Anàlisi de dades massives

Tècniques fonamentals

PID_00250689

Francesc Julbe

Temps mínim de dedicació recomanat: 5 hores

Universitat Oberta de Catalunya

Índex

Introducció

Les eines i tecnologies per a l'anàlisi de dades massives (*big data*) estan en un moment d'alta ebullició. Contínuament apareixen i desapareixen eines per a donar suport a noves necessitats, alhora que les existents van incorporant noves funcionalitats. Per això, l'objectiu principal d'aquest mòdul no és enumerar les tecnologies i eines disponibles, sinó proporcionar les bases teòriques necessàries per a entendre aquest escenari complex i ser capaços d'analitzar de manera autònoma les diferents alternatives existents.

Així i tot, sí que veurem amb cert grau de detall la implementació, el funcionament i les característiques d'algunes de les eines més importants i estables dins d'aquest complex escenari, com poden ser les principals eines de l'ecosistema d'Apache Hadoop o el *framework* Apache Spark.

Iniciarem aquest mòdul revisant els conceptes bàsics de complexitat relacionats amb l'algorísmia. Veurem com es caracteritzen els algorismes seqüencials, paral·lels i els sèrie paral·lels, i també repassarem la notació *Big O* (*Big O notation*), que ens permet caracteritzar l'escalabilitat dels algorismes enfront del volum de dades d'entrada.

A continuació, veurem detalladament un dels principals models de programació en entorns de dades massives, el model MapReduce, que s'implementa en els sistemes Apache Hadoop i veurem les seves característiques bàsiques i els seus principals avantatges i inconvenients.

A l'apartat següent, estudiarem l'aproximació a l'anàlisi de dades utilitzada en l'altre gran *framework* per al processat de dades massives, Apache Spark, veient-ne les seves característiques principals, el seu funcionament i les seves diferències enfront del model MapReduce.

Finalitzarem amb una revisió dels diferents mètodes d'aprenentatge automàtic, fent èmfasi en els algorismes que són suportats per les eines de dades massives, i concretament farem referència a les implementacions que incorporen Apache Hadoop i Apache Spark.

Objectius

Als materials didàctics d'aquest mòdul trobarem les eines indispensables per a assimilar els objectius següents:

1. Comprendre quins són els diferents models de processament distribuït utilitzats en dades massives i en quin escenari és útil cadascun d'ells.
2. Descobrir els principals entorns de treball existents per al processament distribuït, com són Apache Hadoop (i el seu ecosistema) i Apache Spark.
3. Conèixer les diverses eines i serveis existents per al processament distribuït de dades.

1. Processament de dades massives

En aquest apartat discutirem la problemàtica relacionada amb el processament de grans volums de dades i les possibles solucions que es poden desenvolupar des d'un punt de vista algorítmic.

1.1. La problemàtica del processament seqüencial i el volum de dades

Les aproximacions habituals a la computació i a la realització de càlculs complexos de tipus seqüencial (no paral·lelitzables), o el que en llenguatge de computació es tradueix en l'ús d'un sol processador, s'estan convertint en un mecanisme arcaic per a la resolució de problemes complexos per diverses raons:

- El volum de dades que s'ha d'analitzar creix a un ritme molt elevat en l'anomenada «era del *big data*». Així, els problemes que tradicionalment podien solucionar-se amb un codi seqüencial ben implementat i dades emmagatzemades localment en un equip, ara són computacionalment molt costosos i ineficients, la qual cosa es tradueix en problemes irresolubles.

- Actualment els equips posseeixen processadors multinucli capaços de realitzar càlculs en paral·lel. Hauríem de ser capaços d'aprofitar el seu potencial millorant les eines de processament perquè puguin utilitzar tota la capacitat de càlcul disponible.

- A més, la capacitat de càlcul pot ser incrementada més enllà de la que proveeixen els equips individuals. Les noves tecnologies permeten combinar xarxes de computadors, la qual cosa es coneix com a escalabilitat horitzontal. L'escalabilitat horitzontal incrementa la capacitat de càlcul exponencialment i la capacitat d'emmagatzematge, augmentant diversos ordres de magnitud la capacitat de càlcul del sistema per a resoldre problemes que amb una aproximació seqüencial serien absolutament irresolubles, tant per consum de recursos disponibles com per consum de temps.

Escalabilitat horitzontal

Un sistema escala horitzontalment si en agregar-hi més nodes el rendiment millora.

Processadors multinucli

Els processadors multinucli són aquells processadors que combinen un o més microprocessadors en una sola unitat integrada o en un paquet.

No obstant això, tenint en compte que els recursos disponibles per a la resolució de problemes complexos són molt millors i eficients, aquest factor no soluciona per ell mateix el problema. Hi ha una gran quantitat de factors que cal tenir en compte perquè un algorisme pugui ser òptim quan ja s'està executant en un entorn paral·lel.

Aprofitar la millora en els recursos de computació disponibles per executar algorismes paral·lelament requereix bones implementacions d'aquests algorismes. Cal tenir en compte que si un programa dissenyat per a ser executat en un entorn monoprocesador no s'executa ràpidament en aquest entorn, encara serà més lent en un entorn multiprocessador. I és que una aplicació que no ha estat dissenyada per ser executada de manera paral·lela no podrà paral·lelitzar la seva execució encara que l'entorn li permeti (per exemple, en un entorn amb múltiples processadors).

Així, una infraestructura que executarà un algorisme de manera paral·lela i distribuïda ha de tenir en compte el nombre de processadors que s'estan utilitzant, així com l'ús que fan aquests processadors de la memòria disponible, la velocitat de la xarxa d'ordinadors que formen el clúster, etc. Des d'un punt de vista d'implementació, un algorisme pot mostrar una dependència regular o irregular entre les seves variables, recursivitat, sincronisme, etc. En qualsevol cas, és possible accelerar l'execució de l'algorisme sempre que algunes subtasques puguin executar-se simultàniament dins del flux de treball (*workflow*) complet de processos que componen l'algorisme. I és que quan pensem en el desenvolupament d'un algorisme tendim a entendre la seqüència de tasques d'una manera seqüencial, atès que en la majoria d'ocasions un algorisme és una seqüència de tasques que depenen de manera seqüencial unes de les altres. Això es tradueix en un codi no paral·lel quan aquesta aproximació es trasllada a la implementació en programari.

Clúster

En computació, un clúster és un conjunt d'ordinadors connectats entre ells que formen una xarxa de computació distribuïda.

En aquest apartat explorarem els diferents tipus d'algorismes, alguns d'ells molt utilitzats en el món de les dades massives, i també mirarem d'entendre la naturalesa del paral·lelisme per a cadascun d'ells.

1.2. Què és un algorisme?

Encara que pugui semblar una pregunta òbvia, convé concretar què és un algorisme per poder entendre quines són les seves diferents modalitats i quines solucions existeixen per poder accelerar la seva execució en entorns de dades massives; és a dir, per poder executar-los de manera paral·lela i distribuïda.

Un algorisme és la seqüència de processos que han de realitzar-se per resoldre un problema amb un nombre finit de passos. Alguns d'aquests processos poden tenir relacions entre ells i uns altres poden ser totalment independents. La comprensió d'aquest fet és crucial per a entendre com podem paral·lelitzar el nostre algorisme per, posteriorment, implementar-lo i executar-lo sobre un gran volum de dades.

Així, els components d'un algorisme són:

1) tasques;

2) dependències entre les tasques (les sortides, o *outputs*, d'una tasca són les entrades, o *inputs*, de la següent);

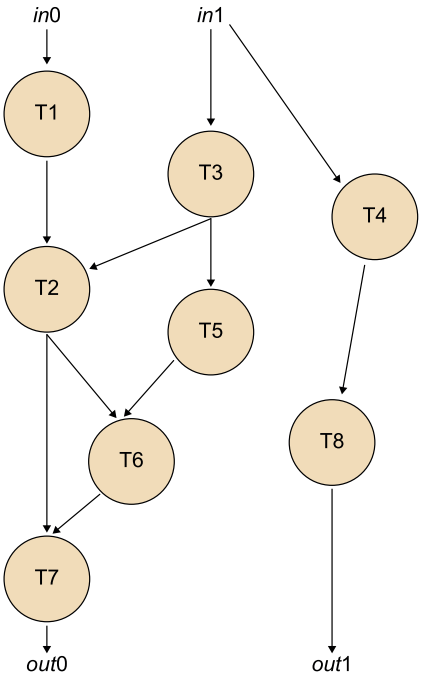
3) entrades necessàries per a l'algorisme;

4) resultat o sortides que generarà l'algorisme.

Aquestes dependències poden descriure's mitjançant un graf dirigit (en anglès, *direct graph*, DG) en què el terme *dirigit* indica que hi ha un ordre en la seqüència de dependències i, per tant, que és necessària la sortida de certa tasca per a iniciar altres tasques dependents.

Un DG és un graf en el qual els nodes representen tasques i els arcs (o arestes) són les dependències entre tasques, tal com s'han definit anteriorment. La figura 1 mostra un exemple d'un algorisme representat pel seu DG.

Figura 1. Representació DG d'un algorisme amb dues entrades *in0* i *in1*, dues sortides *out0* i *out1*, diverses tasques *Ti* i les seves dependències



Un algorisme també pot representar-se mitjançant l'anomenada **matriu d'adjacència**, molt utilitzada en teoria de grafs.

La matriu d'adjacència $M_{N\times N}$ és una matriu d'elements, $N \times N$ en què N és el nombre de tasques. Tal com es mostra a continuació, el valor 1 indica si existeix una dependència entre les tasques corresponents (podrien ser valors diferents d'1, la qual cosa atorgaria pesos diferents a aquestes dependències).

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

En aquesta matriu podem identificar les entrades fàcilment, ja que les seves columnes contenen tots els valors iguals a 0. D'una manera similar, les sortides es poden identificar a partir de les files associades, que presenten tots els valors iguals a 0. La resta de nodes intermedis presenten valors diferents de 0 en les seves files i columnes.

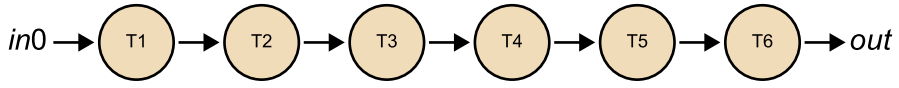
A partir del tipus de dependència entre els processos, podem classificar els algorismes en seqüencials, paral·lels o sèrie paral·lels.

1.2.1. Algorismes seqüencials

És l'algorisme més simple i pot entendre's com aquell en el qual cada tasca ha de finalitzar abans de començar la següent, que depèn del resultat de l'anterior.

Un exemple seria un càlcul en sèrie, en la qual cada element és l'anterior més una quantitat donada. El càlcul de la sèrie de Fibonacci, en què cada element és la suma dels dos anteriors, pot exemplificar aquest cas. La figura 2 mostra el DG d'un algorisme seqüencial.

Figura 2. Representació DG d'un algorisme seqüencial amb una entrada *in0*, una sortida *out0*, les tasques *Ti* i les seves dependències



Un exemple d'implementació de la sèrie de Fibonacci en llenguatge de programació Java s'inclou a continuació:

```
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;
}
```

Només imposant la condició pels dos primers nombres poden calcular-se tots els valors de la sèrie fins a un límit especificat. En aquest cas es resol d'una manera elegant un algorisme netament seqüencial amb l'ús de patrons recursius, molt habituals en el desenvolupament d'algorismes seqüencials.

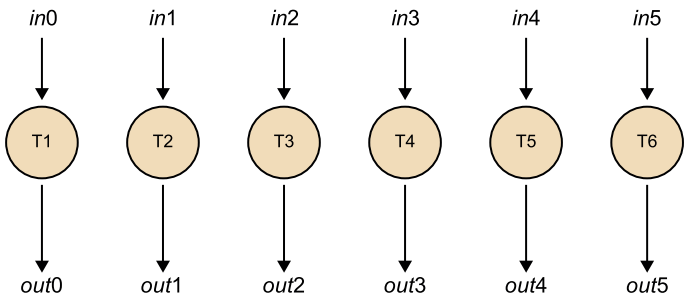
1.2.2. Algorismes paral·lels

A diferència dels anteriors, als algorismes paral·lels les tasques de processament són totes completament independents les unes de les altres, i en cap moment mostren una correlació o dependència.

A tall d'exemple, un algorisme purament paral·lel seria aquell que realitza un processament per a dades segmentades sense realitzar cap tipus d'agregació en finalitzar els processos, com la suma d'elements d'un tipus A i la suma d'elements d'un tipus B, donant com a resultat les dues sumes.

A la figura 3 veiem com cada tasca, que podria ser la suma d'elements segmentats $\{A,B,\ldots,F\}$, es realitza independentment de les altres tasques.

