DISEÑO E IMPLEMENTACIÓN DE CLÚSTER SPARK, Y ANÁLISIS DE DATOS DE TRÁFICO MEDIANTE EL USO DE LA LIBRERÍA MLLIB

ULPGC – Grado de Ingeniería Informática

Memoria Final

Tutores:
Javier Jesús Sánchez Medina
Antonio Ocón Carreras

Félix Cruz Martín
fcruzm95@gmail.com

Resumen

Desarrollo e implementacion de cluster de Big Data basado en Apache Hadoop Spark, sobre multicomputador (MIMD) proporcionado por el CICEI, cuyas máquinas tienen como Sistema Operativo Ubuntu, con énfasis en el estudio de la librería de Machine Learning Spark MLlib aplicada a la predicción del tráfico utilizando un dataset de la plataforma Madrid Open Data. Análisis de Speed-up utilizando diferentes configuraciones de cluster.

Abstract

Apache Hadoop Spark Big Data cluster development, over multicomputer facility provided by CICEI, based on Ubuntu Operating System. The focus of the project is on the Spark Machine Learning library MLlib, applied to traffic prediction, by using the Madrid Open Dataset. Additionally, a Speed-up analysis is developed, comparing different cluster configurations.

Índice

Tabla de Figuras

1. Introducción

1.1. Estado Actual

En la sociedad actualmente somos dependientes de la tecnología a un nivel nunca visto en años anteriores. Esto es debido a que actualmente internet tiene una tasa de penetración de 54.4%, contando con 4.1 mil millones[1] de usuarios humanos. Que constituyen aproximadamente un 48.2% del tráfico total de Internet. Mientras que el 51.8% del tráfico restante es producido por conexiones M2M (*Machine to Machine*). Esto se traduce en un volumen de tráfico de 122.4 exabytes al mes en 2017 aproximadamente.

Con esa cantidad ingente de datos creada de forma diaria, por una población creciente de usuarios, no resulta exagerado decir que el futuro de las organizaciones está en las tecnologías de *Big Data*.

Usando una de las ramas de la Inteligencia Artificial, llamada *Machine Learning* las organizaciones pueden extraer conocimiento útil para su desarrollo, mejorando así la efectividad de su publicidad, hallando formas de optimizar sus procesos, reduciendo gastos en general.

En el panorama actual de las empresas, podemos ver que son las empresas líderes como eBay, Facebook, Twitter y Netflix, las que apuestan por el uso conjunto de las tecnologías de *Big Data* e Inteligencia artificial. Contribuyendo a los proyectos de *Hadoop* y *Spark*, entre otros.

El sector *Big Data* en España está en un estado de crecimiento. Aunque existen algunas grandes empresas relacionadas con esta tecnología. Todavía no se ha extendido a otros sectores como es la educación.

Esta es la razón principal por la que un trabajo de fin de grado usando las tecnologías de *Hadoop* y *Spark* me ha parecido una gran idea. Ya que mediante la implantación de clústeres de computación se puede avanzar a pasos agigantados otros proyectos existentes, así como revolucionar la investigación, implantando nuevas metodologías.

El clúster *Big Data* se desarrollará mediante el uso de las tecnologías *Apache Hadoop* y *Apache Spark*, usando las ventajas proporcionadas por estos tipos de sistemas distribuidos. Hadoop y Spark nos permiten hacer una abstracción de las técnicas de almacenamiento y proceso de los datos en paralelo para así tener mayor facilidad a la hora de crear nuevas aplicaciones.

Tenemos que tener en cuenta que el sistema *Big Data* desarrollado en este trabajo es una demostración conceptual. No está diseñada ni optimizada para la reducción de los tiempos de ejecución, así como tolerancia a fallos ni capacidad para el procesado de grandes cantidades de datos.

1.2. Objetivos

Los objetivos planteados en este trabajo son los siguientes:

1. Estudio de las tecnologías *Big Data* y *Machine Learning.*
2. Diseño de un sistema *Big Data* que se adapte a las necesidades del proyecto usando el hardware disponible del CICEI.
3. Implementación del sistema en base al diseño.
4. Extracción de resultados del procesamiento de los datos.
5. Realización de pruebas de rendimiento del sistema con diferentes configuraciones
6. Extracción de conclusiones del desarrollo del proyecto.
7. Planteamiento de posibles líneas de trabajo dando continuidad a este proyecto.

1.3. Justificación de las competencias específicas cubiertas

TI02. *Capacidad para seleccionar, diseñar, desplegar, integrar, evaluar, construir, gestionar, explotar y mantener las tecnologías de hardware, software y redes, dentro de los parámetros de coste y calidad adecuados.*

A lo largo del desarrollo de este trabajo se ha tenido que desarrollar una infraestructura software que se adaptara a los requisitos puestos por el CICEI, que consistían en el uso del MIMD como plataforma hardware, incluyendo las protecciones necesarias para evitar su compromiso ya que este se encuentra en la red interna de la ULPGC.

TI03. *Capacidad para emplear metodologías centradas en el usuario y la organización para el desarrollo, evaluación y gestión de aplicaciones y sistemas basados en tecnologías de la información que aseguren la accesibilidad, ergonomía y usabilidad de los sistemas*

La infraestructura software que se ha utilizado permite sencillez a la hora de ser usada tanto por los usuarios, como por los encargados de su mantenimiento y capacidad de ampliación de cara al futuro.

TI04. *Capacidad para seleccionar, diseñar, desplegar, integrar y gestionar redes e infraestructuras de comunicaciones en una organización.*

Se ha modificado una parte de la estructura de red del CICEI de forma que los distintos nodos del clúster sean capaces de intercomunicarse, así como la posibilidad de acceder a su interfaz web desde la red interna de la ULPGC.

TI05. *Capacidad para seleccionar, desplegar, integrar y gestionar sistemas de información que satisfagan las necesidades de la organización, con los criterios de coste y calidad identificados.*

Este trabajo ha servido como una prueba a la hora de implementar un clúster Hadoop Spark con el menor coste posible. La prioridad ha sido en reutilizar el hardware disponible por el CICEI para realizar una demostración conceptual de esta plataforma para su estudio posteriormente.

1.4. Aportaciones

A lo largo de la realización de este trabajo se han realizado un número de aportaciones relevantes, las cuales se pasarán a describir en los siguientes párrafos.

La aportación más importante es la de la creación de un sistema multicomputador para el procesado de datos utilizando el sistema operativo *Ubuntu* y una arquitectura basada en *Hadoop*/*Spark*. También se ha trabajado con *Apache Zeppelin* para el manejo de los datos y creación del modelo *Machine Learning*. En el entorno actual español, pocas organizaciones trabajan con *Big Data* y *Machine Learning*, por lo que este trabajo nos permite situarnos como referentes en el uso de estas tecnologías.

La siguiente aportación consiste en la creación de un modelo de Regresión Lineal que se usará para predecir las velocidades medias futuras en los puntos de la M-30 usando la librería *MLlib*. Para llegar a este paso, antes se ha tenido que realizar un trabajo de preprocesamiento de los datos en el que se han eliminado las características superfluas. Para este trabajo de preprocesamiento se ha usado la librería *SparkSQL*, que nos permite usar código Scala para realizar consultas SQL. Este trabajo se ha hecho usando la plataforma de *Apache Zeppelin*, la cual es un *notebook* en el cual nuestro código se puede separar en distintas notas que se pueden ejecutar en cualquier orden, lo que facilita el descubrimiento de errores.

