

UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Ciencias Matemáticas



Trabajo Fin de Grado
Especialidad: Ciencias de la computación

Estudio comparativo de algoritmos paralelos de machine learning e implementación en Hadoop

David Retana Ribeiro

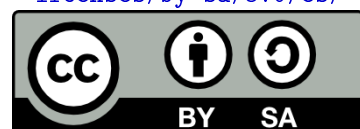
Tutor: Carlos Gregorio Rodríguez

21 de Septiembre de 2017

© Copyright 2017 David Retana Ribeiro

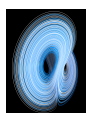
Las imágenes contenidas en este documento así como el código fuente y la información que recoge, se encuentran licenciadas bajo una licencia *Creative Commons*.

<https://creativecommons.org/licenses/by-sa/3.0/es/>



Dedicado a:
Mis padres, Rosa y Pedro, por ese apoyo incondicional a lo largo de toda mi vida
Mis hermanos, Fátima y Carlos, por estar siempre ahí
A mi tutor Carlos Gregorio por dirigirme en este trabajo
A todos mis compañeros de CoreNetworks

SOBRE EL AUTOR



Nombre David Retana Ribeiro
Titulación Grado en Matemáticas
correos davidretanaribeiro@gmail.com
dr4293@outlook.com

LinkedIn <https://www.linkedin.com/in/david-retana-ribeiro-519a56147/>
GitHub <https://github.com/davidRetana>
Kaggle <https://www.kaggle.com/davidretana>

Convenciones en la escritura del documento:

- Se usará *letra en cursiva* para designar aquellos términos en inglés que no son traducidos al español, como por ejemplo *machine learning*, *cluster*... También se usará para designar nombres propios como *Creative Commons* o *Apache*.
- La letra en **negrita** quedará reservada para hacer hincapié en ciertos términos que quieran ser remarcados bien sea porque son importantes para el desarrollo del capítulo o porque en ellos se base la idea a explicar en el capítulo.
- Las notas al pie de página se usan para explicar conceptos de manera breve y concisa, así como evitar confusiones en la utilización de términos ambiguos.
- En numerosas ocasiones aparecerá ejemplos de códigos fuente y comandos de *shell*, éstos aparecerán destacados en un recuadro. Para los comandos *UNIX*, los argumentos encerrados entre signos de desigualdad (<>) indicarán parámetros a completar por el usuario mientras que si están encerrados por corchetes ([]) indica que son opcionales.
- El código fuente escrito en *Python* seguirá el estilo marcado por [PEP8](#).
- En el [Capítulo 5](#), se utiliza un lenguaje matemático para describir con precisión el modelo de algunos algoritmos que se estudian. Al comienzo de dicho capítulo se explica en detalle la notación utilizada.
- Los enlaces a paginas web son marcados en color azul, mientras que las referencias a puntos de este documento están marcadas de color marrón.

Se asume que el lector de este documento tiene una base en matemáticas y estadística así como en algún lenguaje de programación, especialmente *Python*. También es recomendable que el lector este familiarizado con entornos *UNIX* y tenga conocimientos básicos del uso de la terminal. Este documento no esta enfocado a explicar el funcionamiento de los algoritmos de *machine learning* de los que se habla, si no que se centra en desarrollar técnicas que permitan programar estos algoritmos de manera paralela en entornos distribuidos de computación.

Este documento ha sido escrito en L^AT_EX, usando Texmaker 4.5
(compiled with Qt 5.2.1 and Poppler 0.26.5).

Las imágenes han sido realizadas con <https://www.draw.io/>.
El código fuente se encuentra disponible en mi página de [GitHub](#).

Índice general

Objetivos y plan de trabajo	viii
Introducción	ix
I Despliegue de un cluster Hadoop	1
1 ApacheTM Hadoop[®]	2
1.1 ¿Qué es Apache Hadoop?	2
1.2 Arquitectura de un <i>cluster</i>	3
1.3 Topología de un <i>cluster Hadoop</i>	4
2 Instalación y despliegue de un <i>cluster</i>	5
2.1 Instalación de <i>HDFS</i> y <i>YARN</i>	7
2.1.1 Test del <i>cluster</i> desplegado	9
2.2 Instalación de <i>Apache Spark</i>	12
3 Computación paralela	14
3.1 Enfoques procesamiento	14
3.2 Frameworks	15
II Análisis de datos	17
4 Machine Learning	18
4.1 ¿Qué es el machine learning?	18
4.2 <i>Machine Learning</i> distribuido	19
4.3 Machine learning pipelines	19
5 Implementación paralela	22
5.1 Aprendizaje supervisado	22
5.1.1 Regresión lineal	23
5.1.2 Naive-Bayes	26
5.2 Aprendizaje no supervisado	28
5.2.1 Sistema de detección de anomalías	28
5.2.2 K-Means	31
Conclusión	34
III Apéndice	36
A Kaggle y KDD	37
B Cloudera	38
Bibliografía	38
Índice alfabético	39

Índice de figuras

Índice de imágenes	iii
1.1 Logo de <i>Hadoop</i>	2
1.2 Topología de un <i>cluster</i>	3
1.3 Servicios de un <i>cluster Hadoop</i>	4
2.1 Instantánea <i>Cloudera Manager</i>	9
2.2 <i>Overview web namenode</i>	10
2.3 Información web <i>Datanode</i>	10
2.4 Web del servicio <i>Resource Manager</i>	11
2.5 <i>PySpark shell</i>	13
3.1 Conteo de palabras en <i>MapReduce</i>	15
3.2 Logo <i>Apache Spark</i>	16
3.3 Instalación de <i>mrjob</i>	16
4.1 <i>Machine learning pipeline</i>	21
5.1 Regresión lineal en \mathbb{R}^2	23
5.2 División de los datos en el gradiente de descenso	24
5.3 Naive Bayes clasificación	26
5.4 $\mathcal{N}(0, 1)$	28
5.5 Ejemplo de anomalía	29
5.6 Iteraciones del algoritmo k-means	31
B.1 Logo de Cloudera	38

Índice de cuadros

Índice de tablas	iv
2.1 <i>Hardware</i> de las máquinas del <i>cluster</i>	5
2.2 Asignación de roles en el <i>cluster</i>	5
5.1 Diferencias entre <i>Machine Learning</i> y <i>Deep Learning</i>	35

Índice de listados

Índice de códigos fuente	v
5.1 LinearRegression.py	25
5.2 NaiveBayes.py	27
5.3 ComputeMeanVar.py	30
5.4 KMeansSpark.py	32
5.5 KMeansMain	33

Resumen

En este trabajo se aborda el estudio de las tecnologías actuales para el tratamiento de grandes volúmenes de datos, así como la creación de un *cluster* de máquinas utilizando *Apache Hadoop*TM. Posteriormente, se hace uso de él y de modernas técnicas de programación paralela para desarrollar algoritmos de *machine learning* de una manera distribuida, escalable y eficiente. Todos estos conceptos y tecnologías se integran dentro de un área de estudio heterodoxa que se suele denominar como *Big Data*.

Se detalla el despliegue de un *cluster Hadoop* utilizando una herramienta gráfica llamada *Cloudera Manager*. Este software facilita enormemente el trabajo de desplegar un *cluster* ya que se gestiona automáticamente la monitorización de los nodos, la creación de usuarios, ficheros de configuración y demás tareas. Cuando el tamaño del *cluster* se vuelve grande o son muchos los servicios instalados en él, esta es la mejor opción para desplegarlo. *Cloudera* proporciona dos maneras para dicho despliegue que se explican de manera general en el **Apéndice B**.

Este trabajo se enfoca también en desarrollar buenas prácticas en lo que a computación distribuida se refiere y se comparan distintos enfoques para la resolución de un mismo problema. Estos enfoques permiten entender mejor los retos de la computación paralela y la depuración de los algoritmos distribuidos de una manera general, no centrándose específicamente en el *machine learning*.

Los algoritmos de *machine learning* se han desarrollado utilizando dos enfoques de programación paralela, usando el ya clásico paradigma de programación *MapReduce*, o bien utilizando el paradigma *Spark*, más moderno y con una cierta orientación de programación funcional.

La utilización de uno u otro enfoque depende en gran parte de la arquitectura del algoritmo, esto es, para algoritmos iterativos es más conveniente usar *Spark* ya que permite persistir los datos en memoria y por lo tanto reduce enormemente el tiempo de ejecución del algoritmo. Por el contrario, un algoritmo como puede ser calcular la media y la varianza de un conjunto de datos, no requiere mas que una pasada al completo del *dataset*, por lo que con una fase *map* y otra *reduce* es suficiente para calcular dichas variables.

Se tendrá especial atención a la escalabilidad de los algoritmos y el consumo de recursos de un proceso (memoria y *CPU*) ya que un buen rendimiento del algoritmo es clave en la computación distribuida. Los distintos enfoques a la hora de programar un algoritmo se basan en reducir los posibles cuellos de botella que se puedan producir con los datos. Esto es, el uso de *combiners* en trabajos *MapReduce*, ordenes que desencadenen *shuffles*¹ en *Spark*, etc.

Al final de cada sección se ha incluido el **código fuente** de cada algoritmo desarrollado.

Los algoritmos programados usando el paradigma de programación *MapReduce* han sido desarrollados usando la librería *mrjob* del lenguaje de programación *Python*TM. El resto de algoritmos se han desarrollado utilizando el *framework Spark* mediante *Python*, aunque también está disponible en lenguajes como *Java*, *Scala* o *R*. Se hace uso de las librerías *open source numpy*, *scipy*, *matplotlib* y *sklearn*, siempre que sean útiles para el propósito del desarrollo y la visualización de los datos.

Todo el código fuente desarrollado se encuentra disponible en mi página de **GitHub**. (<https://github.com/davidRetana>)

¹movimiento de datos entre nodos a través de la red

Abstract

This paper deals with the study of current technologies for the treatment of large volumes of data, as well as the deployment of a machine's cluster using Apache Hadoop. Afterwards, we will have use of this cluster and modern parallel programming techniques to develop machine learning algorithms in a distributed, scalable and efficient way. All of these concepts and technologies are integrated within a study area called *Big Data*.

The deployment of a **Hadoop cluster** is detailed using a graphical tool called **Cloudera Manager**. This software greatly facilitates the work of deploying a *cluster* because it automatically manages the monitoring of the nodes, the creation of users, configuration files and other tasks. When the size of the *cluster* becomes large or many services are installed on it, this is the best option to deploy it. Cloudera provides two ways for such deployment which are generally explained in [Appendix B](#).

This work also focuses on the development of good practices in what a distributed computing is referred to and compared to other approaches to solving the same problem. These approaches allow a better understanding of the challenges of parallel computing and debugging of the algorithms distributed in a general way, not focusing specifically on machine learning.

Machine learning algorithms have been developed using parallel programming approaches, either using the classic **MapReduce** programming paradigm or using **Spark** programming paradigm, more modern and functional programming oriented.

Using one or other approach depends to a great extent on the architecture of the algorithm, that is, for iterative algorithms it is more convenient to use **Spark** since it allows to persist the data in memory and thereby greatly reduce the execution time of the algorithm. On the contrary, an algorithm as it can calculate the mean and the variance of a data set, does not require more than a complete pass of the *dataset*, so with a map phase and another *reduce* phase is enough to calculate these variables.

Special attention is given to the scalability of the algorithms and the resource consumption of a process (memory and CPU) since a good performance of the algorithm is a key part in distributed computing. The different approaches to programming an algorithm are based on reducing possible bottleneck that can be produced with the data. That is, the use of *combiners* in **MapReduce** jobs, commands that trigger *shuffles*² in **Spark**, etc.

At the end of each section the **source code** of each developed algorithm has been included.

The algorithms programmed using the **MapReduce** programming paradigm have been developed using the **mrjob** library of the **Python**TM programming language. The remaining algorithms have been developed using **Spark** framework through Python, although also Spark is available in other languages such as Java, Scala or R. This report makes use of the open source libraries *numpy*, *scipy*, *matplotlib* and *sklearn*, as long as it is useful for the purpose of development and the visualization of the data.

All the source code developed is available on my [GitHub](#) page.
(<https://github.com/davidRetana>)

²data movement between nodes through the network

Objetivos y plan de trabajo

Con la realización de este proyecto se pretende conseguir desplegar un *cluster* de máquinas instalando el *software Hadoop* en ellas y posteriormente instalar el servicio de *Spark* y la librería *mrjob* de *Python*. Para la realización de este objetivo se usará una herramienta llamada *Cloudera Manager* (ver: **Apéndice B**) que nos guiará en el proceso de instalación.

En cuanto a la parte de algoritmia, esta consistirá en desarrollar algoritmos paralelos de *machine learning* usando el *cluster* construido anteriormente para testar su eficiencia. En total se desarrollarán 4 algoritmos, dos supervisados y dos no supervisados usando *MapReduce* y *Spark* como *frameworks*.

Objetivos

- Instalación de un *cluster Hadoop* de máquinas virtuales.
 - nivel físico: levantar en un servidor las máquinas virtuales con la imagen de *centOS* personalizada.
 - nivel lógico o de *software*: Instalación de *Apache Hadoop*, *Apache Spark* y *mrjob*.
- Desarrollo de algoritmos de *machine learning* de manera distribuida en dicho *cluster*.
 - Algoritmos de aprendizaje supervisado.
 - * Regresión lineal (*Spark*) y *NaiveBayes* (*MapReduce*).
 - Algoritmos de aprendizaje no supervisado.
 - * Detección de anomalías (*MapReduce*) y *K-Means* (*Spark*).

Plan de trabajo

La instalación de un *cluster* desde 0 es un proceso complejo y que requiere de conocimientos tanto a nivel de *hardware* como a nivel de *software* ya que para su realización es necesario una infraestructura física y una infraestructura lógica.

La infraestructura física se realizará levantando varias máquinas virtuales desde un único servidor y en una red local de internet. Estas máquinas virtuales simularán máquinas físicas a efectos prácticos ya que cada una posee su propia dirección *IP*, *CPU*, memoria... Las imágenes de *centOS* (*Community ENTerprise Operating System*, distribución *gnu/linux*) que correrán como sistema operativo se han modificado siguiendo los pasos explicados en https://github.com/davidRetana/custom_centOS para que puedan albergar un *cluster*.

En la **Parte I** (**Despliegue de un cluster Hadoop**) el procedimiento será el siguiente:

Una vez las máquinas *centOS* estén levantadas y funcionando correctamente en el servidor, comenzaremos la instalación del *software Hadoop* en cada uno de los nodos mediante conexiones *SSH* (*Secure SHell*, protocolo criptográfico con conexiones cifradas para acceder a servidores remotos.) , previo reparto de roles entre cada nodo, como se detalla en la **Tabla 2.2**. Hecho esto, ya tendríamos un *cluster* donde a continuación instalaremos el servicio de *Spark* y la librería *mrjob*. Estos dos programas hacen de *framework* de procesamiento y permiten utilizar toda la potencia de cómputo de nuestro *cluster* previamente desplegado.

En la **Parte II** (**Análisis de datos**) el plan consiste en hacer una pequeña introducción al *machine learning*, para que sirva y por que utilizarlo conjuntamente con el *Big Data*. Para la realización de esta sección es necesario el trabajo realizado en la primera ya que utiliza el *cluster* desplegado para desarrollar los dos tipos de algoritmos estudiados: aprendizaje supervisado y aprendizaje no supervisado. Estos algoritmos se intentarán escribir con una sintaxis clara y concisa, y además, se realizarán de manera eficiente dentro de lo posible.

Introducción

Vivimos en la era de los datos, cada día se producen más y más datos que necesitan ser almacenados y procesados para poder sacarles beneficio. En los últimos 10 años se ha generado más información que el acumulado de años anteriores y es por esta razón por la que la manera de almacenar y procesar los datos ha cambiado. El **Big Data** nace como un concepto para hacer referencia a un conjunto de datos masivo, que se origina debido a la incapacidad de los sistemas tradicionales de almacenar y procesar toda la información disponible. La manipulación de grandes cantidades de datos ha de enfrentarse a varios retos, conocidos como las 3 v's del *Big Data*.

- Velocidad (Procesar los datos en un tiempo razonable)
- Volumen (Tener la tecnología suficiente para abordar el volumen de datos existente)
- Variedad (Saber tratar los distintos tipos de datos en sus diversos formatos)

Estos 3 componentes son los que hacen entender el *Big Data* como un concepto nuevo. Adicionalmente se incluyen Veracidad y Valor como nuevos conceptos que deben cumplir los datos. Hemos pasado de hablar en *Gigabytes* o *Terabytes*, a hablar en *Petabytes* o incluso *Exabytes*, magnitudes muy por encima de las soportadas por las máquinas tradicionales.

Para solventar este problema, tradicionalmente se usaban supercomputadores para poder tratar con toda esta información, como por ejemplo el supercomputador *MareNostrum* (<https://es.wikipedia.org/wiki/MareNostrum>). Sin embargo, esta es una opción bastante cara y compleja de mantener. Actualmente, la solución más usada para la gestión de grandes volúmenes de datos es la construcción de *cluster* de máquinas asequible (*commodity hardware*), principalmente porque es una solución sencilla, barata y escalable.

Motivación para la realización de este trabajo

La motivación a la hora de realizar este trabajo ha sido la necesidad de desarrollar algoritmos de *machine learning* de manera paralela, ya que los sistemas tradicionales no soportan el entrenamiento de estos algoritmos bien sea por falta de memoria o por falta de capacidad de computo.

