más importantes son los siguientes:

1) En un entorno de **múltiples computadores** es necesario contar con comunicaciones de altas prestaciones entre ellos, mediante redes de alta velocidad, como por ejemplo Gigabit Ethernet.

2) Desde el punto de vista de la arquitectura de memoria, es una arquitectura híbrida. Un computador o nodo con múltiples procesadores es una máquina de memoria compartida, mientras que cada nodo tiene la memoria exclusiva y no compartida con el resto de nodos, siendo una arquitectura de memoria distribuida. Actualmente, los nuevos entornos de datos masivos simulan las arquitecturas como sistemas de memoria compartida mediante software.

Unidad de procesamiento gráfico (GPU)

La unidad de procesamiento gráfico* (*graphics processor unit*, GPU) es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante para aligerar la carga de trabajo del procesador central. Típicamente las GPU están presentes en las tarjetas gráficas de los ordenadores y su dedicación es exclusiva a tareas altamente paralelizables de procesamiento gráfico, incrementando de forma exponencial la capacidad de procesado del sistema.

*** http://bit.ly/21jWu1e**

Aunque inicialmente las GPU se utilizaron para procesar gráficos (principalmente vértices y píxeles), actualmente su capacidad de cálculo se utiliza en otras aplicaciones en lo que se ha llamado *general-purpose computing on graphics processing units*** (GPGPU). Son especialmente relevantes en aplicaciones del ámbito de redes neuronales *deep learning* y, en general, en entornos de altas prestaciones (*high performance computing*, HPC), de las que hablaremos con más detalle posteriormente.

**** http://bit.ly/2orBoSH**

En un entorno con GPU, el sistema traslada las partes de la aplicación con mayor carga computacional y más altamente paralelizables a la GPU, dejando

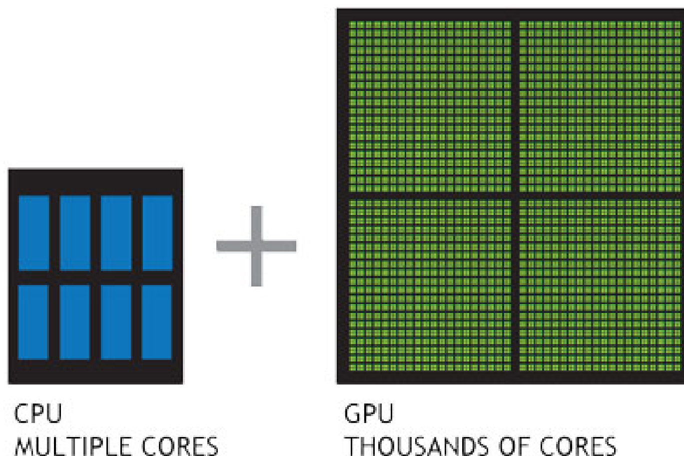
el resto del código ejecutándose en la CPU. Debe tenerse en cuenta que una CPU puede contar con varios núcleos optimizados (cuatro u ocho simulados con tecnología *hyperthreading*), mientras que una GPU cuenta con miles de núcleos diseñados para ejecutar tareas altamente paralelizables. Debido a las diferencias fundamentales entre las arquitecturas de la GPU y la CPU, no cualquier problema se puede beneficiar de una implementación mediante GPU.

Hyperthreading

Hyperthreading es una tecnología de Intel que permite ejecutar programas en paralelo en un solo procesador, simulando el efecto de tener realmente dos núcleos en un solo procesador.

Pese a que cualquier algoritmo que sea implementable en una CPU puede serlo también en una GPU, ambas implementaciones pueden no ser igual de eficientes en las dos arquitecturas. En este sentido, los algoritmos con un altísimo grado de paralelismo, llamados *embarrassingly parallel* (embarazosamente paralelizables en su traducción literal), que no tienen la necesidad de estructuras de datos complejas y con un elevado nivel de cálculo aritmético, son los que mayores beneficios obtienen de su implementación en GPU.

Figura 8. Número de núcleos en un procesador de propósito general (COU) frente a una GPU



Fuente: Nvidia

Inicialmente la programación de GPU se hacía con lenguajes de bajo nivel, como el lenguaje ensamblador o *assembler* u otros más específicos para el procesado gráfico. Sin embargo, actualmente el lenguaje más utilizado es el compute unified device architecture (CUDA)*. CUDA no es un lenguaje en sí, sino un conjunto de extensiones a C y C++ que permite la paralelización de ciertas instrucciones para ser ejecutadas en la GPU del sistema.

* http://bit.ly/1JbXNdB

Lenguaje ensamblador

El lenguaje ensamblador es un lenguaje de programación llamado de bajo nivel, formado por un conjunto de instrucciones básicas para programar microprocesadores y/o microcontroladores.

A continuación, se muestra un fragmento de código estándar en C:

```
void saxpy_serial(int n, float a, float *x, float *y)
{
        for (int i=0; i<n; ++i)
                y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

Seguidamente, se muestra cómo añadiendo algunos parámetros al código se paraleliza el problema presentado en el fragmento de código anterior:

```
__global__
void saxpy_parallel(int n, float a, float *x, float *y)
{
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if(i<n)
                y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n, 2.0, x, y);
```

En este ejemplo*, cada cálculo dentro del bucle no depende de ningún otro cálculo anterior, lo que lo convierte en un caso canónico de incremento de rendimiento por paralelismo en GPU. Sin embargo, un algoritmo que tuviera dependencias entre tareas no sería fácilmente resoluble con CUDA y GPU, a causa de las correlaciones entre tareas.

*** Podéis ver el ejemplo completo en http://bit.ly/2wUi5Zk.**

Como inconvenientes cabe indicar que las GPU tienen latencias altas, lo que presenta un problema en tareas con un corto tiempo de ejecución. Por el contrario, son muy eficientes en tareas de elevada carga de trabajo, con cálculos intensivos y con largo tiempo de ejecución.

1.3.3. Ejemplos de algoritmos y paralelización

A continuación, mostraremos algunos ejemplos de cómo un problema complejo puede particionarse para ser ejecutado en un entorno de computación paralela.

Ejemplo: multiplicación de matrices

Vamos a presentar un problema que iremos revisitando a lo largo del módulo: la multiplicación de matrices.

Dos matrices A y B se pueden multiplicar si el número de filas de la primera (A) es igual al número de columnas de la segunda (B). El primer elemento de la primera fila de la matriz A se multiplica por el primer elemento de la primera columna de B. Su resultado se suma a la multiplicación del segundo elemento de la primera fila de A por el segundo elemento de la primera columna de B, y así sucesivamente.

Supongamos que las matrices A y B son las presentadas a continuación:

$$A_{3\times 2} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix}, \ B_{2\times 3} = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix}$$

Entonces el producto *AB* se representa de la forma siguiente:

$$AB_{3\times 3} = \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} & a_{1,1}b_{1,3} + a_{1,2}b_{2,3} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} & a_{2,1}b_{1,3} + a_{2,2}b_{2,3} \\ a_{3,1}b_{1,1} + a_{3,2}b_{2,1} & a_{3,1}b_{1,2} + a_{3,2}b_{2,2} & a_{3,1}b_{1,3} + a_{3,2}b_{2,3} \end{bmatrix}$$

Y el producto *BA* tal y como mostramos a continuación. Nótese la diferencia en el tamaño de la matriz resultante:

$$BA_{2\times 2} = \begin{bmatrix} b_{1,1}a_{1,1} + b_{1,2}a_{2,1} + b_{1,3}a_{3,1} & b_{1,1}a_{1,2} + b_{1,2}a_{2,2} + b_{1,3}a_{3,2} \\ b_{2,1}a_{1,1} + b_{2,2}a_{2,1} + b_{2,3}a_{3,1} & b_{2,1}a_{1,2} + b_{2,2}a_{2,2} + b_{2,3}a_{3,2} \end{bmatrix}$$

Una implementación con código secuencial del producto de matrices podría ser el siguiente:

```
int[][] a; /* matriz A de NxN elementos */
int[][] b; /* matriz B de NxN elementos */
int[][] c; /* matriz C de NxN elementos */

for(int i=0; i<n; i++) {
  for(int j=0; j<n; j++) {
    for(int k=0; k<n; k++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

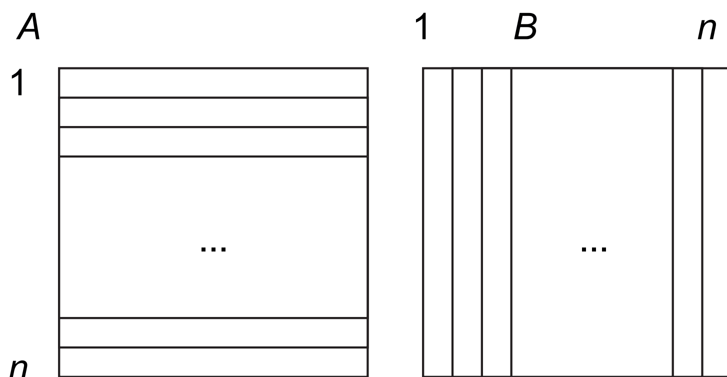
Tal y como se puede ver en el código anterior, al utilizar tres bucles anidados la implementación presenta una complejidad $O(n^3)$.

Sin embargo, el producto de cada par de elementos es independiente de las otras operaciones. Además, la suma y la multiplicación son operaciones conmutativas y asociativas, de modo que el algoritmo de multiplicación de matrices ofrece diferentes aproximaciones de paralelización.

Como regla principal es importante que la estrategia de paralelización no resulte en un problema de mayor complejidad que la implementación secuencial. Así, la complejidad no debe ser mayor que $O(n^3)$.

Una forma de resolver el problema es la división del producto en filas y columnas.

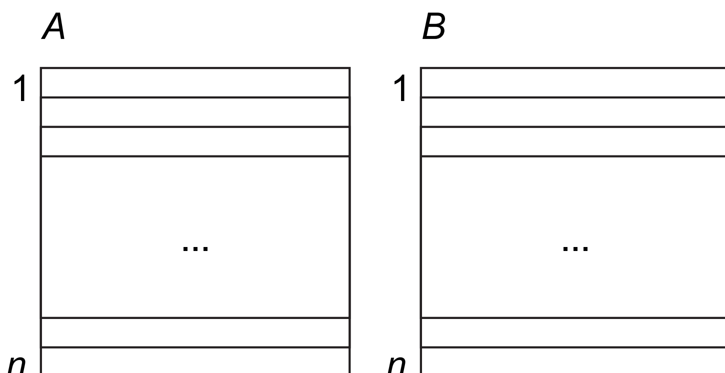
Figura 9. División de tareas por filas y columnas en el producto de matrices



En este caso, existen n^2 tareas, que son la multiplicación de los elementos de cada fila por los elementos de cada columna. Cada una de estas tareas es independiente de las demás, siendo un buen ejemplo de algoritmo paralelizable.