Otra parte destacada de las aportaciones es la creación de una aplicación web diseñada para visualizar los puntos de recogida de datos. En este proyecto web se ha utilizado el *framework* de Javascript NodeJS ya que es una tecnología emergente como *Hadoop* y *Spark*, así como la librería de creación de mapas de *Here.com*[2].

Todas estas aportaciones se han realizado de forma detallada con la intención de que se pueda usar la memoria de este TFG como documentación de inicio para otras personas que quieran empezar a usar estas tecnologías y técnicas.

2. Desarrollo

2.1. Metodología

La metodología usada en este trabajo la podemos dividir en las siguientes fases.

- Investigación
    - Realización del curso Hadoop Spark
    - Investigación de la librería MLlib
    - Introducción a la gestión del tráfico urbano
- Diseño
    - Diseño del clúster Spark
    - Implementación del clúster.
- Evaluación
    - Importación y preprocesamiento del dataset proporcionado por el CICEI.
    - Creación algoritmo MLlib.
    - Análisis y visualización de los datos.
- Documentación
    - Redacción de la memoria
    - Preparación de la defensa.

La fase de investigación se ha dedicado a buscar la información necesaria para el desarrollo de este trabajo de fin de grado. Consiste de 3 sub-fases, en la primera se decidió la realización de un curso sobre las tecnologías de *Hadoop* y *Spark* en la plataforma de e-learning llamada Udemy. Este curso, llamado "Monta un clúster big data de cero", tiene una valoración de 4.5 estrellas sobre 5 y su instructor, llamado *Apasoft Training* goza de una valoración de 4.6 estrellas con más de 1000 valoraciones de sus 4900 estudiantes. El curso, consta de 146 lecciones entre las cuales se destacan, ya que son las útiles para la realización de este trabajo, la introducción a Big Data y Hadoop, preparación de Hadoop y Spark. También se implementará un clúster para la realización de los ejercicios prácticos del curso mediante el uso de máquinas virtuales.

Tras realizar las lecciones adecuadas a este trabajo, se procede a la búsqueda de información de la librería *MLlib*. De la cual podemos encontrar una documentación bastante completa en la página web de *Spark*. En la última sub-fase de investigación, se realizará el estudio de los datos proporcionados por el CICEI.

En la fase de diseño, utilizaremos los computadores disponibles del CICEI para la creación del clúster. Y procederemos a la implementación hardware y software de este.

Para la fase de evaluación, crearemos un algoritmo de *Machine learning* basado en un modelo de regresión lineal para obtener una predicción de la velocidad media en cualquier punto de medida de la autopista M-30 de Madrid. Este modelo se usará para

verificar el correcto funcionamiento del clúster, así como para realizar una comparación de velocidad con distintas configuraciones de este.

Para la visualización de este modelo, se creará una aplicación web basada en el *framework* NodeJS que mostrará una comparativa entre las velocidades medias reales contra las velocidades medias que ha predichas por el algoritmo.

2.2. Conceptos Teóricos

2.2.1. Big Data

En la actualidad estamos más interconectados que nunca. Actualmente Internet cuenta con una tasa de penetración del 56.8%. Lo que significa que más de la mitad de la población mundial, unos 4.3 mil millones de usuarios. Y con predicciones de una tasa de penetración del 59.7% en 2022[4]. Gracias a estas cifras y al aumento de información generada por cada usuario de internet. Según un estudio en domo.com cada día se crean 2.5 Quintillones de bytes. Estos datos nos sirven para poner en contexto la necesidad de las tecnologías denominadas con el término *Big Data*.

El término *Big Data* que se puede traducir al español como "Macrodatos"[5] hace referencia a conjuntos de datos demasiado grandes y complejos para las tecnologías de procesamiento tradicionales. De esta forma, se han desarrollado nuevas tecnologías para el procesamiento de este tipo de conjuntos de datos, que se basan en la computación distribuida.

El uso moderno del término *Big Data* se refiere al análisis de los datos provenientes de los usuarios que interactúan con una aplicación o recurso, extrayendo su valor y usándolos para crear predicciones bien fundamentadas en el comportamiento de estos.

A la hora de trabajar con conjuntos de datos que podemos denominar *Big Data* nos podemos encontrar con una serie de dificultades centradas en la gestión de los datos con los que vamos a trabajar, que describiremos de forma más específica como la recolección y el almacenamiento de los datos, la búsqueda, compartición, análisis y visualización de estos.

Las características que definen el *Big Data* se pueden describir como las 5V's[6]:

2.2.1.1. Volumen

El volumen de los datos generados hoy en día es demasiado masivo. Por eso supone un gran reto técnico y analítico importante para las organizaciones que gestionan los datos.

2.2.1.2. Variedad

El origen de los datos es heterogéneo ya que estos provienen de muchos tipos de sensores (cámaras, micrófonos, dispositivos IoT, ficheros de log, otros tipos de sensores). Los datos recopilados pueden estar estructurados, aunque lo normal es que no tengan estructura. Siendo necesaria la preparación de estos para su posterior procesado.

2.2.1.3. Velocidad

En el ámbito del *Big Data* la información crece de manera desmesurada, un ejemplo nos lo muestra la página domo.com, que según su estudio de datos de 2017 en ese año se generaban 2,5 quintillones de bytes o 2,5 Exabytes diariamente. Por lo que se necesita velocidad a la hora de recopilar, almacenar y procesar los datos.

2.2.1.4. Veracidad

Dado el gran volumen de datos que se pueden generar, lo normal es que dudemos de su veracidad, ya sea por fallos a la hora de recogerlos o errores al almacenarlos. Por eso los datos deben ser limpiados y analizados para asegurar un grado de veracidad adecuado a los objetivos que se pretenden realizar.

2.2.1.5. Valor

Es la característica más importante del *Big Data* ya que, con el valor de los datos, las organizaciones pretenden obtener información útil que les ayude a la hora de realizar decisiones informadas.

El *Big Data* es una herramienta que puede aplicar a una gran variedad de ámbitos como pueden ser empresas con y sin ánimo de lucro, cuerpos gubernamentales, instituciones educativas y medios de comunicación.

Uno de los ejemplos que podemos encontrar hoy en día fácilmente es en el ámbito de las recomendaciones, ya sean en comercio electrónico o plataformas de video o música bajo demanda. Netflix es una empresa líder en su sector de entretenimiento gracias al uso del *Big Data* en forma de un sistema de recomendaciones individualizadas de contenido. De esta forma nos daremos cuenta de que las páginas de inicio de Netflix

de dos cuentas distintas nunca tendrán las mismas recomendaciones en el mismo orden. Ya que estas se basan en una gran cantidad de datos variados que toman de los usuarios de su plataforma.

En el ámbito de los medios de comunicación y publicidad, se abordan los datos como puntos de información sobre millones de personas. Obteniendo así patrones de uso de las plataformas para la distribución de contenido que esté en línea con la mentalidad del usuario.

Otro terreno que se puede beneficiar del *Big Data* es la sanidad. Mediante el estudio de los historiales clínicos y entorno se pueden obtener modelos predictivos que puedan ayudar a los médicos a realizar diagnósticos con un mayor grado de precisión, así como mejoras en la previsión y detección de enfermedades.

Los casos descritos anteriormente son solo unos ejemplos en los que el uso del *Big Data* se puede aprovechar para mejorar las infraestructuras disponibles y así adaptarla a las necesidades de cada ámbito.