La librería *sklearn* de *Python*, el *software Matlab* o el lenguaje de programación *R* (orientado al cálculo estadístico) son ejemplos de sistemas para el desarrollo de algoritmos de *machine learning*. Estos sistemas funcionan muy bien pero solo en un conjunto pequeño de datos ya que si el tamaño del *dataset* crece, se vuelven incapaces de procesarlo. Esto es así porque dichos sistemas meten los datos en memoria para procesarlos por lo que la limitación aquí vendría en la *CPU* y en la memoria máxima de la máquina en cuestión donde se ejecute.

Los algoritmos de *machine learning* se nutren de los datos por lo que cuantos más tengamos más preciso será nuestro modelo construido con dichos datos.

Por esta razón, los objetivos que pretendo conseguir con este trabajo es la realización de estos algoritmos utilizando las distintas herramientas existentes para el manejo de grandes volúmenes de datos y el procesamiento de los mismos, entendiéndose *Hadoop*, *MapReduce*, *Spark*, etc.

Con estas 3 tecnologías será suficiente para realizar los objetivos marcados y abrir unas líneas futuras de investigación para las cuales este trabajo sea de ayuda.

A nivel personal, la motivación para realizar este trabajo radica en la puesta en práctica de mis conocimientos matemáticos adquiridos a lo largo de la carrera en Ciencias Matemáticas así como también los conocimientos aprendidos en el mundo laboral trabajando en temas de *Big data*.

Adicionalmente, este trabajo supone un reto ya que deberé aprender tecnologías actuales tanto de administración como de desarrollo en general.

Estructuración del documento

Este documento se estructura en 3 partes principales:

- **Parte I** (Despliegue de un cluster Hadoop)
- **Parte II** (Análisis de datos)
- **Parte III** (Apéndice)

La **primera parte** consta de una introducción a *Hadoop* y despliegue de un *cluster* con el servicio complementario de *Spark* y la librería *mrjob* de *Python*.

La **segunda parte** se centra en el análisis de datos donde se desarrollaran algoritmos de *machine learning* de manera distribuida, tanto de aprendizaje supervisado como de aprendizaje no supervisado. Cada sección consta de una primera parte donde se explica la utilidad del algoritmo y sus casos de uso, a continuación una segunda parte donde se explica las matemáticas detrás del algoritmo y finalmente una ultima parte donde se desarrolla el código de manera distribuida con la inclusión del código fuente escrito en *Python*.

La **tercera y última parte** consta de un apéndice de información útil acerca de sitios web donde poner en practica los conocimientos adquiridos mediante competiciones y colaboración con otros equipos de científicos de datos. Además se habla de *Cloudera* como proveedor de servicios de *Big Data*.

'Information is the oil of the 21st century, and analytics is the combustion engine'.

Peter Sondergaard

Parte I

Despliegue de un cluster Hadoop

Capítulo 1

ApacheTM Hadoop[®]

1.1 ¿Qué es Apache Hadoop?

Apache Hadoop es un software de procesamiento distribuido que permite almacenar y procesar grandes cantidades de datos sobre un *cluster* (Sección 1.2) de máquinas, también llamadas nodos del *cluster*. *Hadoop* es un proyecto *open source* de la *Apache Software Foundation* creado inicialmente por *Doug Cuttin*, actualmente en desarrollo y mantenido por la comunidad de software libre.

El diseño de *Hadoop* está enfocado a procesar los datos en el mismo nodo donde se encuentran, llevando el código al dato y evitando así el cuello de botella resultante del tráfico de red al transferir los datos. Este diseño es conocido como **data locality**. *Hadoop* es escalable y tolerante a fallos, tanto en el almacenamiento de los datos como en el procesamiento de estos. La tolerancia a fallos la gestiona mediante la replicación de los datos en 3 copias (por defecto, aunque es configurable), cada una en un nodo distinto del *cluster*. De esta manera facilita así las oportunidades de *data locality*, explicado anteriormente.

Apache Hadoop se compone de dos partes fundamentales:

HDFS (*Hadoop Distributed File System*), es el software encargado de almacenar y distribuir los datos a través de las máquinas del *cluster*. Es altamente escalable y tolerante a fallos. Cuando un archivo es subido al *cluster*, este es dividido en bloques de 128mb y replicado por 3 sobre los nodos. Su arquitectura esta basada en el tipo maestro-exclavo:

- *NameNode*(Maestro) Contiene los metadatos de los archivos.
- *NodeManager*(Exclavo) Contiene los datos en sí del archivo.

YARN (*Yet Another Resource Negotiator*), es el encargado de gestionar los recursos del *cluster* (memoria y *CPU* principalmente) e incluye *MapReduce v2* como motor de procesamiento. También es escalable y tolerante a fallos y al igual que *HDFS*, esta diseñado basándose en una arquitectura maestro-exclavo:

- *ResourceManager*(maestro), es el encargado de asignar los contenedores (cajas de memoria y *CPU*) en los diversos nodos del *cluster* para el desarrollo de las tareas.
- *NodeManager*(exclavo), son los encargados de ejecutar propiamente el código.

Hadoop (y más concretamente *YARN*) utiliza por defecto el motor de procesamiento *MapReduce*, que es explicado en más detalle en la sección **Frameworks** (Sección 3.2).

Además *YARN* no se limita solo a *MapReduce* sino que puede ser utilizado como gestor de recursos del *cluster* para otros motores de procesamiento como *Spark* o *Flink* por ejemplo.



Figura 1.1: Logo de *Hadoop*

1.2 Arquitectura de un *cluster*

Un *cluster* es un conjunto de máquinas (ordenadores) conectadas entre sí mediante una red de tráfico de datos, y que trabajan como si fuesen una sola máquina. Cada máquina es independiente del resto, si bien necesitan tener un software instalado en cada una de ellas que permita la comunicación y sincronización entre todas. Además necesitan una serie de elementos físicos para que dicha comunicación sea posible.

A cada máquina del *cluster* se le denomina *nodo*, y estos están agrupados en conjuntos de nodos llamados *racks*. Un centro de datos contiene uno o más *clusters*, cada *cluster* contiene uno o más *racks* y cada *rack* contiene uno o más nodos de máquinas. Los nodos de un mismo *rack* se conectan entre sí mediante un *switch* (*top rack switch*) y cada *rack* se conecta con uno o varios *switch*.

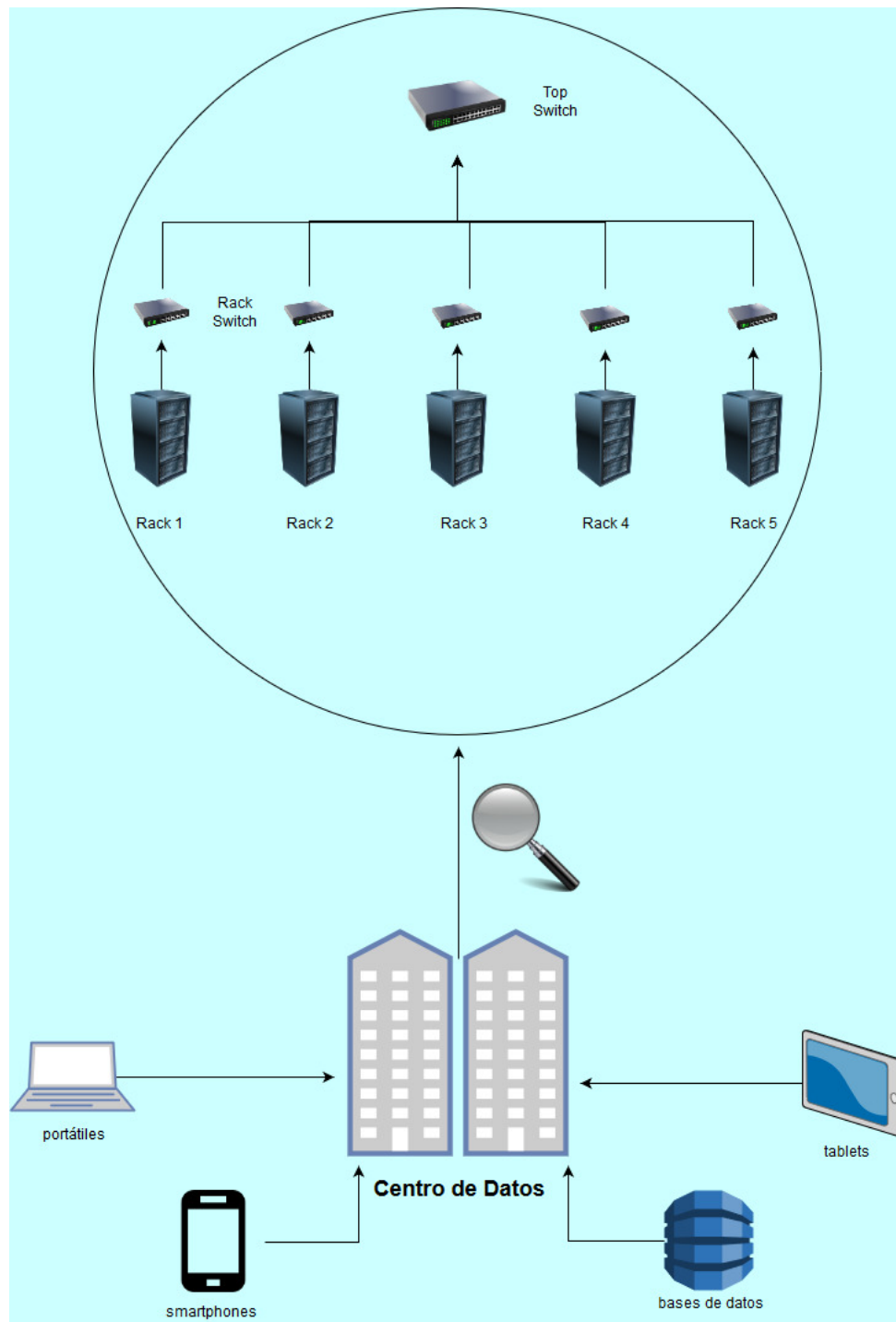


Figura 1.2: Topología de un *cluster*

1.3 Topología de un *cluster Hadoop*

Como se ha mencionado en la [Sección 1.1](#), *Hadoop* se compone de dos partes fundamentales. Cada una de esas partes se compone de subprocesos que se encargan de distintas tareas por lo que en el diseño de un *cluster Hadoop* se deben elegir máquinas con una configuración de *hardware* adecuada a los servicios que se va a desplegar en ella.

En *clusters* destinados a producción, la mejor opción es utilizar *Cloudera Manager* para su despliegue. El asistente gráfico y todos los servicios que lleva por detrás irán instalados en una sola máquina. El resto de servicios propios de *Hadoop* pueden ser instalados en una sola máquina (esto se conoce como modo pseudodistribuido) o en varias máquinas (modo distribuido). Como regla general, se necesitan mínimo dos máquinas *master*, tres máquinas *worker* y una máquina *gateway* para tener un *cluster* plenamente funcional.

Respecto a los servicios de *HDFS* y *YARN* hay que tener ciertas consideraciones, por ejemplo, la máquina designada como *NameNode* necesitará más memoria *RAM* debido a que este guarda toda la información de los metadatos de los archivos en memoria. También, las máquinas que designemos como trabajadoras será recomendable que tengan buenos recursos de *CPU* y memoria así como conexión de red de alta velocidad, de esta manera los trabajos que subamos al *cluster* se ejecutarán más rápido. Las máquinas *worker* será muy recomendable que lleven instalados los servicios de *NodeManager* y *DataNode* conjuntamente, si bien no es obligatorio. Para un conocimiento más profundo acerca de los servicios de *Hadoop*, ver <http://hadoop.apache.org/docs/current/>

A continuación, se muestra un esquema de un *cluster Hadoop* con una configuración básica de los dos servicios mencionados anteriormente.

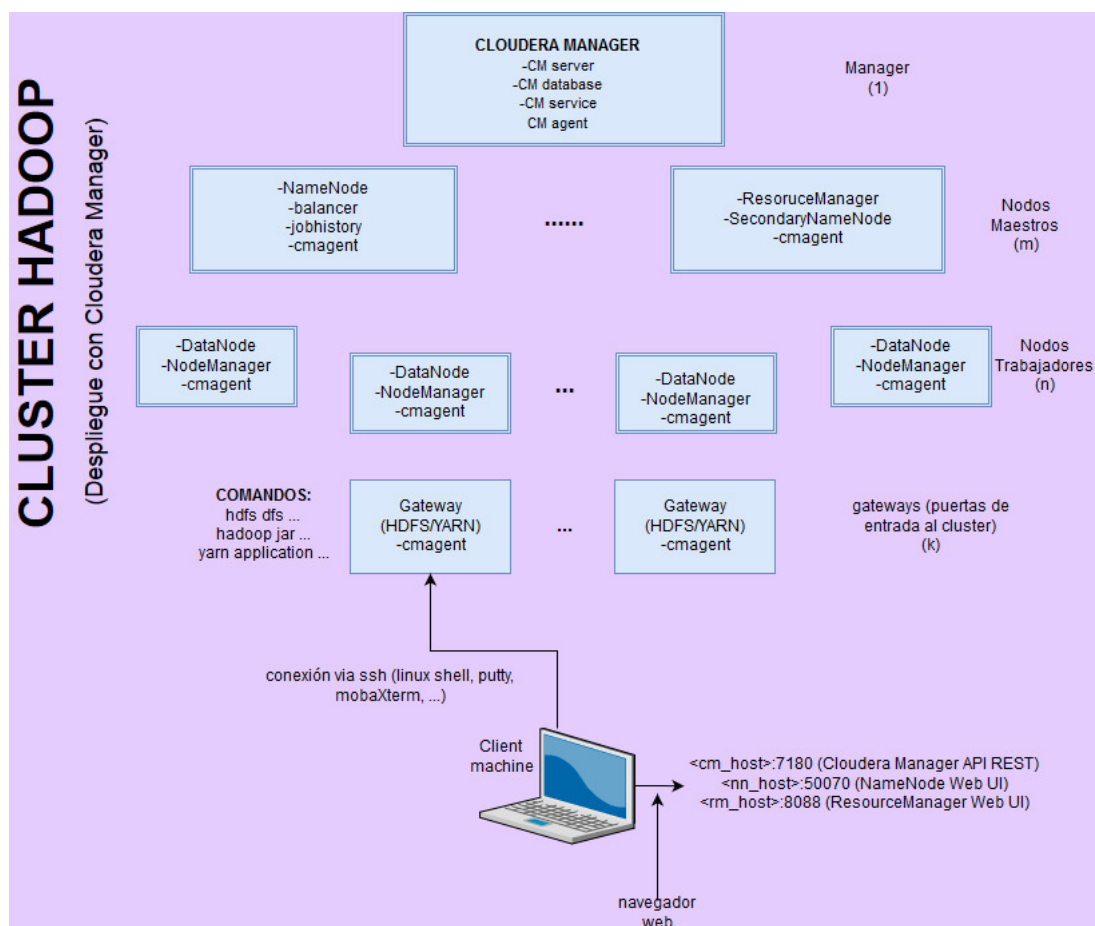


Figura 1.3: Servicios de un *cluster Hadoop*

Capítulo 2

Instalación y despliegue de un *cluster*

En esta sección se cuenta el proceso para acondicionar las máquinas y así poder constituir un *cluster* donde posteriormente instalar el software *Hadoop*. Previamente a la instalación de *Hadoop* hay que configurar los nodos con los requisitos necesarios e instalar distintos paquetes de *software* para el correcto funcionamiento.

La instalación se ha realizado en 7 máquinas con las siguientes características:

	1 <i>Manager</i>	2 <i>masters</i>	3 <i>workers</i>	1 <i>gateway</i>
Sistema Operativo	CentOS 7	CentOS 7	CentOS 7	CentOS 7
<i>CPU</i>	2 vcores	2 vcores	4 vcores	2 vcores
Memoria	8gb	4gb	8gb	4gb
Disco duro	40gb	20gb	80gb	20gb
<i>GPU</i>	None	None	None	None
<i>Java</i>	1.8.0_101	1.8.0_101	1.8.0_101	1.8.0_101
<i>Python</i>	2.7	2.7	2.7	2.7
Versión <i>CDH</i>	5	5	5	5

Cuadro 2.1: Especificaciones de las máquinas

Con los nodos disponibles lo primero que debemos hacer es elegir que máquina albergará cada rol dentro del *cluster*. Esto vendrá dado por los recursos disponibles de cada nodo y el uso que pretendamos hacer. Hay que tener en cuenta que *Hadoop* es escalable por lo que nuestro *cluster* podrá ser ampliado en número de nodos según sea la demanda de recursos que necesitemos.

En nuestro caso, el reparto de roles queda así:

Roles en el <i>cluster</i>							
	Master1	Master2	Worker1	Worker2	Worker3	Gateway	Manager
<i>namenode</i>	✓						
<i>secondary namenode</i>		✓					
<i>datanode</i>			✓	✓	✓		
<i>hdfs gateway</i>						✓	
<i>resource manager</i>		✓					
<i>node manager</i>			✓	✓	✓		
<i>job history server</i>	✓						
<i>yarn gateway</i>						✓	
<i>cm server</i>							✓
<i>cms database</i>							✓
<i>cm service</i>							✓
<i>cm agent</i>	✓	✓	✓	✓	✓	✓	✓

Cuadro 2.2: Asignación de roles entre máquinas

Como pasos previos a la instalación de los servicios del *cluster*, se deben configurar ciertas propiedades del Sistema Operativo tales como parar el servicio *iptables*, instalación de *NTP* para la sincronización de relojes, instalar el repositorio de *Cloudera*... Además de todo esto, debemos instalar Java ya que *Hadoop* lo requiere para poder funcionar.