Figura 3. Representació DG d'un algorisme paral·lel amb una entrada *in0*, una sortida *out0* i diverses tasques *Ti*

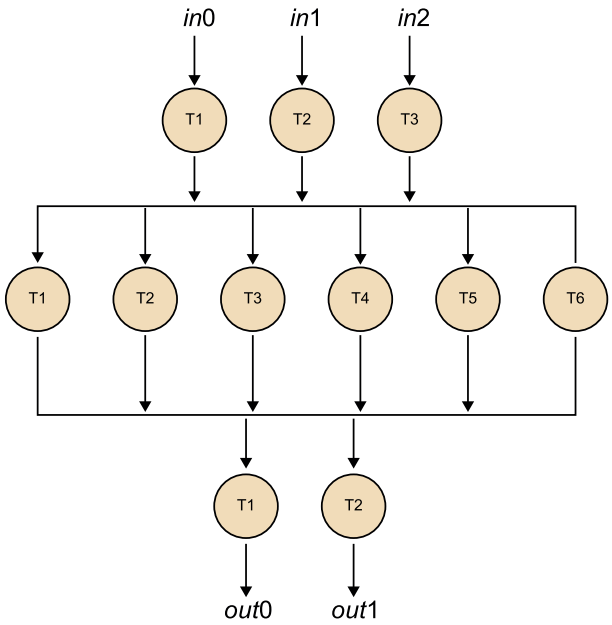


1.2.3. Algorismes sèrie paral·lels

A partir dels dos tipus d'algorismes ja descrits, podem introduir un tercer tipus, que es compon de tasques executades de manera paral·lela i tasques executades de manera seqüencial.

Així, certes tasques es descomponen en subtasques independents entre elles i, després de la seva finalització, s'agrupen els resultats de manera síncrona. Si és necessari, es torna a dividir l'execució en noves subtasques d'execució en paral·lel, tal com mostra la figura 4.

Figura 4. Algorisme amb conjunts de diferents tasques executades en paral·lel en diferents cicles seqüencials

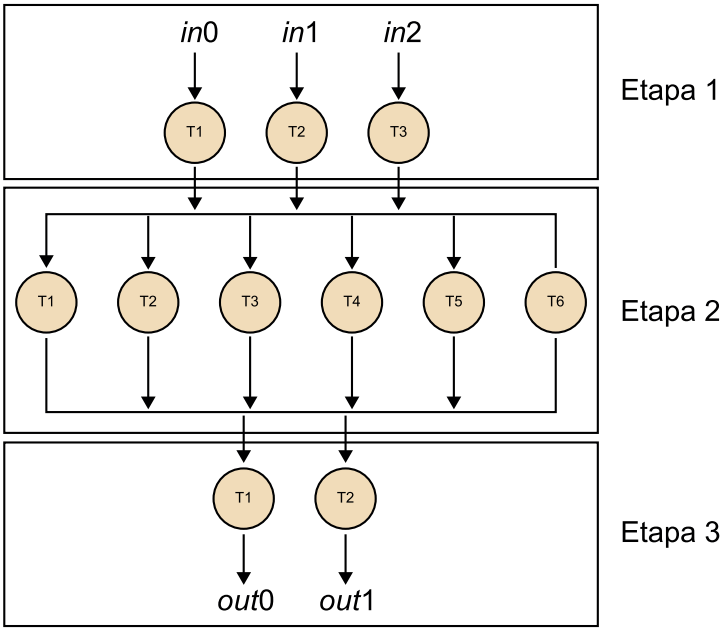


Podem afegir un nivell d'abstracció a aquest tipus d'algorisme, en el qual diverses tasques que s'executen en paral·lel componen una etapa (*stage*) el resultat de la qual és l'entrada per a un altre conjunt de tasques que s'executaran en paral·lel, formant una altra etapa. Així, les tasques individuals s'executen de manera paral·lela, però les etapes s'executen de manera seqüencial, la qual cosa acaba generant l'anomenat *pipeline*. La figura 5 mostra un possible esquema d'aquesta estructura.

Més endavant en aquest mòdul tornarem a analitzar aquest tipus d'algorismes, ja que constitueixen la metodologia de processament d'Apache Spark.

Un exemple d'aquests algorismes seria el paradigma MapReduce. Les tasques Map són independents entre elles i el seu resultat s'agrupa en un conjunt (típicament menor) de tasques *reduce*, seguint la metodologia «divideix i venceràs» (*divide & conquer*).

Figura 5. Algorisme sèrie paral·lel amb etapes seqüencials



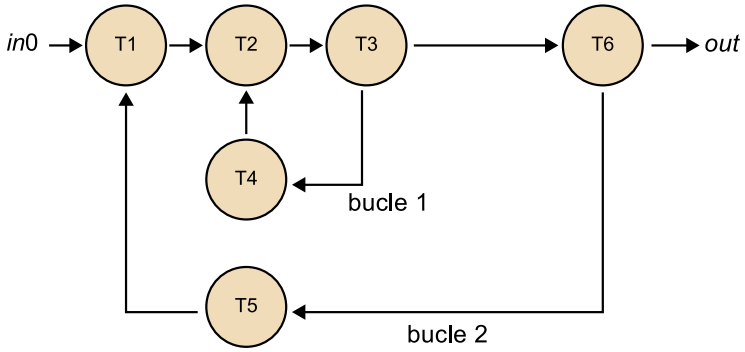
1.2.4. Altres algorismes

Hi ha algorismes que no poden ser classificats en cap de les categories anteriors, ja que el seu DG no segueix cap patró. En aquest cas, podem distingir dos tipus d'algorismes:

- *Directed acyclic graph* (DAG): al DAG no hi ha cicles i les tasques s'executen amb una ordenació que defineix una «direcció», amb un principi i final; no obstant això, no hi ha un sincronisme clar entre tasques. La figura 1 és un exemple de DAG. Es tracta d'un tipus d'algorisme important perquè és el graf que representa la metodologia d'execució de tasques d'Apache Spark, que s'estudiarà més endavant.

- *Directed cyclic graph* (DCG): són aquells algorismes que contenen cicles en el seu processament. És habitual trobar implementacions en sistemes de processament de senyals, en el camp de les telecomunicacions o de la compressió de dades, en què hi ha etapes de predicció i correcció. La figura 6 mostra un exemple d'aquesta estructura.

Figura 6. Representació d'un algorisme amb dependències cícliques



1.3. Eficiència en la implementació d'algorismes

L'ús d'entorns de computació distribuïts requereix una implementació en programari adequada dels algorismes perquè pugui ser executada de manera eficient. Executar un algorisme en un entorn distribuït requereix una anàlisi prèvia del disseny d'aquest algorisme per comprendre l'entorn en el qual s'executarà.

Les tasques que componen aquest algorisme han de descompondre's en subtasques més petites per poder executar-se en paral·lel i, moltes vegades, l'execució d'aquestes subtasques ha de ser síncrona, la qual cosa permetrà l'agrupació dels resultats finals de cada tasca.

Per tant, paral·lelitzar algorismes està fortament relacionat amb una arquitectura que pugui executar-los de manera eficient. No és possible paral·lelitzar l'execució d'un algorisme seqüencial perquè el processador no sabrà quines tasques pot distribuir entre els seus nuclis.

1.3.1. Notació *Big O*

La notació *Big O* permet quantificar la complexitat d'un algorisme a partir del volum de dades d'entrada, que generalment s'identifica amb la lletra n. Vegem alguns dels casos més rellevants:

- $O(1)$: és el cas en el qual un algorisme requereix un temps constant per executar-se, independentment del volum de dades que cal tractar. Sol ser bastant inusual. Un exemple seria accedir a un element d'un vector.

- $O(n)$: en aquest cas, la complexitat de l'algorisme s'incrementa de manera lineal amb el volum de l'entrada de dades. És a dir, el temps de càlcul requerit augmenta de forma lineal respecte a la quantitat de dades que ha de tractar. Per exemple, escollir el valor màxim d'un vector de valors tindria aquesta complexitat, ja que s'ha de recórrer una vegada tot el vector per poder determinar quin és el valor màxim. Generalment, càlculs que impliquin un bucle `for` que iteri sobre els n elements d'entrada sol tenir aquesta complexitat.

- $O(log(n))$: també coneguda com a «complexitat logarítmica». Aquest tipus d'algorismes són aquells en els quals el temps no s'incrementa de forma lineal, sinó que aquest increment és menor a mesura que augmenta el conjunt de dades. Generalment, aquest tipus de complexitat es troba en algorismes de cerca en arbres binaris, on no s'analitzen totes les dades de l'arbre, només els continguts en certes branques. D'aquesta manera, la profunditat de l'arbre que haurem d'analitzar no creix proporcionalment al volum de les entrades.

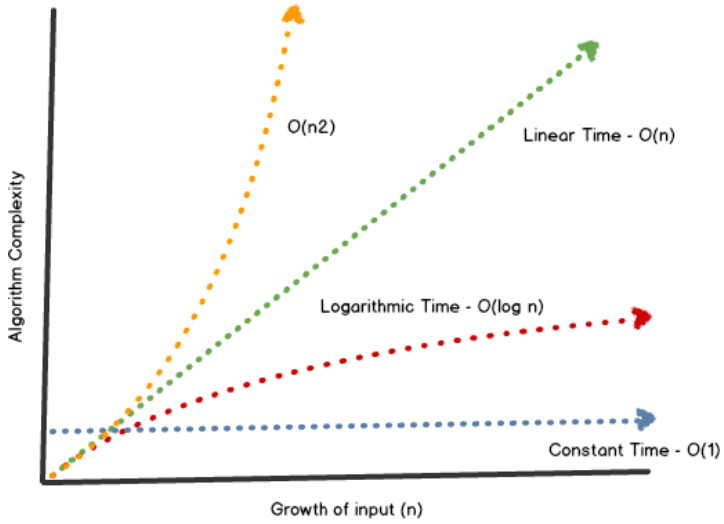
- $O(n^2)$: aquest cas és conegut com a complexitat quadràtica. Indica que el temps de càlcul requerit creixerà exponencialment a partir del valor d'n,

és a dir, del volum de dades d'entrada. Generalment, càlculs que impliquin un doble bucle `for` que iteri sobre els n elements d'entrada sol tenir aquesta complexitat.

Els algorismes amb aquest tipus de complexitat no solen ser aplicables en dades massives, ja que en aquests casos el valor d'n sol ser molt gran. Per tant, no és computacionalment possible resoldre problemes amb dades massives emprant algorismes d'ordre $O(n^2)$ o superior, com per exemple «cúbics» $O(n^3)$.

La figura 7 mostra, aproximadament, com creix el temps de càlcul necessari segons el volum de dades de l'entrada.

Figura 7. Representació de la complexitat dels algorismes segons la notació *Big O*



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

Un algorisme pot paral·lelitzar-se i d'aquesta manera millorar el seu temps d'execució, aprofitar millor els recursos disponibles, etc. No obstant això, l'eficiència en l'execució d'un algorisme també depèn en gran mesura de la seva implementació. Un exemple clar i fàcil d'entendre són els diferents mecanismes per a ordenar elements en una col·lecció d'elements desordenats.

Exemple: ordenació de valors d'una col·lecció

La implementació més habitual per a ordenar un vector d'elements n de major a menor consisteix a iterar dues vegades el mateix vector, utilitzant-ne una còpia i comparant cada valor amb el següent. Si és major intercanvien les seves posicions, si no es deixen tal com estan. És un algorisme anomenat «ordenament de bombolla» (*bubble sort*).* Malgrat que la seva implementació és molt senzilla, també és molt ineficient, ja que ha de recórrer el vector n vegades. En el cas òptim (ja ordenat), la seva complexitat és $O(n)$, però en el pitjor dels casos és $O(n^2)$. L'ordenament de bombolla, a més, és més difícil de paral·lelitzar a causa de la seva implementació seqüencial, ja que recorre el vector de manera ordenada per a cada element, per la qual cosa és doblement ineficient.

* http://bit.ly/1bfggqp

Hi ha altres mecanismes per a ordenar vectors molt més eficients, com el *Quicksort*,** probablement un dels més eficients i ràpids. Aquest mecanisme es basa en el divideix i venceràs. El mecanisme de l'algorisme no és trivial i la seva explicació està fora dels objectius d'aquest mòdul; així i tot, és interessant indicar que aquest algorisme ofereix una complexitat de l'ordre d'$O(nlog(n))$. A més, en aproximar el problema dividint el vector en petits problemes és més fàcilment paral·lelitzable.

** https://goo.gl/chh3so

Un altre algorisme d'ordenament molt popular és l'«ordenament per barreja» (*merge sort*),* que divideix el vector en dues parts de manera iterativa, fins que només queden vectors de dos elements, fàcilment ordenables. Aquest problema també és fàcilment paral·lelitzable, ja que les operacions en cada vector acaben sent independents entre elles. Aquests exemples d'ordenació serveixen per a entendre que un mateix algorisme pot implementar-se de maneres diferents i, a més, el mecanisme adequat pot permetre la paral·lelització, la qual cosa millora la seva eficiència final.

*** https://goo.gl/kkmFj7**

1.3.2. Computació paral·lela

En un entorn de computació paral·lela existeixen diversos factors rellevants que configuren la seva capacitat de paral·lelització.

Entorns multiprocessador

A nivell d'un sol computador, alguns dels factors més importants són:

1) Nombre de processadors i **nombre de nuclis** per processador.