2.2.2. Machine Learning[7]

El aprendizaje automático o Machine Learning está formado por la unión de los campos de las ciencias de computación y una rama de la inteligencia artificial, que se dedica a desarrollar técnicas que permitan el proceso de inducción del conocimiento a las computadoras. Podemos definir la definición de aprendizaje cuando el desempeño de su tarea mejora con la experiencia. Los modelos o programas resultantes deben de ser capaces de generalizar comportamientos e inferencias para un conjunto de datos potencialmente infinito.

Este proceso del aprendizaje automático se solapa a menudo con el campo de la estadística inferencial, ya que estos se basan en el análisis de datos. Otra forma de ver el aprendizaje automático es la intención de automatizar partes del metodo cientifico usando métodos matemáticos.

El aprendizaje automático se puede utilizar de muchas maneras, incluyendo motores de búsqueda, clasificación de secuencias de ADN, reconocimiento del habla y lenguaje escrito, juegos y diagnósticos.

Los algoritmos de Machine Learning se encargan de la creación de modelos matemáticos basados en un conjunto de datos llamados "datos de entrenamiento".

Antes de seguir explicando conceptos de Machine Learning se pasará a la explicación de lo que son las Regresiones, y específicamente las Regresiones Lineales, las cuales se usarán como modelo de entrenamiento para la predicción de velocidades medias en cada punto de la M-30. Las Regresiones son procesos estadísticos para estimar las relaciones entre variables, de las cuales una es dependiente y puede haber una o más variables independientes. El análisis de regresión ayuda a entender cómo el

valor de la variable dependiente varía al cambiar el valor de una de las variables independientes, manteniendo el valor de las otras variables independientes fijas. El análisis de regresión se usa mayoritariamente en tareas como la predicción y previsión de resultados, así como para entender la relación que hay entre las variables dependientes e independientes.

Al aplicar este modelo de regresión Lineal con el que se han creado con un dataset de entrenamiento, se obtienen generalizaciones sobre estos sin haber sido programadas de forma explícita.

Los tipos de algoritmos de Aprendizaje Automático más usados son:

2.2.2.1. Aprendizaje Supervisado.

Este tipo de aprendizaje se basa en establecer una relación entre una serie de datos de entrada del sistema con otra serie de salidas deseadas establecidas. Dependen de una serie de datos etiquetados anteriormente, como puede ser el reconocimiento de objetos, detección de spam, reconocimiento de escritura entre otros. Es decir, problemas que ya se han resuelto, pero que seguirán existiendo en el futuro. Este tipo de problemas se pueden clasificar en dos tipos: algoritmos de clasificación y de regresión. La principal diferencia entre estos dos tipos de algoritmos es que en los de clasificación el conjunto de salidas deseadas es discreto, mientras que, en los algoritmos de regresión, el conjunto de salidas deseadas puede tener cualquier valor numérico dentro de un rango.

2.2.2.2. Aprendizaje No Supervisado.

Los algoritmos de aprendizaje no supervisado se basan en un conjunto de datos con un mayor volumen, pero no etiquetados. Este tipo de algoritmos se basa en buscar patrones existentes basados en las características que se encuentran en esos datos. Se usa este tipo de aprendizaje para detectar morfología en las oraciones, clasificar información, etc.

2.2.2.3. Aprendizaje por Refuerzo.

En el caso de los algoritmos de aprendizaje por refuerzo, el sistema basa su aprendizaje en el método de prueba y error con el conjunto de datos previamente etiquetados que se le suministra. Aunque el algoritmo conoce los resultados desde un principio, tiene que deducir cuáles son las decisiones para llegar a ellos, asociando patrones de éxito y desechando patrones de fallos. Esto se realiza de forma iterativa hasta perfeccionar el modelo.

Existen otros enfoques de aprendizaje más complejos, pero no se van a mencionar ya que no es el objetivo principal de este trabajo.

2.3. Tecnologías Utilizadas

2.3.1. Apache Hadoop[8]

Apache Hadoop es un entorno distribuido de datos y procesos fácilmente escalable con hardware relativamente barato. Permite trabajar con una gran cantidad de nodos y de datos (del orden de 1000 nodos y petabytes). Se inspiró en los documentos de Google para *MapReduce* y *Google File System*. *Hadoop* implementa el procesamiento en paralelo a través de nodos de datos con un sistema de ficheros distribuido mediante el uso de nodos maestros que se encargan de la gestión del clúster y nodos esclavos que se encargan del almacenamiento y procesado de los datos.

Sus componentes principales son:

- *Hadoop Common*.
- *MapReduce*/*YARN*.
- *Hadoop Distributed File System* (HDFS).

Un clúster típico *Hadoop* incluye un nodo maestro y varios nodos esclavos. Se debe mencionar que los subsistemas de *Hadoop* que veremos en este trabajo disponen de una WebUI para su gestión.

El paquete *Hadoop Common*, proporciona acceso a los sistemas de archivos soportados por Hadoop. Este contiene los archivos *jar* y *scripts* necesarios para la ejecución de *Hadoop*. Su funcionalidad clave es que cada sistema de ficheros debe proporcionar su ubicación, indicando el nombre del *rack* donde está el nodo esclavo. Las aplicaciones usan esta información para ejecutar el trabajo en el nodo donde se encuentran los datos, reduciendo el tráfico de la red principal. El sistema de ficheros HDFS utiliza esta información a la hora de replicar los datos, manteniendo copias en distintos racks para reducir el impacto en el clúster cuando se produzcan fallos energéticos.

2.3.2. HDFS

HDFS es un sistema de ficheros usado por *Hadoop* que tiene una alta tolerancia a fallos y que puede almacenar una gran cantidad de datos. Su característica distribuida es la más destacada a la hora de almacenar un gran volumen de datos, ya que no es común que se almacenen terabytes o petabytes de datos en la misma máquina. Escala de forma incremental y puede sobrevivir a fallos de hardware sin perder datos. En caso

de que falle un nodo, el clúster puede continuar trabajando sin perder datos o interrumpir el trabajo, esto se produce ya que el clúster redistribuye el trabajo entre los nodos restantes.

El que sea distribuido proporciona ventajas, ya que se puede escalar el espacio de almacenamiento de datos con gran facilidad, para esto sólo tenemos que añadir nuevos nodos al clúster existente. También proporciona redundancia porque se almacenan los ficheros varias veces y en varios equipos distintos. Por defecto se duplican los ficheros en 3 nodos distintos, aunque se puede cambiar el número de réplicas en los ficheros de configuración[1].

En HDFS los archivos se descomponen en bloques. Cada uno de ellos del mismo tamaño que por defecto es de 128 MB. Los bloques que pertenecen al mismo fichero pueden ser almacenados en distintos nodos, lo que permite escribir ficheros de mayor tamaño, así como que se puedan leer de forma paralela.

Al igual que *Hadoop* y *Spark*, HDFS posee una arquitectura maestro/esclavo, los cuales se denominan *Namenode* y *Datanode* respectivamente.

Los *Namenode* actúan como maestros y almacena la estructura de directorios, ficheros y los metadatos necesarios para recomponer dichos ficheros a partir de sus bloques.

Los *Datanode* se limitan a almacenar los bloques que componen cada fichero, así de proporcionarlo a los clientes que lo necesitan.

2.3.3.YARN[9]

MapReduce es un modelo de computación que usa *Hadoop* para realizar tareas de procesado de datos en paralelo sobre un gran volumen de datos. *Hadoop* usa dos métodos de procesamiento de datos distintos, los cuales se llaman *MapReduce* y *YARN*.