Todo este trabajo, si bien no es complicado, excede la longitud y los objetivos de este documento, por lo que he omitido la inclusión de esta parte a la cual se puede acceder a través del siguiente link: https://github.com/davidRetana/custom_centOS

Antes de instalar el *software*, tenemos que dar nombre a cada una de las máquinas que formaran el *cluster*. Esto se hace mapeando en el archivo `/etc/hosts` cada ip de la máquina con el nombre que la queramos dar.

```
$ sudo vi /etc/hosts
```

```
127.0.0.1      localhost
127.0.1.1      aceraspire

# The following lines are desirable for IPv6 capable hosts
::1           ip6-localhost ip6-loopback
fe00::0       ip6-localnet
ff00::0       ip6-mcastprefix
ff02::1       ip6-allnodes
ff02::2       ip6-allrouters

# nodos del cluster
10.164.79.110  master1
10.164.79.111  master2
10.164.79.112  worker1
10.164.79.113  worker2
10.164.79.114  worker3
10.164.79.115  gateway
10.164.79.116  manager
```

A continuación copiamos el archivo a cada máquina del *cluster*:

```
$ sudo scp /etc/hosts root@<ip_maquina_destino>:/etc/hosts
```

Para cada máquina del *cluster* hacer:

```
$ sudo hostname <nombre_de_la_maquina>
$ sudo vi /etc/sysconfig/network
```

En el archivo `/etc/sysconfig/network` cambiar `HOSTNAME=<nombre máquina>`

Para probar que los nodos se reconocen por su nombre lanzar el siguiente comando:

```
$ ping <nombre_nodo_destino>
```

Si el nodo responde, lo hemos configurado bien.

Para comenzar el despliegue del *cluster*, instalaremos los dos servicios fundamentales para su correcto funcionamiento y sobre ellos se podrán añadir más servicios en un futuro. Sin embargo, este no es el objetivo del documento, y nos centraremos en instalar **HDFS**, **YARN** y posteriormente **Spark**.

2.1 Instalación de *HDFS* y *YARN*

En la máquina designada como *Cloudera Manager* instalamos el servicio de *cloudera-manager-server*

```
$ sudo yum install cloudera-manager-server
```

Este comando nos instala el servicio en la máquina en cuestión. Una vez finalizado, se nos habrá creado el directorio `/usr/share/cmf` dentro del cual tenemos que lanzar un *script* que nos configura la base de datos que usa *Cloudera Manager* por debajo

```
$ sudo /usr/share/cmf/schema/scm-prepare-database.sh <base_de_datos> <nombre_db> \
<usuario> <password>
```

A modo de ejemplo:

```
$ sudo /usr/share/cmf/schema/scm-prepare-database.sh mysql cloudera root training
```

Si todo a funcionando correctamente, accedemos a *mysql*¹ y vemos que se hayan creado correctamente las tablas

```
$ mysql -uroot -ptraining

mysql > show databases;
mysql > exit;
```

Deberemos ver como las bases de datos *amon* y *rman* están creadas. Finalmente arrancamos el servicio del *cloudera-manager-server*

```
$ sudo service cloudera-scm-server start
```

Este proceso tarda en terminar de ejecutarse pero una vez finalizado abrimos un navegador desde nuestro portátil y escribimos en el campo de la url: `<ip_nodo_cloudera_manager>:7180`. A partir de ahora, será el asistente gráfico el que nos guíe a través del proceso de instalación del *cluster* en el resto de las máquinas. Para hacer el login inicial escribimos usuario:admin y password:admin, de esta manera ya estamos dentro del asistente gráfico.

Como se comento previamente, *Cloudera Manager* es un software gratuito pero en su versión *Cloudera Express*, sin embargo, tiene una versión de pago llamada *Cloudera Enterprise* con funciones avanzadas para la gestión del *cluster* además de soporte. Esta opción está disponible durante un mes a modo de prueba gratuita.

A lo largo de toda la instalación nos pedirá diversas opciones de configuración como por ejemplo el uso de remesas de Cloudera, que son sencillamente abstracciones a nivel de repositorios de paquetes para que podamos tener diferentes versiones de un mismo software sin que entren en conflicto y poder elegir que versión usar en cada momento.

Como *Cloudera Manager* necesita acceder a los nodos para desplegar paquetes de software e instalar agentes, necesita acceder por *SSH* a las máquinas por lo que nos pedirá la contraseña de los nodos del *cluster*. En este caso dicha contraseña deberá ser igual en todos los nodos.

¹Sistema de gestión de base de datos relacional

Instalación de clúster

Proporcionar credenciales de inicio de sesión de SSH.

Es necesario el acceso a raíz a los hosts para instalar los paquetes de Cloudera. Este instalador se conectará a los hosts mediante SSH e iniciará sesión directamente como raíz o como otro usuario con privilegios sudo/pbrun sin contraseña para convertirse en raíz.

Iniciar sesión en todos los hosts como: ☒ root ☐ Otro usuario

Puede realizar la conexión mediante autenticación por contraseña o clave pública para el usuario seleccionado anteriormente.

Método de autenticación: ☒ Todos los hosts aceptan la misma contraseña ☐ Todos los hosts aceptan la misma clave privada.

Introducir contraseña:

Confirmar contraseña:

Puerto SSH:

Número de instalaciones simultáneas: (Ejecutar un gran número de instalaciones a la vez puede consumir una gran cantidad de ancho de banda y otros recursos del sistema)

[Volver](#) 1 2 3 4 5 6 7 [Continuar](#)

A continuación se comenzará a descargar los paquetes de código y distribuirlos por las máquinas

Instalación de clúster

Instalando parcels seleccionados

Los parcels seleccionados se están descargando e instalando en todos los hosts del clúster.

▼ CDH 5.9.0-1.cdh5.9.0.p0.23	Descargado: 63%	Distribuido: 0/0	Desempaquetado: 0/0	Activado: 0/0

Si todo funciona bien deberemos ver una pantalla como la siguiente:

Instalación de clúster

Inspeccionar hosts para comprobar si son correctos. [Volver a ejecutar](#)

Validaciones

✓	Se ha ejecutado el inspector en los 10 hosts.
✓	Los hosts individuales han resuelto sus propios nombres de hosts correctamente.
✓	No se han encontrado errores al buscar scripts de inicio (init) conflictivos.
✓	No se han encontrado errores al comprobar /etc/hosts.
✓	Todos los hosts han resuelto localhost en 127.0.0.1.
✓	Todos los hosts comprobados han resuelto los nombres de hosts recíprocos correctamente y a tiempo.
✓	Los relojes de los hosts están sincronizados aproximadamente (en diez minutos).
✓	Las zonas horarias de host son consistentes en el clúster.
✓	No falta ningún grupo ni ningún usuario.
✓	No se han detectado conflictos entre paquetes y parcels.
✓	No se está ejecutando ninguna versión de kernel incorrecta.
✓	Todos los hosts tienen /proc/sys/vm/swappiness definido en 0.
✓	No hay asuntos de rendimiento con los ajustes de Transparent Huge Pages.
✓	La dependencia de versión CDH 5 de Python para Hue es satisfactoria.
✓	0 hosts están ejecutando CDH 4 y 10 hosts están ejecutando CDH 5.
✓	Todos los hosts comprobados en cada clúster están ejecutando la misma versión de los componentes.
✓	Todos los hosts gestionados poseen versiones consistentes de Java.
✓	Todas las versiones comprobadas de los demonios de Cloudera Manager son consistentes con el servidor.
✓	Todas las versiones comprobadas de los Cloudera Manager Agents son consistentes con el servidor.

Resumen de la versión

[Volver](#) 1 2 3 4 5 6 7 [Finalizar](#)

Esta pantalla nos indica que todo ha salido correctamente. Comprueba entre otras cosas que los requisitos mencionados al inicio del capítulo se cumplen y el despliegue del software ha sido satisfactorio.

Posteriormente, en la elección de los servicios a desplegar seleccionamos *HDFS* y *YARN* e indicamos los nodos donde queremos que se instalen acorde a la [Tabla 2.2](#).

A continuación rellenamos las propiedades de los directorios donde los servicios de *HDFS* y *YARN* dejarán los datos y habremos finalizado la instalación del *cluster*.

Configuración de clúster

✓ Primera ejecución Comando

Estado: **Finalizado** Hora de inicio: nov. 23, 1:14:38 PM Duración: 2.4m

Finished First Run of the following services successfully: HDFS, YARN (MR2 Included), Cloudera Management Service.

Detalles Completado(s) 4 de 4 paso(s). ⦿ Todo ⦿ Solo erróneos ⦿ Solo en ejecución

Paso	Contexto	Hora de inicio	Duración	Acciones
➤ ✓ Ejecutar 1 pasos en paralelo 1 pasos completados correctamente.		nov. 23, 1:14:38 PM	206ms	
➤ ✓ Implementando la configuración de cliente Successfully deployed all client configurations.	Cluster 1	nov. 23, 1:14:38 PM	16.08s	
➤ ✓ Iniciar Cloudera Management Service, HDFS 2 pasos completados correctamente.		nov. 23, 1:14:54 PM	70.42s	
➤ ✓ Iniciar YARN (MR2 Included) 1 pasos completados correctamente.		nov. 23, 1:16:05 PM	58.4s	

[Volver](#) 1 2 3 4 5 6 [Continuar](#)

Si todo fue satisfactoriamente, deberemos ser redirigidos a la página principal de administración de *Cloudera Manager*. Esta página será el punto de partida para cualquier configuración que queramos hacer en el *cluster*, así como añadir nuevos nodos, desplegar nuevos servicios, habilitar la alta disponibilidad (HA por sus siglas en inglés *High Availability*²)... Además permite obtener métricas del uso de *CPU*, I/O de disco, uso de red, etc.

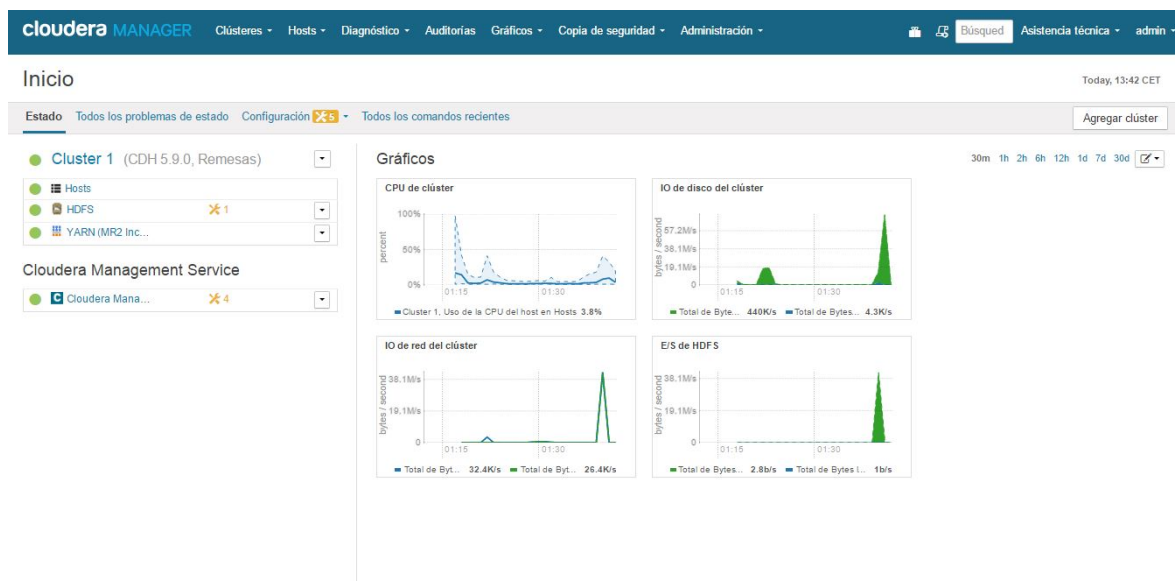


Figura 2.1: Instantánea de la pagina principal de *Cloudera Manager*

2.1.1 Test del *cluster* desplegado

Para comprobar el correcto funcionamiento del *cluster* instalado con *Cloudera Manager*, vamos a subir un archivo a *HDFS*, veremos como se distribuye entre los nodos (en porciones de 128 *MB* por defecto) y luego ejecutaremos un trabajo *MapReduce* que es el motor por defecto que utiliza *YARN v2*.

²Consiste en habilitar un segundo *namenode* en el *cluster* para evitar que un fallo en el *namenode* principal deje fuera de servicio al *cluster* entero.

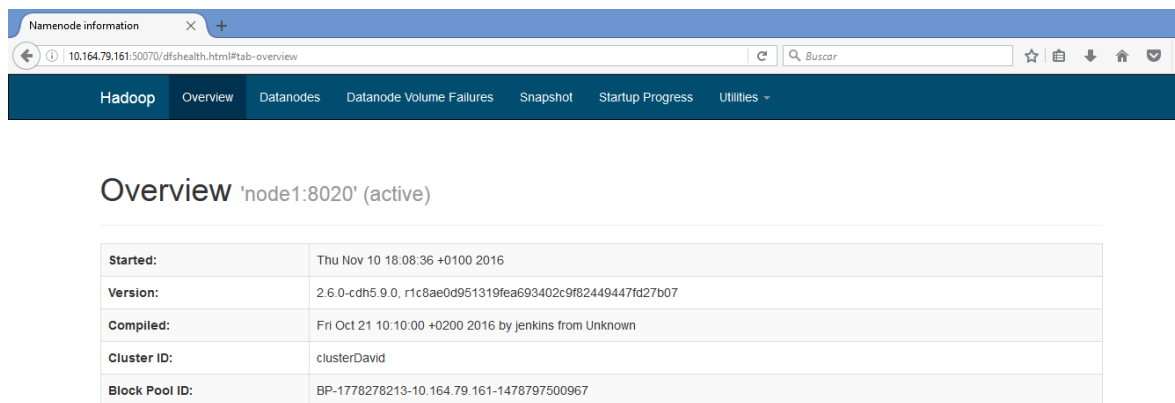
Logeados en nuestra máquina *gateway*, lo primero que debemos hacer es crearnos un directorio de trabajo en *HDFS* para nuestro usuario y darle permisos. Además vamos a crear un directorio temporal donde todos los usuarios puedan escribir.

```
$ # creacion de la home del usuario
$ sudo -u hdfs hdfs dfs -mkdir -p /user/<nombre_usuario>
$ sudo -u hdfs hdfs dfs -chown -R <nombre_usuario> /user/<nombre_usuario>
$ # creacion del directorio temporal
$ sudo -u hdfs hdfs dfs -mkdir -p /tmp
$ sudo -u hdfs hdfs dfs -chmod -R 1777 /tmp
```

Hecho esto, lanzamos el comando para subir un archivo que esta en local a *HDFS* y luego comprobamos que realmente se ha subido:

```
$ hdfs dfs -put <path_archivo_local> <path_en_hdfs>
$ hdfs dfs -ls
```

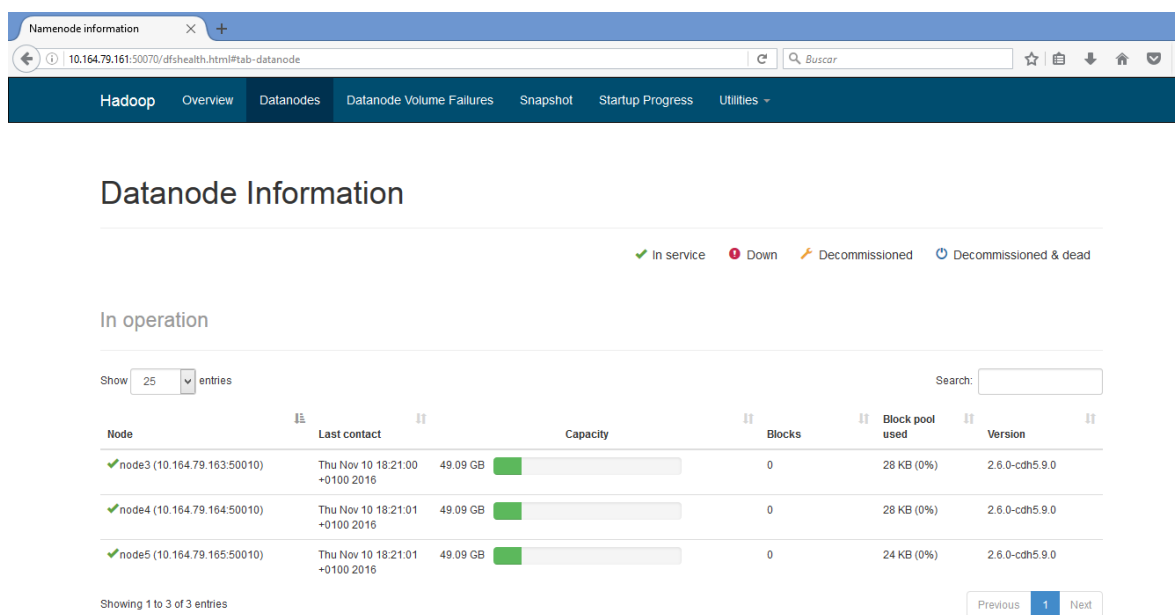
Si todo ha funcionado correctamente, abrimos un navegador y escribimos `<ip_namenode>:50070`



Started:	Thu Nov 10 18:08:36 +0100 2016
Version:	2.6.0-cdh5.9.0, r1c8ae0d951319fea693402c9f82449447fd27b07
Compiled:	Fri Oct 21 10:10:00 +0200 2016 by jenkins from Unknown
Cluster ID:	clusterDavid
Block Pool ID:	BP-1778278213-10.164.79.161-1478797500967

Figura 2.2: *Overview* del servicio web del *namenode*

Si nos vamos a la pestaña de *Datanodes* veremos los nodos trabajadores que tenemos activos y su capacidad de almacenamiento entre otras cosas.