También podríamos distribuir la carga por fila en ambas matrices, tal y como se muestra en la figura 10.

Figura 10. Producto de matrices dividiendo las tareas por filas de ambas matrices



En este caso, se multiplica cada fila por el primer elemento de cada columna de la matriz B, lo que va generando resultados parciales, hasta que el algoritmo se ha completado. Es menos intuitivo que el caso anterior, pero igualmente paralelizable.

Algoritmos no paralelizables

Existen algoritmos que por su naturaleza secuencial no son fácilmente paralelizables. Sin embargo, se han desarrollado estrategias para paralelizar algoritmos que *a priori* tienen un carácter secuencial. El ejemplo ya presentado de la serie de Fibonacci es un claro candidato de algoritmo secuencial no fácilmente paralelizable, ya que existen dependencias con valores calculados anteriormente, de modo que la tarea i depende de la $(i-1)$. Sin embargo, existen implementaciones paralelas de muchos algoritmos, añadiendo etapas de sincronización y agregación de resultados.

2. Apache Hadoop y MapReduce

MapReduce es un modelo de programación para dar soporte a la computación paralela sobre grandes conjuntos de datos en clústeres de computadoras. MapReduce ha sido adoptado mundialmente, ya que existe una implementación de código abierto denominada Hadoop. Su desarrollo fue liderado inicialmente por Yahoo! y actualmente lo realiza el proyecto Apache.

2.1. Paradigma MapReduce

El nombre del entorno está inspirado en los nombres de dos importantes métodos o funciones en programación funcional: *map* y *reduce*.

2.1.1. Abstracción

Las dos tareas principales que realiza este modelo son:

- ***Map***: esta tarea es la encargada de «etiquetar» o «clasificar» los datos que se leen desde disco, típicamente de HDFS, en función del procesamiento que estemos realizando.
- ***Reduce***: esta tarea es la responsable de agregar los datos etiquetados por la tarea Map. Puede dividirse en dos etapas, la *shuffle* y el propio *reduce* o agregado.

Todo el intercambio de datos entre tareas utiliza estructuras llamadas parejas <clave, valor> (<*key*, *value*> en inglés) o tuplas.

En el ejemplo siguiente vamos a mostrar, desde un punto de vista funcional, en qué consiste una tarea MapReduce.

Ejemplo conceptual de proceso MapReduce

Imaginemos que tenemos tres ficheros con los datos siguientes:

Fichero 1:

```
Carlos, 31 años, Barcelona
María, 33 años, Madrid
Carmen, 26 años, Coruña
```

Fichero 2:

```
Juan, 12 años, Barcelona
Carmen, 35 años, Madrid
José, 42 años, Barcelona
```

Fichero 3:

```
María, 78 años, Sevilla
Juan, 50 años, Barcelona
Sergio, 33 años, Madrid
```

Dados estos ficheros, podríamos preguntarnos: ¿cuántas personas hay en cada ciudad?

Para responder a esa pregunta definiremos una tarea *map* que leerá las filas de cada fichero y «etiquetará» cada una en función de la ciudad que aparece, que es el tercer campo de cada una de las líneas que siguen la estructura: nombre, edad, ciudad.

Para cada línea, nos devolverá una tupla de la forma: <ciudad, cantidad>

Al final de la ejecución de la tarea *map*, en cada fichero tendremos:

- Fichero 1: (Barcelona, 1), (Madrid, 1), (Coruña, 1)
- Fichero 2: (Barcelona, 1), (Madrid, 1), (Barcelona, 1)
- Fichero 3: (Sevilla, 1), (Barcelona, 1), (Madrid, 1)

Como vemos, nuestras tuplas están formadas por una clave (*key*), que es el nombre de la ciudad, y un valor (*value*) que representa el número de veces que aparece en la línea, que siempre es 1.

La tarea *reduce* se ocupará de agrupar los resultados según el valor de la clave. Así, recorrerá todas las tuplas agregando los resultados por una misma clave y nos devolverá:

```
(Barcelona, 4)
(Sevilla,1)
(Madrid, 3)
(Coruña, 1)
```

Obviamente, en este ejemplo la operación no requería un entorno distribuido. Sin embargo, en un entorno con millones de registros de personas una operación de estas características sería muy efectiva.

Conectando con lo descrito en subapartados anteriores, una operación MapReduce es un procesado en modo de lotes (*batch*), ya que recorre de una vez todos los datos disponibles y no devuelve resultados hasta que ha finalizado.

Es importante tener en cuenta que las funciones de agregación (*reducer*) deben ser conmutativas y asociativas.

Propiedad asociativa

Una operación $\oplus$ es asociativa si cumple la propiedad $(a \oplus b) \oplus c = a \oplus (b \oplus c)$. Por ejemplo, la suma es una operación asociativa, mientras que la resta no lo es.

2.1.2. Implementación MapReduce en Hadoop

Desde un punto de vista de proceso en entorno Hadoop, vamos a describir cuáles son los componentes de una tarea MapReduce:

1) Generalmente, un clúster está formado por varios nodos, controlados por un nodo maestro. Cada nodo almacena ficheros localmente y son accesibles mediante el sistema de ficheros HDFS (típicamente es HDFS, aunque no es un requisito necesario). Los ficheros se distribuyen de forma homogénea en todos los nodos. La ejecución de un programa MapReduce implica la ejecución de las tareas de *map()* en muchos o todos los nodos del clúster. Cada una de estas tareas es equivalente, es decir, no hay tareas *map()* específicas o distintas a las otras. El objetivo es que cualquiera de dichas tareas pueda procesar cualquier fichero que exista en el clúster.

2) Cuando la fase de *map* ha finalizado, los resultados intermedios (tuplas <clave, valor>) deben intercambiarse entre las máquinas para enviar todos los

valores con la misma clave a un solo *reduce*. Las tareas *reduce* también se ejecutan en los mismos nodos que las *map*, siendo este el único intercambio de información entre tareas (ni las tareas *map* ni las *reduce* intercambian información entre ellas, ni el usuario puede interferir en este proceso de intercambio de información). Este es un componente importante de la fiabilidad de una tarea MapReduce, ya que si un nodo falla, reinicia sus tareas sin que el estado del procesamiento en otros nodos dependa de él.

Sin embargo, en el proceso de intercambio de información entre *map* y *reducer*, podemos introducir dos nuevos conceptos: partición (*partition*) y *shuffle*.

Cuando una tarea *map* ha finalizado en algunos nodos, otros nodos todavía pueden estar realizando tareas del mismo tipo. Sin embargo, el intercambio de datos intermedios ya se inicia y estos son mandados a la tarea *reduce* correspondiente. El proceso de mover datos intermedios desde los *mapper* a los *reducers* se llama *shuffle*. Se crean subconjuntos de datos, agrupados por <clave>, lo que da lugar a las particiones, que son las entradas a las tareas *reduce*. Todos los valores para una misma clave son agregados o reducidos juntos, indistintamente de la que fuera su tarea *mapper*. Así, todos los *mapper* deben ponerse de acuerdo en dónde mandar las diferentes piezas de datos intermedios.

Cuando el proceso *reduce* ya se ha completado, agregando todos los datos, estos son guardados de nuevo en disco. La figura 11 nos muestra de forma gráfica el ejemplo descrito.

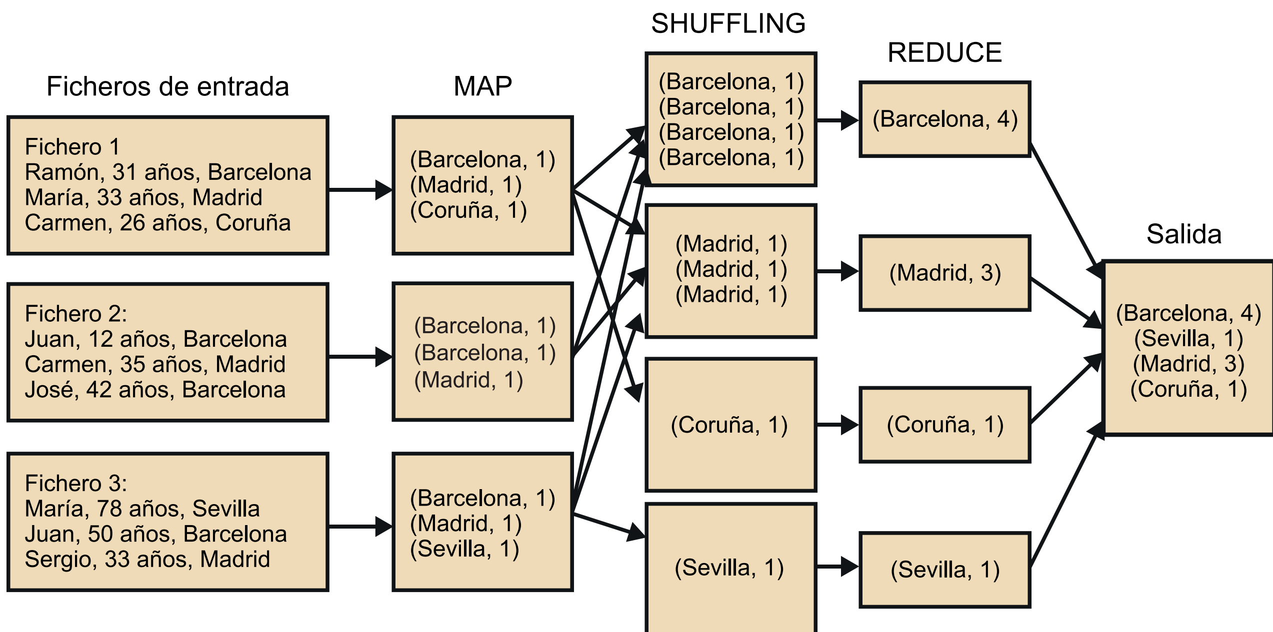


Figura 11. Diagrama de flujo con las etapas *map* y *reduce* del ejemplo descrito

2.1.3. Limitaciones de Hadoop

Hadoop es la implementación del paradigma MapReduce que solucionaba la problemática del cálculo distribuido utilizando las arquitecturas predominantes hace una década. Sin embargo, actualmente presenta importantes limitaciones:

- Es complicado aplicar el paradigma MapReduce en muchos casos, ya que una tarea debe descomponerse en subtareas *map* y *reduce*. En algunos casos, no es fácil esta descomposición.