No todos los procesos pueden ser abordados mediante el *framework MapReduce*. Solamente son abordables los que pueden implementar los métodos de *map* y *reduce*. La función de *map* se encarga del mapeo y se aplica de forma paralela para cada objeto en la entrada de datos. Lo que produce una lista de pares que se van a usar en la función *reduce*. *Reduce* produce un valor[2] a partir de la lista devuelta por la función de *map*. Gracias a este conjunto de funciones, *MapReduce* transforma una lista de pares en un valor único que combina todos los valores de la lista.

MapReduce utiliza varios procesos para su correcto funcionamiento en un clúster, estos se llaman: *Resource Manager*, *Node Manager* y *Application Master*. El proceso de *Resource Manager* se ejecuta en el maestro y se encarga de gestionar los recursos del clúster, mientras que *Node Manager* se ejecuta en los nodos esclavos y se

[1] ./hadoop/etc/hadoop/hdfs-site.xml
[2] Este valor puede ser nulo o no.

encarga de gestionar los recursos del nodo en el que se ejecuta. Por último, el proceso de *Application Master* gestiona el ciclo de vida y planificación de la aplicación, siendo único por aplicación.

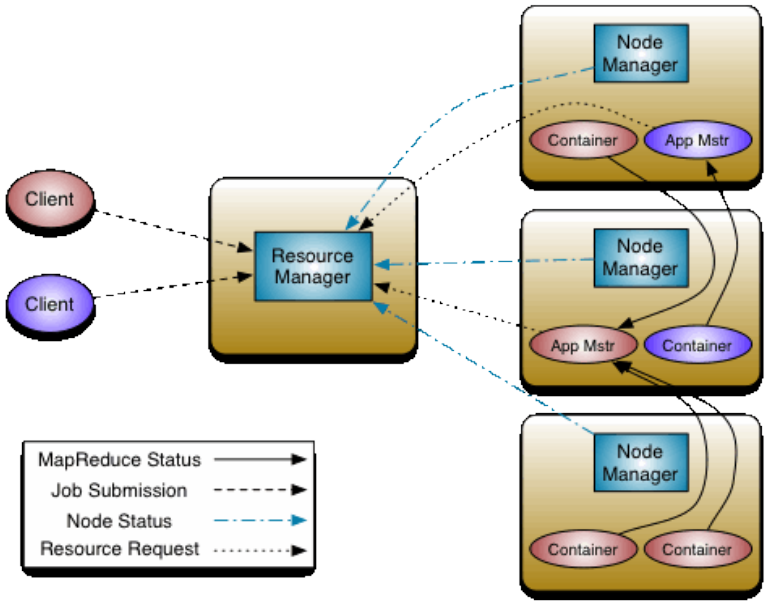

ILUSTRACIÓN 1:GESTIÓN DE PROCESOS MAPREDUCE. FUENTE: CURSO HADOOP LECCIÓN 44.

En la versión 1.0 de *Hadoop*, se usaba el *framework MapReduce* como sistema de procesado de datos y de gestión de clúster, lo que limitaba la capacidad para usar *Hadoop* porque solo podía resolver un tipo de problemas, además de reducir el rendimiento del clúster. Por eso en la versión 2.0 de Hadoop se pasó a usar el *framework YARN* como su componente central, ya que divide la gestión del clúster y el procesado de datos. Actualmente se utiliza *MapReduce* como un componente de procesado de datos de *YARN*.

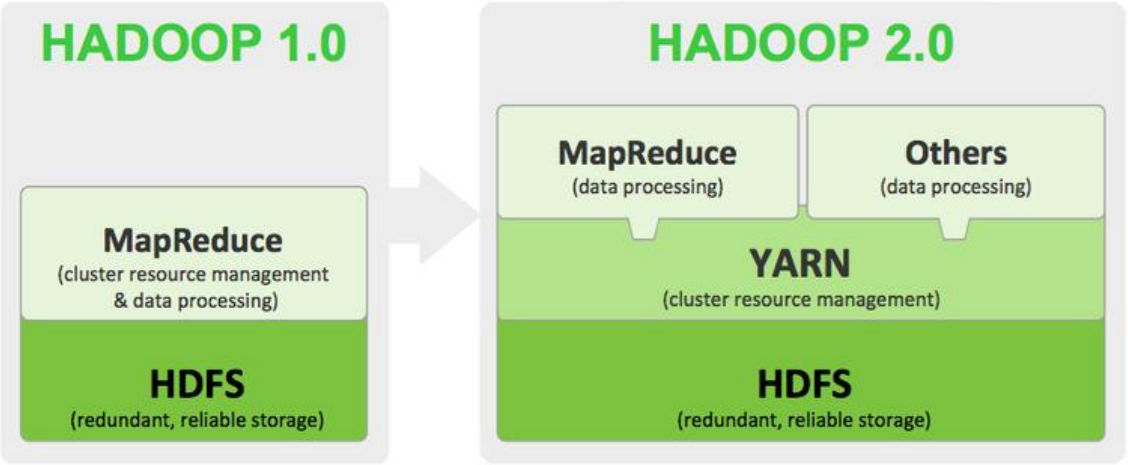

ILUSTRACIÓN 2:CAMBIOS ENTRE HADOOP 1.0 Y HADOOP 2.0. FUENTE: CURSO HADOOP LECCIÓN 44.

2.3.4. Apache Spark[10]

Apache Spark es un entorno de procesamiento distribuido y paralelo que trabaja en memoria volátil. Permite el análisis de grandes conjuntos de datos. Integra diferentes entornos como Bases de Datos NoSQL, *Real Time*, *Machine Learning*, análisis de grafos, etc. *Spark* permite gran escalabilidad, facilidad de uso y velocidad a la hora de resolver problemas de aprendizaje automático. *Spark* es hasta 100 veces más rápido que *MapReduce* ejecutado en memoria o hasta 10 veces más rápido que *MapReduce* ejecutándose en disco.

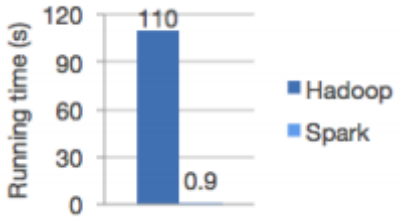

ILUSTRACIÓN 3: COMPARATIVA ENTRE HADOOP Y SPARK EN EL TIEMPO DE EJECUCIÓN DE UN PROCESO. FUENTE: SPARK

Al contrario que MapReduce que trabaja con procesos de tipo *batch*, Spark está orientado al trabajo *in-memory*. *Spark* es compatible con *Hadoop*, lo que significa que se puede ejecutar sobre HDFS, en el mismo clúster que *MapReduce*, las aplicaciones *Spark* se pueden lanzar sobre *YARN* y también es posible la mezcla de aplicaciones *Spark* y *MapReduce* para distintos tipos de resolución de problemas. También soporta múltiples fuentes de datos como: HIVE, JSON, CSV, Parquet y muchas más.

Spark consiste en un *Core* y en un conjunto de librerías escritas en Scala, aunque se pueden escribir aplicaciones en otros lenguajes de programación. Dispone de un *Shell* interactivo.