Node	Last contact	Capacity	Blocks	Block pool used	Version
node3 (10.164.79.163:50010)	Thu Nov 10 18:21:00 +0100 2016	49.09 GB	0	28 KB (0%)	2.6.0-cdh5.9.0
node4 (10.164.79.164:50010)	Thu Nov 10 18:21:01 +0100 2016	49.09 GB	0	28 KB (0%)	2.6.0-cdh5.9.0
node5 (10.164.79.165:50010)	Thu Nov 10 18:21:01 +0100 2016	49.09 GB	0	24 KB (0%)	2.6.0-cdh5.9.0

Figura 2.3: Información de los *datanodes*

Para testear *YARN*, vamos a lanzar un trabajo *MapReduce* en dos fases, una primera donde solo se ejecute la fase *map* y luego otra donde solo se ejecute la fase *reduce*.

```
$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.6.0-cdh5.9.0.jar \
  teragen 10000 output_prueba_teragen
```

Veremos una salida parecida a esta:

```
[training@node1 ~]$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.6.0-cdh5.9.0.jar teragen 10000 /user/training/pruebateragen
16/11/11 13:33:32 INFO client.RMProxy: Connecting to ResourceManager at node2/10.164.79.162:8032
16/11/11 13:33:33 INFO terasort.TeraSort: Generating 10000 using 2
16/11/11 13:33:33 INFO mapreduce.JobSubmitter: number of splits:2
16/11/11 13:33:34 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1478865278521_0002
16/11/11 13:33:34 INFO impl.YarnClientImpl: Submitted application application_1478865278521_0002
16/11/11 13:33:35 INFO mapreduce.Job: The url to track the job: http://node2:8088/proxy/application_1478865278521_0002/
16/11/11 13:33:35 INFO mapreduce.Job: Running job: job_1478865278521_0002
16/11/11 13:33:49 INFO mapreduce.Job: Job job_1478865278521_0002 running in uber mode : false
16/11/11 13:33:49 INFO mapreduce.Job: map 0% reduce 0%
16/11/11 13:33:57 INFO mapreduce.Job: map 50% reduce 0%
16/11/11 13:33:59 INFO mapreduce.Job: map 100% reduce 0%
16/11/11 13:33:59 INFO mapreduce.Job: Job job_1478865278521_0002 completed successfully
16/11/11 13:34:00 INFO mapreduce.Job: Counters: 32
```

Ahora ejecutaremos la fase *reduce* donde recibirá como entrada la salida del trabajo anterior

```
$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.6.0-cdh5.9.0.jar \
  terasort output_prueba_teragen output_prueba_terasort
```

Donde nuevamente veremos una salida parecida a esta:

```
[training@node1 ~]$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.6.0-cdh5.9.0.jar terasort pruebateragen pruebaterasort
16/11/11 13:42:21 INFO terasort.TeraSort: starting
16/11/11 13:42:25 INFO input.FileInputFormat: Total input paths to process : 2
Spent 162ms computing base-splits.
Spent 2ms computing TeraScheduler splits.
Computing input splits took 165ms
Sampling 2 splits of 2
Making 1 from 10000 sampled records
Computing partitions took 598ms
Spent 767ms computing partitions.
16/11/11 13:42:25 INFO client.RMProxy: Connecting to ResourceManager at node2/10.164.79.162:8032
16/11/11 13:42:27 INFO mapreduce.JobSubmitter: number of splits:2
16/11/11 13:42:27 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1478865278521_0003
16/11/11 13:42:28 INFO impl.YarnClientImpl: Submitted application application_1478865278521_0003
16/11/11 13:42:28 INFO mapreduce.Job: The url to track the job: http://node2:8088/proxy/application_1478865278521_0003/
16/11/11 13:42:28 INFO mapreduce.Job: Running job: job_1478865278521_0003
```

Si nos vamos a un navegador y escribimos `<ip_resourceanager>:8088`, veremos de una manera gráfica los trabajos que se están ejecutando en nuestro *cluster*, el historial de trabajos subidos, la memoria utilizada, los contenedores asignados...

Llegados a este punto, ya tendremos un *cluster* totalmente operativo con los servicios de *HDFS* Y *YARN* instalados correctamente.

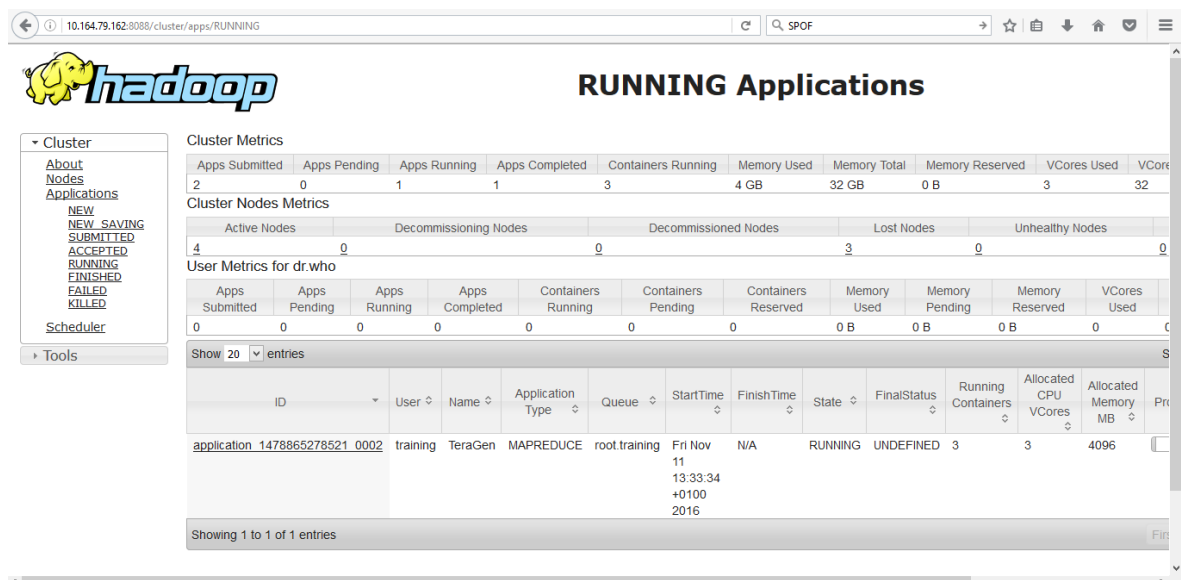


Figura 2.4: Web del servicio *resource manager*

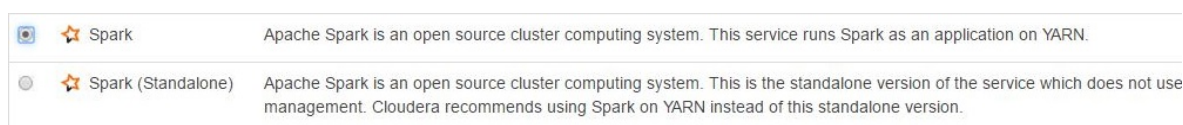
2.2 Instalación de *Apache Spark*



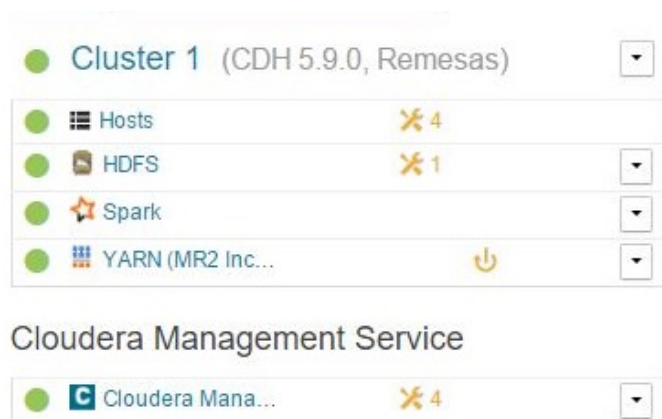
Para la instalación de *Apache Spark*, desde la página principal de administración de *Cloudera Manager* seleccionamos la pestaña 'Agregar un servicio' en el desplegable de *cluster*. En esta pestaña están disponible todos los servicios soportados por Cloudera y que por lo tanto son compatibles con nuestro *cluster*.

Nota: Ya se encuentra disponible la versión 2.0 de *Spark*, pero su instalación es algo diferente ya que debe hacerse desde las *parcels* de Cloudera. Esto es así porque las versiones 2.x.x son incompatibles con las versiones 1.x.x

La opción de *Spark* que cogeremos será aquella que utilice *YARN* como gestor de recursos del *cluster*. Notese que la opción *Spark (Standalone)*³ también está disponible.



Una vez hecho esto, elegimos la máquina que hemos etiquetado como *gateway* para desplegar el servicio de *Spark* ya que será desde esta máquina donde subiremos nuestros trabajos *spark-submit* o *pyspark* al *cluster*. Después de todo este proceso de instalación, nuestra página principal de *Cloudera Manager* debería lucir los 3 servicios que hemos instalado.



Lo último que nos queda por hacer es reiniciar el servicio de *YARN* para que hagan efecto los cambios que hemos realizado en el *cluster*.



Seguimos el asistente de reinicio y habremos terminado la instalación de *Spark*.

Para comprobar su correcto funcionamiento iniciamos un intérprete *pyspark* desde la máquina *gateway*

```
$ pyspark --master local[*]
```

³ *Standalone* es un modo de despliegue autónomo, no requiere de ningún gestor de recursos del *cluster*.

```

avid@aceraspire:~/spark-1.6.1-bin-hadoop2.6/bin$ ./pyspark --master local[*]
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
17/08/21 09:13:10 INFO SparkContext: Running Spark version 1.6.1
17/08/21 09:13:11 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/08/21 09:13:12 WARN Utils: Your hostname, aceraspire resolves to a loopback address: 127.0.1.1; using 10.164.77.211 instead (on interface p2s0f0)
17/08/21 09:13:12 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
17/08/21 09:13:12 INFO SecurityManager: Changing view acls to: david
17/08/21 09:13:12 INFO SecurityManager: Changing modify acls to: david
17/08/21 09:13:12 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(david); users with modify permissions: Set(david)
17/08/21 09:13:13 INFO Utils: Successfully started service 'sparkDriver' on port 42515.
17/08/21 09:13:14 INFO SLF4JLogger: SLF4JLogger started
17/08/21 09:13:14 INFO Remoting: Starting remoting
17/08/21 09:13:14 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriverActorSystem@10.164.77.211:38249]
17/08/21 09:13:14 INFO Utils: Successfully started service 'sparkDriverActorSystem' on port 38249.
17/08/21 09:13:15 INFO SparkEnv: Registering MapOutputTracker
17/08/21 09:13:15 INFO SparkEnv: Registering BlockManagerMaster
17/08/21 09:13:15 INFO DiskBlockManager: Created local directory at /tmp/blockmgr-14b04ff3-43ed-4439-86bd-630a97b5111d
17/08/21 09:13:15 INFO MemoryStore: MemoryStore started with capacity 511.1 MB
17/08/21 09:13:15 INFO SparkEnv: Registering OutPutCommitCoordinator
17/08/21 09:13:15 INFO Utils: Successfully started service 'SparkUI' on port 4040.
17/08/21 09:13:15 INFO SparkUI: Started SparkUI at http://10.164.77.211:4040
17/08/21 09:13:16 INFO Executor: Starting executor ID driver on host localhost
17/08/21 09:13:16 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 38717.
17/08/21 09:13:16 INFO NettyBlockTransferService: Server created on 38717
17/08/21 09:13:16 INFO BlockManagerMaster: Trying to register BlockManager
17/08/21 09:13:16 INFO BlockManagerMasterEndpoint: Registering block manager localhost:38717 with 511.1 MB RAM, BlockManagerId(driver, localhost, 38717)
17/08/21 09:13:16 INFO BlockManagerMaster: Registered BlockManager
Welcome to
  ____
 /  __ \
/   /  \
/_____/
version 1.6.1

Using Python version 2.7.12 (default, Nov 19 2016 06:48:10)
SparkContext available as sc, HiveContext available as sqlContext.
>>>

```

Figura 2.5: *PySpark shell*

En esta terminal tendremos creadas por defecto las variables `sc` (*Spark Context*) y `sqlContext`. La primera es el punto de partida para una aplicación *Spark*, mientras que la segunda habilita características de SQL para el tratamiento de datos.

Si queremos comprobar que todo funciona correctamente podemos ejecutar las siguientes ordenes de *pyspark*:

```
>>> rdd = sc.parallelize(range(100))
>>> rdd.count()
```

Con estas ordenes lo que estamos diciendo es que se paralelice la lista de enteros que va desde el 0 hasta el 99, distribuyéndola entre los nodos, y luego desencadenando una acción que es el contar el número de elementos. Si la orden se completa, hemos instalado correctamente *Spark* en nuestro *cluster*.

Conclusión del despliegue

Esto concluye el despliegue de un *cluster* realizado con *Cloudera Manager*, donde hemos instalado los servicios básicos de *Hadoop* y además el motor de procesamiento *Spark*.

En el siguiente capítulo se comentará las diferencias entre los distintos enfoques de procesamiento y se explica en que consisten los *frameworks* de *MapReduce* y *Spark*. En la **Parte II (Análisis de datos)** se utiliza el *cluster* construido en esta sección para entrenar los modelos de *machine learning* y reducir sus tiempos de ejecución.

Capítulo 3

Computación paralela

La computación paralela consiste en dividir el flujo de procesamiento en instrucciones que se ejecutan simultáneamente y de manera asíncrona con el fin de reducir los tiempos de ejecución de un programa. Es especialmente importante hoy en día debido a la gran cantidad de datos que tratan las aplicaciones desarrolladas en el área del *Big data*.

Este enfoque de programación obliga a rediseñar los algoritmos de procesamiento, en el libro [3] se encuentra una buena introducción al diseño de algoritmos paralelos.

3.1 Distintos enfoques de procesamiento

A nivel de programación o código hay 3 enfoques de procesamiento de los datos:

- Secuencial
- Concurrente
- Paralelo

A nivel de procesador también existe el concepto de paralelismo ya que las nuevas arquitecturas de *microchips* incorporan varios núcleos físicos, además cada núcleo puede manejar varios procesos a la vez (lo que se conoce como concurrente). Dentro de cada hilo de ejecución, las instrucciones son procesadas una a una en el orden en que aparecen (procesamiento secuencial).

Podemos programar de manera secuencial con *Python*, de manera concurrente usando *Python* y librerías como *numpy* o *multiprocessing*, o podemos programar de manera totalmente paralela usando *Python* y *Apache Spark* o *MapReduce* (en este último, a través de *mrjob*).

Al ejecutar un programa básico *Python*, este se ejecuta como un solo proceso. Si nuestro ordenador tiene por ejemplo 2 núcleos físicos (y 4 virtuales) esto quiere decir que al ejecutarse nuestro código este consumirá un solo núcleo virtual (*virtual core* o *vcore*), es decir, consumirá el 25 % de la capacidad de procesamiento.

Si queremos aprovechar el 100 % de la *CPU* deberemos reescribir nuestro código haciendo uso de librerías tales como *multiprocessing*, para que el programa soporte la concurrencia y así acelerar los tiempos. Esto por regla general no es sencillo y solo es paralelizable hasta el máximo de vcores de la *CPU*. Algunas librerías como *numpy* o *scipy* están optimizadas para aprovechar el máximo de la *CPU* de nuestro ordenador de manera totalmente transparente al programador.

En un sistema *UNIX*, la mejor manera de comprobar la cantidad de recursos que están siendo utilizados por el sistema es desde una terminal (*CTRL + ALT + T* si estamos en Ubuntu) y ejecutar el siguiente comando:

```
$ top
```

Este comando muestra el consumo de recursos de cada proceso que esta corriendo en nuestra máquina. Para monitorizar el consumo de recursos cuando se lance un proceso hay que tener en cuenta que los trabajos en un *cluster* se distribuyen a través de los nodos, por lo que el comando *top* mostrado anteriormente solo nos da información de la máquina donde se ha lanzado.

3.2 Frameworks de procesamiento paralelo

En este trabajo nos centraremos especialmente en dos *frameworks*¹: *MapReduce* y *Spark*. Ambos son motores de procesamiento distribuido aunque con un diseño muy diferente, que dependiendo de la naturaleza del problema a resolver será más conveniente utilizar uno u otro.

MapReduce: es un *framework* de procesamiento inspirado en dos de las principales funciones de la programación funcional: *map* y *reduce*.

- *map*: fase donde se produce el procesamiento en paralelo de los datos. Recibe como entrada tuplas (clave, valor) (k_x, v_x) y genera como salida una lista de tuplas de (clave, valor) también: $[(k_1, v_1), \dots, (k_n, v_n)]$
- *shuffle and sort*: fase donde se produce la mezcla de las claves (las claves iguales van a parar a un mismo reduce) y el ordenamiento de las mismas.
- *reduce*: fase donde se procesa el conjunto de valores para una misma clave.

Adicionalmente, se pueden incorporar más fases para optimizar los trabajos *MapReduce* como por ejemplo la utilización de *combiners*².

MapReduce está incluido en *YARN v2* como motor por defecto. Además, cuenta con implementaciones en diversos lenguajes de programación como *Java*, *Scala* o *Python*.