- Es lento. Una tarea puede requerir la ejecución de varias etapas MapReduce y, en ese caso, el intercambio de datos entre etapas se llevará a cabo utilizando ficheros, haciendo uso intensivo de lectura y escritura en disco.

- Hadoop es un entorno esencialmente basado en Java que requiere la descomposición en tareas *map* y *reduce*. No obstante, esta aproximación al procesamiento de datos resultaba muy dificultosa para el científico de datos sin conocimientos de Java, de modo que diversas herramientas han aparecido para flexibilizar y abstraer al científico del paradigma MapReduce y su programación en Java. Entre estas herramientas podemos citar:

– **Apache Flume*** para almacenar datos en *streaming* hacia HDFS.

* https://flume.apache.org/

– **Apache Sqoop**** para el intercambio de datos entre bases de datos relacionales y HDFS.

** http://sqoop.apache.org/

– **Apache Hive***** o **Apache Impala****** para realizar consultas del tipo SQL sobre datos almacenados en HDFS.

*** https://hive.apache.org/
**** https://impala.apache.org

– **Apache Pig******* para definir cadenas de procesado sobre datos distribuidos.

***** https://pig.apache.org

– **Apache Giraph*** para análisis de grafos.

* http://giraph.apache.org

– **Apache Mahout**** para desarrollar tareas de aprendizaje automático.

** http://mahout.apache.org

– **Apache HBase***** como sistema de almacenamiento NoSQL.

*** https://hbase.apache.org

– **Apache Oozie****** como gestor de flujos de trabajo.

**** http://oozie.apache.org

Así, el científico de datos debe conocer un amplio conjunto de herramientas, cada una con propiedades distintas, lenguajes distintos y muy poco (o ninguna) compatibilidad entre ellas.

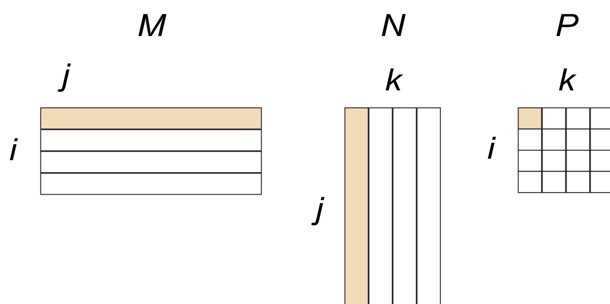
2.2. Ejemplo de aplicación de MapReduce

A continuación, recuperamos un ejercicio introducido anteriormente: la multiplicación de matrices.

Ejemplo de multiplicación de matrices con MapReduce

Vamos a resolver la multiplicación de dos matrices M y N, que da como resultado la matriz P utilizando el modelo MapReduce. Tomemos por ejemplo dos matrices, M y N, mostradas en la figura 12.

Figura 12. Representación abstracta del producto de dos matrices



$$M_{3\times 4} = \begin{bmatrix} 3 & 4 & 3 & 2 \\ 1 & 2 & 1 & 3 \\ 5 & 4 & 3 & 1 \end{bmatrix}, N_{4\times 3} = \begin{bmatrix} 6 & 4 & 3 \\ 5 & 4 & 3 \\ 3 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix}$$

Tal y como se ha descrito con anterioridad, el proceso funciona de modo que cada elemento de la primera fila de la primera matriz M se multiplica por cada elemento de la primera columna N. El resultado numérico del ejercicio es el siguiente:

$$M_{3\times 4} \cdot N_{4\times 3} = P_{3\times 3} = \begin{bmatrix} 51 & 33 & 26 \\ 25 & 16 & 13 \\ 61 & 40 & 31 \end{bmatrix}$$

Es un proceso altamente paralelizable, ya que cada cálculo es independiente del otro, tanto la multiplicación entre filas y columnas como entre elementos de cada una, al ser una operación conmutativa y asociativa.

El algoritmo de multiplicación debe componerse de etapas *map* y *reduce*, así que se debe buscar la estrategia necesaria para poder realizar los cálculos adecuándolos a estas dos etapas, y devolviendo el resultado deseado.

Map

La etapa *map* debe formar estructuras con la forma <clave,valor>que nos permitan agrupar los resultados en la etapa *reduce*.

Para entender la formulación del ejercicio debemos definir primero la notación, especialmente en lo referente a los índices:

$$M_{i\times j} \cdot N_{j\times k} = P_{i\times k}$$

donde:

- i es el número de filas en la matriz M,
- j es el número de columnas de la matriz M (para poder realizar el producto de matrices, debe coincidir con el número de filas de la matriz N),
- k es el número de columnas de la matriz N.

Para cada elemento $m_{i,j}$ de la matriz M, crearemos pares como:

$$((i,k),(M,j,m_{i,j}))$$

donde k toma valores desde 1 hasta el número de columnas.

Para cada elemento $n_{j,k}$ de la matriz N, crearemos pares como:

$$((i,k),(N,j,n_{jk}))$$

donde i toma valores desde 1 hasta el número de filas de la matriz.

Aplicando esta fórmula al ejemplo, encontramos los siguientes pares de valores para la matriz M:

$M_{1,1} = 3$
k=1: (1,1), (M,1,3)
k=2: (1,2), (M,1,3)
k=3: (1,3), (M,1,3)
k=4: (1,4), (M,1,3)
$M_{1,2} = 4$
k=1: (1,1), (M,2,4)
k=2: (1,2), (M,2,4)
k=3: (1,3), (M,2,4)
k=4: (1,4), (M,2,4)
. . .
$M_{4,3} = 1$
k=1: (1,1), (M,4,1)
k=2: (1,2), (M,4,1)
k=3: (1,3), (M,4,1)
k=4: (1,4), (M,4,1)

Este será el multiplicador para cada primer elemento k de las columnas de la matriz N, que será:

$N_{1,1} = 6$
i=1: (1,1), (N,1,6)
i=2: (2,1), (N,1,6)
i=3: (3,1), (N,1,6)
i=4: (4,1), (N,1,6)
$N_{2,1} = 5$
i=1: (1,1), (N,2,5)
i=2: (2,1), (N,2,5)
i=3: (3,1), (N,2,5)
i=4: (4,1), (N,2,5)
. . .
$N_{4,3} = 1$
i=1: (1,3), (N,3,1)
i=2: (2,3), (N,3,1)
i=3: (3,3), (N,3,1)
i=4: (4,3), (N,3,1)

Vemos que hemos creado cuatro «copias» de cada valor, ya que cada uno de ellos va a multiplicarse con cada valor de la fila correspondiente de la matriz N, esto es, cuatro productos distintos.

Para empezar a entender la notación, vamos a agrupar todas las expresiones generadas por clave, siguiendo la fórmula siguiente:

$$((i,k),[(M,j,m_{i,j}),(M,j,m_{i,j}),\ldots,(N,j,n_{j,k}),(N,j,n_{j,k}),\ldots])$$

Numéricamente queda reflejado como:

(1,1) [(M,1,3), (M,2,4), (M,3,3), (M,4,2), (N,1,6), (N,2,5), (N,3,3), (N,4,2)]
(2,1) [(M,1,1), (M,2,2), (M,3,1), (M,4,3), (N,1,6), (N,2,5), (N,3,3), (N,4,2)]
(3,1) [(M,1,5), (M,2,4), (M,3,3), (M,4,1), (N,1,6), (N,2,5), (N,3,3), (N,4,2)]

(1,2) [(M,1,3), (M,2,4), (M,3,3), (M,4,2), (N,1,4), (N,2,4), (N,3,1), (N,4,1)]
(2,2) [(M,1,1), (M,2,2), (M,3,1), (M,4,3), (N,1,4), (N,2,4), (N,3,1), (N,4,1)]
(3,2) [(M,1,5), (M,2,4), (M,3,3), (M,4,1), (N,1,4), (N,2,4), (N,3,1), (N,4,1)]

(1,3) [(M,1,3), (M,2,4), (M,3,3), (M,4,2), (N,1,3), (N,2,3), (N,3,1), (N,4,1)]
(2,3) [(M,1,1), (M,2,2), (M,3,1), (M,4,3), (N,1,3), (N,2,3), (N,3,1), (N,4,1)]
(3,3) [(M,1,5), (M,2,4), (M,3,3), (M,4,1), (N,1,3), (N,2,3), (N,3,1), (N,4,1)]

Reduce

El proceso de *reduce* toma los valores agrupados por cada clave y debe encontrar el mecanismo de agrupación. Tomamos todos los valores de cada clave y los separamos en dos listas, los correspondientes a la matriz M y los de la matriz N:

Por ejemplo, para la clave (1,1):
(M,1,3), (M,2,4), (M,3,2), (M,4,2)
(N,1,6), (N,2,5), (N,3,3), (N,4,2)

Así, tomamos cada elemento ordenado por el segundo índice, y multiplicamos cada valor por el correspondiente de la otra lista según la ordenación, sumando el resultado.

Para la clave (1,1), la suma de los productos de dichos valores es:

$$3 \times 6 + 4 \times 5 + 3 \times 3 + 2 \times 2 = 51$$

El proceso de *reduce* debe repetir la misma operación para todas las claves disponibles.

Podemos concluir que, aun siendo un paradigma estricto, nada impide crear pares de <clave, valor> complejas que el *reducer* pueda procesar de modo que se consiga la agrupación deseada.

2.3. Apache Mahout

Apache Mahout* es una herramienta que proporciona algoritmos de aprendizaje automático distribuidos o escalables, enfocados principalmente a las áreas de filtrado colaborativo, de clusterización y de clasificación. Muchas de las implementaciones usan la plataforma Apache Hadoop, aunque Mahout también proporciona bibliotecas de Java para operaciones matemáticas comunes (centradas en álgebra lineal y estadísticas) y colecciones primitivas de Java.