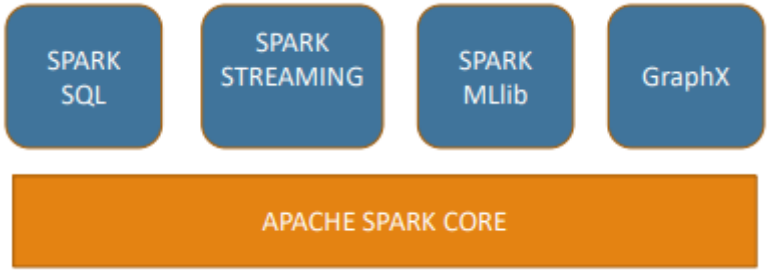

ILUSTRACIÓN 4: SPARK CORE Y SUS LIBRERÍAS. CURSO HADOOP

Spark Core es el motor base para el procesamiento de los datos. Aunque está construido en Scala, existen APIs para Python, Java y R. Este se encarga de la gestión de la memoria, recuperación ante fallos, planificación y distribución de trabajos en el clúster, monitorización del trabajo y de acceder a los sistemas de almacenamiento. *Spark Core* usa una estructura de datos especial denominada RDD. Los RDDs son colecciones de registros inmutables y particionados que pueden ser manejados en paralelo. Pueden contener cualquier clase de objetos Python, Scala y Java. Los RDDs se crean habitualmente transformándolos de otros RDD o cargándolos de una fuente de datos externa, como por ejemplo HDFS.

La aplicación *Spark* es un conjunto de procesos que se van a lanzar dentro de un clúster y que son coordinador por el proceso *SparkContext*. Este componente se conecta al gestor del clúster, sea cual sea, una vez está conectado al clúster, se va a reservar un espacio en cada uno de los nodos para un componente llamado *Executor*. Este proceso es el cual ejecutará los programas enviados al clúster, esto se hace mediante la creación de procesos *Task* y una cache. Lo que al ser ejecutado *In-Memory* le proporciona la velocidad a Spark. Cada nodo esclavo, tiene un conjunto de *Executors* y cada uno de ellos contiene sus *Task* que realizarán los procesos lanzados. Por último, a la hora de usar los datos, *Spark* puede acceder a sus propios datasets o al sistema de ficheros HDFS.

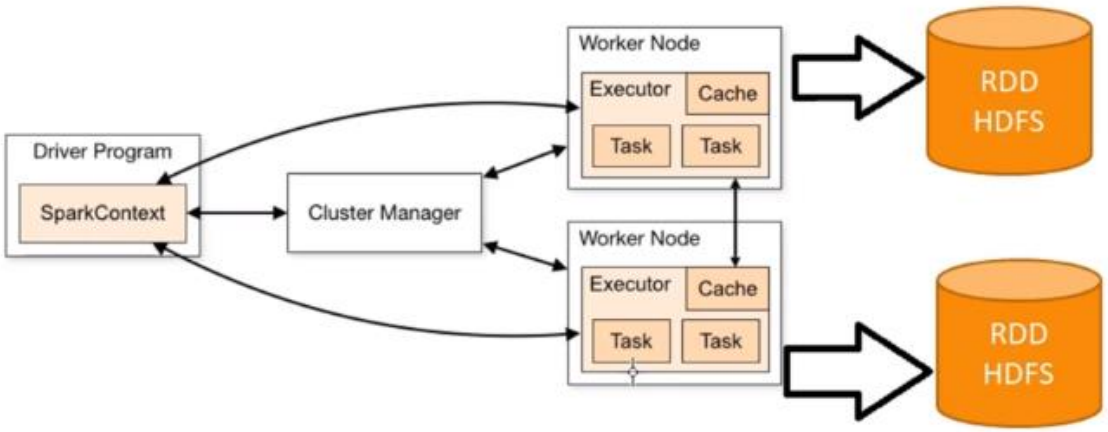

ILUSTRACIÓN 5:FLUJO DE EJECUCIÓN DE UNA APLICACIÓN SPARK. FUENTE: SPARK

A lo largo de este trabajo usaremos 2 librerías de las 4 que forman Spark, estas son *Spark SQL* y *MLlib*, por lo cual será en estas en las que profundizaremos.

2.3.5.SparkSQL

Spark SQL es una librería de *Spark* que introdujo un tipo de abstracción de datos llamado *DataFrame*, lo que permitió el soporte para datos semi-estructurados. *Spark SQL* da soporte para hacer consultas SQL, así como un lenguaje específico para manipular *DataFrames* en Scala, Java o Python.

2.3.6. MLlib[11]

MLlib es una librería que se usa para *Machine Learning* en *Spark*. Dispone de una gran variedad de algoritmos *Machine Learning* que han sido implementados directamente en esta librería. También se dispone de algoritmos para la generación de datos aleatoria, contraste de hipótesis, clasificación y regresión, filtrado colaborativo, clusterización, extracción de características y algoritmos de transformación, así como algoritmos de optimización.

En la librería *MLlib* se trabaja con el concepto de ML Pipelines. Las ML Pipelines proporcionan APIs de alto nivel que ayuda en la creación y optimización de algoritmos de *Machine Learning.* Esta API se basa en varios conceptos inspirados por el proyecto *scikit-learn*.

DataFrame

La API de *MLlib* usa el *DataFrame* de *Spark SQL* como un dataset de *Machine Learning*, gracias a esto podemos almacenar una gran variedad de tipos de datos, como pueden ser texto, valores numéricos, vectores de característica, predicciones, etc.

Transformer

Un *Transformer* es un algoritmo capaz de convertir un *DataFrame* en otro *DataFrame*. Técnicamente los *Transformers* disponen de un método llamado *transform*, que convierte el *DataFrame* en otro, generalmente añadiendo una o más columnas. Por ejemplo, un modelo de aprendizaje automático es un transformador que convierte un dataset de características en otro que contiene predicciones.

Estimator

Un *Estimator* es un algoritmo que ajusta un *Dataframe* para producir un *Transformer*. Implementa un método *fit*, el cual acepta un *DataFrame* y produce un modelo, que es un *Transformer*. Por ejemplo, un algoritmo de aprendizaje como una regresión lineal es un *Estimator* llamando a su función *fit*, la cual entrena un *LinealRegressionModel*.

Parameters

Los *Estimator* y *Transformers* de *MLlib* utilizan una API uniforme para la especificación de *Parameters*. Esto se puede hacer de 2 formas distintas. La primera consiste en la definición directa de los parámetros en la instancia de cada *Estimator* o *Transformer*. Y la segunda forma en pasar un *ParamMap*, el cual consiste en un set de (parámetro, valor), al método *fit* o *transform*.

Pipeline

Un *Pipeline* es un objeto que encadena múltiples *Transformer* y *Estimators* para especificar un flujo de aprendizaje automático. Un ejemplo sería el flujo de trabajo necesario para procesar un documento de texto simple:

1. Separa el documento en palabras.
2. Convierte cada una de las palabras del documento en un vector de características numéricas.
3. Aprende un modelo de predicción usando los vectores de características y sus etiquetas.

Un *Pipeline* está especificado como una secuencia de etapas, en la cual cada una de ellas es un *Transformer* o *Estimator*. Estas etapas son ejecutadas en orden y los *DataFrames* de entrada son transformados y pasan a la siguiente etapa del *Pipeline*. Por cada estado transformador, se llama al método *transform* del *DataFrame*. En el caso de las etapas de estimación, se llama al método *fit* para producir un *Transformer* (que se convierte en un *PipelineModel*).