2) Comunicació entre processadors: busos de comunicacions en origen, actualment mitjançant el sistema de comunicacions Network On-Xip (NoC).

Network-On-Chip (NoC)

Network-On-Chip (NoC) és un subsistema de comunicacions en un circuit integrat.

3) Memòria: les aplicacions poden particionarse de manera que diversos processos comparteixin memòria (*threads* o fils), o de manera que cada procés gestioni la seva memòria.

Els sistemes de memòria compartida ofereixen la possibilitat que tots els processadors comparteixin la memòria disponible. Per tant, un canvi en la memòria és visible per tots ells, amb la qual cosa comparteixen el mateix espai de treball (per exemple, variables).

Existeixen arquitectures de memòria distribuïda entre processadors, en la qual cadascun d'ells té el seu espai de memòria reservat i exclusiu. Aquestes arquitectures són escalables, però més complexes des d'un punt de vista de gestió de memòria.

4) Accés a dades: cada procés pot accedir a una partició de les dades o, per contra, pot haver-hi concurrència en l'accés a dades, la qual cosa implica una capacitat de sincronització de l'algorisme, incloent-hi mecanismes d'integritat de dades.

Entorns multinode

En xarxes de múltiples ordinadors, alguns dels factors més importants són els següents:

1) En un entorn de **múltiples ordinadors** és necessari disposar de comunicacions d'altes prestacions entre ells, mitjançant xarxes d'alta velocitat, com per exemple Gigabit Ethernet.

2) Des del punt de vista de l'arquitectura de memòria, és una arquitectura híbrida. Un ordinador o node amb múltiples processadors és una màquina de memòria compartida, mentre que cada node té la memòria exclusiva i no compartida amb la resta de nodes, la qual cosa fa que sigui una arquitectura de memòria distribuïda. Actualment, els nous entorns de dades massives simulen les arquitectures com a sistemes de memòria compartida mitjançant programari.

Unitat de processament gràfic (GPU)

La unitat de processament gràfic* (*graphics processor unit*, GPU) és un coprocessador dedicat al processament de gràfics o operacions de coma flotant per alleugerir la càrrega de treball del processador central. Típicament les GPU estan presents en les targetes gràfiques dels ordinadors i la seva dedicació és exclusiva de tasques altament paral·lelitzables de processament gràfic, fet que incrementa exponencialment la capacitat de processament del sistema.

* https://goo.gl/fauqgt

Tot i que inicialment les GPU es van utilitzar per a processar gràfics (principalment vèrtexs i píxels), actualment la seva capacitat de càlcul s'utilitza en altres aplicacions en el que s'ha anomenat *general-purpose computing on graphics processing units*** (GPGPU). Són especialment rellevants en aplicacions de l'àmbit de xarxes neuronals *deep learning* i, en general, en entorns d'altes prestacions (*high performance computing*, HPC), de les quals parlarem amb més detall posteriorment.

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

En un entorn amb GPU, el sistema trasllada les parts de l'aplicació amb més càrrega computacional i més altament paral·lelitzables a la GPU, deixant la resta del codi executant-se a la CPU. Cal tenir en compte que una CPU pot tenir diversos nuclis optimitzats (quatre o vuit simulats amb tecnologia *hyperthreading*), mentre que una GPU té milers de nuclis dissenyats per a executar tasques altament paral·lelitzables. A causa de les diferències fonamentals entre les arquitectures de la GPU i la CPU, no qualsevol problema es pot beneficiar d'una implementació mitjançant GPU.

Hyperthreading

Hyperthreading és una tecnologia d'Intel que permet executar programes en paral·lel en un sol processador, simulant l'efecte de tenir realment dos nuclis en un sol processador.

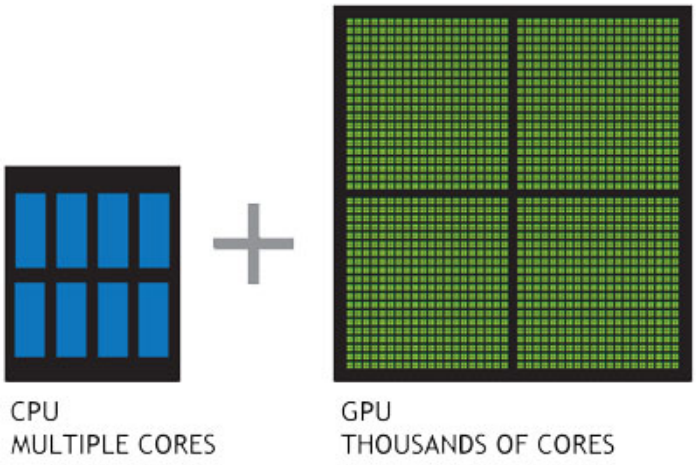
Malgrat que qualsevol algorisme que sigui implementable en una CPU pot ser-ho també en una GPU, ambdues implementacions poden no ser igual d'eficients en les dues arquitectures. En aquest sentit, els algorismes amb un altíssim grau de paral·lelisme, anomenats *embarrassingly parallel* (vergonyosament paral·lelitzables, traduït literalment), que no tenen la necessitat d'estructures de dades complexes i amb un elevat nivell de càlcul aritmètic, són els que obtenen més beneficis de la seva implementació a GPU.

Inicialment la programació de GPU es feia amb llenguatges de baix nivell, com el llenguatge assemblador o *assembler* o altres més específics per al processament gràfic. No obstant això, actualment el llenguatge més utilitzat és el *compute unified device architecture* (CUDA).* CUDA no és un llenguatge en si,

* https://goo.gl/ci8dAg

sinó un conjunt d'extensions a C i C++ que permet la paral·lelització de certes instruccions per a ser executades a la GPU del sistema.

Figura 8. Nombre de nuclis en un processador de propòsit general (COU) enfront d'una GPU



Font: Nvidia

Llenguatge d'assemblador

El llenguatge d'assemblador és un llenguatge de programació anomenat de baix nivell, format per un conjunt d'instruccions bàsiques per a programar microprocessadors i/o microcontroladors.

A continuació, es mostra un fragment de codi estàndard en C:

```
void saxpy_serial(int n, float a, float *x, float *i)
{
        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, i);
```

Seguidament, es mostra com afegint alguns paràmetres al codi es paral·lelitza el problema presentat al fragment de codi 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 aquest exemple* cada càlcul dins del bucle no depèn de cap altre càlcul anterior, la qual cosa el converteix en un cas canònic d'increment de rendiment per paral·lelisme a GPU. No obstant això, un algorisme que tingués dependèn-

*** Podeu veure l'exemple complet a https://goo.gl/XaA99U.**

cies entre tasques no seria fàcilment resoluble amb CUDA i GPU, a causa de les correlacions entre tasques.

Com a inconvenients cal indicar que les GPU tenen latències altes, la qual cosa presenta un problema en tasques amb un curt temps d'execució. Per contra, són molt eficients en tasques d'elevada càrrega de treball, amb càlculs intensius i amb un llarg temps d'execució.

1.3.3. Exemples d'algorismes i paral·lelització

A continuació, mostrarem alguns exemples de com un problema complex pot particionar-se per a ser executat en un entorn de computació paral·lela.

Exemple: multiplicació de matrius

Ara presentarem un problema que anirem revisant al llarg del mòdul: la multiplicació de matrius.

Dues matrius A i B es poden multiplicar si el nombre de files de la primera (A) és igual al nombre de columnes de la segona (B). El primer element de la primera fila de la matriu A es multiplica pel primer element de la primera columna de B. El seu resultat se suma a la multiplicació del segon element de la primera fila d'A pel segon element de la primera columna de B, i així successivament.

Suposem que les matrius A i B són les presentades a continuació:

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

Llavors el producte AB es representa de la manera següent:

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

I el producte BA, tal com mostrem a continuació. Noteu la diferència en la grandària de la matriu resultant:

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

Una implementació amb codi seqüencial del producte de matrius podria ser el següent:

```
int[][] a; /* matriu A de NxN elements */
int[][] b; /* matriu B de NxN elements */
int[][] c; /* matriu C de NxN elements */

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];
      }
    }
  }
```

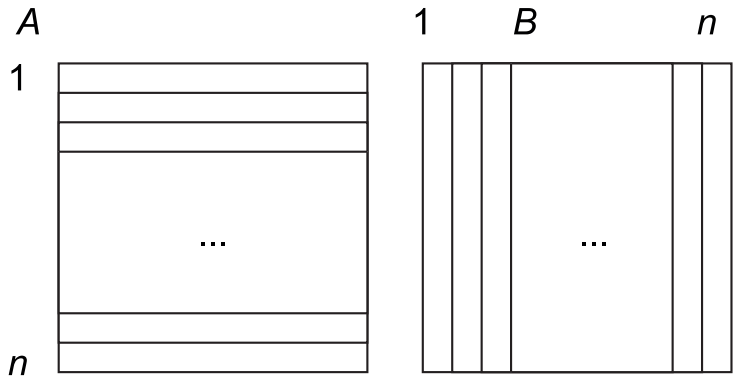
Tal com es pot veure al codi anterior, en utilitzar tres bucles niats la implementació presenta una complexitat $O(n^3)$.

No obstant això, el producte de cada parell d'elements és independent de les altres operacions. A més, la suma i la multiplicació són operacions commutatives i associatives, de manera que l'algorisme de multiplicació de matrius ofereix diferents aproximacions de paral·lelització.

Com a regla principal, és important que l'estratègia de paral·lelització no resulti en un problema de més complexitat que la implementació seqüencial. Així, la complexitat no ha de ser més gran que $O(n^3)$.

Una manera de resoldre el problema és la divisió del producte en files i columnes.

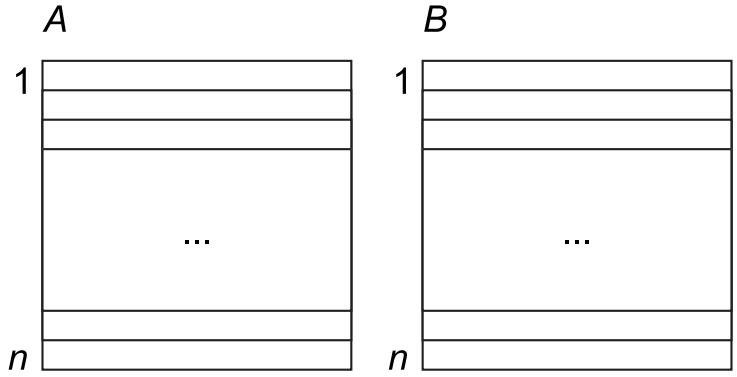
Figura 9. Divisió de tasques per files i columnes al producte de matrius



En aquest cas, existeixen n^2 tasques, que són la multiplicació dels elements de cada fila pels elements de cada columna. Cadascuna d'aquestes tasques és independent de les altres, i són un bon exemple d'algorisme paral·lelitzable.

També podríem distribuir la càrrega per fila en ambdues matrius, tal com es mostra a la figura 10.

Figura 10. Producte de matrius dividint les tasques per files d'ambdues matrius.



En aquest cas, es multiplica cada fila pel primer element de cada columna de la matriu *B*, per la qual cosa va generant resultats parcials fins que l'algorisme s'ha completat. És menys intuïtiu que el cas anterior, però igualment paral·lelitzable.

Algorismes no paral·lelitzables

Existeixen algorismes que per la seva naturalesa seqüencial no són fàcilment paral·lelitzables. No obstant això, s'han desenvolupat estratègies per paral·lelitzar algorismes que *a priori* tenen un caràcter seqüencial. L'exemple ja presentat de la sèrie de Fibonacci és un clar candidat d'algorisme seqüencial no fàcilment paral·lelitzable, ja que existeixen dependències amb valors calculats anteriorment, de manera que la tasca i depèn de $(i-1)$. No obstant això, hi ha implementacions paral·leles de molts algorismes, afegint etapes de sincronització i agregació de resultats.

2. Apache Hadoop i MapReduce

MapReduce és un model de programació per donar suport a la computació paral·lela sobre grans conjunts de dades en clústers d'ordinadors. MapReduce ha estat adoptat mundialment, ja que existeix una implementació de codi obert denominada Hadoop. El seu desenvolupament va ser liderat inicialment per Yahoo! i actualment el realitza el projecte Apache.

2.1. Paradigma MapReduce

El nom de l'entorn està inspirat en els noms de dos mètodes o funcions importants en programació funcional: *map* i *reduce*.

2.1.1. Abstracció

Les dues tasques principals que duu a terme aquest model són:

- ***Map***: aquesta tasca és l'encarregada d'«etiquetar» o «classificar» les dades que es llegeixen des de disc, típicament d'HDFS, en funció del processament que estiguem realitzant.
- ***Reduce***: aquesta tasca és la responsable d'agregar les dades etiquetades per la tasca *map*. Pot dividir-se en dues etapes, la *shuffle* i el mateix *reduce* o agregat.

Tot l'intercanvi de dades entre tasques utilitza estructures anomenades parelles <clau, valor> (<*key*, *value*> en anglès) o tuples.