*** http://mahout.apache.org**

Los algoritmos centrales de Mahout para la clusterización, la clasificación y el filtrado colaborativo basado en procesamiento por lotes (*batch processing*) se implementan sobre Apache Hadoop utilizando el paradigma MapReduce. Por

lo tanto, se busca la forma de poder implementar los algoritmos existentes en las diferentes áreas del aprendizaje automático para poder ser procesados en forma de etapas *map* y *reduce*, aunque sea de forma iterativa, empleando los resultados de un proceso como entradas de un segundo proceso, en un estructura de *pipeline* o cadenas de procesos.

2.4. Conclusiones

MapReduce fue un paradigma muy novedoso para poder explorar grandes archivos en entornos distribuidos. Es un algoritmo que puede desplegarse, en su implementación Hadoop, en equipos de bajo coste o convencionales (o también llamado *commodity*), que lo hizo muy atractivo. Varias herramientas que ofrecen funcionalidades diversas en el ecosistema de datos masivos como Hive o Sqoop se han desarrollado implementando MapReduce. Sin embargo, el algoritmo presenta algunas limitaciones importantes:

- Utiliza un modelo forzado para cierto tipo de aplicaciones, que obliga a crear etapas adicionales *map* o *reduce* para ajustar la aplicación al modelo. Puede llegar a emitir valores intermedios extraños o inútiles para el resultado final, e incluso creando funciones *map* o *reduce* de tipo identidad, es decir, que no son necesarias para la operación que se va a realizar, pero que se deben crear para adecuar el cálculo al modelo.

- MapReduce es un proceso *file-based*, lo que significa que el intercambio de datos se realiza utilizando ficheros. Esto produce un elevado flujo de datos en lectura y escritura de disco, y afecta a su velocidad y rendimiento general si un procesamiento concreto está formado por una cadena de procesos MapReduce.

Para solucionar estas limitaciones apareció Apache Spark, que cambia el modelo de ejecución de tareas, haciéndolo más ágil y polivalente, como se presentará más adelante.

3. Apache Spark

Como hemos visto en el apartado anterior, MapReduce es un potente algoritmo para procesar datos distribuidos, altamente escalable y relativamente simple. Sin embargo, presenta muchas debilidades en cuanto a rendimiento y versatilidad para implementar algoritmos más complejos. Así, en este apartado exploraremos cuáles son los entornos más utilizados para el procesado de grandes volúmenes de datos para tareas de análisis de datos y aplicación a algoritmos complejos, centrando nuestra atención en Apache Spark.

Apache Spark* es un entorno para el procesamiento de datos distribuidos que ofrece un alto rendimiento y es compatible con todos los servicios que forman parte del ecosistema de Apache Hadoop. Entre sus principales funcionalidades destacamos:

*** https://spark.apache.org**

- **SparkSQL**: API para trabajar con datos distribuidos utilizando estructuras abstractas, llamadas *dataframes*, similares a los *dataframes* de R** o Pandas*** de Python.

**** https://www.r-project.org**
***** http://pandas.pydata.org**

- **MLlib**: Spark también ofrece una API para realizar tareas de aprendizaje automático, de la que hablaremos con más detalle en subapartados posteriores.

- **GraphX**: API para desarrollar aplicaciones de análisis de datos basados en grafos (*graph mining*).

- **Spark Streaming**: API para desarrollar aplicaciones de datos masivos con flujos de datos continuos (*streaming data*).

En este apartado nos centraremos en el modo interno de funcionamiento de Spark y cómo paraleliza sus tareas, convirtiéndolo en el entorno más popular para análisis en entornos de datos masivos.

3.1. Resilient distributed dataset (RDD)

Para entender el procesamiento distribuido de Spark, debemos conocer uno de sus componentes principales, el *resilient distributed dataset* (RDD en adelante) sobre el que se basa su estructura de paralelización de trabajo.

Un RDD es una entidad abstracta que representa un conjunto de datos distribuidos por el clúster y que representa una capa de abstracción encima de

todos los datos que componen nuestro corpus de trabajo, independiente de su volumen, ubicación en el clúster, etc.

Las principales características de los RDD son las siguientes:

- Es inmutable, una condición importante porque evita que uno o varios hilos (*threads*) actualicen el conjunto de datos con el que se trabaja.

- Es *lazy loading*, es decir, solo se accede a los datos y estos se cargan cuando es necesario.

- Puede almacenarse en memoria caché.

El RDD se construye a partir de bloques de memoria distribuidos en cada nodo, dando lugar a las llamadas particiones. Este particionamiento lo realiza Spark y es transparente para el usuario, aunque también puede tener control sobre él. Si los datos se leen del sistema HDFS, cada partición se corresponde con un bloque de datos del sistema HDFS.

El concepto partición tiene un peso importante para comprender cómo se paraleliza en Spark. Cuando se lee un fichero, si este es mayor que el tamaño de bloque definido por el sistema HDFS (típicamente 64 MB o 128 MB), se crea una partición para cada bloque (aunque el usuario puede definir el número de particiones por crear cuando está leyendo el fichero, si lo desea).

Así, nos encontramos con lo siguiente:

- Un RDD es un *array* de referencias a particiones en nuestro sistema.

- La partición es la unidad básica para entender el paralelismo en Spark y cada partición se relaciona con bloques de datos físicos en nuestro sistema de almacenaje o de memoria.

- Las particiones se asignan con criterios como la localidad de datos (*data locality*) o la minimización de tráfico en la red interna.

- Cada partición se carga en memoria volátil (típicamente, RAM) antes de ser procesada.

Un RDD se puede construir a partir de:

1) Creación de un objeto «colección» paralelo:

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

Esto creará una colección paralela en el nodo del controlador, hará particiones y las distribuirá entre los nodos del clúster, concretamente, en la memoria.

2) Creación de RDD desde fuentes externas, como por ejemplo HDFS o S3 de Amazon.

Se crearán particiones por bloque de datos HDFS en los nodos donde los datos están físicamente disponibles.

3) Al ejecutar cualquier operación sobre un RDD existente. Al ser inmutable, cuando se aplica cualquier operación sobre un RDD existente, se creará un nuevo RDD.

Sobre un RDD se pueden aplicar transformaciones o acciones, pero no se accede a los datos que componen un RDD hasta que se ejecuta sobre ellos una acción, siguiendo el modelo *lazy loading* ya mencionado. Así, cuando se trabaja con RDD a los cuales se aplican transformaciones, en realidad se está definiendo una secuencia de procesos que aplicar sobre el RDD que no se pondrán en marcha hasta la invocación de una acción sobre ella.

Tomemos la siguiente secuencia de comandos en Python para Spark:

```
data = spark.textFile("hdfs://...")
words = data.flatMap(lambda line : line.split("␣"))
        .map(lambda word : (word, 1))
                .reduceByKey(lambda a, b : a + b)
words.count()
```

Esta porción de código muestra cómo se carga un RDD (variable *data*) a partir de un fichero ubicado en el sistema distribuido HDFS. A continuación, se aplican hasta tres transformaciones (*flatMap*, *map* y *reduceByKey*). No obstante, ninguna de estas transformaciones se va a ejecutar de forma efectiva hasta la llamada al método *count()*, que nos devolverá el número de elementos en el RDD «words». Es decir, hasta que no se ejecute una acción sobre los datos, no se ejecutan las operaciones que nos los proporcionan.

El programa en el cual se ejecuta el código mostrado es llamado *driver*, y es el encargado de controlar la ejecución, mientras que las transformaciones son las operaciones que se paralelizan y se ejecutan de forma distribuida entre los nodos que componen el clúster. Los responsables de la ejecución en los nodos son conocidos como *executors*. Es entonces cuando se accede a los datos distribuidos que componen el RDD. Así, si hay algún problema en alguna de las transformaciones solo será posible detectarla una vez se ejecute la acción que desencadena las transformaciones asociadas.

3.2. Modelo de ejecución Spark

El *directed acyclic graph* (DAG, o grafo dirigido acíclico) es un estilo de programación para sistemas distribuidos, que se presenta como una alternativa al paradigma MapReduce. Mientras que MapReduce tiene solo dos pasos (*map* y *reduce*), lo que limita la implementación de algoritmos complejos, DAG puede tener múltiples niveles que pueden formar una estructura de árbol, con más funciones tales como *map*, *filter*, *union*, etc.

En un modelo DAG no hay ciclos definidos ni un patrón de ejecución previamente definido, aunque sí existe un orden o una dirección de ejecución. Apache Spark hace uso del modelo DAG para la ejecución de sus tareas en entornos distribuidos.

El DAG de Spark está compuesto por arcos (o aristas) y vértices, donde cada vértice representa un RDD y los ejes representan operaciones que aplicar sobre el RDD.

Las transformaciones sobre RDD pueden ser categorizadas como:

- ***Narrow operation***: se utiliza cuando los datos que se necesitan tratar están en la misma partición del RDD y no es necesario realizar una mezcla de dichos datos para obtenerlos todos. Algunos ejemplos son las funciones *filter*, *sample*, *map* o *flatMap*.

- ***Wide operation***: se utiliza cuando la lógica de la aplicación necesita datos que se encuentran en diferentes particiones de un RDD y es necesario mezclar dichas particiones para agrupar los datos necesarios en un RDD determinado. Ejemplos de *wide transformation* son: *groupByKey* o *reduceByKey*. Estas suelen ser las tareas de agregación y los datos provenientes de diferentes particiones se agrupan en un conjunto menor de particiones.

Cada RDD mantiene una referencia a uno o más RDD originales, junto con metadatos sobre qué tipo de relación tiene con ellos. Por ejemplo, si ejecutamos el código siguiente:

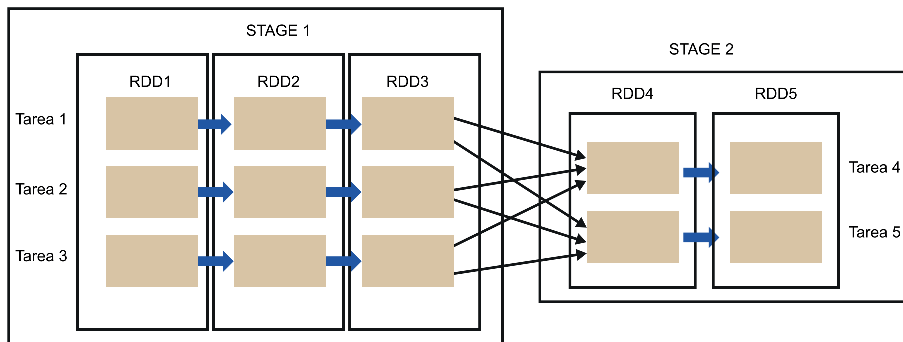
```
val b = a.map()
```

El RDD *b* mantiene una referencia a su padre, el RDD *a*. Es el llamada RDD *lineage*.