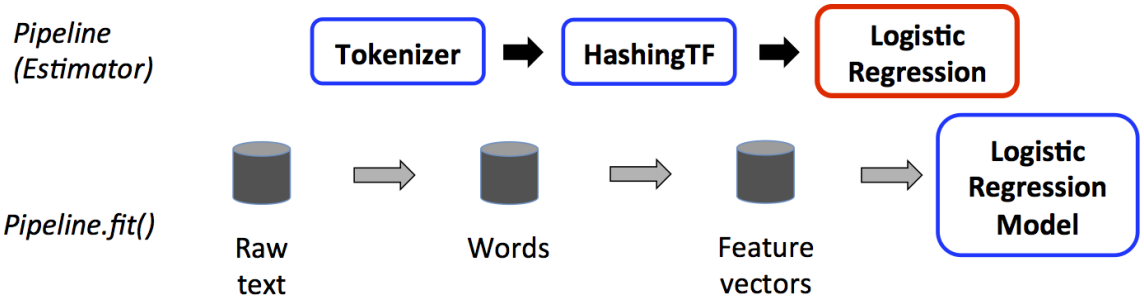

ILUSTRACIÓN 6: EJEMPLO DE FLUJO DE EJECUCIÓN DE UNA PIPELINE EN SPARK. FUENTE:SPARK

Para conseguir un mayor grado de precisión en los algoritmos de aprendizaje automático y *Pipelines*, podemos utilizar una de las herramientas que disponemos en *MLlib*, que consiste en el ajuste de *hyper-parameters*. Este ajuste se puede hacer mediante algoritmos de validación cruzada como *Cross-Validation* y *Train-Validation Split*.

El método de *Cross-Validation* se basa en la separación del dataset en un conjunto de *folds* que son usados como datasets de entrenamiento y validación separados. Por ejemplo, con k=3 *folds*, el algoritmo de validación cruzada creará 3 parejas de datasets de entrenamiento y validación, cada una de las cuales usará dos tercios de los datos para entrenamiento y un tercio para validación. Para evaluar un *ParamMap* particular, *CrossValidator* haya la media de la métrica de validación para los 3 modelos producidos al entrenar el estimador en los 3 *folds* de parejas de datasets. Después de la identificación del *ParamMap*, *CrossValidator* re-entrena el *Estimator*, usando el mejor *ParamMap* y el dataset en su totalidad. La principal diferencia entre los algoritmos de *Cross-Validation* y el *Train-Validation Split* es que en el caso del *Train-Validation Split*, el número de pliegues es k=1. Lo que ahorra tiempo de ejecución y recursos, en caso de que esto sea necesario.

2.3.7. Apache Zeppelin[12]

Apache Zeppelin es una aplicación web de tipo notebook que permite realizar análisis de datos de forma interactiva y producción colaborativa de documentos mediante el uso de varias tecnologías entre las cuales encontramos *Spark*.

Apache Zeppelin permite la creación y ejecución de programas en el clúster *Spark* de forma sencilla. La forma con la cual se comunica con *Spark* es mediante el uso de intérpretes. Los intérpretes son *plugins* que comunican los motores de procesamiento, como puede ser *Spark*, con las fuentes de datos de *Zeppelin UI*.

Se ha decidido utilizar esta aplicación para el preprocesado de los datos con los que se trabajará y también en la creación del modelo de Regresión Lineal. Esto se debe a que Zeppelin se integra de forma muy sencilla a Spark y proporciona muchas facilidades a la hora de crear código, como es la posibilidad de dividir el código en distintas partes que se pueden ejecutar por separado ayudando así a la hora de depurar errores.

2.3.8. Scala[13]

Scala es un lenguaje de programación que integra características de los lenguajes funcionales y de los orientados a objetos. Su implementación actual usa la máquina virtual de Java, lo que lo hace compatible con las aplicaciones Java.

Este lenguaje de programación se ha usado para el preprocesado del dataset, la creación del modelo de regresión lineal y para un paquete que se ejecuta sobre el cluster que permite predecir la velocidad media en cada uno de los puntos de medida en 15 minutos posteriores a la fecha pasada por parámetros.

2.3.9.SBT[14]

SBT es una herramienta Open-source que se usa para empaquetar aplicaciones Scala. Proporciona una consola mediante la cual se pueden importar librerías o empequetar el software

2.3.10. Node.js[15]

Node.js es un entorno multiplataforma y de código abierto que ejecuta código JavaScript de forma nativa. *Node* está diseñado para crear aplicaciones escalables. Utiliza *Node Package Manage* o npm como gestor de paquetes, lo que nos permite instalar y administrar librerías fácilmente. Los paquetes que se han utilizado en la creación de la aplicación web son:

- Express.js: *Framework* diseñado para la creación de aplicaciones web y APIs.
- Body-parser: *Middleware* que analiza
- Jsonfile: Módulo usado para leer y escribir ficheros json.
- Nodemon: Monitor usado para el desarrollo de aplicaciones.
- Shelljs: Permite la ejecución de comandos *Shell* desde el servidor *node*.
- Ejs: Permite incrustar código *javascript* en archivos *html*.

2.3.11. GitHub[16]

Git es una herramienta de código abierto que utilizada para el control de versiones. Se ha creado un repositorio en el cual se encuentra el código generado para la aplicación web desarrollada en este trabajo.

2.3.12. R[17]

R es un lenguaje de programación enfocado al análisis estadístico. Aunque se trata de uno de los lenguajes preferidos para la investigación científica, minería de datos, investigación biomédica y matemáticas financieras. Se ha utilizado este lenguaje para la realización de un análisis *Speed-Up* del clúster donde se comparan los tiempos de ejecución usando distintas configuraciones de este.

2.4. Trabajo Realizado

2.4.1. Creación del clúster en modo *Standalone*.

En el momento del diseño del clúster se eligió la distribución de Ubuntu 17.10 Server ya que era la más actual del momento. Pero existe un problema, que es que ha terminado su vida útil según la comunidad de Ubuntu. Lo que significa que no es posible instalar nuevos paquetes. Así que se utilizará la versión 18.04.1 LTS Server. Por eso se creará una máquina virtual con las siguientes características:

- 2 CPU
- 6 GB de RAM
- 40 GB de Disco Duro
- 1 interfaz de red en modo NAT
- Ubuntu 18.04.1 Server como sistema operativo.

Procederemos a una instalación típica, seleccionando el idioma español, y como nombre de la máquina usaremos el nombre "nodo1", para diferenciarla de las otras que crearemos, y el resto de las opciones por defecto. Al terminar de hacer la instalación de Ubuntu, tendremos un sistema con interfaz CLI y cuyo *hostname* es "nodo1"

2.4.1.1. Configuración del entorno

Primero nos descargaremos el JDK de Java de su página oficial. Para ello usaremos el siguiente código, el cual nos instalará el JDK en la máquina virtual.

```
sudo apt install default-jdk

java -version
```

Después de haber instalado JAVA, procederemos a la configuración del entorno creando una cuenta de usuario nueva con la cual realizaremos la configuración y gestión del clúster *Hadoop*. Esta cuenta se llamará "hadoop".

Modificamos el fichero **".bashrc"**, que lo encontramos en el directorio del usuario **"/home/hadoop"**, incorporando el acceso a los ficheros de JAVA. Para esto añadimos esta línea al final del fichero.

```
export JAVA_HOME=/usr/lib/jvm/default-java
```

Por último, para terminar la configuración del entorno, configuraremos el servicio de *ssh* ya que Hadoop hace las conexiones a los nodos, incluso a sí mismo, mediante *ssh*. Para ello usaremos el comando **ssh-keygen** para la creación de un par de claves asimétricas

y finalmente copiaremos la clave pública al fichero **"/home/hadoop/.ssh/authorized_keys"**.