A l'exemple següent mostrem, des d'un punt de vista funcional, en què consisteix una tasca MapReduce.

Exemple conceptual de procés MapReduce

Imaginem que tenim tres fitxers amb les dades següents:

Fitxer 1:

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

Fitxer 2:

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

Fitxer 3:

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

Donats aquests fitxers, podríem preguntar-nos: quantes persones hi ha a cada ciutat?

Per respondre a aquesta pregunta definirem una tasca *map*, que llegirà les files de cada fitxer i «etiquetarà» cadascuna en funció de la ciutat que apareix, que és el tercer camp de cadascuna de les línies que segueixen l'estructura: nom, edat, ciutat.

Per a cada línia, ens retornarà una tupla de la forma: <ciutat, quantitat>.

Al final de l'execució de la tasca *map*, a cada fitxer tindrem:

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

Com veiem, les nostres tuples estan formades per una clau (*key*), que és el nom de la ciutat, i un valor (*value*), que representa el nombre de vegades que apareix a la línia, que sempre és 1.

La tasca *reduce* s'ocuparà d'agrupar els resultats segons el valor de la clau. Així, recorrerà totes les tuples agregant els resultats per una mateixa clau i retornarà el següent:

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

Òbviament, en aquest exemple l'operació no requeria un entorn distribuït. No obstant això, en un entorn amb milions de registres de persones una operació d'aquestes característiques seria molt efectiva.

Connectant amb el que s'ha descrit en subapartats anteriors, una operació MapReduce és un processament en lots (*batch*), ja que recorre d'un sol cop totes les dades disponibles i no retorna resultats fins que ha finalitzat.

És important tenir en compte que les funcions d'agregació (*reducer*) han de ser commutatives i associatives.

Propietat associativa

Una operació $\oplus$ és associativa si cumpleix la propietat $(a \oplus b) \oplus c = a \oplus (b \oplus c)$. Per exemple, la suma és una operació associativa, mentre que la resta no ho és.

2.1.2. Implementació MapReduce a Hadoop

Des d'un punt de vista de procés en entorn Hadoop, descriurem quins són els components d'una tasca MapReduce:

1) Generalment, un clúster està format per diversos nodes, controlats per un node mestre. Cada node emmagatzema fitxers localment i són accessibles mitjançant el sistema de fitxers HDFS (típicament és HDFS, tot i que no és un requisit necessari). Els fitxers es distribueixen homogèniament en tots els nodes. L'execució d'un programa MapReduce implica l'execució de les tasques de `map()` en molts o tots els nodes del clúster. Cadascuna d'aquestes tasques és equivalent, és a dir, no hi ha tasques `map()` específiques o diferents de les altres. L'objectiu és que qualsevol d'aquestes tasques pugui processar qualsevol fitxer que existeixi al clúster.

2) Quan la fase de *map* ha finalitzat, els resultats intermedis (tuples <clau, valor>) han d'intercanviar-se entre les màquines per enviar tots els valors amb

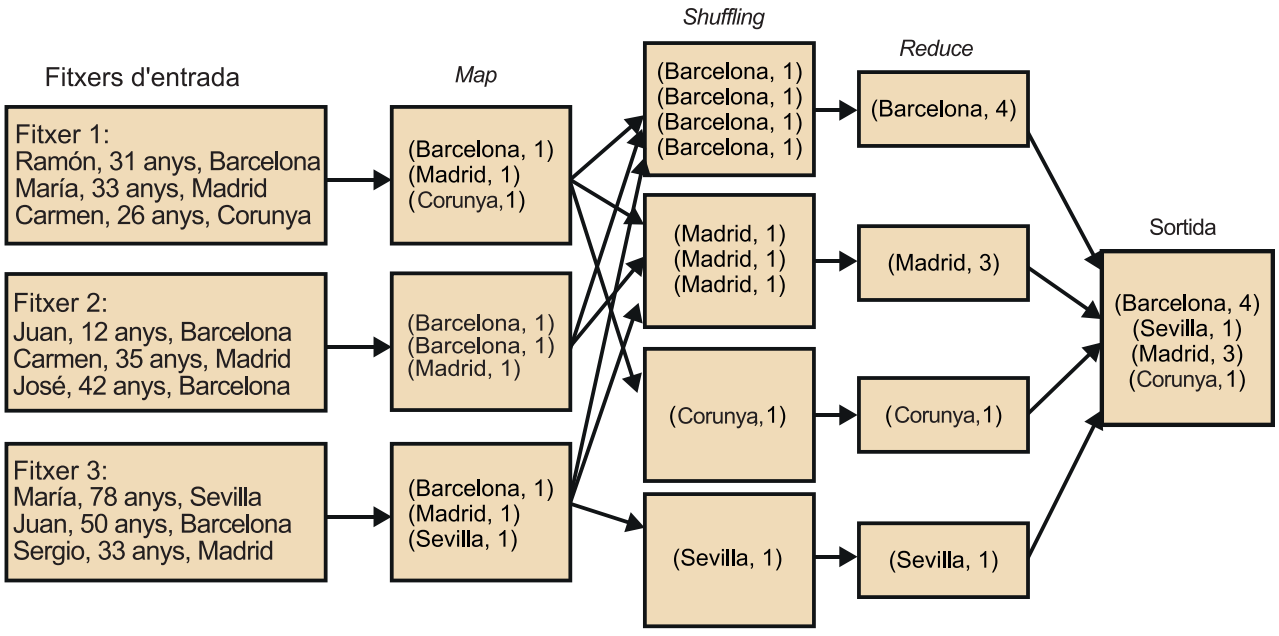
la mateixa clau a un sol *reduce*. Les tasques *reduce* també s'executen als mateixos nodes que les *map*; aquest és l'únic intercanvi d'informació entre tasques (ni les tasques *map* ni les *reduce* intercanvien informació entre elles, ni l'usuari pot interferir en aquest procés d'intercanvi d'informació). Aquest és un component important de la fiabilitat d'una tasca MapReduce, ja que si un node falla, reinicia les seves tasques sense que l'estat del processament en altres nodes depengui d'ell.

No obstant això, en el procés d'intercanvi d'informació entre *map* i *reduce*, podem introduir dos conceptes nous: partició (*partition*) i *shuffle*.

Quan una tasca *map* ha finalitzat en alguns nodes, altres nodes encara poden estar realitzant tasques del mateix tipus. No obstant això, l'intercanvi de dades intermèdies ja s'inicia i les dades s'envien a la tasca *reduce* corresponent. El procés de moure dades intermèdies des dels *mappers* als *reducers* es diu *shuffle*. Es creen subconjunts de dades, agrupats per <clau>, la qual cosa dona lloc a les particions, que són les entrades a les tasques *reduce*. Tots els valors per a una mateixa clau són agregats o reduïts junts, indistintament de la seva tasca *mapper*. Així, tots els *mappers* han de posar-se d'acord sobre on enviar les diferents peces de dades intermèdies.

Quan el procés *reduce* ja s'ha completat, agregant totes les dades, es guarden de nou en disc. La figura 11 ens mostra gràficament l'exemple descrit.

Figura 11. Diagrama de flux amb les etapes *map* i *reduce* de l'exemple descrit



2.1.3. Limitacions de Hadoop

Hadoop és la implementació del paradigma MapReduce que solucionava la problemàtica del càlcul distribuït utilitzant les arquitectures predominants fa una dècada. No obstant això, actualment presenta limitacions importants:

- És complicat aplicar el paradigma MapReduce en molts casos, ja que una tasca ha de descompondre's en subtasques *map* i *reduce*. En alguns casos, aquesta descomposició no és fàcil.

- És lent. Una tasca pot requerir l'execució de diverses etapes MapReduce i, en aquest cas, l'intercanvi de dades entre etapes es durà a terme utilitzant fitxers, fent un ús intensiu de lectura i escriptura en disc.

- Hadoop és un entorn essencialment basat en Java que requereix la descomposició en tasques *map* i *reduce*. No obstant això, aquesta aproximació al processament de dades resultava molt dificultosa per al científic de dades sense coneixements de Java, de manera que han aparegut diverses eines per flexibilitzar i abstreure el científic del paradigma MapReduce i la seva programació en Java. Entre aquestes eines podem citar:

  – **Apache Flume*** per a emmagatzemar dades en *streaming* cap a HDFS.

  – **Apache Sqoop**** per a l'intercanvi de dades entre bases de dades relacionals i HDFS.

  – **Apache Hive***** o **Apache Impala****** per a realitzar consultes del tipus SQL sobre dades emmagatzemades a HDFS.

  – **Apache Pig******* per a definir cadenes de processament sobre dades distribuïdes.

  – **Apache Giraph*** per a l'anàlisi de grafs.

  – **Apache Mahout**** per a desenvolupar tasques d'aprenentatge automàtic.

  – **Apache HBase***** com a sistema d'emmagatzematge NoSQL.

  – **Apache Oozie****** com a gestor de fluxos de treball.

* https://flume.apache.org/

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

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

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

* http://giraph.apache.org

** http://mahout.apache.org

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

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

Així, el científic de dades ha de conèixer un ampli conjunt d'eines, cadascuna amb propietats diferents, llenguatges diferents i molt poca (o cap) compatibilitat entre elles.

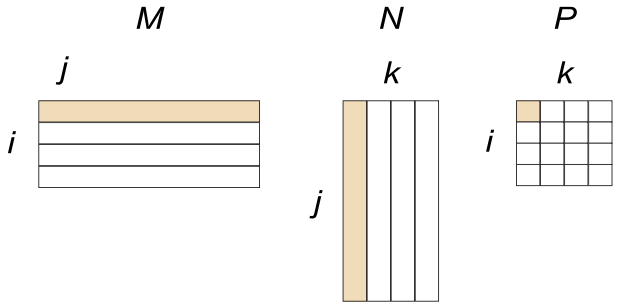
2.2. Exemple d'aplicació de MapReduce

A continuació, recuperem un exercici introduït anteriorment: la multiplicació de matrius.

Exemple de multiplicació de matrius amb MapReduce

Ara resoldrem la multiplicació de dues matrius *M* i *N* que dona com resultat la matriu *P* utilitzant el model MapReduce. Prenem com a exemple les dues matrius *M* i *N*, mostrades a la figura 12.

Figura 12. Representació abstracta del producte de dues matrius



$$M_{3\times4} = \begin{bmatrix} 3 & 4 & 3 & 2 \\ 1 & 2 & 1 & 3 \\ 5 & 4 & 3 & 1 \end{bmatrix}, N_{4\times3} = \begin{bmatrix} 6 & 4 & 3 \\ 5 & 4 & 3 \\ 3 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix}$$

Tal com s'ha descrit amb anterioritat, el procés funciona de manera que cada element de la primera fila de la primera matriu *M* es multiplica per cada element de la primera columna *N*. El resultat numèric de l'exercici és el següent:

$$M_{3\times4} \cdot N_{4\times3} = P_{3\times3} = \begin{bmatrix} 51 & 33 & 26 \\ 25 & 16 & 13 \\ 61 & 40 & 31 \end{bmatrix}$$

És un procés altament paral·lelitzable, ja que cada càlcul és independent de l'altre, tant la multiplicació entre files i columnes com entre elements de cadascuna, en ser una operació commutativa i associativa.

L'algorisme de multiplicació ha de compondre's d'etapes *map* i *reduce*, així que s'ha de buscar l'estratègia necessària per a poder realitzar els càlculs adequant-los a aquestes dues etapes i retornant el resultat desitjat.

Map

L'etapa *map* ha de formar estructures amb la forma <clau,valor>, que ens permetin agrupar els resultats en l'etapa *reduce*.

Per entendre la formulació de l'exercici hem de definir primer la notació, especialment referent als índexs:

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

on:

- i és el nombre de files a la matriu M;
- j és el nombre de columnes de la matriu M (per poder realitzar el producte de matrius, ha de coincidir amb el nombre de files de la matriu N);
- k és el nombre de columnes de la matriu N.

Per a cada element $m_{i,j}$ de la matriu M, crearem parelles com:

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

En aquest cas, k pren valors des d'1 fins al nombre de columnes.

Per a cada element $n_{j,k}$ de la matriu N, crearem parelles com:

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

En aquest cas, i pren valors des d'1 fins al nombre de files de la matriu.

Aplicant aquesta fórmula a l'exemple, trobem les parelles següents de valors per a la matriu M:

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

Aquest serà el multiplicador per a cada primer element k de les columnes de la matriu N, que serà:

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

Veiem que hem creat quatre «còpies» de cada valor, ja que cadascun d'ells es multiplicarà per cada valor de la fila corresponent de la matriu N, és a dir, quatre productes diferents.

Per començar a entendre la notació, agruparem totes les expressions generades per clau, seguint la fórmula següent:

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

Numèricament queda reflectit de la manera següent:

(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 procés de *reduce* pren els valors agrupats per cada clau i ha de trobar-ne el mecanisme d'agrupació. Prenem tots els valors de cada clau i els separem en dues llistes, els corresponents a la matriu *M* i els de la matriu *N*:

Per exemple, per a la clau (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)

Així, prenem cada element ordenat pel segon índex i multipliquem cada valor pel corresponent de l'altra llista segons l'ordenació, sumant-hi el resultat.