Al conjunto de operaciones sobre las que se realizan transformaciones preservando las particiones del RDD (*narrow*) se las llama «etapa» (*stage*). El final de un *stage* se presenta cuando se reconfiguran las particiones debido a operacio-

nes de transformación tipo *wide*, que inician el llamado intercambio de datos entre nodos (*shuffling*), iniciando otro *stage*.

Figura 13. Modelo de procesado de RDD en Spark, diferenciando tareas y *stages* o etapas en función del tipo de transformación que se va a aplicar



Las limitaciones de MapReduce se convirtieron en un punto clave para introducir el modelo de ejecución DAG en Spark.

Con la aproximación de Spark, la secuencia de ejecuciones, formadas por los *stages* y las transformaciones, las acciones por aplicar y su orden queda reflejado en un grafo DAG que optimiza el plan de ejecución y minimiza el tráfico de datos entre nodos.

3.2.1. Funcionamiento

Cuando una acción es invocada para ser ejecutada sobre un RDD, Spark envía el DAG con los *stages* y las tareas que se van a realizar al DAG Scheduler, el cual transforma un plan de ejecución lógica (es decir, RDD de dependencias construidas mediante transformaciones) a un plan de ejecución físico (utilizando *stages* o etapas y los bloques de datos).

El DAG Scheduler es un servicio que se ejecuta en el *driver* de nuestro programa. Tiene dos tareas principales:

1) Determina las ubicaciones preferidas para ejecutar cada tarea y calcula la secuencia de ejecución óptima:

- primero se analiza el DAG para determinar el orden de las transformaciones,
- con el fin de minimizar el mezclado de datos, primero se realizan las transformaciones *narrow* en cada RDD,
- finalmente se realiza la transformación *wide* a partir de los RDD sobre los que se han realizado las transformaciones *narrow*.

2) Maneja los posibles problemas y fallos debido a una posible corrupción de datos en algún nodo: cuando cualquier nodo queda no operativo (bloqueado o caído) durante una operación, el administrador del clúster asigna otro nodo para continuar el proceso. Este nodo operará en la partición particular del RDD y la serie de operaciones que tiene que ejecutar sin haber pérdida de datos.

Así, es el DAG Scheduler el que calcula el DAG de etapas para cada trabajo, realiza un seguimiento de qué RDD y salidas de etapa se materializan y encuentra el camino mínimo para ejecutar los trabajos. A continuación, envía las etapas al Task Scheduler.

Las operaciones por ejecutar en un DAG están optimizadas por el DAG Optimizer, que puede reorganizar o cambiar el orden de las transformaciones si con ello se mejora el rendimiento de la ejecución. Por ejemplo, si una tarea tiene una operación *map* seguida de una operación *filter*, el DAG Optimizer cambiará el orden de estos operadores, ya que el filtrado reducirá el número de elementos sobre los que realizar el *map*.

3.3. Apache Spark MLlib

Entre las varias API de trabajo de Spark, probablemente la más popular y potente es la API de aprendizaje automático (*machine learning*). La comunidad de contribuidores a esta API es la más grande del ecosistema Spark y en cada nueva versión se incluyen nuevas técnicas.

El análisis de datos es uno de los campos de trabajo más atractivos en el sector tecnológico y uno de los que tienen mayor demanda profesional. En dicho campo se trabaja de forma intensiva utilizando las herramientas de minería de datos más completas, como R, Python o Matlab.

Estas plataformas presentan un alto grado de madurez, incorporan multitud de librerías de aprendizaje automático y, también, muchos casos de uso y experiencias desarrollados y acumulados a lo largo de los años. Sin embargo, Spark ofrece una implementación paralela.

3.3.1. ML Pipelines

Como entorno para trabajar con la API de aprendizaje automático de Spark, se ofrece la herramienta *pipeline*, que permite crear cadenas de trabajo propias de una tarea de minería de datos. Una *pipeline* consta de varios componentes:

- *dataframe* de entrada
- *estimator* (responsable de generar un modelo)
- parámetros para generar los modelos
- *transformer* (modelo generado a partir de unos parámetros concretos)

Una *pipeline* consta de una cadena de estimadores y transformaciones. El DAG de una ML Pipeline no tiene por qué ser secuencial, donde la entrada de cada *stage* es la salida de la anterior, pero debe poder ser representado mediante ordenación topológica.*

*** http://bit.ly/2CuyQtW**

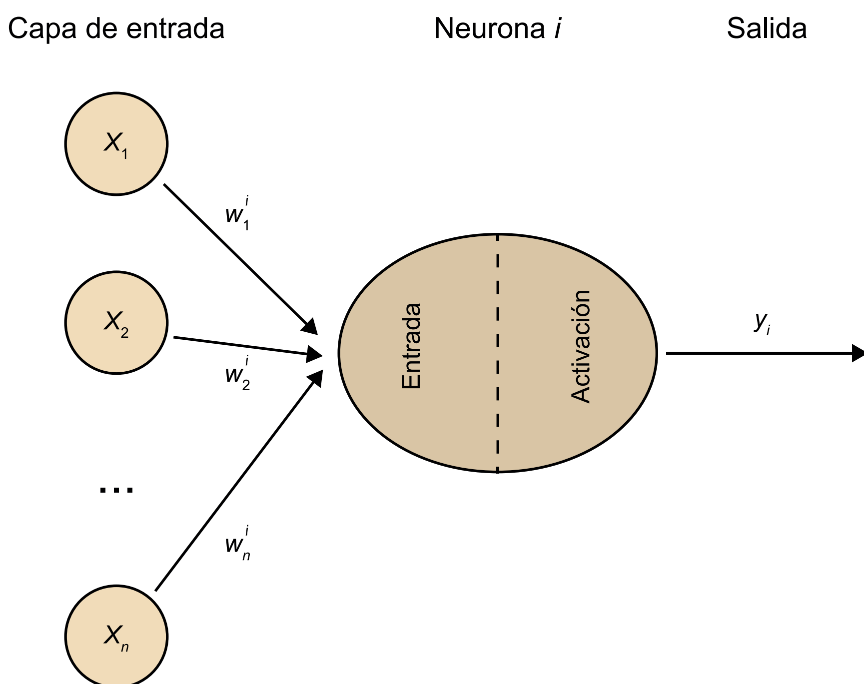
4. TensorFlow

Actualmente, el aprendizaje automático es una de las disciplinas más atractivas de estudio y con mayor proyección profesional, especialmente tras la explosión de los datos masivos y su ecosistema tecnológico, que ha abierto un enorme panorama de potenciales aplicaciones que no eran viables de resolver hasta ahora. Pero el mayor desafío del aprendizaje automático viene dado de las técnicas de clasificación no supervisada, esto es, que la máquina aprende por sí misma a partir de unos datos no etiquetados previamente.

4.1. Redes neuronales y *deep learning*

En el enfoque basado en las redes neuronales, se usan estructuras lógicas que se asemejan en cierta medida a la organización del sistema nervioso de los mamíferos, teniendo capas de unidades de proceso (llamadas neuronas artificiales) que se especializan en detectar determinadas características existentes en los objetos percibidos, permitiendo que dentro del sistema global haya redes de unidades de proceso que se especialicen en la detección de determinadas características ocultas en los datos.

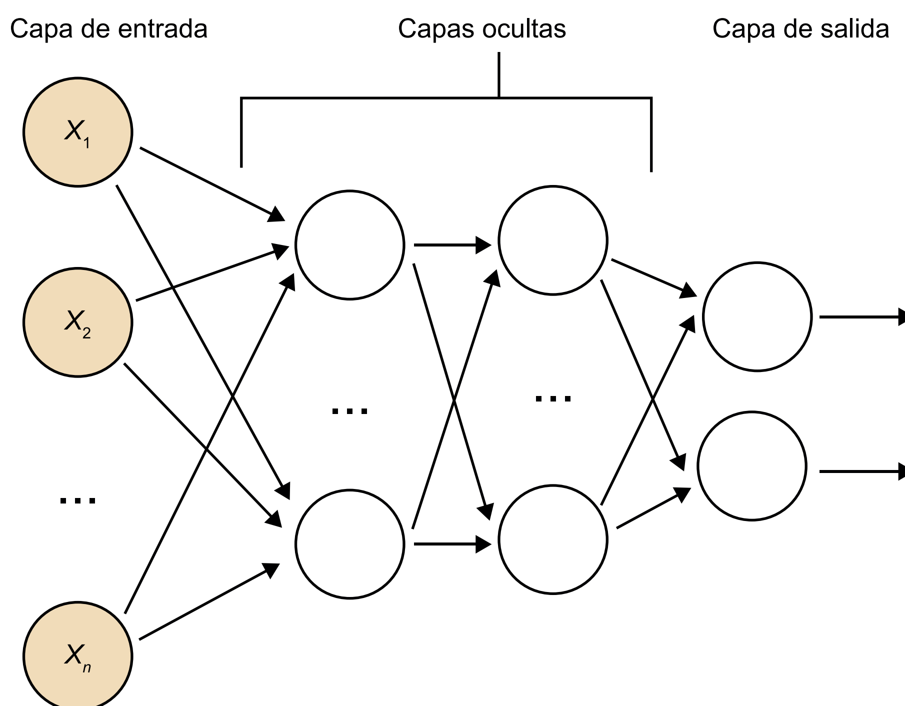
Figura 14. Esquema de una neurona



Las redes neuronales artificiales (ANN, *artificial neural networks*) están formadas por un conjunto de neuronas distribuidas en distintas capas. Cada una de estas neuronas realiza un cálculo u operación sencilla sobre el conjunto de valores de entrada de la neurona, que en esencia son entradas de datos o las salidas de las neuronas de la capa anterior, y calcula un único valor de salida, que a su vez será un valor de entrada para las neuronas de la capa siguiente o bien formará parte de la salida final de la red.

Las neuronas se agrupan formando capas, que son estructuras de neuronas paralelas. Las salidas de unas capas se conectan con las entradas de otras, de forma que se crea una estructura secuencial. La figura 15 presenta un esquema básico de una red neuronal con la capa de entrada, múltiples capas ocultas y la capa de salida.