Los algoritmos implementados en este trabajo se realizarán utilizando la librería *mrjob* de *Python*, accesible a través de <https://pythonhosted.org/mrjob/>

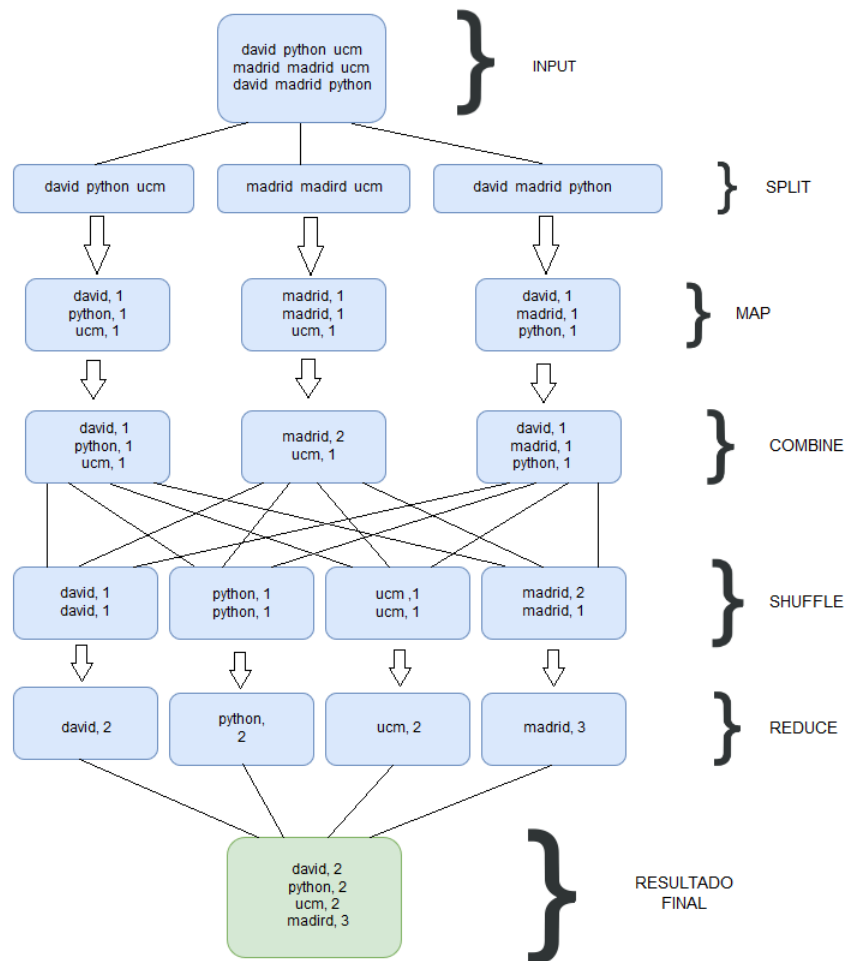


Figura 3.1: Funcionamiento interno de un *word count* en *MapReduce*

¹<https://es.wikipedia.org/wiki/Framework>.

²fase reduce ejecutada localmente en un nodo para reducir el coste del movimiento de datos a través de la red.

Apache Spark: es un *framework* de procesamiento distribuido para grandes cantidades de datos. Su diseño se basa en funciones del paradigma de programación funcional, realizando los cálculos en memoria, lo que le da mayor rapidez que los motores basados en disco como *MapReduce*.

Spark pretende sustituir a *MapReduce* como motor de procesamiento debido a su mayor rendimiento, sobre todo en algoritmos iterativos, lo que acelera mucho el entrenamiento de algoritmos de *machine learning*. *Spark* consta de varias librerías como *ML* o *MLlib* para *machine learning* y *GraphX* para el trabajo con grafos. La abstracción de procesamiento en *Spark* es el **RDD** (*Resilient Distributed Dataset*), siendo altamente escalable y tolerante a fallos (mediante puntos de control o *checkpoints*). Además, *Spark* soporta *YARN* como gestor de recursos del *cluster*. *Apache Spark* es un proyecto de la *Apache Software Foundation* de código abierto, descargable a través de <https://spark.apache.org/>



Figura 3.2: Logo de *Apache Spark*

La instalación de *Spark* en el *cluster* se detalla en la [Sección 2.2](#), mientras que para la instalación de *mrjob* en la máquina que hemos designado como *gateway*, solo tendremos que lanzar un sencillo comando.

```
$ sudo pip install mrjob
```

*Pip*³, el gestor de paquetes de *Python*, se encargará de descargar todas las dependencias necesarias para su correcta instalación.

```
david@aceraspire:~$ sudo pip install mrjob
The directory '/home/david/.cache/pip/http' or its parent directory is not owned by the current user and the cache has been disabled. Please check the permissions and owner of that directory. If executing pip with sudo, you may want sudo's -H flag.
The directory '/home/david/.cache/pip' or its parent directory is not owned by the current user and caching wheels has been disabled. Check the permissions and owner of that directory. If executing pip with sudo, you may want sudo's -H flag.
Collecting mrjob
  Downloading mrjob-0.5.10-py2.py3-none-any.whl (305kB)
    100% |#####| 307kB 2.8MB/s
Collecting boto>=2.35.0 (from mrjob)
  Downloading boto-2.48.0-py2.py3-none-any.whl (1.4MB)
    100% |#####| 1.4MB 927kB/s
Collecting PyYAML>=3.08 (from mrjob)
  Downloading PyYAML-3.12.tar.gz (253kB)
    100% |#####| 256kB 3.5MB/s
Collecting google-api-python-client>=1.5.0 (from mrjob)
  Downloading google-api-python-client-1.6.2-py2.py3-none-any.whl (52kB)
    100% |#####| 61kB 8.7MB/s
Collecting filechunkio (from mrjob)
  Downloading filechunkio-1.8.tar.gz
Collecting six>=1.0.1 (from google-api-python-client>=1.5.0->mrjob)
  Downloading six-1.10.0-py2.py3-none-any.whl
Collecting uritemplate>=3.0.0 (from google-api-python-client>=1.5.0->mrjob)
  Downloading uritemplate-3.0.0-py2.py3-none-any.whl
Collecting oauth2client<5.0.0dev,>=1.5.0 (from google-api-python-client>=1.5.0->mrjob)
  Downloading oauth2client-4.1.2-py2.py3-none-any.whl (99kB)
    100% |#####| 102kB 5.2MB/s
Collecting httplib2>=0.9.2 (from google-api-python-client>=1.5.0->mrjob)
  Downloading httplib2-0.10.3.tar.gz (204kB)
    100% |#####| 204kB 3.9MB/s
Collecting rsa>=3.1.4 (from oauth2client<5.0.0dev,>=1.5.0->google-api-python-client>=1.5.0->mrjob)
  Downloading rsa-3.4.2-py2.py3-none-any.whl (46kB)
    100% |#####| 51kB 7.2MB/s
Collecting pyasn1-modules>=0.0.5 (from oauth2client<5.0.0dev,>=1.5.0->google-api-python-client>=1.5.0->mrjob)
  Downloading pyasn1-modules-0.0.11-py2.py3-none-any.whl (60kB)
    100% |#####| 61kB 7.9MB/s
Collecting pyasn1>=0.1.7 (from oauth2client<5.0.0dev,>=1.5.0->google-api-python-client>=1.5.0->mrjob)
  Downloading pyasn1-0.3.2-py2.py3-none-any.whl (63kB)
    100% |#####| 71kB 8.2MB/s
Installing collected packages: boto, PyYAML, six, uritemplate, pyasn1, rsa, httplib2, pyasn1-modules, oauth2client, google-api-python-client, filechunkio, mrjob
Running setup.py install for PyYAML ... done
Running setup.py install for httplib2 ... done
Running setup.py install for filechunkio ... done
```

Figura 3.3: Instalación de *mrjob*

³Python Package Index, <https://pypi.python.org/pypi/pip>.

Parte II

Análisis de datos

Capítulo 4

Machine Learning

Los datos son una fuente de valor por lo que analizarlos y tomar decisiones en función a estos se ha convertido en algo esencial. Cada vez son más las compañías autodenominadas *data-driven company*, es decir, empresas que toman decisiones de futuro e inversión en función al análisis que hacen de sus datos.

El *machine learning* se nutre de los datos ya que permite construir modelos sobre estos, que posteriormente se usarán para tomar dichas decisiones de futuro de la compañía. A modo de ejemplo, estas decisiones pueden ser: ¿Dónde construir un nuevo centro de datos?, ¿Cómo me anticipo a la posible baja de un cliente de mis servicios móviles?, ¿Cómo minimizo el coste de mantenimiento de las infraestructuras?...

Por estas y más razones, el *machine learning* y las grandes cantidades de datos manejadas en el *Big Data* están estrechamente relacionadas.

4.1 ¿Qué es el machine learning?

El *Machine Learning* (ML) es una rama de la inteligencia artificial (IA) que se centra en desarrollar métodos para hacer posible que los sistemas aprendan sin ser programados explícitamente para ello. El denominado aprendizaje automático se basa en los datos de entrada o datos de entrenamiento, que son utilizados para ajustar los modelos tanto predictivos, como clasificatorios o de clusterización.

Podemos clasificar los modelos de *machine learning* en dos clases:

- **Aprendizaje supervisado:** los datos de entrada del algoritmo van etiquetados con la salida esperada, es decir, cada dato de entrada lleva consigo la clase a la que pertenece. De esta manera se pretende que el algoritmo sea capaz de identificar patrones entre los datos de una misma clase para que cuando vea un dato sin etiquetar, este sea capaz de asignarle una clase en función de los patrones aprendidos en la fase de entrenamiento. Unos ejemplos de esta clase de algoritmos serían: regresión lineal, regresión logística, *support vector machine*, redes neuronales...
- **Aprendizaje no supervisado:** los datos pasados al algoritmo no llevan asignados una etiqueta, es el propio algoritmo el que debe aprender patrones similares entre todo el conjunto de datos de entrada.
Algunos ejemplos de aprendizaje no supervisado serían: *KMeans*, *Principal Component Analysis*, *k-nearest neighbors*...

Aunque el concepto de *machine learning* apareció a mediados del siglo XX, es ahora cuando su evolución ha crecido en importancia debido al aumento de la capacidad de computo de los ordenadores. Lo esencial en *machine learning* son los datos, los modelos se alimentan de ellos y es por lo que cuantos más datos tengamos a disposición, más preciso será nuestro modelo entrenado a coste de un mayor tiempo de procesamiento ([1]). Esta afirmación es matizable, ya que en ciertos casos (ver: https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff) la inclusión de más datos a tu algoritmo no aumentara su desempeño.

El ML tiene campos de aplicación muy diversos tales como procesamiento del lenguaje natural, robótica, desarrollo de motores de búsqueda y recomendación, detección de fraude...

4.2 ¿Por qué desarrollar *machine learning* de manera distribuida?

En la era de la generación masiva de datos, el mejor aliado del *Big Data* es el *machine learning*. Gracias a esto podemos generar modelos artificiales en muy diversas áreas para sacar un beneficio mayor a nuestros datos. Podemos crear modelos predictivos, probabilísticos, clasificatorios, sistemas de detección de fraude y anomalías, algoritmos de compresión de datos...

Para entrenar estos modelos necesitamos datos, tradicionalmente, este proceso de entrenamiento del algoritmo se hacía de manera secuencial en una sola máquina. Hay muchos lenguajes de programación, librerías y herramientas para llevar a cabo este proceso de entrenamiento y modelización como por ejemplo *R*, *Matlab*, *Octave* o *Python* mediante la librería *sklearn*. Estas herramientas funcionan muy bien pero solo en conjuntos pequeños de datos.

Si queremos desarrollar un modelo con un buen rendimiento necesitamos muchos datos, tantos que a veces superan ampliamente la memoria disponible del ordenador así como la capacidad de los procesadores para poder procesar toda la información en un tiempo razonable. A medida que el conjunto de datos de entrenamiento crece, las herramientas tradicionales se quedan inservibles para este propósito, por esta razón, estamos en la necesidad de desarrollar algoritmos paralelos que den solución a estos problemas.

En el artículo de investigación [15], se puede comprobar los efectos de la paralelización (en cuanto a términos de rendimiento) de algunos algoritmo de *machine learning*. Esto pone de manifiesto más aún si cabe la necesidad de distribuir los cómputos entre varias máquinas. Como se ve en la [Sección 4.3](#), el proceso de creación de un modelo de *machine learning* se basa en la prueba y error por lo que normalmente es necesario entrenar el modelo sobre el mismo conjunto de datos varias veces, hasta que se obtengan las métricas deseadas. Esto aumenta más aún si cabe el tiempo necesario de entrenamiento, que en entornos secuenciales pueden llegar a ser inviables.

En el [Capítulo 5](#) se estudiará, analizará e implementará de manera paralela algunos de los algoritmos más populares y usados hoy en día de *machine learning*.

4.3 Machine learning pipelines

El desarrollo de un modelo de *machine learning* pasa por una serie de fases o procesos de desarrollo, desde el preprocesamiento de los datos crudos hasta la utilización de dicho modelo en un entorno de producción.

En la vida real los datos provienen de diversas fuentes como sensores, redes sociales o bases de datos. Estos datos no siempre vienen formateados de manera numérica por lo que hay diversos motivos por los cuales el preprocesamiento es algo fundamental.

Nos podemos encontrar con que nos faltan valores en un determinado registro y una determinada columna que deben ser rellenados o eliminados con el fin de utilizarlos posteriormente en nuestro modelo. En los datos crudos suele ser muy común encontrarse con características que son categóricas en vez de numéricas, esto es, un registro puede contener el sexo de una persona (hombre, mujer), el color de un coche (rojo, negro, azul...). Todas estas características deben ser convertidas a un formato numérico (escalar o vector de escalares) para ser procesadas posteriormente.

También puede ser que nuestro *dataset* crudo tal vez contenga *outliers*¹ que queramos eliminar o transformar, etc.

Debido a estos motivos, necesitamos un preprocesamiento de los datos con el fin de limpiarlos y prepararlos para ser consumidos por el modelo matemático.

Posteriormente, en la mayoría de los casos es necesario escalar los datos para un desempeño óptimo, debido a que si no lo hacemos, las características de los datos con un valor numérico más alto tendrán más peso en la función objetivo. Este escalado previo es necesario en modelos como las redes neuronales o la regresión logística.

¹los outliers son valores atípicos respecto al resto de observaciones de una muestra.

De manera esquemática, los pasos a seguir son:

1. Preprocesamiento

- (a) Limpieza y purga: quitar o rellenar registros vacíos, tratamiento de valores nulos y N/A , convenciones en formato de fechas...
- (b) *Feature Engineering*: convertir variables categóricas a numéricas, estandarización o escalado de los datos...
- (c) División de los datos: dividir el conjunto de datos en conjunto de entrenamiento y conjunto de test (generalmente en proporciones 70 – 30 % o similar).

2. Procesamiento

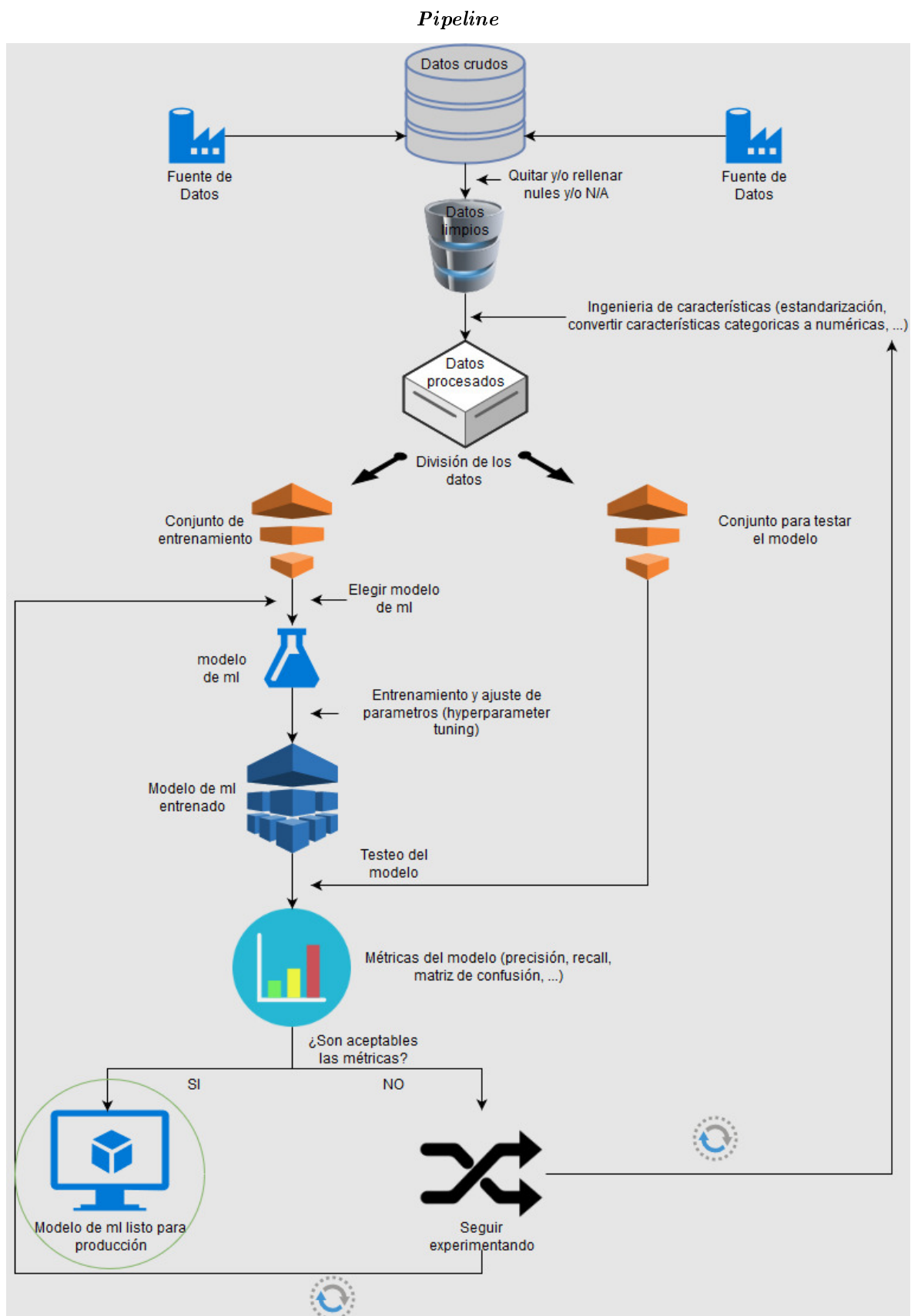
- (a) Elección de modelo que mejor se ajuste a nuestro problema (*Support Vector Machine*, Redes Neuronales, regresión lineal, etc.)
- (b) Entrenamiento: *Hyperparameter tuning*, es decir, ajuste de los parámetros variables del modelo (regularización, grado, tolerancia...).
- (c) Testar el modelo con el conjunto de datos de test. Esto nos arroja las métricas necesarias para identificar cuan de bien se desempeña nuestro algoritmo en datos nunca vistos antes (precisión, *recall*, matriz de confusión...)