2.4.1.2. Descarga e Instalación de *Hadoop*

Accedemos como usuario "root". Nos descargamos la versión 2.9.2 de *Hadoop* desde la página principal usando otro ordenador y lo transferimos a la máquina virtual usando un programa de transferencia de ficheros. Movemos el fichero, lo desempaquetamos en **"/opt"** y cambiamos el propietario y grupo al que pertenece.

```
su -
mv hadoop-2.9.2-src.tar.gz /opt
cd /opt
tar xvf hadoop-2.9.2-src.tar.gz
mv hadoop-2.9.2-src hadoop
chown hadoop:hadoop hadoop
```

Despues de ejecutar los comandos mostrados anteriormente, vamos a proceder a incorporar los ficheros al *PATH* del sistema operativo. Para ello, volveremos a editar el fichero "**.bashrc**" que encontraremos en el directorio "**/home/hadoop**". A este fichero añadiremos las líneas mostradas a continuación:

```
export HADOOP_HOME=/opt/hadoop
export PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
. ./.bashrc
```

2.4.1.3. Comprobación del funcionamiento de *Hadoop*

Para la comprobación del funcionamiento de *Hadoop*, ejecutaremos un paquete de ejemplos en modo *Standalone*. Para esto ejecutaremos los siguientes comandos:

```
cd /opt/hadoop/share/hadoop/mapreduce/
mkdir /tmp/entrada
cp /opt/hadoop/etc/hadoop/*.xml /tmp/entrada
hadoop jar hadoop-mapreduce-examples-2.9.2.jar grep /tmp/entrada /tmp/salida
'kms[a-z.]+'
```

Con estos comandos, estamos ejecutando el programa **grep** contenido en el fichero **"hadoop-mapreduce-examples-2.9.2.jar"** sobre los ficheros que se encuentran en

“/tmp/entrada”, creando un archivo que contiene los resultados en directorio **“/tmp/salida”.**

Si hemos seguido los pasos correctamente, nos encontraremos varios ficheros. Los resultados los encontramos en el fichero llamado **“part-r-00000”.**

2.4.1.4. Configuración de HDFS

Como se ha mencionado anteriormente, HDFS es el sistema de ficheros que usa *Hadoop*, ahora pasaremos a la configuración de los ficheros más importantes para su funcionamiento. Los archivos por modificar son: **“core-site.xml”** y **“hdfs-site.xml”**.

Primero modificaremos el archivo “**core-site.xml**” añadiendo las siguientes líneas:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://nodo1:9000</value>
    </property>
</configuration>
```

Lo que nos permite especificar usar comandos **dfs** sin tener que usar la ruta completa como parámetro. Por ejemplo, nos permite usar el comando **“hdfs dfs -ls /”** en vez de **“hdfs dfs -ls hdfs:///”**. También se modificará el fichero “**hdfs-site.xml**” al cuál se le añadirán las líneas siguientes:

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/datos/namenode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/datos/datanode</value>
    </property>
</configuration>
```

En este caso, estamos modificando 3 propiedades del clúster: *replication*, *namenode.name.dir* y *datanode.data.dir*. La primera propiedad indica el número de veces que se replicará cada uno de los bloques que forman los datos. Como estamos configurando el clúster en modo *Standalone*, tiene un valor de 1. Mientras que *namenode.name.dir* y *datanode.data.dir* indican los directorios que contienen la información del nodo maestro y de los nodos esclavos respectivamente.

Despues de haber realizado esta configuración, tenemos que crear los directorios **“/datos”**, **“/datos/namenode”** y **“/datos/datanode”,** y usando el comando **chown**

cambiar su propietario y grupo propietario al mismo del usuario “hadoop”. Por último, para terminar la preparación del sistema de ficheros HDFS, utilizaremos el siguiente comando:

```
hadoop namenode -format
```

Una vez hecho esto, ya podemos iniciar el servicio de HDFS y comprobar su funcionamiento con un par de comandos:

```
start-dfs.sh

echo “esto es un documento de prueba” > prueba.txt

hdfs dfs -put prueba.txt /
```

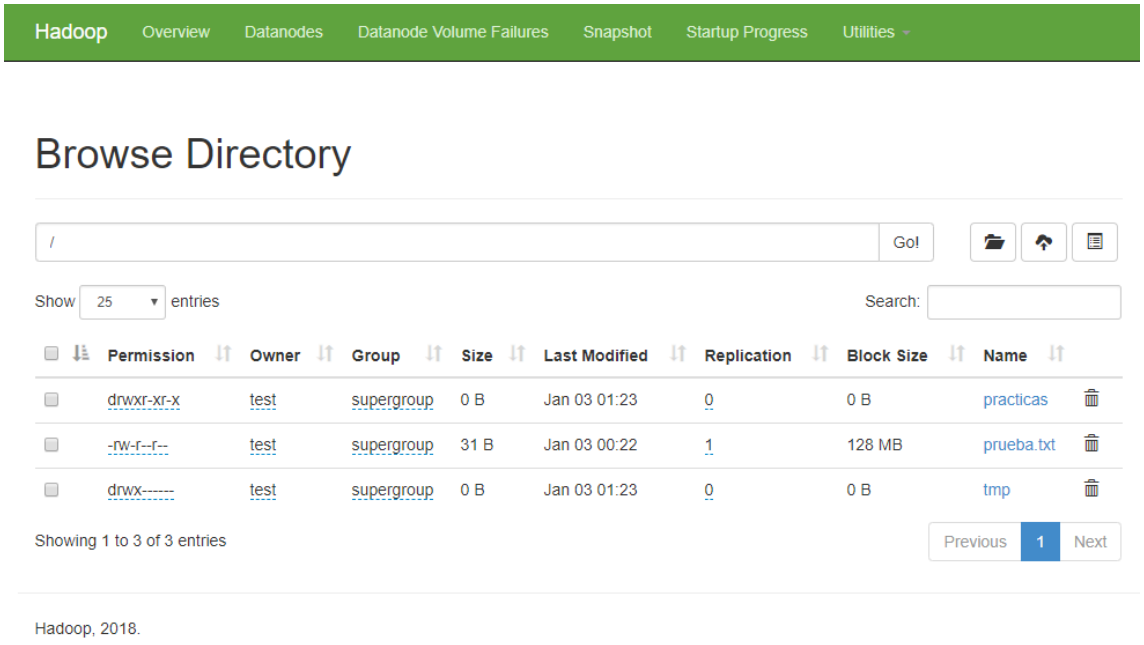

ILUSTRACIÓN 7: COMPROBACIÓN DEL FUNCIONAMIENTO DE HDFS. FUENTE: ELABORACIÓN PROPIA

Cuando terminemos de usar HDFS, lo paramos con el siguiente comando:

```
stop-dfs.sh
```

2.4.1.5. Configuración de YARN

Para la configuración de YARN nos basta con modificar 2 ficheros: **“mapred-site.xml”** y **“yarn-site.xml”**. Aunque **“mapred-site.xml”** no existe, lo podemos crear fácilmente copiando **“mapred-sit.xml.template”** y añadiendo las siguientes líneas:

```
<configuration>
       <property>
           <name>mapreduce.framework.name</name>
           <value>yarn</value>
       </property>
</configuration>
```

Lo que nos permite usar YARN para la gestión y ejecución de procesos. También tenemos que modificar el fichero **"yarn-site.xml"** añadiendo:

```
<configuration>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>localhost</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>
</configuration>
```

Con estos dos ficheros modificados ya estamos preparados para usar los comandos de inicio y parada de YARN:

```
start-yarn.sh

stop-yarn.sh
```

Para comprobar que se ha configurado correctamente los archivos, realizaremos un ejercicio sencillo que se basa en un algoritmo para contar palabras de un fichero de texto que contiene un extracto del Quijote. Este algoritmo lo encontramos en el fichero **"hadoop-mapreduce-examples-2.9.2.jar"**. Para realizar la prueba necesitaremos ejecutar estos comandos:

```
hdfs dfs -mkdir /practicas

hdfs dfs -put /home/hadoop/Downloads/quijote.txt /practicas

hadoop jar /opt/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-
2.9.2.jar wordcount /practicas/quijote.txt /practicas/resultado

hdfs dfs -get /practicas/resultado/part-r-00000 /tmp/palabras_quijote.txt
```