Figura 15. Esquema de red neuronal con múltiples capas ocultas



Las redes neuronales no son un concepto nuevo y, aunque su funcionamiento siempre ha sido satisfactorio en cuanto a resultados, el consumo de recursos para entrenar una red neuronal siempre ha sido muy elevado, lo que ha impedido en parte su completo desarrollo y aplicación. Sin embargo, tras la irrupción de las GPU han vuelto a ganar protagonismo gracias al elevado paralelismo requerido para entrenar cada neurona y el potencial beneficio que ofrecen las GPU para este tipo de operaciones.

Estas redes se suelen inicializar con valores aleatorios, y requieren de un proceso de entrenamiento con un conjunto de datos para poder «aprender» una tarea o función concreta. Este proceso de aprendizaje se realiza utilizando el método conocido como Backpropagation*. En esencia, este método calcula el

* http://bit.ly/2jMEOBe

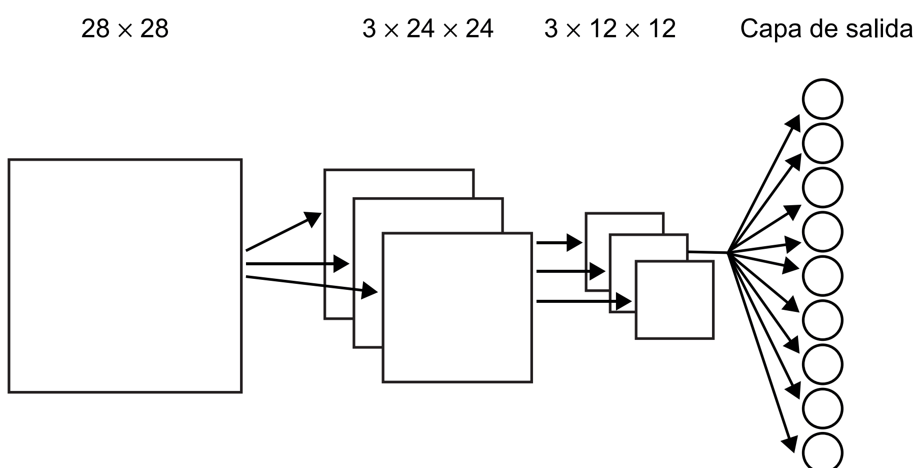
error que comete la red en la predicción del valor para un ejemplo dado e intenta modificar los parámetros de todas las neuronas de la red para reducir dicho error.

Este tipo de algoritmos tuvo su época de esplendor hace ya unas décadas. Su limitación principal se encuentra en el proceso de aprendizaje que se da en las redes con cierto número de capas ocultas (capas intermedias, es decir, que se encuentran entre la entrada de datos y la salida o respuesta final de la red). En estos casos, se produce lo que se conoce como el problema de la desaparición o explosión del gradiente,** que básicamente provoca problemas en el proceso de aprendizaje de la red.

** http://bit.ly/2DERG0J

Estos problemas han sido superados años más tarde, dando inicio a lo que se conoce actualmente como *deep learning*. Se ha modificado la estructura básica de las redes neuronales, creando, por ejemplo, redes convolucionales que permiten crear distintas capas donde el conocimiento se va haciendo más abstracto. Es decir, las primeras capas de la red se pueden encargar de identificar ciertos patrones en los datos, mientras que las capas posteriores identifican conceptos más abstractos a partir de estos patrones más básicos. Por ejemplo, si queremos que una red neuronal pueda detectar cuándo aparece una cara en una imagen, este enfoque buscaría que las primeras capas de la red se encargaran de detectar la presencia de un ojo en alguna parte de la imagen, de una boca, etc. Así, las siguientes capas se encargarían de combinar esta información e identificar una cara a partir de la existencia de las partes que la forman (ojos, boca, nariz, etc). De esta forma vamos avanzando desde una información más básica hacia un conocimiento más abstracto. La figura 16 muestra un esquema, aunque muy general, de una red convolucional.

Figura 16. Esquema general de una red neuronal convolucional



Existen varios entornos y bibliotecas para trabajar con redes neuronales y *deep learning* que se ejecutan en las potentes GPU modernas, generalmente mediante la API CUDA. Probablemente, una de las más famosas es el entorno TensorFlow de Google.

4.2. TensorFlow

TensorFlow* es una API de código abierto desarrollada por Google para construir y entrenar redes neuronales. TensorFlow es una potente herramienta para hacer uso extensivo de los recursos locales de un ordenador, ya sea CPU o GPU.

* https://www.tensorflow.org

Así, TensorFlow es un sistema de programación en el que representamos cálculos en forma de grafos, donde cada nodo del grafo (llamados *ops*) realiza una operación sobre uno o varios «tensores» (que puede ser un vector, un valor numérico o una matriz) y como resultado devuelve de nuevo un tensor.

El grafo que representa el TensorFlow describe los cálculos que se van a realizar, de ahí la terminología *flow*. El grafo se lanza en el contexto de una sesión de TensorFlow (*session*), que encapsula en entorno de operación de nuestra tarea, incluyendo la ubicación de las tareas en nuestro sistema, ya sea CPU o GPU.

Aunque TensorFlow dispone de implementaciones paralelizables también puede ser distribuido** por medio de un clúster de ordenadores, en el cual cada nodo se ocupa de una tarea del grafo de ejecución.

** http://bit.ly/2EnN3JY

Existen implementaciones de TensorFlow en varios lenguajes de programación, como Java, Python, C++ y Go.*** Asimismo, permite el uso de GPU, lo que puede resultar en un incremento importante de rendimiento en el entrenamiento de redes neuronales profundas, por ejemplo.

*** https://golang.org

En este punto, nos preguntamos: ¿podríamos combinar lo mejor del procesado distribuido (Spark) y una herramienta de aprendizaje automático tan potente como TensorFlow?

Podríamos utilizar modelos entrenados con herramientas como scikit-learn y combinarlos con las funcionalidades de Spark y su capacidad de propagar variables a los nodos mediante la opción *broadcast***** para enviar el modelo mediante múltiples nodos y dejar que cada nodo aplique el modelo a un corpus de datos.

**** http://bit.ly/2oukWCh

En la actualidad hay muchas iniciativas en las que se combina Spark para la evaluación de modelos y TensorFlow para realizar tareas de entrenamiento, validación cruzada de modelos o aplicación y predicción del modelo.

Recientemente ha aparecido una nueva API, llamada Deep Learning Pipelines,***** desarrollada por Databricks (los mismos desarrolladores de Apache Spark), para combinar técnicas de *deep learning*, principalmente TensorFlow, con Apache Spark.

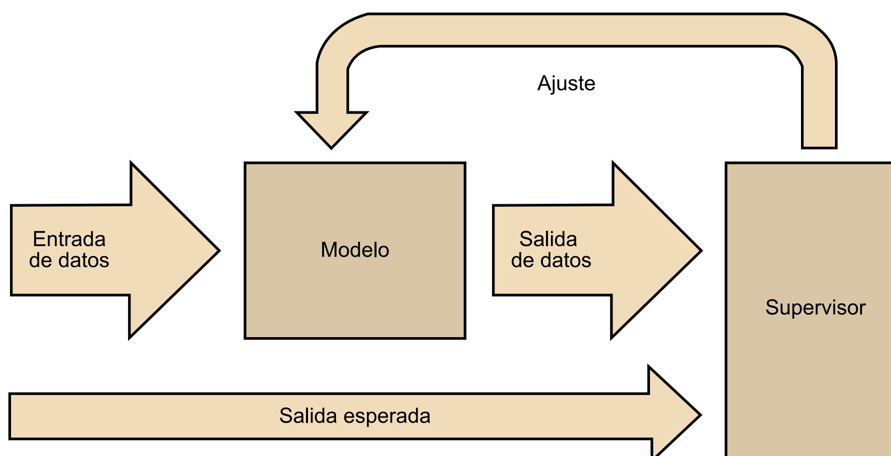
***** http://bit.ly/2Ci1l1h

5. Aprendizaje autónomo

El aprendizaje autónomo (también conocido por su denominación en inglés, *Machine Learning*) es el conjunto de métodos y algoritmos que permiten a una máquina aprender de manera automática en base a experiencias pasadas.

Generalmente, un algoritmo de aprendizaje autónomo debe construir un modelo sobre la base de un conjunto de datos de entrada que representan el conjunto de aprendizaje, lo que se conoce como *conjunto de entrenamiento*. Durante esta fase de aprendizaje, el algoritmo va comparando la salida de los modelos en construcción con la salida ideal que deberían tener estos modelos, para ir ajustándolos y aumentando la precisión. Esta comparación que forma la base del aprendizaje en sí y este aprendizaje puede ser **supervisado** o **no supervisado**. En el aprendizaje supervisado (figura 17), hay un componente externo que compara los datos obtenidos por el modelo con los datos esperados por este, y proporciona retroalimentación al modelo para que vaya ajustándose. Para ello, pues, será necesario proporcionar al modelo un conjunto de datos de entrenamiento que contenga tanto los datos de entrada como la salida esperada para cada uno de esos datos.

Figura 17. Aprendizaje supervisado

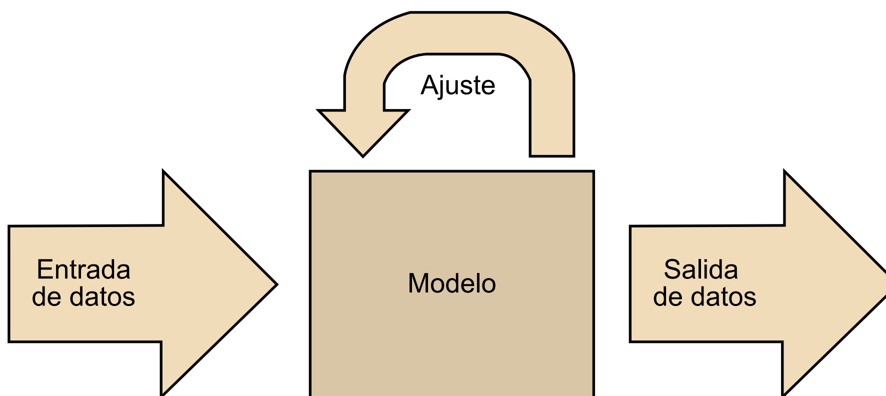


Todas ellas se basan en el paradigma del aprendizaje inductivo. La esencia de cada una de ellas es derivar inductivamente a partir de los **datos** (que representan la información del entrenamiento), un **modelo** (que representa el

conocimiento) que tiene utilidad predictiva, es decir, que puede aplicarse a nuevos datos.