3. Producción

- (a) Preprocesamiento usando los mismos criterios que en la primera etapa con datos nunca antes vistos.
- (b) Aplicación del modelo ya entrenado a los datos anteriores para sacar los resultados.

A pesar de que las fórmulas matemáticas en las que se apoyan los modelos de *machine learning* son exactas, el proceso de creación de un modelo no es una ciencia exacta o no hay unas reglas universales. El desarrollo se basa en la experimentación a base de prueba y error sobre el conjunto de datos de entrenamiento. Si las métricas que obtenemos al entrenar un modelo no son las deseadas, no hay un teorema que nos indique exactamente donde se puede mejorar el modelo, debemos ser nosotros como programadores los que tengamos que decidir que hacer.

En la **Figura 4.1** se muestra el flujo de desarrollo de un proyecto de *machine learning*.

Figura 4.1: Diagrama de flujo de un proyecto de *machine learning*

Capítulo 5

Implementación paralela de algoritmos

En este capítulo se implementarán algunos de los algoritmos más populares de *machine learning* pero con un enfoque distribuido, con el fin de minimizar los tiempos de ejecución a medida que el *dataset* se vuelve más grande. Cada sección consistirá en explicar de una manera breve y concisa la problemática a estudiar, el algoritmo a desarrollar y la publicación del código fuente del algoritmo en paralelo.

Para comenzar, vamos a establecer una serie de convenciones para la notación, con el fin de explicar las matemáticas que hay detrás de los algoritmos de *machine learning*.

El numero total de datos de entrenamiento se marcara con la letra m , mientras que es numero total de características de los datos se denotara con la letra n .

Un dato de entrenamiento sera un vector fila donde cada componente del vector será una característica de dicho dato. A cada dato del conjunto de entrenamiento se le denotara con la letra x y un superíndice, mientras que las características se denotaran con la letra x y subíndices.

En caso de que el dato lleve aparejado una clase a la que pertenece, ésta será denotada con la letra y . Así pues, para un primer ejemplo de entrenamiento (supervisado) sería:

$$x^{(1)} = \underbrace{(x_1^{(1)}, x_2^{(1)}, \dots, x_n^{(1)})}_{n \text{ características}} \in \mathbb{R}^n; y^{(1)} \in \mathbb{R}$$

Al conjunto total de datos lo denotaremos con la letra X que representará la matriz que contiene todos los datos de entrenamiento, donde cada ejemplo será un vector fila de la matriz. La letra y será un vector columna conteniendo las clases de sus respectivos datos (aparejados por el índice), por lo que a la fila i -ésima de la matriz le corresponde la clase i -ésima del vector y . Si estamos hablando de un problema de aprendizaje no supervisado, no tendría sentido hablar de las clases aparejadas a cada dato, con lo cual el vector y no existiría.

A modo de ejemplo, el conjunto de entrenamiento para un problema de aprendizaje supervisado quedaría así:

$$X = \begin{pmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ & \vdots & \\ - & x^{(m)} & - \end{pmatrix} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \dots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & x_3^{(m)} & \dots & x_n^{(m)} \end{pmatrix}; y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}$$

De manera análoga si hubiera que hacer distinción entre los datos de entrenamiento y los datos de test, se denotarán X_{train}, y_{train} y X_{test}, y_{test} .

5.1 Aprendizaje supervisado

En el aprendizaje supervisado necesitamos que cada dato de entrenamiento lleve aparejado una clase a la que pertenece. De esta manera nuestro algoritmo intentara adaptarse a los datos lo mejor posible corrigiendo los errores en función de la clase real de cada ejemplo. En esta sección se estudiaran los algoritmos de Regresión Lineal y *NaiveBayes*.

5.1.1 Regresión lineal

La **Regresión Lineal** es un potente algoritmo de *machine learning* que permite predecir el comportamiento de una variable dependiente $y \in \mathbb{R}$ a partir de los valores de la variable independiente $x \in \mathbb{R}^n$. El modelo se puede expresar como una ecuación o recta de regresión

$$h_{\theta}(x) = \theta^T x = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

de la cual se debe ajustar el parámetro desconocido $\theta \in \mathbb{R}^n$ para que el error de las predicciones sea el menor posible.

Dicho error viene representado por la función de perdida cuadrática

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)})^2$$

donde $\theta^T x^{(i)}$ representa el valor predicho para el ejemplo i -ésimo del *dataset*, y la variable $y^{(i)}$ representa el valor real del ejemplo i -ésimo.

Gráficamente, en \mathbb{R}^2 , el problema consiste en ajustar lo mejor posible una recta a los puntos que representan los ejemplos de entrenamiento



Figura 5.1: Recta de regresión

Gradiente de descenso

Uno de los algoritmos más populares de optimización de funciones es el algoritmo del Gradiente de Descenso (*Gradient Descent*). Este algoritmo consiste en actualizar los pesos o variables $\theta_i \forall i = 1, \dots, n$ de manera que con cada iteración se vaya reduciendo el error $J(\theta)$. Las actualizaciones en cada iteración están dadas por la siguiente fórmula (vease libro [1])

$$\theta := \theta - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x^{(i)} \quad (5.1)$$

donde α es un parámetro que controla el salto que se produce en cada iteración del algoritmo, también denominado tasa de aprendizaje o *learning rate*. A valores pequeños del parámetro la iteración se vuelve más lenta pero más segura, por el contrario, valores grandes del parámetro producen que el algoritmo se acelere de manera considerable pero a costa de correr el riesgo de que incluso no llegue a converger.

Una manera lógica de paralelizar este computo es dividir el trabajo en el conjunto de entrenamiento de tal manera que cada máquina trabaje sobre un cierto numero de ejemplos de entrenamiento y no sobre el conjunto total. Supongamos que nuestro conjunto de entrenamiento X tiene un total de $m = 4 \cdot 10^8$ ejemplos. Una iteración del algoritmo de descenso de gradiente tiene que recorrer todos los ejemplos y realizar un calculo con cada uno de ellos, sin embargo, definamos lo siguiente:

$$temp^{(1)} = \sum_{i=1}^{10^8} (h_{\theta}(x^{(i)}) - y^{(i)})x^{(i)}$$

Este cálculo sería realizado en una máquina de tal manera que esta máquina en particular solo contendría un cuarto del calculo total a realizar. De manera análoga definimos

$$temp^{(2)} = \sum_{i=10^8+1}^{2 \cdot 10^8} (h_{\theta}(x^{(i)}) - y^{(i)})x^{(i)}$$

que sería el calculo a realizar por la segunda máquina. Así sucesivamente, hemos dividido el trabajo en 4 porciones que se desarrollan de manera totalmente independiente y paralela.

Sin embargo, el calculo de una iteración del gradiente de descenso no acaba aquí, ya que necesitamos combinar los resultados de estas cuatro máquinas.

Esto de nuevo en sencillo, ya que un ultimo cálculo requeriría juntar las cuatro porciones de la siguiente manera:

$$\theta := \theta - \alpha \frac{1}{4 \cdot 10^8} (temp^{(1)} + temp^{(2)} + temp^{(3)} + temp^{(4)})$$

De manera gráfica, para una paralelización en n partes, cada partición de los datos se alojaría en un nodo como un bloque de datos, el cual sería procesado y luego enviado a través de la red a un nodo que recibiría todos los cómputos y haría la agregación o combinación de los resultados.

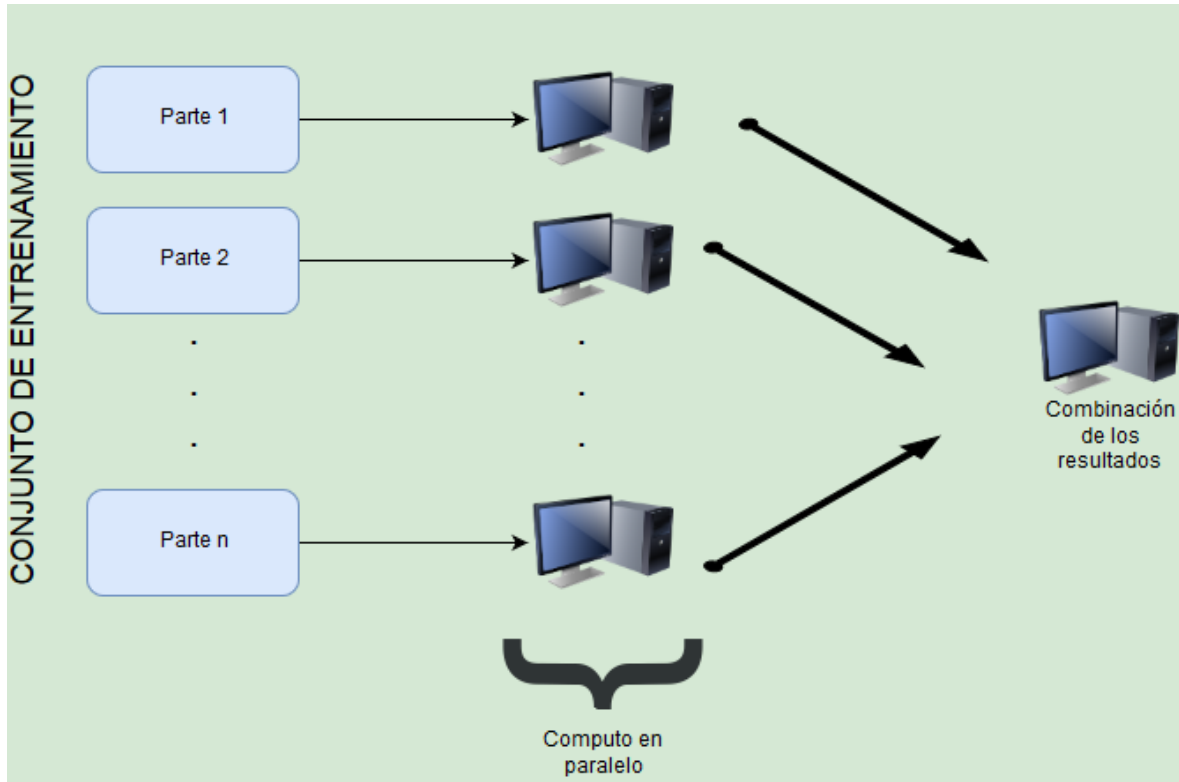


Figura 5.2: Computo en paralelo y agregación de los datos

Código *Spark* Regresión lineal

```

1 from __future__ import print_function, division
import numpy as np
3 from pyspark import SparkContext

5
class LinearRegression():
7
    def __init__(self, alpha=1, tolerance=5, max_steps=10):
9         self.theta = None
        self.alpha = alpha
11        self.tolerance = tolerance
        self.max_steps = max_steps
13
    def map_function(self, t):
15        x, y = t
        error = self.theta.T.dot(x) - y
17        return (error**2, error * x)

19    def reduce_function(self, t1, t2):
        error_squared_1, error_dot_x_1 = t1[0], t1[1]
21        error_squared_2, error_dot_x_2 = t2[0], t2[1]
        return (error_squared_1+error_squared_2, error_dot_x_1+error_dot_x_2)
23
    def fit(self, rdd): # rdd = x; y
25        tuples_x_y = rdd.map(lambda arr: (arr[0:-1], arr[-1]))
        tuples_x_y.persist()
27
        m = tuples_x_y.count() # numero de ejemplos de entrenamiento
29        n = tuples_x_y.first()[0].shape[0] # numero de características
        self.theta = np.random.rand(n, 1) # Inicializamos theta aleatoriamente
31        J = np.inf
        steps = 0
33
        while J > self.tolerance and steps < self.max_steps:
35            errors = tuples_x_y.map(self.map_function)
            sum_errors = errors.reduce(self.reduce_function)
37            sum_root_error = sum_errors[0][0]
            root_error_dot_x = sum_errors[1].reshape((-1,1))
39            J = (1 / (2*m)) * sum_root_error
            self.theta = self.theta - (self.alpha / m) * root_error_dot_x
41            steps += 1

43 if __name__ == '__main__':
45     sc = SparkContext()
47     sep = ','
    alpha = 1
49     tolerance = 5
    max_steps = 10
51     lr = LinearRegression(alpha, tolerance, max_steps)

53     data = sc.textFile("file:///home/training/Desktop/data3.txt")\
        .map(lambda s: np.fromstring(s, dtype=np.float64, sep=sep))
55     lr.fit(data)

```

Listado 5.1: LinearRegression.py

```
$ spark-submit --master yarn-client LinearRegression.py
```

El porqué del uso del *framework Spark* para desarrollar este código ha sido su capacidad para cachear los datos en memoria. Esto es especialmente importante en este tipo de algoritmos ya que al ser iterativos sobre el mismo conjunto de datos, el acceso intensivo a memoria es mucho más rápido que el acceso a disco.

Las diversas iteraciones del algoritmo se centran en actualizar el vector de valores θ para que el error total al ajustar los parámetros θ_i sea el menor posible. El error se calcula en la función *map_function* y no es mas que el valor predicho ($self.theta.dot(x)$) menos el valor real y . A continuación se propaga este error sumandolo al resto de errores generados por cada ejemplo de entrenamiento del *dataset*. El algoritmo se detiene cuando dicho error acumulado es menor que una cierta tolerancia prefijada.

5.1.2 Naive-Bayes

Un clasificador **NaiveBayes** es un clasificador probabilístico que se apoya en el *Teorema de Bayes* para clasificar las entradas. Es un modelo que asume que las características de las variables de entrada son independientes entre sí, esto es, el valor de una cierta variable no influye para nada en el valor de otra.

La idea general detrás del algoritmo es calcular la media y la varianza de cada clase y cada característica. Una vez realizado esto, se pueden utilizar los valores obtenidos para predecir la clase de un nuevo dato de entrada. El modelo lo etiquetará con la clase que más se parezca de las vistas en el conjunto de entrenamiento.

Como se ha mencionado anteriormente, el algoritmo utiliza el teorema de Bayes para asignar la probabilidad de un suceso condicionado a la ocurrencia de otro. En el ejemplo explicado mas abajo dichos sucesos serían la probabilidad a priori y a posteriori de ser hombre o mujer.

Teorema 5.1.1 (Teorema de Bayes) Sean A_1, A_2, \dots, A_n sucesos con $P(A_i) \neq 0 \quad \forall i = 1, 2, \dots, n$. Sea B un suceso cualquiera del que se conocen $P(B|A_i) \quad \forall i = 1, 2, \dots, n$. Entonces:

$$P(A_i|B) = \frac{P(B|A_i) \cdot P(A_i)}{P(B)}$$

Supongamos que queremos clasificar a una persona en hombre o mujer a través de características como la altura, el peso y el numero de pie. Aquí nuestras características serían $n = 3$ y el objetivo sería predecir la variable $y = 0$ si es hombre o $y = 1$ si es mujer. A partir de un *dataset* de ejemplos etiquetados con hombre o mujer, el concepto de aprendizaje para el algoritmo sería calcular la media y la varianza de la altura, el peso y la talla de pie de todos los hombres y mujeres por separado. Con estos datos, estamos en disposición de calcular la probabilidad a posteriori y las probabilidades condicionadas que servirán al algoritmo para realizar sus predicciones.

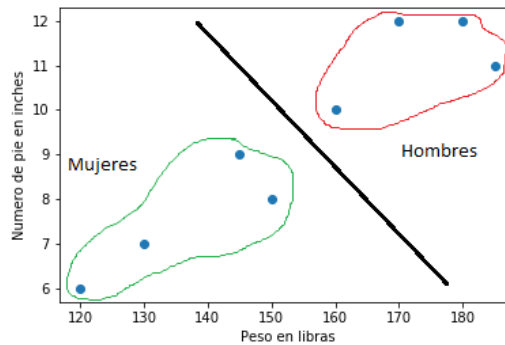


Figura 5.3: Clasificación entre hombres y mujeres

Debido a la simplicidad de los cálculos que usa para entrenarse, *NaiveBayes* se desempeña muy bien en conjuntos de datos muy grandes dando lugar a modelos muy precisos.

Sin embargo, como todo modelo, tiene sus ventajas e inconvenientes que hacen de *NaiveBayes* un modelo propicio para unos determinados tipos de problemas.

- Ventajas:

1. Funciona muy bien en problemas multiclase. Dado un dato, es rápido y fácil predecir su clase.
2. Asumiendo la independencia de las clases, un clasificador *NaiveBayes* obtiene un mejor desempeño comparado con otros métodos como la regresión logística. Además, necesita menos datos de entrenamiento.

- Inconvenientes:

1. Si en los datos de test hay una característica nunca antes vista en los datos de entrenamiento, el modelo le asignará una probabilidad de 0 y será incapaz de hacer una predicción. Esto es conocido como frecuencia cero o *zero frequency*.
2. En la vida real, es casi imposible encontrar un *dataset* donde todas las características sean completamente independientes unas de las otras.