Per a la clau (1,1), la suma dels productes d'aquests valors és:

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

El procés de *reduce* ha de repetir la mateixa operació per a totes les claus disponibles.

Podem concloure que, tot i que és un paradigma estricte, res no impedeix crear parelles de <clau, valor> complexes que el *reducer* pot processar de manera que s'aconsegueixi l'agrupació desitjada.

2.3. Apache Mahout

Apache Mahout* és una eina que proporciona algorismes d'aprenentatge automàtic distribuïts o escalables, enfocats principalment a les àrees de filtrat col·laboratiu, de clusterització i de classificació. Moltes de les implementacions fan servir la plataforma Apache Hadoop, tot i que Mahout també proporciona biblioteques de Java per a operacions matemàtiques comunes (centrades en àlgebra lineal i estadística) i col·leccions primitives de Java.

* http://mahout.apache.org

Els algorismes centrals de Mahout per la clusterització, la classificació i el filtrat col·laboratiu basat en processament per lots (*batch processing*) s'implementen sobre Apache Hadoop utilitzant el paradigma MapReduce. Per tant, es busca la manera de poder implementar els algorismes existents a les diferents àrees

de l'aprenentatge automàtic per poder ser processats en forma d'etapes *map* i *reduce*, encara que sigui de manera iterativa, emprant els resultats d'un procés com a entrades d'un segon procés, en un estructura de *pipeline* o cadenes de processos.

2.4. Conclusions

MapReduce va ser un paradigma molt nou que permetia explorar grans arxius en entorns distribuïts. És un algorisme que pot desplegar-se, en la seva implementació Hadoop, en equips de baix cost o convencionals (o també anomenat *commodity*), cosa que el va fer molt atractiu. Diferents eines que ofereixen funcionalitats diverses en l'ecosistema de dades massives com Hive o Sqoop s'han desenvolupat implementant MapReduce. No obstant això, l'algorisme presenta algunes limitacions importants:

- Utilitza un model forçat per a cert tipus d'aplicacions, fet que obliga a crear etapes addicionals *map* o *reduce* per ajustar l'aplicació al model. Pot arribar a emetre valors intermedis estranys o inútils per al resultat final, i fins i tot crear funcions *map* o *reduce* de tipus identitat, és a dir, que no són necessàries per a l'operació que es realitzarà, però que s'han de crear per adequar el càlcul al model.

- MapReduce és un procés *file-based*, la qual cosa significa que l'intercanvi de dades es realitza utilitzant fitxers. Això produeix un elevat flux de dades en lectura i escriptura de disc i afecta la seva velocitat i rendiment general si un processament concret està format per una cadena de processos MapReduce.

Per solucionar aquestes limitacions va aparèixer Apache Spark, que canvia el model d'execució de tasques i el fa més àgil i polivalent, com exposarem més endavant.

3. Apache Spark

Com hem vist a l'apartat anterior, MapReduce és un algorisme potent per a processar dades distribuïdes, altament escalable i relativament simple. No obstant això, presenta moltes febleses quant a rendiment i versatilitat per a implementar algorismes més complexos. Així, en aquest apartat explorarem quins són els entorns més utilitzats per al processament de grans volums de dades per a tasques d'anàlisi de dades i aplicació d'algorismes complexos i centrarem la nostra atenció en Apache Spark.

Apache Spark* és un entorn per al processament de dades distribuïdes que ofereix un alt rendiment i és compatible amb tots els serveis que formen part de l'ecosistema d'Apache Hadoop. Entre les seves funcionalitats principals destaquem:

* https://spark.apache.org

- **SparkSQL**: API per a treballar amb dades distribuïdes mitjançant estructures abstractes, anomenades *dataframes*, similars als *dataframes* d'R** o Pandas*** de Python.

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

- **MLlib**: Spark també ofereix una API per a realitzar tasques d'aprenentatge automàtic, de la qual parlarem amb més detall en subapartats posteriors.

- **GraphX**: API per a desenvolupar aplicacions d'anàlisi de dades basades en grafs (*graph mining*).

- **Spark Streaming**: API per a desenvolupar aplicacions de dades massives amb fluxos de dades contínues (*streaming data*).

En aquest apartat ens centrarem en la manera interna de funcionament de Spark i com paral·lelitza les seves tasques, cosa que el converteix en l'entorn més popular per a l'anàlisi en entorns de dades massives.

3.1. *Resilient distributed dataset* (RDD)

Per entendre el processament distribuït de Spark, hem de conèixer un dels seus components principals, el *resilient distributed dataset* (RDD, d'ara endavant), sobre el qual es basa la seva estructura de paral·lelització.

Un RDD és una entitat abstracta que representa un conjunt de dades distribuïdes pel clúster i una capa d'abstracció per sobre de totes les dades que componen el nostre corpus de treball, independentment del seu volum, ubicació al clúster, etc.

Aquestes són les característiques principals dels RDD:

- Són immutables, una condició important perquè evita que un o diversos fils (*threads*) actualitzin el conjunt de dades amb el qual es treballa.
- Són *lazy loading*, és a dir, només cal que accedim a les dades i elles es carreguen quan és necessari.
- Poden emmagatzemar-se en memòria cau.

L'RDD es construeix a partir de blocs de memòria distribuïts en cada node, fet que dona lloc a les anomenades particions. Spark duu a terme aquest particionament, que és transparent per a l'usuari (tot i que també pot tenir-ne cert control). Si les dades es llegeixen del sistema HDFS, cada partició es correspon amb un bloc de dades del sistema HDFS.

El concepte de partició té un pes important per a comprendre com es paral·lelitza a Spark. Quan es llegeix un fitxer, si aquest és més gran que la grandària de bloc definida pel sistema HDFS (normalment 64 MB o 128 MB), es crea una partició per a cada bloc (malgrat que l'usuari pot definir el nombre de particions que s'han de crear quan està llegint el fitxer, si vol).

Així, ens trobem amb el següent:

- Un RDD és un *array* de referències a particions al nostre sistema.
- La partició és la unitat bàsica per a entendre el paral·lelisme a Spark i cada partició es relaciona amb blocs de dades físiques en el nostre sistema d'emmagatzematge o de memòria.
- Les particions s'assignen amb criteris com la localitat de les dades (*data locality*) o la minimització de trànsit a la xarxa interna.
- Cada partició es carrega en memòria volàtil (típicament, RAM) abans de ser processada.

Un RDD es pot construir de la manera següent:

1) Creació d'un objecte «col·lecció» paral·lel:

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

Això crearà una col·lecció paral·lela al node del controlador, farà particions i les distribuirà entre els nodes del clúster, concretament a la memòria.

2) Creació d'un RDD des de fonts externes, com per exemple HDFS o S3 d'Amazon.

Es crearan particions per bloc de dades HDFS als nodes en què les dades estiguin físicament disponibles.

3) Creació d'un RDD sobre un altre que ja existeix. Atès que és immutable, quan s'aplica qualsevol operació sobre un RDD existent se'n crearà un de nou.

Sobre un RDD es poden aplicar transformacions o accions, però no s'accedeix a les dades que componen un RDD fins que s'executa sobre ells una acció, seguint el model *lazy loading* ja esmentat. Així, quan es treballa amb RDD sobre els quals s'apliquen transformacions, en realitat s'està definint una seqüència de processos per aplicar sobre l'RDD que no s'iniciaran fins que s'hi hagi iniciat una acció.