En el aprendizaje no supervisado (figura 18), el algoritmo de entrenamiento aprende sobre los propios datos de entrada, descubriendo y agrupando patrones, características, correlaciones, etc.

Figura 18. Aprendizaje no supervisado



Algunos algoritmos de aprendizaje autónomo requieren de un gran conjunto de datos de entrada para poder converger hacia un modelo preciso y fiable, siendo pues los datos masivos un entorno ideal para dicho aprendizaje. Tanto Hadoop (con su paquete de aprendizaje autónomo, Mahout) y Spark (con su solución MLlib) proporcionan un extenso conjunto de algoritmos de aprendizaje autónomo. A continuación los describiremos, indicando su clasificación y para qué plataformas están disponibles.

5.1. Clasificación

La clasificación (*classification*) es uno de los procesos cognitivos importantes, tanto en la vida cotidiana como en los negocios, donde podemos clasificar clientes, empleados, transacciones, tiendas, fábricas, dispositivos, documentos o cualquier otro tipo de instancias en un conjunto de clases o categorías predefinidas con anterioridad.

La tarea de clasificación consiste en asignar instancias de un dominio dado, descritas por un conjunto de atributos discretos o de valor continuo, a un conjunto de clases, que pueden ser consideradas valores de un atributo discreto seleccionado, generalmente denominado **clase**. Las etiquetas de clase correctas son, en general, desconocidas, pero se proporcionan para un subconjunto del dominio. Por lo tanto, queda claro que es necesario disponer de un subconjunto de datos correctamente etiquetado, y que se usará para la construcción del modelo.

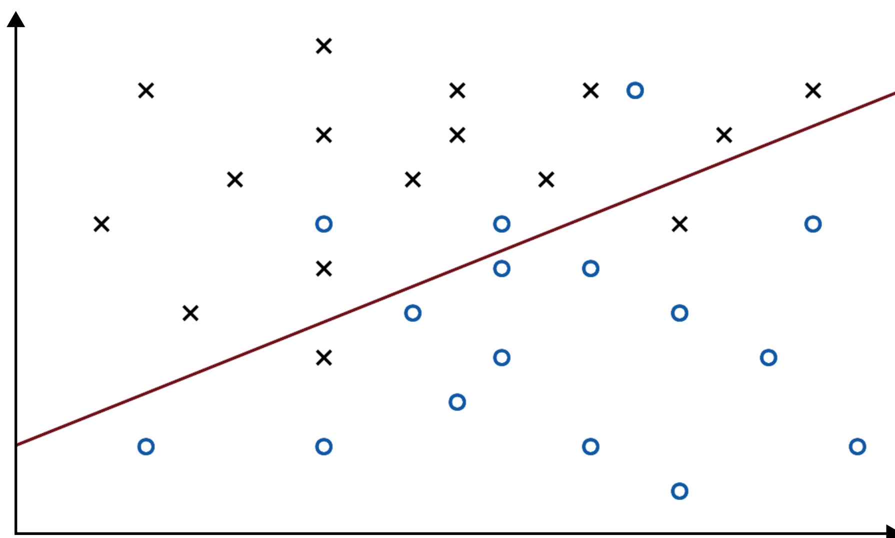
La función de clasificación puede verse como:

$$c : X \rightarrow C \quad (1)$$

donde c representa la función de clasificación, X el conjunto de atributos que forman una instancia y C la etiqueta de clase de dicha instancia.

Un tipo de clasificación particularmente simple, pero muy interesante y ampliamente estudiado, hace referencia a los problemas de clasificación binarios, es decir, problemas con un conjunto de datos pertenecientes a dos clases, es decir, $C = \{0,1\}$. La figura 19 muestra un ejemplo de clasificación binaria, donde las cruces y los círculos representan elementos de dos clases, y se pretende dividir el espacio de forma tal que separe a la mayoría de elementos de clases diferentes.

Figura 19. Ejemplo de clasificación



Tanto MLlib como Mahout proporcionan diversos métodos de aprendizaje para la clasificación, tales como árboles de clasificación y regresión, clasificadores bayesianos *naïve*, bosques de árboles de decisión, perceptrones multicapa y modelos ocultos de Markov.

Las técnicas de clasificación supervisada incluidas en Spark* son las más completas del entorno y se dividen en varios tipos de clasificadores, que podemos agrupar en:

* http://bit.ly/2lqh43M

- clasificador basado en regresión logística, tanto binaria como multidimensional,
- clasificador basado en árboles de decisión (*decision trees*),
- clasificador *random forest*,

- clasificador basado en árbol con degradación de gradiente,
- clasificador basado en redes neuronales (perceptrón multicapa),
- máquinas de vectores soporte (SVM, *support vector machine*),
- clasificador One-vs-Rest o One-vs-All (clasificador que se construye encima de un clasificador binario),
- clasificador bayesiano ingenuo (*naïve Bayes*).

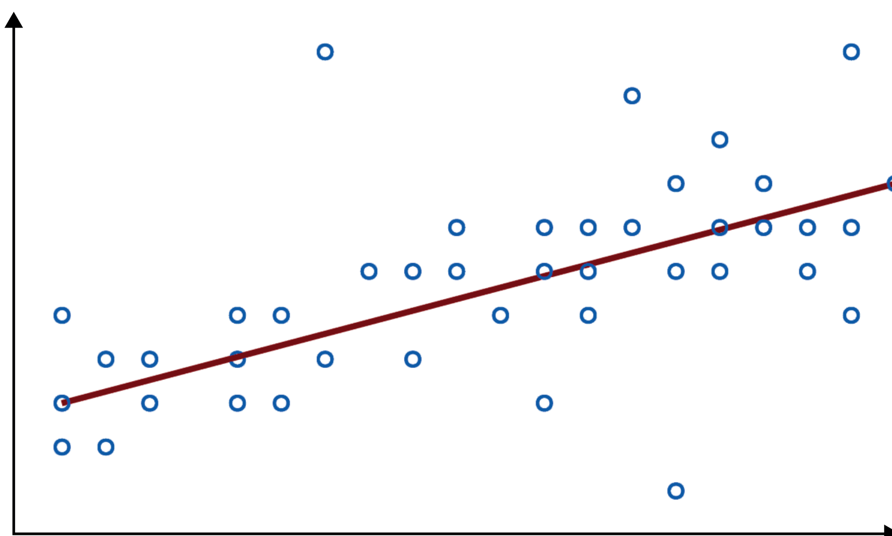
En Spark existe tanto el clasificador como el modelo de regresión de cada una de estas técnicas. Las técnicas de regresión retornan una variable de salida con valores continuos, mientras que en la clasificación la variable de salida toma etiquetas (*labels*) que representan una clase, resultado de la clasificación.

5.2. Regresión

Igual que la clasificación, la regresión (*regression*) es una tarea de aprendizaje inductivo que ha sido ampliamente estudiada y utilizada. Se puede definir, de forma informal, como un problema de «clasificación con clases continuas». Es decir, los modelos de regresión predicen valores numéricos en lugar de etiquetas de clase discretas. A veces también nos podemos referir a la regresión como «predicción numérica».

La tarea de regresión consiste en asignar valores numéricos a instancias de un dominio dado, descritos por un conjunto de atributos discretos o de valor continuo, como se muestra en la figura 20, donde los puntos representan los datos de aprendizaje y la línea representa la predicción sobre futuros eventos. Se supone que esta asignación se aproxima a alguna función objetivo, generalmente desconocida, excepto para un subconjunto del dominio. Este subconjunto se puede utilizar para crear el modelo de regresión.

Figura 20. Ejemplo de regresión lineal



En este caso, la función de regresión se puede definir como:

$$f : X \to \mathbb{R} \quad (2)$$

donde f representa la función de regresión, X el conjunto de atributos que forman una instancia y $\mathbb{R}$ un valor en el dominio de los números reales.

Es importante remarcar que una regresión no pretende devolver una predicción exacta sobre un evento futuro, sino una aproximación (como muestra la diferencia entre la línea y los puntos de la figura). Por lo general, datos más dispersos resultarán en predicciones menos ajustadas.

Actualmente, MLlib y Mahout proporcionan métodos para la regresión logística (aquellas que pueden ser aproximadas mediante una función logística).* Además, MLlib proporciona métodos para regresiones lineales y regresiones isotónicas (definidas por partes).**

*** http://bit.ly/2CtYLSu**
**** http://bit.ly/2zSJg3z**

5.3. Agrupamiento

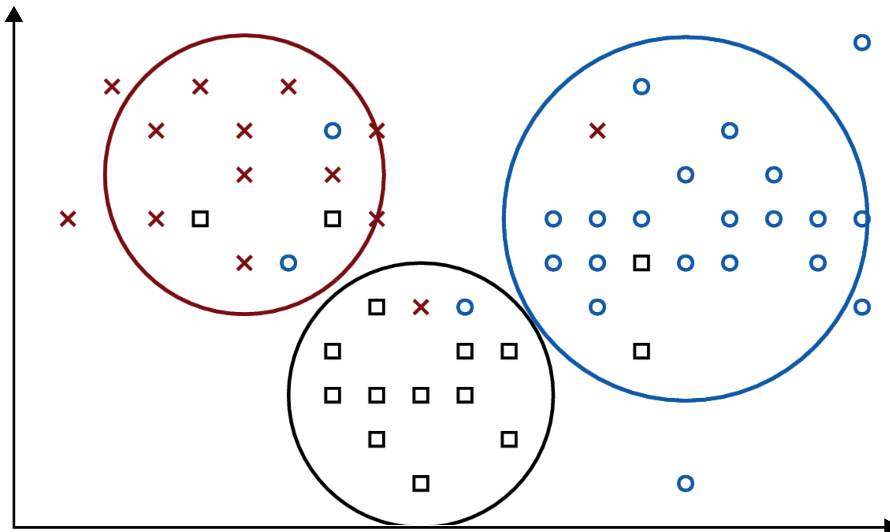
El agrupamiento (*clustering*) es una tarea de aprendizaje inductiva que, a diferencia de las tareas de clasificación y regresión, no dispone de una etiqueta de clase por predecir. Puede considerarse como un problema de clasificación, pero en el que no existen un conjunto de clases predefinidas, y estas se «descubren» de forma autónoma por el método o algoritmo de agrupamiento, basándose en patrones de similitud identificados en los datos.