Código MapReduce NaiveBayes

```

1 from __future__ import print_function, division
2 from mrjob.job import MRJob
3 import sys
4
5
6
7 class NaiveBayes(MRJob):
8
9     regex = "," # separador de campos
10
11     def mapper(self, _, line):
12         fields = line.split(self.regex)
13         x = fields[:-1] # características
14         y = fields[-1] # clase o etiqueta
15
16         for i in range(len(x)):
17             yield((float(y), i), (float(x[i]), 1))
18
19     def reducer(self, key, values):
20         m = 0.0 # numero de registros
21         sum_features = 0.0
22         sum_features_squared = 0.0
23         for feature, i in values:
24             sum_features += feature
25             sum_features_squared += feature**2
26             m += 1
27         muj = sum_features / m
28         sj2 = (sum_features_squared + m*muj**2 - 2*muj*sum_features) / m
29         yield (key, (muj, sj2))
30
31 if __name__ == '__main__':
32     if len(sys.argv) != 2:
33         print('Usage naiveBayes: <input_file>', file=sys.stderr)
34         exit(-1)
35     print('Starting parallelized NaiveBayes computation')
36     path = sys.argv[1]
37     job = NaiveBayes(args=[path])
38     runner = job.make_runner()
39     runner.run()
40     tmp_output = []
41     for line in runner.stream_output():
42         tmp_output.append(line.split("\t"))
43     for i in tmp_output:
44         print('element: ', i)

```

Listado 5.2: NaiveBayes.py

```
$ python NaiveBayes.py <input_file> [-r hadoop]
```

Para el desarrollo de este código se ha elegido el *framework MapReduce* debido a que la arquitectura del algoritmo es altamente paralelizable, esto es consecuencia de la asociatividad de las operaciones que se calculan.

En la fase *map* se parsea la línea y se separa en campos (divididos por coma), las características se asignan a la variable x y el último campo se asigna a la variable y , que es la clase.

Como clave se emite una tupla que tiene por valor la clase (y) y la posición del campo (i), mientras que como valor se emite otra tupla que contiene el valor del propio campo ($x[i]$) y un 1 que servirá para hacer un conteo de los datos. Esta elección ha sido así ya que se consigue la máxima paralelización al dividir las claves (*key*) en tantas como campos tengan los registros.

En la fase *reduce* se crean 3 variables: m , $sum_features$ y $sum_features_squared$. La primera sirve para hacer la agregación del conteo de registros, la segunda y tercera sirven como variables auxiliares para posteriormente calcular la media μ_j y la varianza σ^2 .

5.2 Aprendizaje no supervisado

En el aprendizaje no supervisado es el propio algoritmo el que debe sacar los patrones de comportamiento de todo el conjunto de datos. En esta sección se estudiarán los algoritmos de Detección de anomalías y *K-Means*.

5.2.1 Sistema de detección de anomalías

Un sistema de detección de anomalías es un software capaz de detectar comportamientos anómalos a partir de comportamientos previamente establecidos como normales. La base de este modelo es la distribución Gaussiana y requiere que nuestro conjunto de datos tenga variables o características que se distribuyan según una normal de media μ y varianza σ^2 , es decir, $\mathcal{N}(\mu, \sigma^2)$.

Este modelo es usado frecuentemente en detección de intrusos de una red, monitorización de las máquina de un data center, detección de fraude en el uso de tarjetas de crédito...

La distribución Gaussiana (o distribución normal) es una distribución de probabilidad que aparece con mucha frecuencia en fenómenos reales, lo cual es ideal para poder modelar estos fenómenos desde un punto de vista estadístico.

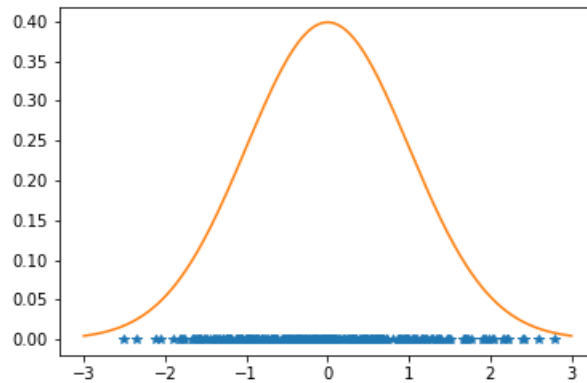


Figura 5.4: Distribución normal de media 0 y desviación típica 1

Esta función será nuestro punto de partida para construir nuestro modelo, el cual asume que las características de los datos siguen dicha distribución, es decir, $\forall j = 1, \dots, n; \quad x_j \sim \mathcal{N}(\mu_j, \sigma_j^2)$.

El objetivo es modelizar una función que denotaremos $p(x)$, la cual, dado un ejemplo nos devuelva la probabilidad de que dicho ejemplo sea anómalo. Más adelante se definirá esta función de manera concreta.

Lo primero que debemos hacer es calcular las medias y varianzas de cada característica de nuestros ejemplos de entrenamiento (matriz X), esto nos da una serie de parámetros $\mu = (\mu_1, \dots, \mu_n)$ y $\sigma^2 = (\sigma_1^2, \dots, \sigma_n^2)$ que se utilizarán posteriormente para predecir la probabilidad de que un nuevo dato sea anómalo. Más concretamente:

$$\mu_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)} \quad \sigma_j^2 = \frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2$$

Una vez que tenemos nuestros parámetros calculados ya podemos definir $p(x; \mu, \sigma^2)$ como:

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2} \frac{x-\mu}{\sigma^2}}$$

Esta fórmula nos da la probabilidad de que un valor de una determinada característica se comporte de manera anómala.

Una vez calculadas todas las curvas gaussianas para cada característica del *dataset*, definimos $p(x)$ como:

$$p(x) = \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2) = p(x_1; \mu_1, \sigma_1^2) \cdots p(x_n; \mu_n, \sigma_n^2)$$

La función $p(x)$ se comporta como un detector de irregularidades ya que si alguna de las funciones $p(x_j; \mu_j, \sigma_j^2)$ para cierto j arroja un valor fuera de lo normal, este quedará reflejado en el valor final de $p(x)$.

Llegados a este punto, debemos establecer un cierto umbral ϵ que nos marque la frontera para considerar un ejemplo como normal o anómalo, es decir, marcaremos un ejemplo x como anómalo si $p(x) < \epsilon$ y será considerado normal si por el contrario $p(x) \geq \epsilon$. Esta elección del parámetro ϵ no es algo universal sino que depende del problema en cuestión (Figura 5.5) y el *dataset* utilizado para modelar $p(x)$. Existen ciertas directrices así como reglas generales para una buena elección de ϵ ¹, pero están fuera de los objetivos de este documento.

Gráficamente, la función $p(x)$ establece una bola n-dimensional con un cierto centro y radio que dependerá del epsilon elegido y el *dataset* utilizado. Toda representación de un ejemplo en el espacio \mathbb{R}^n que quede dentro de dicha bola, será considerado normal, si por el contrario dicha representación queda fuera de la bola, entonces será considerado una anomalía.

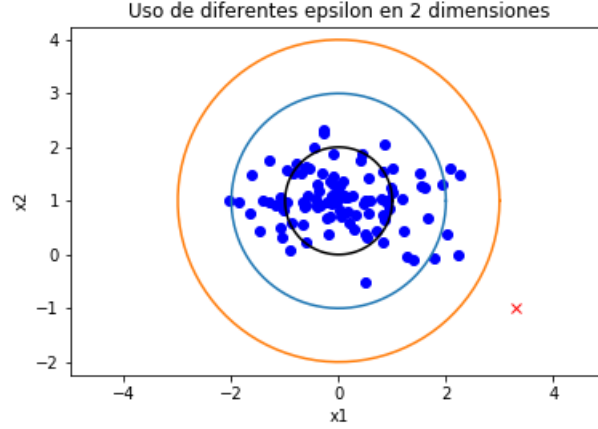


Figura 5.5: Ejemplo de anomalías para distintos valores de ϵ

En la figura los valores se distribuyen según $x_1 \sim \mathcal{N}(0, 1)$; $x_2 \sim \mathcal{N}(1, 0.5)$. Los círculos concéntricos representan la elección de distintos valores del parámetro ϵ , centrados en el punto $\mu = (\mu_1, \mu_2) = (0, 1)$. Si tomamos como referencia el círculo más grande, la x marcada en rojo sería un ejemplo anómalo en ese conjunto de datos

¹<https://www.coursera.org/learn/machine-learning/lecture/Mwrni/developing-and-evaluating-an-anomaly-detection-system>

Código MapReduce para el computo de la media y la varianza

```

1  from __future__ import print_function, division
3  from mrjob.job import MRJob
   import sys
5
7  class ComputeMeanVar(MRJob):
9
11     regex = ","
13
14     def mapper(self, _, line):
15         fields = line.split(self.regex)
16         for i in range(len(fields)):
17             yield(i, (float(fields[i]), 1))
18
19     def reducer(self, key, values):
20         m = 0.0 # numero de registros
21         sum_features = 0.0
22         sum_features_squared = 0.0
23         for feature, i in values:
24             sum_features += feature
25             sum_features_squared += feature**2
26             m += i
27         muj = sum_features / m
28         sj2 = (sum_features_squared + m*muj**2 - 2*muj*sum_features) / m
29         yield (key, (muj, sj2))
30
31 if __name__ == '__main__':
32     if len(sys.argv) != 2:
33         print('Usage computeMeanVar: <input_file>', file=sys.stderr)
34         exit(-1)
35     print('Starting parallelized computation of mean and var')
36     path = sys.argv[1]
37     job = ComputeMeanVar(args=[path])
38     runner = job.make_runner()
39     runner.run()
40     tmp_output = []
41     for line in runner.stream_output(): #stream_output es un generador
42         tmp_output.append(line.split("\t"))
43     for i in tmp_output:
44         print('element: ', i)

```

Listado 5.3: ComputeMeanVar.py

```
$ python ComputeMeanVar.py <input_file> [-r hadoop]
```

La elección de *MapReduce* para desarrollar este algoritmo ha sido debido a que para calcular la media y la varianza de un conjunto de datos, solo es necesario un escaneo completo del *dataset*. Como se ve gráficamente en la Figura 3.1, en la fase *map* se produce el parseo de los datos donde cada línea se divide separando los campos por coma (.). Los valores emitidos son: la posición del campo (*i*) como clave y el valor del campo (*fields[i]*) y un 1 como valor. Este 1 sirve para hacer un conteo de los datos en la fase *reduce*.

En dicha fase *reduce*, se itera sobre los valores recibidos en la fase *map* y se descompone el calculo de la media en 2 variables, aparte de crear otra variable *m* que será un contador de los registros totales por cada campo. Una vez terminado la iteración del *for*, se utilizan las variables *sum_features* y *sum_features_squared* para calcular la media μ_j y la varianza σ^2 .

Para el despliegue de la aplicación, se puede hacer de manera local (usado principalmente para depuración del código) o de manera distribuida haciendo uso de un *cluster Hadoop* para que la computación se produzca en paralelo (*-r hadoop*). Este último modo de despliegue sube el archivo *input_file* a *HDFS* para su posterior ejecución con *MapReduce*. También se puede indicar que el archivo ya se encuentra en *HDFS* poniendo delante el prefijo de *hdfs* en el path: *hdfs://<input_file_in_hdfs>*

5.2.2 K-Means

K-Means es uno de los algoritmos de *clusterización* más extendidos y usados en la actualidad. La idea principal del algoritmo es agrupar los datos de entrada en distintos conjuntos o *clusters*² coherentes, esto es, los puntos dentro de una mismo *cluster* son más parecidos entre sí que los puntos de otro *cluster* cualquiera.

Nuestros datos de entrada son puntos $x^{(i)} \in \mathbb{R}^n$ y un cierto numero $k \in \mathbb{N}$ de *clusters* en los que vamos a agrupar nuestros datos.

Comenzamos estableciendo k puntos en lugares aleatorios de nuestros datos, estos puntos serán los centros de los *clusters* c_1, c_2, \dots, c_k , y los llamaremos *centroides*. Una vez hecho esto, el algoritmo recorrerá cada punto x_i y encontrará el centroide c_j más cercano (en términos de la distancia euclídea) a nuestro punto. Por lo cual, al punto x_i se le asigna el *cluster* j .

Una vez que tengamos todos los puntos asignados a sus respectivos centroides más cercanos, actualizamos los centroides con la media de todos los puntos que pertenecen al *cluster* de dicho centroide.

Las iteraciones del algoritmo se detendrán cuando en dos iteraciones sucesivas ninguno de los puntos sea asignado a otro *cluster* distinto al anterior o cuando la norma del vector que componen la resta de los centroides de dos iteraciones consecutivas sea menos que un cierto ϵ prefijado.

Pasos del algoritmo K-Means:

Input: Puntos en \mathbb{R}^n y un numero $k \in \mathbb{N}$ de *clusters* en los que agrupar nuestros datos.

1. Insertar k centroides c_1, c_2, \dots, c_k en localizaciones aleatorias.
2. Calcular la distancia entre cada punto $x^{(i)}$ y cada centroide c_i .
3. Asignar a cada punto $x^{(i)}$ al *cluster* cuya distancia al centroide sea menor que la distancia al resto de centroides.
4. Recalcular los nuevos centroides usando la formula $c_i = \frac{1}{m_i} \sum_{j=1}^{m_i} x^{(i)}$ donde m_i es el número de puntos que pertenecen al *cluster* i y $x^{(i)}$ son todos los puntos de dicho cluster, más concretamente $x^{(i)} \in \{p \in \mathbb{R}^n \mid d(p, c_j) < d(p, c_s) \forall s \in 1, \dots, k; s \neq j\}$.
5. Calcular la norma entre los centroides anteriores y los nuevos centroides $\|(c_1^i, c_2^i, \dots, c_k^i) - (c_1^{i-1}, c_2^{i-1}, \dots, c_k^{i-1})\|_\infty$ donde el superíndice i indica la iteración i -ésima.
6. Si dicha norma es menor que un cierto ϵ prefijado entonces parar (se ha llegado a la convergencia), en caso contrario volver al paso 2.

Output: k centroides c_1, c_2, \dots, c_k .

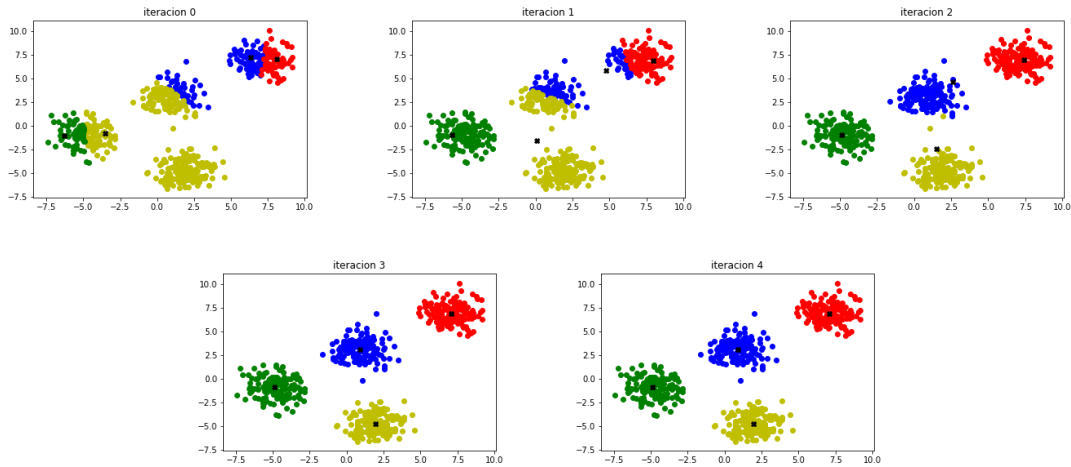


Figura 5.6: Iteraciones del algoritmo k-means

²Notese que no se debe confundir el uso de la palabra *cluster* para hacer referencia a un *cluster* de maquinas, o cuando se usa para referirnos a un conjunto de datos agrupados.