Imagineu que prenem la seqüència següent d'ordre a Python per a 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()
```

Aquesta porció de codi mostra com es carrega un RDD (dades variables) a partir d'un fitxer situat al sistema distribuït HDFS. A continuació, s'apliquen fins a tres transformacions (*flatMap*, *map* i *reduceByKey*). No obstant això, cap d'aquestes transformacions s'executarà de manera efectiva fins a la crida al mètode `count ()`, que ens retornarà el nombre d'elements a l'RDD «words». És a dir, fins que no s'executi una acció sobre les dades, no s'executen les operacions que ens les proporcionen.

El programa en què s'executa el codi mostrat és anomenat *driver*, i és l'encarregat de controlar-ne l'execució, mentre que les transformacions són les operacions que es paral·lelitzen i s'executen de manera distribuïda entre els nodes que componen el clúster. Els responsables de l'execució als nodes són coneguts com a *executors*. Com només s'accedeix a les dades de l'RDD en executar una acció, si hi ha algun problema en alguna de les transformacions només serà possible detectar-lo una vegada s'executi l'acció que desencadena les transformacions associades.

3.2. Model d'execució de Spark

El *directed acyclic graph* (DAG, o graf dirigit acíclic) és un estil de programació per a sistemes distribuïts que es presenta com a alternativa al paradigma MapReduce. Mentre que MapReduce té només dos passos (*map* i *reduce*), la qual cosa limita la implementació d'algorismes complexos, DAG pot tenir múltiples nivells que poden formar una estructura d'arbre, amb més funcions com *map*, *filter*, *union*, etc.

En un model DAG no hi ha ni cicles definits ni un patró d'execució definit prèviament, tot i que sí existeix un ordre o una adreça d'execució. Apache

Spark fa ús del model DAG per a l'execució de les seves tasques en entorns distribuïts.

El DAG de Spark està compost per arcs (o arestes) i vèrtexs, on cada vèrtex representa un RDD i els eixos representen operacions per aplicar sobre l'RDD.

Les transformacions sobre RDD poden ser categoritzades com:

- ***Narrow operation***: s'utilitza quan les dades que s'han de tractar estan en la mateixa partició de l'RDD i no cal barrejar aquestes dades per a obtenir-les totes. Alguns exemples són les funcions *filter*, *sample*, *map* o *flatMap*.

- ***Wide operation***: s'utilitza quan la lògica de l'aplicació necessita dades que es troben en diferents particions d'un RDD i cal barrejar aquestes particions per a agrupar les dades necessàries en un RDD determinat. Exemples de *wide transformation* són *groupByKey* o *reduceByKey*. Aquestes tasques solen ser d'agregació i les dades provinents de diferents particions s'agrupen en un conjunt menor de particions.

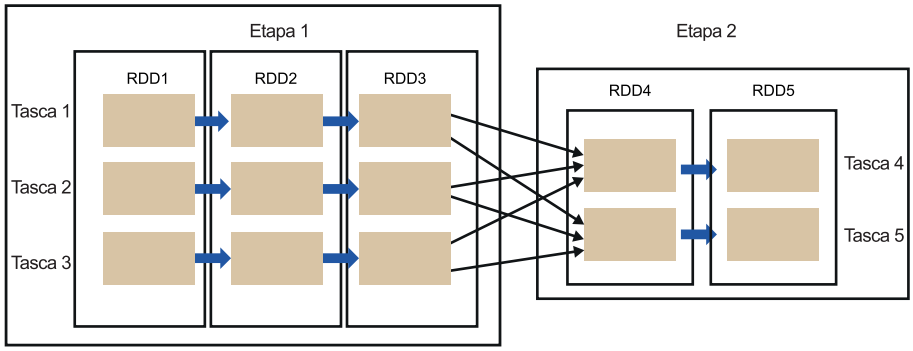
Cada RDD manté una referència a un o més RDD originals, juntament amb metadades sobre quin tipus de relació té amb ells. Per exemple, si executem el codi següent:

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

L'RDD *b* manté una referència al seu pare, l'RDD *a*. És l'anomenada RDD *lineage*.

El conjunt d'operacions sobre les quals es realitzen transformacions preservant les particions de l'RDD (*narrow*) s'anomenen «etapes» (*stage*). El final d'un *stage* es presenta quan es reconfiguren les particions gràcies a operacions de transformació de tipus *wide*, que inicien l'anomenat intercanvi de dades entre nodes (*shuffling*) i comencen d'aquesta manera un altre *stage*.

Figura 13. Model de processat d'RDD a Spark en què es diferencien tasques i etapes (*stages*) en funció del tipus de transformació que s'aplicarà.



Les limitacions de MapReduce es van convertir en un punt clau per a introduir el model d'execució DAG a Spark.

Amb l'aproximació de Spark, la seqüència d'execucions —formades pels *stages* i les transformacions—, les accions per aplicar i el seu ordre queden reflectits en un graf DAG que optimitza el pla d'execució i minimitza el trànsit de dades entre nodes.

3.2.1. Funcionament

Quan una acció és invocada per ser executada sobre un RDD, Spark envia el DAG amb els *stages* i les tasques que es realitzaran al DAG Scheduler, que transforma un pla d'execució lògica (és a dir, RDD de dependències construïdes mitjançant transformacions) en un pla d'execució físic (utilitzant *stages* o etapes i els blocs de dades).

El DAG Scheduler és un servei que s'executa al *driver* del nostre programa. Té dues tasques principals:

1) Determina les ubicacions preferides per a executar cada tasca i calcula la seqüència d'execució òptima:

- en primer lloc, s'analitza el DAG per determinar l'ordre de les transformacions;
- a continuació, amb la finalitat de minimitzar la mescla de dades, es realitzen les transformacions *narrow* en cada RDD;
- finalment, es realitza la transformació *wide* a partir dels RDD sobre els quals s'han realitzat les transformacions *narrow*.

2) Maneja els possibles problemes i les fallades a causa d'una possible corrupció de dades en algun node: quan algun node queda inoperatiu (bloquejat o caigut) durant una operació, l'administrador del clúster assigna un altre node per continuar el procés. Aquest node operarà en la partició particular de l'RDD i la sèrie d'operacions que ha d'executar sense que hi hagi una pèrdua de dades.

Així, és el DAG Scheduler el que calcula el DAG d'etapes per a cada treball, fa un seguiment de quin RDD i quines sortides d'etapa es materialitzen i troba el camí mínim per a executar els treballs. A continuació, envia les etapes al Task Scheduler.

Les operacions que s'executen en un DAG estan optimitzades pel DAG Optimizer, que pot reorganitzar o canviar l'ordre de les transformacions si així es millora el rendiment de l'execució. Per exemple, si una tasca té una operació

map seguida d'una operació *filter*, el DAG Optimizer canviarà l'ordre d'aquests operadors, ja que el filtratge reduirà el nombre d'elements sobre els quals es realitza el *map*.

3.3. Apache Spark MLlib

Entre les diverses API de treball de Spark, probablement la més popular i potent sigui l'API d'aprenentatge automàtic (*machine learning*). La comunitat de contribuïdors a aquesta API és la més gran de l'ecosistema Spark i en cada nova versió s'inclouen noves tècniques.

L'anàlisi de dades és un dels camps de treball més atractius en el sector tecnològic i forma part dels que tenen més demanda professional. En aquest camp es treballa de manera intensiva utilitzant les eines de mineria de dades més completes, com a R, Python o Matlab.

Aquestes plataformes presenten un alt grau de maduresa, incorporen multitud de llibreries d'aprenentatge automàtic i, també, molts casos d'ús i experiències desenvolupats i acumulats al llarg dels anys. No obstant això, Spark ofereix una implementació paral·lela.

3.3.1. ML Pipelines

Com a entorn per a treballar amb l'API d'aprenentatge automàtic de Spark, s'ofereix l'eina *pipeline*, que permet crear cadenes de treball pròpies d'una tasca de mineria de dades. Un *pipeline* consta de diversos components:

- *dataframe* d'entrada
- *estimator* (responsable de generar un model)
- paràmetres per a generar els models
- *transformer* (model generat a partir d'uns paràmetres concrets)

Un *pipeline* consta d'una cadena d'estimadors i transformacions. El DAG d'una ML Pipeline no té per què ser seqüencial, que és el tipus en què l'entrada de cada *stage* és la sortida de l'anterior, però ha de poder ser representat mitjançant ordenació topològica.*

* http://bit.ly/2CuyQtW

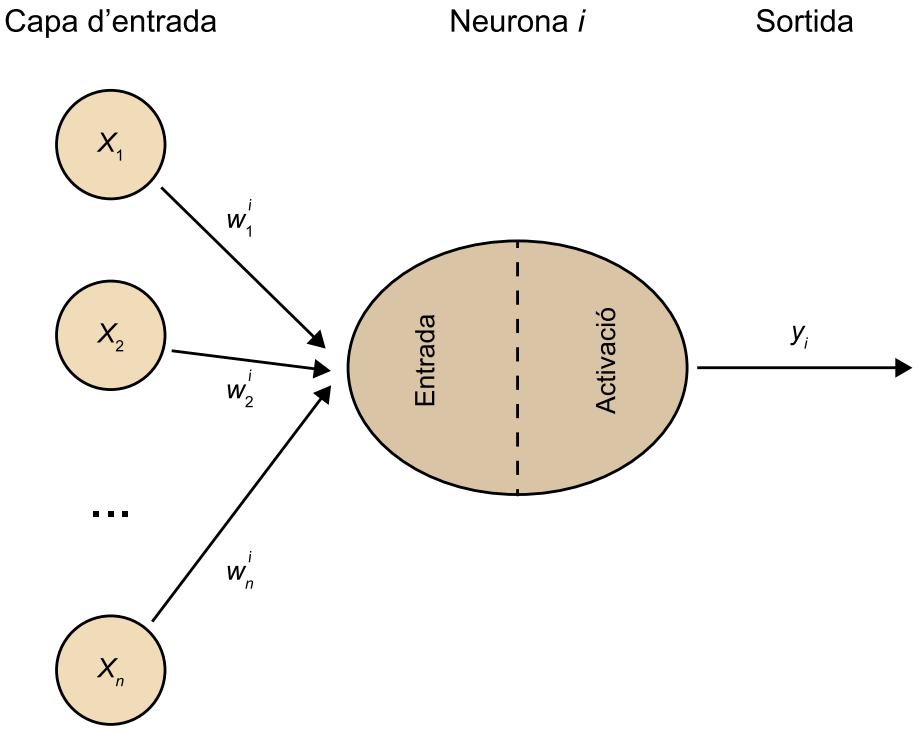
4. TensorFlow

Actualment, l'aprenentatge automàtic és una de les disciplines més atractives d'estudi i amb més projecció professional, especialment després de l'explosió de les dades massives i el seu ecosistema tecnològic, que ha obert un enorme panorama d'aplicacions potencials que no eren viables de resoldre fins ara. Però el major desafiament de l'aprenentatge automàtic ve a partir de les tècniques de classificació no supervisada, és a dir, que la màquina aprèn per si mateixa a partir d'unes dades no etiquetades prèviament.

4.1. Xarxes neuronals i *deep learning*

En l'enfocament basat en les xarxes neuronals, s'utilitzen estructures lògiques que s'assemblen en certa mesura a l'organització del sistema nerviós dels mamífers, amb capes d'unitats de procés (anomenades neurones artificials) que s'especialitzen a detectar determinades característiques existents en els objectes percebuts, i que permeten que dins del sistema global hi hagi xarxes d'unitats de procés que s'especialitzin en la detecció de determinades característiques ocultes en les dades.

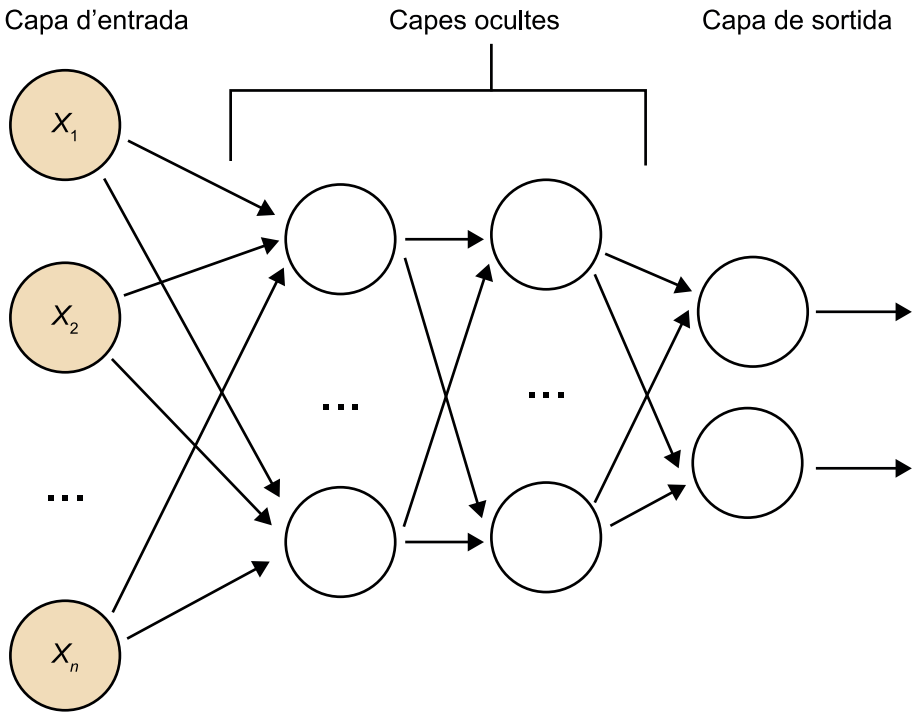
Figura 14. Esquema d'una neurona



Les xarxes neuronals artificials (ANN, *artificial neural networks*) estan formades per un conjunt de neurones distribuïdes en diferents capes. Cadascuna d'aquestes neurones realitza un càlcul o una operació senzilla sobre el conjunt de valors d'entrada de la neurona —que en essència són o bé entrades de dades o bé les sortides de les neurones de la capa anterior— i calcula un únic valor de sortida, que, al seu torn, serà un valor d'entrada per a les neurones de la capa següent o bé formarà part de la sortida final de la xarxa.

Les neurones s'agrupen formant capes, que són estructures de neurones paral·leles. Les sortides d'unes capes es connecten amb les entrades d'unes altres, de manera que es crea una estructura seqüencial. La figura 15 presenta un esquema bàsic d'una xarxa neuronal amb la capa d'entrada, múltiples capes ocultes i la capa de sortida.

Figura 15. Esquema de xarxa neuronal amb múltiples capes ocultes



Les xarxes neuronals no són un concepte nou i, malgrat el seu funcionament sempre ha estat satisfactori quant a resultats, el consum de recursos per a entrenar una xarxa neuronal sempre ha estat molt elevat, la qual cosa ha impedit en part el seu desenvolupament complet i la seva aplicació. No obstant això, després de la irrupció de les GPU les xarxes neuronals han tornat a guanyar protagonisme gràcies a l'elevat paral·lelisme requerit per a entrenar cada neurona i el benefici potencial que ofereixen les GPU per a aquest tipus d'operacions.

Aquestes xarxes se solen inicialitzar amb valors aleatoris i requereixen un procés d'entrenament amb un conjunt de dades per poder «aprendre» una tasca o funció concreta. Aquest procés d'aprenentatge es realitza utilitzant el mètode conegut com a Backpropagation.* En essència, aquest mètode calcula l'error

* http://bit.ly/2jMEOBe

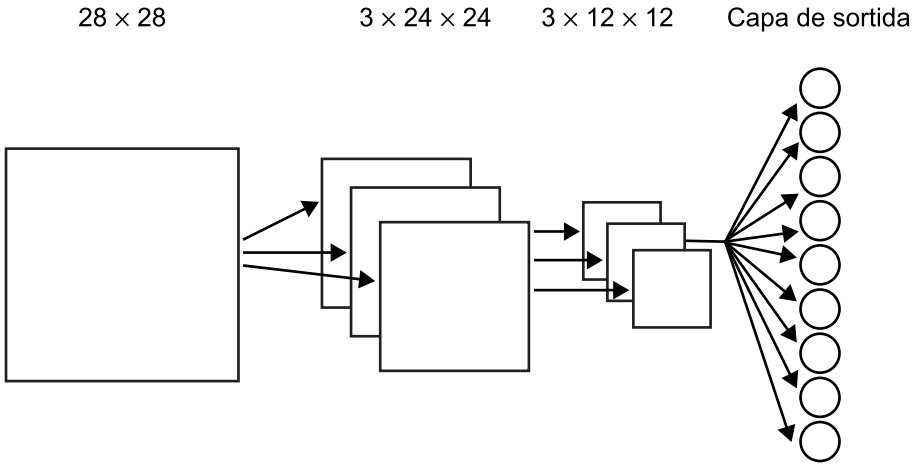