La tarea de agrupamiento consiste en dividir un conjunto de instancias de un dominio dado, descrito por un número de atributos discretos o de valor continuo, en un conjunto de grupos (*clusters*) basándose en la similitud entre las instancias, y crear un modelo que puede asignar nuevas instancias a uno de estos grupos o *clusters*. La figura 21 muestra un ejemplo de agrupamiento, donde las cruces, círculos y cuadros pertenecen a tres clases de elementos distintos que pretendemos agrupar.

Un proceso de agrupación puede proporcionar información útil sobre los patrones de similitud presentes en los datos como, por ejemplo, segmentación de clientes o creación de catálogos de documentos. Otra de las principales utilidades del agrupamiento es la detección de anomalías. En este caso, el método de agrupamiento permite distinguir instancias que con un patrón absolutamente distinto a las demás instancias «normales» del conjunto de datos, de

forma que facilita la detección de anomalías y posibilita la emisión de alertas automáticas para nuevas instancias que no tienen ningún clúster existente.

Figura 21. Ejemplo de agrupamiento



La función de agrupamiento o *clustering* se puede modelar mediante:

$$h : X \rightarrow C_h \quad (3)$$

donde h representa la función de agrupamiento, X el conjunto de atributos que forman una instancia y C_h un conjunto de grupos o *clusters*. Aunque esta definición se parece mucho a la tarea de clasificación, hay una diferencia aparentemente pequeña pero muy importante consistente en que el conjunto de «clases» no está predeterminado ni es conocido *a priori*, sino que se identifica como parte de la creación del modelo.

El algoritmo de agrupamiento más conocido es *K-means*, el cual va formando grupos iterativamente a partir de un conjunto dado. Este es implementado en diferentes variantes tanto por MLlib como por Mahout.

MLlib de Apache Spark incorpora el algoritmo *K-means*.* También incorpora implementaciones de Latent Dirichlet Allocation (LDA), Bisecting *k-means*, una técnica de agrupamiento jerárquica que va creando subclústeres dentro de clústeres mayores, y Gaussian Mixture Model (GMM).** Sin embargo, hay otras implementaciones que no vienen incluidas en la API de Spark que son contribuciones de terceros y que incluyen otras técnicas como el popular DBScan o la integración con TensorFlow, del que hemos hablado anteriormente.

*** http://bit.ly/2lrObEq**
**** http://bit.ly/2lrObEq**

Mahout también implementa el algoritmo *K-means*.* Además, implementa el algoritmo Canopy** de preclustering y agrupamiento espectral.***

* http://bit.ly/2q7CxU0
** http://bit.ly/2CgmRmV
*** http://bit.ly/2q0J4j2

5.4. Reducción de dimensionalidad

La reducción de dimensionalidad se basa en la reducción del número de variables que se vayan a tratar en un espacio multidimensional. Este proceso ayuda a reducir la aleatoriedad y el ruido a la hora de realizar otras operaciones de aprendizaje autónomo, puesto que se eliminan aquellas dimensiones que tengan escasa correlación con los resultados que se pretenden estudiar.

El problema de la dimensionalidad es uno de los más importantes en la minería de datos y existen varias técnicas para descartar los parámetros que tienen menos peso en nuestro análisis. Las dos técnicas implementadas son la descomposición en valores singulares (SVD, *singular-value decomposition*) y el análisis de componentes principales (PCA, *principal component analysis*), probablemente la más popular.

En ocasiones, reduciendo la dimensionalidad no solo se ahorrará en cálculos a la hora de realizar regresiones o clasificaciones, sino que incluso estas serán más precisas, al reducir ruidos del conjunto de entrenamiento.

MLlib* proporciona métodos de descomposición en valores singulares (*singular value decomposition*, SVD),** descomposición QR y análisis de componentes principales (PCA). Spark también implementa algunas técnicas para extraer características que en muchos aspectos son similares a las técnicas de reducción de dimensionalidad. Algunas de ellas son muy específicas del dominio de trabajo, como por ejemplo la Tf-idf (*term frequency - inverse document frequency*) o el Word2vec, técnicas de aplicación en la caracterización y clasificación de documentos en tareas de minería de texto.

* http://bit.ly/2EnZu8s
** http://bit.ly/2lycNdV

Por otro lado, Mahout también proporciona los métodos SVD,* descomposición QR** y PCA.*** Además, Mahout proporciona otros algoritmos como SVD estocástica**** o Lanczos.

* http://bit.ly/2Cgnzkd
** http://bit.ly/2Cjq3hu
*** http://bit.ly/2ltsgws
**** http://bit.ly/2C3Heji

5.5. Filtrado colaborativo/sistemas de recomendación

Los sistemas de recomendación tratan de predecir los gustos o las intenciones de un usuario sobre la base de sus valoraciones previas y las valoraciones de otros usuarios con gustos similares. Son muy usados en sistemas de recomendación de música, películas, libros y tiendas en línea.

Cuando se crea un perfil de gustos del usuario, se crea utilizando dos formas o métodos en la recolección de características: implícitas o explícitas. Formas ex-

plícitas podrían ser solicitar al usuario que evalúe un objeto o tema particular, o mostrarle varios temas/objetos y que escoja su preferido. Formas implícitas serían tales como guardar un registro de artículos que el usuario ha visitado en una tienda, canciones que ha escuchado, etc.

Por ejemplo, imaginaos la siguiente situación en un sistema de películas en red:

- El usuario A ha valorado como excelente *El Padrino* y como mediocre *La guerra de las galaxias*
- El usuario B ha valorado como mala *El Padrino*, como excelente *La guerra de las galaxias* y como excelente *Star Trek*
- El usuario C ha valorado como excelente *El Padrino*, como aceptable *La guerra de las galaxias* y como excelente *Casablanca*

Basándose en la afinidad del usuario A con el resto de usuarios, un sistema recomendador probablemente recomendaría *Casablanca* al usuario A.

Tanto Mahout como MLlib proveen sistemas de recomendación basados en factorización de matrices con *alternate least squares**. El objetivo de esta técnica es encontrar la información «no existente» de un elemento a partir de la información de otros elementos con características similares. La definición de estas características es la que permitirá analizar su similitud. La técnica que subyace tras el *collaborative filtering* es la factorización matricial** en sistemas recomendadores.

*** http://bit.ly/2C3HRJG**
**** http://bit.ly/2Itth7K**

Mahout también provee filtrado colaborativo basado en ítems***.

***** http://bit.ly/2IrYrfP**

Resumen

En este módulo didáctico hemos presentado los conceptos elementales de complejidad computacional, que nos permiten entender la problemática relacionada con el tratamiento de los datos masivos.

A partir de ahí, hemos introducido uno de los primeros paradigmas utilizados para lidiar con grandes volúmenes de datos. Estamos hablando de MapReduce, un paradigma que permite descomponer un problema en dos tareas básicas (el *map* y el *reduce*) que permiten ejecutarse en entornos de datos masivos, como por ejemplo en Apache Hadoop.

A partir de las limitaciones que presenta el paradigma MapReduce, hemos introducido el entorno de trabajo Apache Spark, que nace para superar las limitaciones que tiene el modelo MapReduce. En este sentido, hemos abordado el funcionamiento interno de Spark, así como su librería de aprendizaje automático (*machine learning*), conocida como MLlib.

Finalmente, hemos revisado los distintos métodos de aprendizaje automático, haciendo énfasis en los algoritmos que son soportados por las herramientas de datos masivos, concretamente haciendo referencia a las implementaciones que incorporan los entornos Apache Hadoop y Apache Spark.

Glosario

escalabilidad horizontal *f* Acción que ocurre si un sistema mejora su rendimiento al agregar más nodos al mismo.

hilo *m* Proceso que se ejecuta en paralelo o de forma asíncrona a otros procesos que forman una aplicación. *en* thread **Hyperthreading** *sust.* Tecnología de Intel que permite ejecutar programas en paralelo en un solo procesador, simulando tener dos núcleos en un solo procesador.

lenguaje ensamblador *m* Lenguaje de programación llamado de bajo nivel, formado por un conjunto de instrucciones básicas para programar microprocesadores y/o microcontroladores.

Network-On-Chip (NoC) *sust.* Subsistema de comunicación en un circuito integrado.

propiedad asociativa *f* Operación $\oplus$ que cumple la propiedad $(a \oplus b) \oplus c = a \oplus (b \oplus c)$. Por ejemplo, la suma es una operación asociativa, mientras que la resta no lo es.

thread *m* Véase **hilo**.

Bibliografía

Boehm y otros (2016). *SystemML: Declarative Machine Learning on Spark*. Actos del archivo del VLDB Endowment VLDB Endowment Hompage (vol. 9, apartado 13, págs. 1425-1436).

Gebali, F. (2011). *Algorithms and Parallel Computing*. Nueva York: John Wiley & Sons, Inc.

Hearst, M. (2003). *Untangling Text Data Mining*. Actos del ACL'99: el 37.º encuentro anual de la Asociación de Lingüística Computacional.

Julbe Lopez, F. (2016). *Big data frameworks: Frameworks para el procesamiento distribuido de datos masivos*. Barcelona: Editorial UOC.

Kamburugamuve, S.; Wickramasinghe, P.; Ekanayake, S.; Fox, G. (2017). «Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink». *The International Journal of High Performance Computing Applications* (vol. 32, Issue 1, págs. 61-73). https://doi.org/10.1177/1094342017712976

Meng, X.; Bradley, J.; Yavuz, B.; Sparks, E.; Venkataraman, S.; Liu, D; Freeman, J.; Tsai, DB.; Amde, M; Owen, S.; Xin, D.; Xin, R.; Franklin, M. J.; Zadeh, R.; Zaharia, M.; Talwalkar, A. (2015). *MLlib: Machine Learning in Apache Spark*. Databricks. arXiv:1505.06807

White, T. (2015). *Hadoop: The Definitive Guide, 4th Edition*. Boston: O'Reilly Media.

Zaharia, M.; Chambers, B. (2017). *Spark: The Definitive Guide, Big Data Processing Made Simple*. Boston: O'Reilly Media.