Código *Spark* para el algoritmo de *K-Means*

```

1  from pyspark import SparkContext
3  import numpy as np
   from time import time
5
7  class KMeansSpark():
9
10     def __init__(self, k, epsilon=1, max_iter=10):
11         self.k = k # numero de clusters
12         self.epsilon = epsilon # tolerancia para el criterio de parada
13         self.max_iter = max_iter # maximo numero de iteraciones
14
15     def distance_squared(self, p, q):
16         # Distancia al cuadrado entre dos puntos
17         return np.sum((p - q)**2)
18
19     def closest_centroid(self, p):
20         # para un punto p devuelve el indice del centroide mas proximo a p
21         index = np.argmin(np.linalg.norm(self.centroids - p, axis=1))
22         return index
23
24     def fit(self, X): # X es un RDD de np.array
25         # ajusta los centroides a los datos
26         init_time = time()
27         X.persist() # solo si tenemos suficiente memoria
28         # se empieza con k puntos del dataset eleccionados aleatoriamente
29         initial_centroids = np.asarray(X.takeSample(False, self.k, 42))
30         self.centroids = initial_centroids
31
32         distance = np.inf
33         it = 0 # iteracion
34         while (distance > self.epsilon and it < max_iter):
35
36             # Para cada punto, encontrar el indice del centroide mas proximo
37             # mapearlo a (index, (point, 1))
38             points_clusterized = X.map(lambda p: (self.closest_centroid(p), \
39                                                     (p, 1) ))
40
41             # para cada key (k-point index), hacer una agregacion
42             # de las coordenadas y el numero de puntos
43             clusters_set = points_clusterized\
44                 .reduceByKey(lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]) )
45
46             # para cada key (k-point index), encontrar los nuevos centroides
47             # calculando la media de los puntos de un mismo cluster (centroid)
48             new_centroids = clusters_set\
49                 .mapValues(lambda pair: (pair[0] / pair[1]) )\
50                 .sortBy(lambda t: t[0])\
51                 .map(lambda pair: pair[1]).collect()
52             new_centroids = np.asarray(new_centroids)
53
54             # calculamos la distancia entre los nuevos centroides y los anteriores
55             distance = np.linalg.norm(self.centroids - new_centroids, ord=np.inf)
56
57             #Asignamos los nuevos centroides al array de centroides de la clase
58             self.centroids = new_centroids
59             it += 1
60             time_passed = time() - init_time
61             if it == max_iter:
62                 print('Maximum number of iterations reached')
63                 print('{} iterations terminated in {} seconds'.format(it, time_passed))
64             else:
65                 print('Convergence successfull')
66                 print('{} iterations terminated in {} seconds'.format(it, time_passed))

```

Listado 5.4: KMeansSpark.py

Con el fin de testar el código anteriormente escrito:

```

1 if __name__ == '__main__':
2
3     sc = SparkContext(appName='KMeansSpark')
4     k = 4
5     epsilon = 0.1
6     max_iter = 10
7
8     from_file = False
9
10    # Testarlo localmente
11    if not from_file:
12        np.random.seed(42)
13        D, N = 2, 150
14        mu0 = np.array([1, 3])
15        X0 = (np.random.randn(D, N) + mu0[:, np.newaxis]).T
16        mu1 = np.array([7, 7])
17        X1 = (np.random.randn(D, N) + mu1[:, np.newaxis]).T
18        mu2 = np.array([2, -5])
19        X2 = (np.random.randn(D, N) + mu2[:, np.newaxis]).T
20        mu3 = np.array([-5, -1])
21        X3 = (np.random.randn(D, N) + mu3[:, np.newaxis]).T
22        X = np.vstack((X0, X1, X2, X3))
23        X = sc.parallelize(X)
24    else:
25        number_partitions = 2
26        sep = " "
27        X = sc.textFile('/path/to/your/textfile', number_partitions)\
28            .map(lambda s: np.fromstring(s, dtype=np.float64, sep=sep))
29
30    k_means = KMeansSpark(k, epsilon)
31    k_means.fit(X)
32
33    for centroid in k_means.centroids:
34        print(centroid)

```

Listado 5.5: KMeansMain

```

$ # master puede ser local[*] o yarn-client
$ spark-submit --master yarn-client KMeansSpark.py

```

Para el desarrollo de este código se ha elegido el *framework Spark* debido principalmente a que el algoritmo de *K-Means* es un algoritmo iterativo sobre los mismos datos para realizar los computos. Debido a la capacidad de *Spark* de cachear los datos en memoria, los tiempos de ejecución se reducirán considerablemente.

Los datos de entrada del algoritmo es un *RDD* que contiene como registros objetos *numpy arrays*. Lo primero que se hace es cachear los datos en memoria y a continuación inicializar los centroides en posiciones aleatoria de los datos de entrada.

Dentro del bucle *while*, la variable *points_clusterized* contiene un mapeo de los datos originales a una tupla que contiene el centroide más cercano al punto *p* en cuestión y un 1 que permite realizar posteriormente el conteo de los puntos totales. La variable *clusters_set* realiza una agregación que suma los puntos de entrada y los unos anteriores para realizar la media de cada *cluster* de puntos.

La variable *new_centroids* realiza un mapeo de los valores para calcular la división de la suma de los puntos y el conteo, de esta manera se obtienen los nuevos centroides que se guardan en la variable *new_centroids*. Por último, se calcula la distancia entre los nuevos centroides y los anteriores con el fin de calcular la distancia para el criterio de parada (del bucle *while*).

Las iteraciones se detendrán cuando la distancia entre los centroides de dos iteraciones consecutivas sea menor que un cierto ϵ prefijado (o cuando se superen las máximas iteraciones permitidas).

El modo de despliegue de la aplicación depende de si como *master* ponemos que se ejecute en local (*local[*]*) o en distribuido (*yarn-client*). El primero no distribuye ningún cálculo en absoluto mientras que el segundo es un cliente del gestor de recursos del *cluster*, *YARN*.

Conclusión y líneas de trabajo futuras

Hemos visto como desplegar un *cluster* de máquinas utilizando el software *Apache Hadoop*, y posteriormente utilizarlo para desarrollar algoritmos paralelos de *machine learning*. Dichos algoritmos se han desarrollado tanto en *MapReduce* como en *Spark*.

Respecto a los conocimientos necesarios para abordar este trabajo, del grado en **Ciencias Matemáticas** cabe destacar por su especial utilidad las asignaturas de programación paralela, geometría computacional y programación declarativa entre otras. Todas ellas pertenecientes al itinerario de **ciencias de la computación**. Adicionalmente a estos conocimientos, también ha sido especialmente necesario aprender el funcionamiento de los sistemas *UNIX* (en particular de *LINUX*), sobre todo su uso a través de la línea de comandos (*CLI*, por sus siglas en inglés *Command Line Interface*). Si bien no hay una asignatura específica para aprender estos sistemas operativos en la carrera de Matemáticas, el libro [2] es un buen punto de partida para comenzar.

Evaluación de los objetivos

Los objetivos expuestos en la sección de **Objetivos y plan de trabajo** se han realizado siguiendo el plan de trabajo establecido. A modo de evaluación vamos a repasarlos:

- Instalación de un *cluster Hadoop* de máquinas virtuales.

El despliegue del *cluster* se ha realizado sobre máquinas virtuales usando *Cloudera Manager* como herramienta principal. En la **Subsección 2.1.1** se comprobó como la instalación se realizó correctamente y todos los servicios desplegados (*HDFS*, *YARN*...) funcionaban bien.

En esta parte la mayor dificultad radica en la instalación de todos los componentes que necesita *Hadoop* para su correcto funcionamiento, es decir, *Java*, *MySQL*, *NTP*...

Como *Cloudera Manager* es un asistente gráfico, facilita enormemente el trabajo que va por detrás ya que lo gestiona automáticamente. Sin embargo, para realizar dicho despliegue conviene tener muy clara la teoría detrás de los servicios de *Hadoop*, donde viene muy bien explicada en el libro [16].

Respecto a los *frameworks* de procesamiento que hemos instalado, el proceso ha sido bastante sencillo debido a las herramientas de *Cloudera* y al gestor de paquetes de *Python*. Esta parte no supuso mayor complicación.

- Desarrollo de algoritmos de *machine learning* de manera paralela.

Los algoritmos desarrollados cumplen con las características que todo algoritmo distribuido debe cumplir, especialmente en lo que se refiere a la **escalabilidad**. El incremento de los datos a procesar solo penaliza el rendimiento en cuanto a tiempo de ejecución y nunca llega a colapsar el programa. Tanto los algoritmos desarrollados en *MapReduce* como en *Spark* cumplen con dichas condiciones de escalabilidad si bien el diseño de cada uno de ellos es diferente debido a su arquitectura interna.

La clave a la hora de conseguir este objetivo es tener en mente que el diseño de los algoritmos paralelos se basa en no guardar variables en memoria que puedan colapsar la capacidad del nodo trabajador. El proceso de desarrollo de algoritmos distribuidos implica cambiar la mentalidad a la hora de escribir el código ya que los datos se encuentran repartidos en distintas máquinas que funcionan de manera asíncrona. El reto en este objetivo era desarrollar estos algoritmos en lo que a buenas prácticas se refiere.

Líneas de trabajo futuras

Este proyecto puede servir como base para futuros trabajos relacionados con los temas que aquí se tratan, entiendase *Big Data*, *Machine Learning*, *Apache Hadoop*, *Apache Spark*...

Como primera línea de trabajo futuro se puede relacionar con temas de **Deep Learning**, que es una rama encuadrada dentro del *machine learning* y está enfocada en las redes neuronales convolucionales, que por su naturaleza estas toman un gran tiempo de entrenamiento.

Dentro del *deep learning*, se puede avanzar en el estudio y desarrollo de técnicas paralelas para poder entrenar redes neuronales convolucionales (*CNN*, por sus siglas en inglés *Convolutional Neural Network*) en un *cluster* y así reducir los tiempos de ejecución. En la tabla **Tabla 5.1** se muestran las principales diferencias conceptuales entre *machine learning* y *deep learning*.

	<i>Machine Learning</i>	<i>Deep Learning</i>
Conjunto de entrenamiento	medio	grande
Ingeniería de características	manual	automática
Clasificadores disponibles	muchos	pocos
Tiempo de entrenamiento	medio	grande

Cuadro 5.1: Diferencias entre *Machine Learning* y *Deep Learning*

Otra posible línea de investigación reside en el uso de **GPU**³ para la aceleración del proceso de entrenamiento de una red neuronal, y en concreto de una *CNN*. Este proceso paralelo se puede ejecutar bien sea en una sola máquina (como puede ser un ordenador personal) o bien en un *cluster* de máquinas donde cada nodo lleve incorporado una Unidad de Procesamiento Gráfico.

Los procesadores gráficos de **NVIDIA** poseen una arquitectura de cálculo paralela llamada **CUDA** (<http://www.nvidia.es/object/cuda-parallel-computing-es.html>) que permite aprovechar dicha tarjeta gráfica para realizar cálculos. Es especialmente útil en la multiplicación de matrices ya que esencialmente una red neuronal se compone de matrices distinguidas en varias capas. Este trabajo puede servir de base para futuros proyectos acerca de la utilización de *GPU's* en *clusters* de máquinas.

Una tercera línea de investigación posible es la utilización de algoritmos para otros fines de los que inicialmente fueron destinados. Para la compresión de imágenes se pueden utilizar distintos algoritmos de *machine learning* como por ejemplo *KMeans* o *PCA*⁴ (vease [4]).

Un tipo especial de redes neuronales denominadas autocodificadores o *autoencoders* son aquellas que tienen 3 capas (una de entrada, una oculta y otra de salida) donde la capa de entrada y de salida son la misma y la capa oculta posee menos neuronas que las otras dos para así obligar a la red que aprenda a codificar los datos de entrada en un formato más comprimido.

³Graphic Procesing Unit

⁴*Principal Component Analysis*

Parte III

Apéndice

Apéndice A

Kaggle y KDD

Kaggle es una plataforma que aloja datos de diversas fuentes y organiza competiciones para que todo aquel que desee pueda desarrollar sus modelos predictivos y analíticos con el objetivo de conseguir el mayor desempeño. Esta página pone en contacto diversos perfiles de personas (científicos de datos, mineros de datos...) con diversos problemas que a menudo exponen compañías y premian a aquellos equipos de personas que obtengan la mejor puntuación. La plataforma tiene una serie de conceptos sobre los que se desarrolla:

Competiciones Permite a las empresas ponerse en contacto con los científicos de datos de la comunidad Kaggle para resolver determinados problemas para su propio beneficio. Los mejores equipos reciben una compensación económica así como puntos para el ranking interno de Kaggle. Las competiciones están clasificadas por nivel de dificultad, variando desde un nivel principiante para iniciarse en el mundo de la ciencia de datos hasta un nivel experto en el cual obliga a los participantes a desarrollar un proyecto de *machine learning* de inicio a final (*end to end*), esto es, preprocesamiento, ingeniería de características, elección del modelo, evaluación...

Datasets La plataforma aloja datos de muy diversas fuentes a disposición de todo aquel que quiera usarlos para entrenar sus modelos de *machine learning*. Nos encontramos con datos que van desde clasificación o regresión hasta clusterización.

Kernels Los kernels en Kaggle son *scripts* de código que pueden ser ejecutados en la nube y sirven de ayuda al resto de la comunidad para iniciarse en un cierto problema, preprocesar un conjunto de datos, construir un modelo... Los kernels permiten ser valorados por el resto de usuarios que pueden premiar tu trabajo con votos y comentarios.

Además de todo lo mencionado anteriormente, Kaggle dispone de un foro para discutir las diversas problemáticas que puedan surgir a cada usuario. Conecta a miles de *Data Scientist*s de todo el mundo para que intercambien ideas y conocimientos. La plataforma fue fundada por [Anthony Goldbloom](#) en el año 2010 y se puede acceder a través de <https://www.kaggle.com/>

KDD viene de sus siglas en inglés *Knowledge Discover Dataset*, y se refiere al hecho de sacar información útil de un *dataset*, es decir, extraer conocimiento de los datos. En su página web <http://www.kdd.org/> podemos encontrar toda la información relacionada con lo que hacen y a que se dedican. Organizan conferencias y eventos acerca de *KDD*, publican *papers*¹ y noticias acerca de temas de innovación, investigaciones y demás temas en relación con el *KDD*. Cada año lanza una competición abierta al público para que todo aquel interesado pueda descargarse el conjunto de datos que proporciona y realizar la tarea que se busca. Una de las *KDD Cup* más famosas fue la del año 1999 <http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>.

Algunos repositorios de *datasets* de interés:

<https://www.kaggle.com/datasets>

<http://archive.ics.uci.edu/ml/index.php>

<http://deeplearning.net/datasets/>

¹documentos que contienen trabajos científicos

Apéndice B

Cloudera

Cloudera es una compañía que proporciona software basado en *Apache Hadoop*, formación y soporte técnico. De dicho software, en este trabajo se ha usado *Cloudera Manager*, aquí se explica de manera general las 2 posibles opciones para desplegar un *cluster*

- *CDH* (*Cloudera Distribution Hadoop*) es una distribución de *Hadoop* modificada por *Cloudera* que permite instalar *Hadoop* con una serie de paquetes para abstraer al programador de tareas como la instalación manual de todos los servicios de un *cluster*, creación de usuarios, mantenimiento...
- *Cloudera Manager* (CM) es un asistente gráfico que mediante una API REST permite crear y gestionar *clusters* de máquinas de una manera sencilla y visual. Con esta herramienta se pueden desplegar servicios en el *cluster*, montar seguridad (*Kerberos*, *Sentry*...), acceso unificado a los *logs*¹ y demás opciones. También ofrece métricas y estadísticas del *cluster* tales como uso de *CPU*, tráfico de red, I/O de disco... Esta opción sería la más sensata cuando el *cluster* tiene muchos nodos o tiene muchos servicios instalados en él, ya que mantenerlo se volvería una tarea bastante tediosa y propensa a fallos.

Tanto *CDH* como *CM* son de código abierto y cualquiera puede acceder a ellos bajo licencia de *Cloudera*. En este documento se ha detallado la instalación de un *cluster* usando *Cloudera Manager* en la [Sección 2.1](#)



Figura B.1: Logo de Cloudera

¹ Archivos que solo permiten añadir contenido al final del mismo y sirven para saber de manera más detallada lo que pasa en la ejecución de un programa

Bibliografía

- [1] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007.
- [2] Rob Pike Brian W. Kernighan. *The UNIX programming environment*. Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1984.
- [3] Zbigniew J. Czech. *Introduction to Parallel Computing*. Cambridge University Press, 2017.
- [4] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer series in statistics. Springer, 2009.
- [5] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.
- [6] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.
- [7] Holden Karau. *Learning Spark - lightning-fast data analysis, 1st Edition*. O’Reilly, 2015.
- [8] Holden Karau and Rachel Warren. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O’Reilly Media, 1 edition, 6 2017.
- [9] Donald Miner and Adam Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O’Reilly Media, 1 edition, 12 2012.
- [10] Andrew NG. Aprendizaje automático. <https://es.coursera.org/learn/machine-learning>, 2011.
- [11] Mahmoud Parsian. *Data Algorithms: Recipes for Scaling Up with Hadoop and Spark*. O’Reilly Media, Inc., 1st edition, 2015.
- [12] David Peñas. Una pequeña introducción a latex. Pdf, 10 2015.
- [13] Fernando Pérez and Brian E. Granger. Ipython: A system for interactive scientific computing, computing in science and engineering, 2007.
- [14] S. Chris Colbert Stéfan van der Walt and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation, computing in science and engineering, 2011.
- [15] Cheng tao Chu, Sang K. Kim, Yi an Lin, Yuanyuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y. Ng. Map-reduce for machine learning on multicore. In P. B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2007.
- [16] Tom White. *Hadoop: The Definitive Guide*. O’Reilly Media, Inc., 1st edition, 2009.

Índice alfabético

Apache
 Hadoop, 2
 MapReduce, 15
 Spark, 16
 Api Rest, 38
 Aprendizaje
 no supervisado, 18
 supervisado, 18
 Archivo log, 38

 Big Data, ix

 Centroides, 31
 Cloudera, 38
 CDH, 38
 Manager, 38
 Clusterización, 31
 Combiner, 15
 Comodity Hardware, ix

 Data locality, 2
 Data-driven company, 18
 Deep Learning, 35
 Distribución
 Gaussiana, 28
 Normal, 28

 Feature engineering, 20
 Flink, 2

 GPU, 35
 Gradiente de descenso, 23

 HA, 9
 Hadoop
 HDFS, 2
 YARN, 2
 Hyperparameter tuning, 20

 IP, 6

 K-Means, 31
 Kaggle, 37
 KDD, 37
 Kernels, 37

 Machine Learning, 18
 MySql, 7

 NaiveBayes, 26

 Outlier, 19

 PCA, 35

 Pseudodistribuido, 4
 Pyspark, 12

 Recall, 20
 Regresión
 Lineal, 23

 Sistema Operativo, 5
 Spark Context, 13
 Spark-submit, 12
 SQLContext, 13
 SSH, viii
 Standalone, 12

 Teorema de Bayes, 26
 Top, 14

 Vcore, 14