que comet la xarxa en la predicció del valor per a un exemple donat i intenta modificar els paràmetres de totes les neurones de la xarxa per reduir aquest error.

Aquest tipus d'algorismes va tenir la seva època d'esplendor ja fa unes dècades. La seva limitació principal està en el procés d'aprenentatge que es dona a les xarxes amb un cert nombre de capes ocultes (capes intermèdies, és a dir, que es troben entre l'entrada de dades i la sortida o resposta final de la xarxa). En aquests casos, es produeix el que es coneix com el problema de la desaparició o l'explosió del gradient,* que bàsicament provoca problemes en el procés d'aprenentatge de la xarxa.

* http://bit.ly/2DERG0J

Aquests problemes s'han superat anys més tard i han iniciat el que es coneix actualment com a *deep learning*. S'ha modificat l'estructura bàsica de les xarxes neuronals, creant, per exemple, xarxes convolucionals que permeten originar diferents capes en què el coneixement es va fent més abstracte. És a dir, les primeres capes de la xarxa es poden encarregar d'identificar certs patrons a les dades, mentre que les capes posteriors identifiquen conceptes més abstractes a partir d'aquests patrons més bàsics. Per exemple, si volem que una xarxa neuronal pugui detectar quan apareix una cara en una imatge, aquest enfocament buscaria que les primeres capes de la xarxa s'encarreguessin de detectar la presència d'un ull, d'una boca, etc. en alguna part de la imatge. Així, les capes següents s'encarregarien de combinar aquesta informació i identificar una cara a partir de l'existència de les parts que la formen (ulls, boca, nas, etc). D'aquesta manera anem avançant des d'una informació més bàsica cap a un coneixement més abstracte. La figura 16 mostra un esquema, tot i que molt general, d'una xarxa convolucional.

Figura 16. Esquema general d'una xarxa neuronal convolucional



Hi ha diversos entorns i biblioteques per a treballar amb xarxes neuronals i *deep learning* que s'executen en les potents GPU modernes, generalment mitjançant una API anomenada CUDA. Probablement, l'entorn TensorFlow de Google sigui un dels més famosos.

4.2. TensorFlow

TensorFlow* és una API de codi obert desenvolupada per Google per construir i entrenar xarxes neuronals. TensorFlow és una eina potent per a fer ús extensiu dels recursos locals d'un ordinador, ja sigui CPU o GPU.

* https://www.tensorflow.org

Així, TensorFlow és un sistema de programació en el qual representem càlculs en forma de grafs; cada node dels grafs (anomenats *ops*) realitza una operació sobre un o diversos «tensors» (que poden ser un vector, un valor numèric o una matriu) i, com a resultat, retorna un tensor.

El graf que representa el TensorFlow descriu els càlculs que realitzaran, per això es fa servir la terminologia *flow*. El graf es llança en el context d'una sessió de TensorFlow (*session*), que encapsula l'entorn d'operació de la nostra tasca, incloent-hi la ubicació de les tasques en el nostre sistema, ja sigui CPU o GPU.

Tot i que TensorFlow disposa d'implementacions paral·lelitzables també pot ser distribuït** per mitjà d'un clúster d'ordinadors, en el qual cada node s'ocupa d'una tasca del graf d'execució.

** https://goo.gl/94m9Fp

Existeixen implementacions de TensorFlow en diversos llenguatges de programació, com Java, Python, C++ i Go.*** També permet l'ús de GPU, la qual cosa pot donar com a resultat un increment important del rendiment en l'entrenament de xarxes neuronals profundes, per exemple.

*** https://golang.org

En aquest punt, ens preguntem: podríem combinar el millor del processament distribuït (Spark) i una eina d'aprenentatge automàtic tan potent com TensorFlow?

Podríem utilitzar models entrenats amb eines com scikit-learn i utilitzar les funcionalitats de Spark i la seva capacitat de propagar variables als nodes mitjançant l'opció *broadcast*,**** per tal d'enviar el model mitjançant múltiples nodes i deixar que cada node l'apliqui a un corpus de dades.

**** https://goo.gl/HLT8SB

Actualment hi ha moltes iniciatives en les quals es combina Spark per a l'avaluació de models i TensorFlow per dur a terme tasques d'entrenament, validació creuada de models o aplicació i predicció del model.

Recentment ha aparegut una nova API, anomenada Deep Learning Pipelines,***** desenvolupada per Databricks (els mateixos desenvolupadors d'Apache Spark), per combinar tècniques de *deep learning*, principalment TensorFlow, amb Apache Spark.

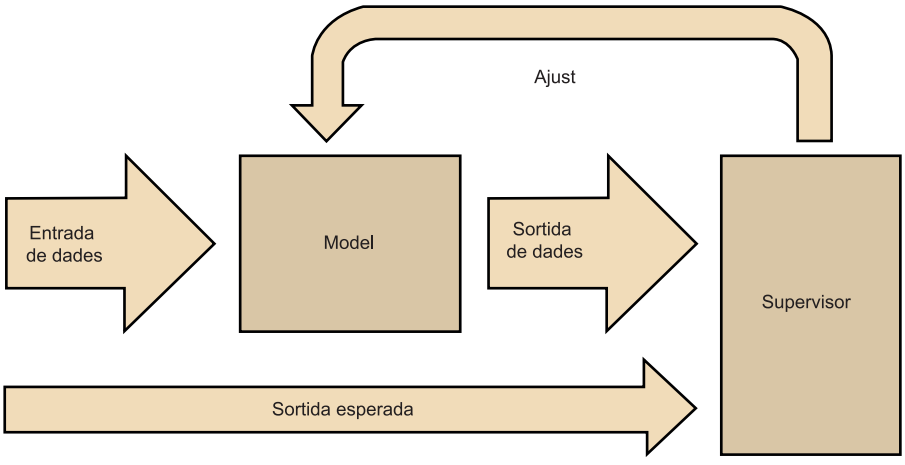
***** https://goo.gl/tbM3rt

5. Aprenentatge autònom

L'aprenentatge autònom (també conegut per la seva denominació en anglès, *machine learning*) és el conjunt de mètodes i algorismes que permeten a una màquina aprendre de manera automàtica sobre la base d'experiències passades.

Generalment, un algorisme d'aprenentatge autònom ha de construir un model sobre la base d'un conjunt de dades d'entrada que representen el conjunt d'aprenentatge, la qual cosa es coneix com a conjunt d'entrenament. Durant aquesta fase d'aprenentatge, l'algorisme va comparant la sortida dels models en construcció amb la sortida ideal que haurien de tenir aquests models, per anar ajustant-los i augmentant-ne la precisió. Aquesta comparació forma la base de l'aprenentatge en si, i aquest aprenentatge pot ser **supervisat** o **no supervisat**. En l'aprenentatge supervisat (figura 17), hi ha un component extern que compara les dades obtingudes pel model amb les dades esperades per aquest, i proporciona retroalimentació al model perquè vagi ajustant-se. Per fer-ho, doncs, serà necessari proporcionar al model un conjunt de dades d'entrenament que contingui tant les dades d'entrada com la sortida esperada per a cadascuna d'aquestes dades.

Figura 17. Aprenentatge supervisat

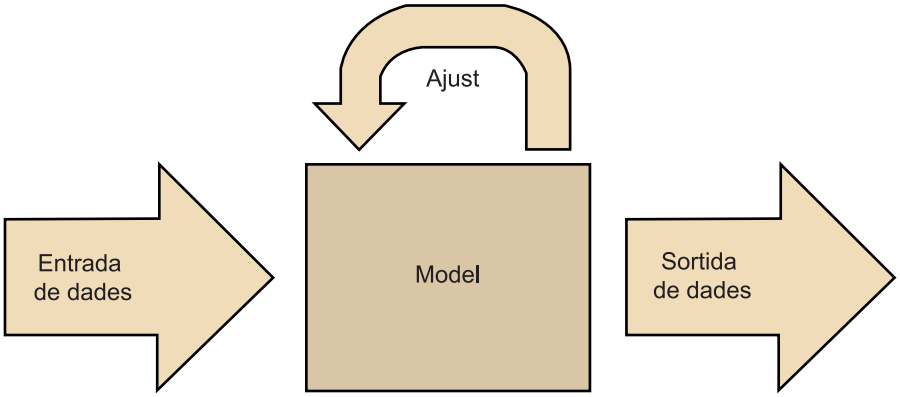


Totes elles es basen en el paradigma de l'aprenentatge inductiu. L'essència de cadascuna d'aquestes és derivar inductivament a partir de les **dades** (que representen la informació de l'entrenament) i un **model** (que representa el

coneixement) que té utilitat predictiva, és a dir, que pot aplicar-se a noves dades.

En l'aprenentatge no supervisat (figura 18), l'algorisme d'entrenament aprèn sobre les pròpies dades d'entrada, descobrint i agrupant patrons, característiques, correlacions, etc.

Figura 18. Aprenentatge no supervisat



Alguns algorismes d'aprenentatge autònom requereixen d'un gran conjunt de dades d'entrada per poder convergir cap a un model precís i fiable, de manera que les dades massives són un entorn ideal per a aquest aprenentatge. Tant Hadoop (amb el seu paquet d'aprenentatge autònom Mahout) i Spark (amb la seva solució MLlib) proporcionen un extens conjunt d'algorismes d'aprenentatge autònom. A continuació, els descriurem i indicarem la seva classificació i per a quines plataformes estan disponibles.

5.1. Classificació

La classificació (*classification*) és un dels processos cognitius més importants, tant en la vida quotidiana com en els negocis, on podem classificar clients, empleats, transaccions, botigues, fàbriques, dispositius, documents o qualsevol altre tipus d'instàncies en un conjunt de classes o categories predefinides amb anterioritat.

La tasca de classificació consisteix a assignar instàncies d'un domini concret, descrites per un conjunt d'atributs discrets o de valor continu, a un conjunt de classes, que poden ser considerades valors d'un atribut discret seleccionat, generalment denominat **classe**. Les etiquetes de classe correctes són, en general, desconegudes, però es proporcionen per a un subconjunt del domini. Per tant, queda clar que és necessari disposar d'un subconjunt de dades correctament etiquetat i que s'utilitzarà per a la construcció del model.

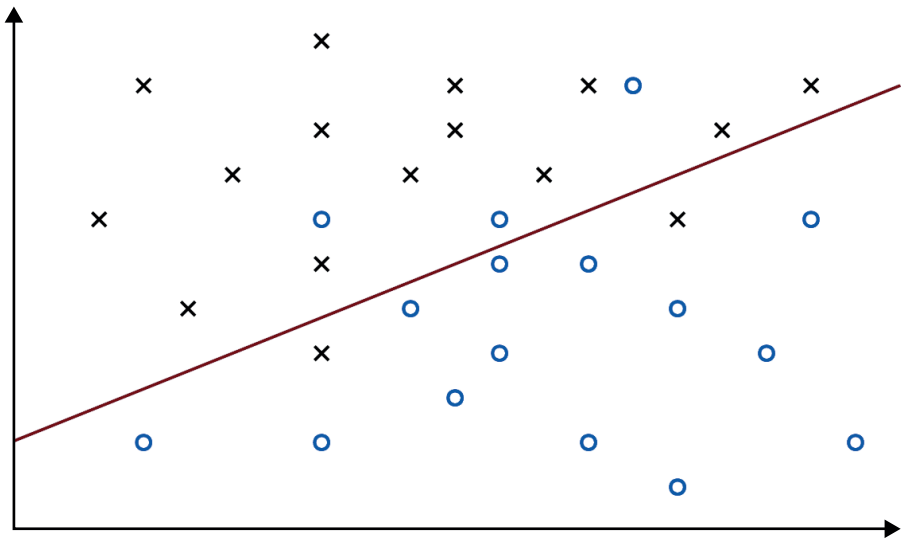
La funció de classificació es pot veure com:

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

on c representa la funció de classificació, X el conjunt d'atributs que formen una instància i C l'etiqueta de classe d'aquesta instància.

Un tipus de classificació particularment simple, però molt interessant i àmpliament estudiat, fa referència als problemes de classificació binaris, és a dir, problemes amb un conjunt de dades pertanyents a dues classes: $C = \{0,1\}$. La figura 19 mostra un exemple de classificació binària, en què les creus i els cercles representen elements de dues classes i es pretén dividir l'espai de manera que separi la majoria d'elements de classes diferents.

Figura 19. Exemple de classificació



Tant MLlib com Mahout proporcionen diversos mètodes d'aprenentatge per a la classificació, com arbres de classificació i regressió, classificadors bayesians ingenus, boscos d'arbres de decisió, perceptrons multicapa i models ocults de Markov.

Les tècniques de classificació supervisada incloses a Spark* són les més completes de l'entorn i es divideixen en diversos tipus de classificadors, que podem agrupar en:

* https://goo.gl/1RAEYo

- classificador basat en regressió logística, tant binària com multidimensional;
- classificador basat en arbres de decisió (*decision trees*);
- classificador *random forest*;
- classificador basat en arbre amb degradació de gradient;

- classificador basat en xarxes neuronals (perceptró multicapa);
- màquines de vectors de suport (SVM, *support vector machine*);
- classificador *one-vs-rest* o *one-vs-all* (classificador que es construeix per sobre d'un classificador binari);
- classificador bayesià ingenu (*naïve Bayes*).

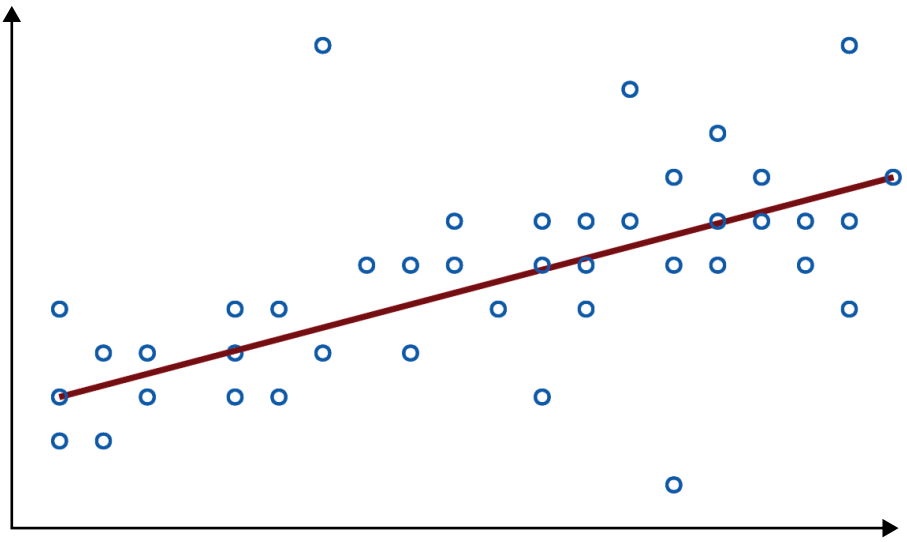
A Spark existeix tant el classificador com el model de regressió de cadascuna d'aquestes tècniques. Les tècniques de regressió retornen una variable de sortida amb valors continus, mentre que en la classificació la variable de sortida pren etiquetes (*labels*) que representen una classe com a resultat de la classificació.

5.2. Regressió

Igual que la classificació, la regressió (*regression*) és una tasca d'aprenentatge inductiu que ha estat àmpliament estudiada i que s'utilitza sovint. Es pot definir, de manera informal, com un problema de «classificació amb classes contínues». És a dir, els models de regressió prediuen valors numèrics en comptes d'etiquetes de classe discretes. De vegades també ens podem referir a la regressió com a «predicció numèrica».

La tasca de regressió consisteix a assignar valors numèrics a instàncies d'un domini donat, descrits per un conjunt d'atributs discrets o de valor continu, com es mostra en la figura 20, en què els punts representen les dades d'aprenentatge i la línia representa la predicció sobre futurs esdeveniments. Se suposa que aquesta assignació s'aproxima a alguna funció objectiu, generalment desconeguda, excepte per a un subconjunt del domini. Aquest subconjunt es pot utilitzar per a crear el model de regressió.

Figura 20. Exemple de regressió lineal



En aquest cas, la funció de regressió es pot definir com:

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

on f representa la funció de regressió, X el conjunt d'atributs que formen una instància i $\mathbb{R}$ un valor en el domini dels nombres reals.

És important remarcar que una regressió no pretén retornar una predicció exacta sobre un esdeveniment futur, sinó una aproximació (com mostra la diferència entre la línia i els punts de la figura). En general, dades més disperses donaran com a resultat prediccions menys ajustades.

Actualment, MLlib i Mahout proporcionen mètodes per a la regressió logística (és a dir, el tipus de regressió que es pot aproximar mitjançant una funció logística).* A més, MLlib proporciona mètodes per a regressions lineals i regressions isotòniques (definides per parts).**

* https://goo.gl/JGjVZb
** https://goo.gl/gRh1hm

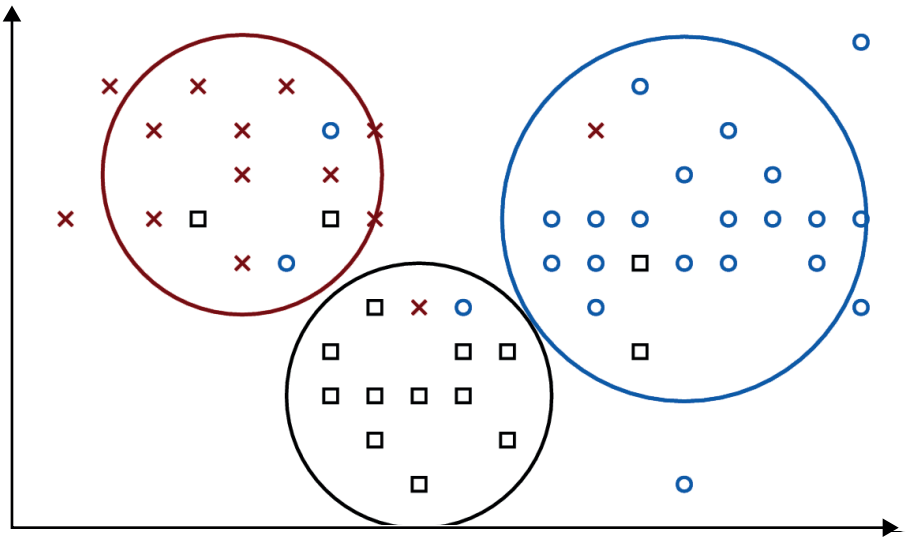
5.3. Agrupament

L'agrupament (*clustering*) és una tasca d'aprenentatge inductiu que, a diferència de les tasques de classificació i de regressió, no té una etiqueta de classe que s'ha de predir. Pot considerar-se com un problema de classificació, però en el qual no existeixen un conjunt de classes predefinides, sinó que es «descobreixen» de manera autònoma pel mètode o algorisme d'agrupament basant-se en patrons de similitud identificats a les dades.

La tasca d'agrupament consisteix a dividir un conjunt d'instàncies d'un domini donat, descrit per un nombre d'atributs discrets o de valor continu, en un conjunt de grups (*clusters*) basant-se en la similitud entre les instàncies, i a crear un model que pot assignar noves instàncies a un d'aquests grups o *clusters*. La figura 21 mostra un exemple d'agrupament en què les creus, els cercles i els quadres pertanyen a tres classes d'elements diferents que pretenem agrupar.

Un procés d'agrupació pot proporcionar informació útil sobre els patrons de similitud presents a les dades, com, per exemple, segmentació de clients o creació de catàlegs de documents. Una altra de les principals utilitats de l'agrupament és la detecció d'anomalies. En aquest cas, el mètode d'agrupament permet distingir instàncies amb un patró absolutament diferent a les altres instàncies «normals» del conjunt de dades, fet que facilita la detecció d'anomalies i possibilita l'emissió d'alertes automàtiques per a noves instàncies que no tenen cap clúster existent.

Figura 21. Exemple d'agrupament



La funció d'agrupament o *clustering* es pot modelar mitjançant:

$$h : X \rightarrow C_h \tag{3}$$

on h representa la funció d'agrupament, X el conjunt d'atributs que formen una instància i C_h un conjunt de grups o *clusters*. Tot i que aquesta definició s'assembla molt a la tasca de classificació, hi ha una diferència aparentment petita però molt important, i és que el conjunt de «classes» no està predeterminat ni és conegut *a priori*, sinó que s'identifica com a part de la creació del model.

L'algorisme d'agrupament més conegut és *k-means*, que va formant grups iterativament a partir d'un conjunt donat. Aquest conjunt s'implementa en diferents variants tant per MLlib com per Mahout.

MLlib d'Apache Spark incorpora l'algorisme *k-means*.* També incorpora implementacions de Latent Dirichlet Allocation (LDA), de *bisecting k-means* (una tècnica d'agrupament jeràrquica que va creant subclústers dins de clústers més grans) i del model mixt gaussià (*Gaussian mixture model*, GMM).** No obstant això, hi ha altres implementacions que no venen incloses a l'API de Spark que són contribucions de tercers i que inclouen altres tècniques, com el popular DBScan o la integració amb TensorFlow, del qual hem parlat anteriorment.

* https://goo.gl/jJ8io6
** https://goo.gl/HJrgpZ

Mahout també implementa l'algorisme *k-means*.* A més, implementa l'algorisme Canopy** de *preclustering* i agrupament espectral.***

* https://goo.gl/pvr5sg
** https://goo.gl/okAd2J
*** https://goo.gl/isFkVn

5.4. Reducció de dimensionalitat

La reducció de dimensionalidad es basa en la reducció del nombre de variables que es tractaran en un espai multidimensional. Aquest procés ajuda a reduir l'aleatorietat i el soroll a l'hora de realitzar altres operacions d'aprenentatge autònom, ja que s'eliminen aquelles dimensions que tenen escassa correlació amb els resultats que es pretenen estudiar.

El problema de la dimensionalitat és un dels més importants en la mineria de dades i existeixen diverses tècniques per a descartar els paràmetres que tenen menys pes en la nostra anàlisi. Les dues tècniques implementades són la descomposició en valors singulars (SVD, *singular-value decomposition*) i l'anàlisi de components principals (PCA, *principal component analysis*), probablement la més popular.

De vegades, reduint la dimensionalitat no només estalviarem en càlculs a l'hora de realitzar regressions o classificacions, sinó que fins i tot aconseguirem que siguin més precises, ja que reduïm sorolls del conjunt d'entrenament.

MLlib* proporciona mètodes de descomposició en valors singulars (*singular value decomposition*, SVD), descomposició QR i anàlisi de components principals (PCA). Spark també implementa algunes tècniques per extreure característiques que en molts aspectes són similars a les tècniques de reducció de dimensionalitat. Algunes d'elles són molt específiques del domini de treball, com per exemple la tf-idf (*term frequency - inverse document frequency*) o el Word2vec, tècniques d'aplicació en la caracterització i classificació de documents en tasques de mineria de text.

* https://goo.gl/sASmxt

D'altra banda, Mahout també proporciona els mètodes SVD,** descomposició QR i PCA. A més, proporciona altres algorismes com SVD estocàstica o Lanczos.

** https://goo.gl/7xmMxL

5.5. Filtrat col·laboratiu / sistemes de recomanació

Els sistemes de recomanació miren de predir els gustos o les intencions d'un usuari sobre la base de les seves valoracions prèvies i les valoracions d'altres usuaris amb gustos similars. Són molt usats en sistemes de recomanació de música, pel·lícules, llibres i botigues en línia.

Quan es crea un perfil de gustos de l'usuari, s'utilitzen dues formes o mètodes en la recol·lecció de característiques: implícites o explícites. Una exemple de forma explícita podria ser sol·licitar a l'usuari que avaluï un objecte o un tema particular, o mostrar-li diversos temes/objectes i que esculli el seu preferit. Un exemple de forma implícita seria guardar un registre d'articles que l'usuari ha visitat en una botiga, cançons que ha escoltat, etc.

Per exemple, imagineu-vos la situació següent en un sistema de pel·lícules en xarxa:

- L'usuari A ha valorat com a excel·lent *El Padrí* i com a mediocre *La guerra de les galàxies*.
- L'usuari B ha valorat com a dolenta *El Padrí*, com a excel·lent *La guerra de les galàxies* i com a excel·lent *Star Trek*.
- L'usuari C ha valorat com a excel·lent *El Padrí*, com a acceptable *La guerra de les galàxies* i com a excel·lent *Casablanca*.

Basant-se en l'afinitat de l'usuari A amb la resta d'usuaris, un sistema de recomanació probablement donaria *Casablanca* com a opció a l'usuari A.

Tant Mahout com MLlib proveeixen sistemes de recomanació basats en factorització de matrius amb *alternate least squares*.* L'objectiu d'aquesta tècnica és trobar la informació «no existent» d'un element a partir de la informació d'altres elements amb característiques similars. La definició d'aquestes característiques és el que permetrà analitzar la seva similitud. La tècnica que apareix després del *collaborative filtering* és la factorització matricial** en sistemes de recomanació.

*** https://goo.gl/5k3iRp**
**** https://goo.gl/bbwn9B**

Cal destacar que Mahout també proveeix filtrat col·laboratiu basat en ítems.***

***** https://goo.gl/sPd8mz**

Resum

En aquest mòdul didàctic hem presentat els conceptes elementals de complexitat computacional que ens permeten entendre la problemàtica relacionada amb el tractament de les dades massives.

A partir d'aquí, hem introduït un dels primers paradigmes utilitzats per tractar grans volums de dades. Estem parlant de MapReduce, un paradigma que permet descompondre un problema en dues tasques bàsiques (*map* i *reduce*) que permeten executar-se en entorns de dades massives, com per exemple a Apache Hadoop.

A partir de les limitacions que presenta el paradigma MapReduce, hem introduït l'entorn de treball Apache Spark, que neix per superar les limitacions que té el model MapReduce. En aquest sentit, hem abordat el funcionament intern de Spark i la seva llibreria d'aprenentatge automàtic (*machine learning*), coneguda com a MLlib.

Finalment, hem revisat els diferents mètodes d'aprenentatge automàtic i hem fet èmfasi en els algorismes suportats per les eines de dades massives, concretament fent referència a les implementacions que incorporen els entorns Apache Hadoop i Apache Spark.

Glossari

escalabilitat horitzontal *f* Acció que té lloc si un sistema millora el seu rendiment en agregar-hi més nodes.

fil *m* Procés que s'executa en paral·lel o de manera asíncrona a altres processos que formen una aplicació.
en thread

hyperthreading *sust.* Tecnologia d'Intel que permet executar programes en paral·lel en un sol processador, ja que simula tenir dos nuclis en un sol processador.

llenguatge assemblador *m* Llenguatge de programació anomenat «de baix nivell» format per un conjunt d'instruccions bàsiques per a programar microprocessadors i/o microcontroladors.

Network-On-Xip (NoC) *sust.* Subsistema de comunicació en un circuit integrat.

propietat associativa *f* Operació $\oplus$ que compleix la propietat $(a \oplus b) \oplus c = a \oplus (b \oplus c)$. Per exemple, la suma és una operació associativa, mentre que la resta no ho és.

thread *m* Vegeu **fil**.

Bibliografia

Boehm i altres (2016). *SystemML: Declarative Machine Learning on Spark*. Actes de l'arxiu del VLDB Endowment Hompage (vol. 9, apartat 13, pàg. 1425-1436).

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

Hearst, M. (2003). *Untangling Text Data Mining*. Actes de l'ACL'99: la 37a. trobada anual de l'Associació 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, tema 1, pàg. 61-73). https://doi.org/10.1177/1094342017712976

Meng, X.; Bradley, J.; Yavuz, B.; Sparks, I.; 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 Apatxe 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.