Despues de terminar la ejecución de estos comandos, encontraremos un archivo llamado **“palabras_quijote.txt”** en la carpeta **“/tmp”**, este archivo contiene:

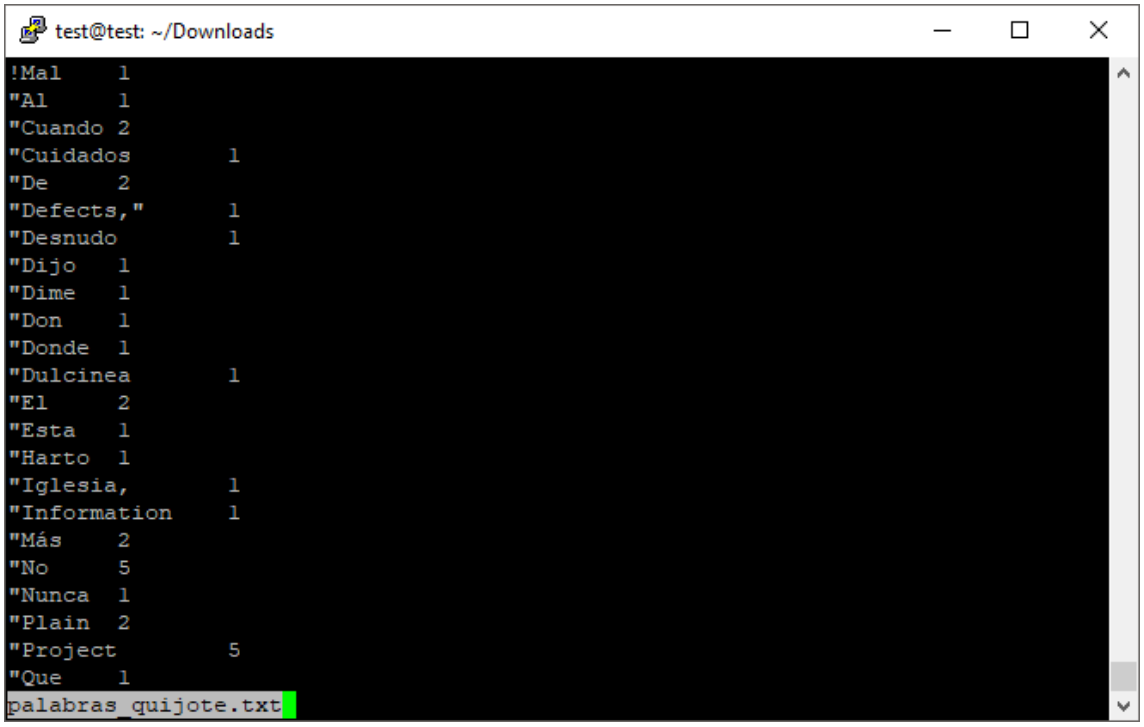

ILUSTRACIÓN 8: PALABRAS_QUIJOTE.TXT. FUENTE: ELABORACIÓN PROPIA

2.4.1.6. Instalación de *Spark*

Spark es un módulo que se puede usar de varias formas, en este trabajo se utilizará sobre el clúster *Hadoop* para así aprovechar los beneficios de la computación en paralelo y del sistema de ficheros HDFS. Primero nos descargaremos *Spark* de la página oficial donde se elegirá la versión 2.3.0 y el tipo de paquete como *Pre-build with user-provided Apache Hadoop*. Se da el caso de que hay versiones de Spark que están pre-empaquetadas con una versión de *Hadoop*, estas se usan en caso de que no se haya configurado previamente un entorno *Hadoop*. También existen las versiones que no contienen un entorno *Hadoop*, las cuales se usan en caso de usar otro tipo de tecnología clúster. Después de haber descargado la versión indicada, se procederá a transferir el archivo a la máquina virtual y a su posterior instalación.

Para instalar *Spark* en un *cluster Hadoop*, hay que desempaquetar el fichero siguiendo los pasos que se indican a continuación:

```
cp spark-2.3.0-bin-without-hadoop.tgz /opt/hadoop

tar xvf spark-2.3.0-bin-without-hadoop.tgz

mv spark-2.3.0-bin-without-hadoop spark
```

Al haber terminado la instalación de *Spark*, se pasará a la configuración del entorno. Para esto tendremos que modificar el fichero **"/home/hadoop/.bashrc"**, añadiendo:

```
export
PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:/opt/hadoop/spark
/bin:/opt/hadoop/spark/sbin

export SPARK_DIST_CLASSPATH=$(Hadoop classpath)
```

Para arrancar y parar los servicios de *Spark* se usan los comandos:

```
start-all.sh


stop-all.sh
```

Para comprobar el funcionamiento de *Spark*, se usará el fichero de **"/opt/hadoop/spark/bin/spark-shell"**, en el cual se ejecutarán comandos para realizar un ejercicio de contar palabras de un archivo de texto.

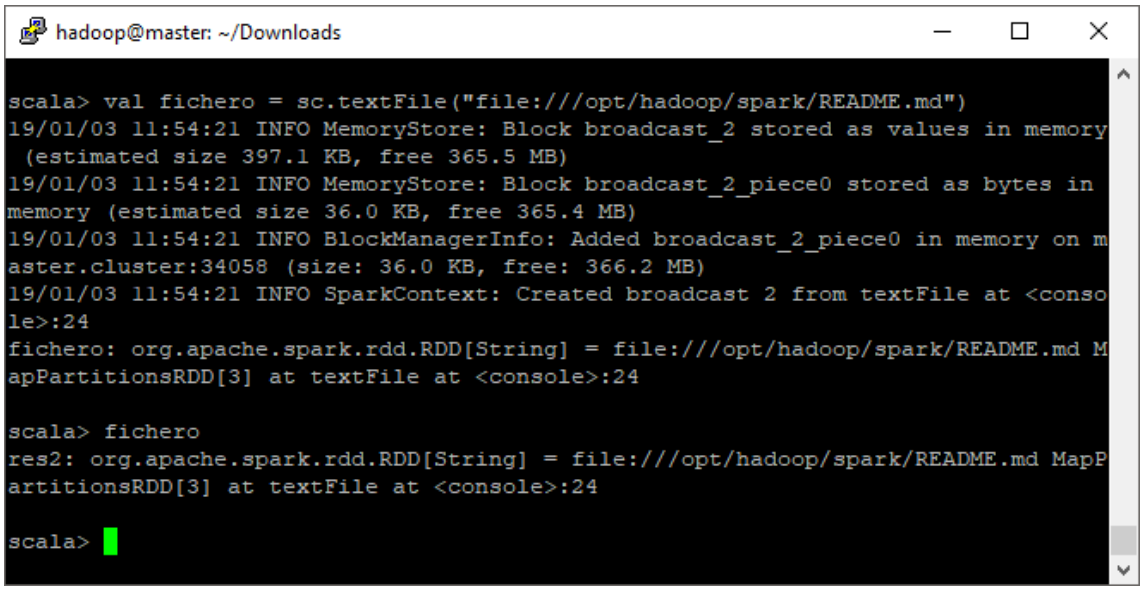

ILUSTRACIÓN 9: CARGA EN MEMORIA DE UN FICHERO. FUENTE: ELABORACIÓN PROPIA

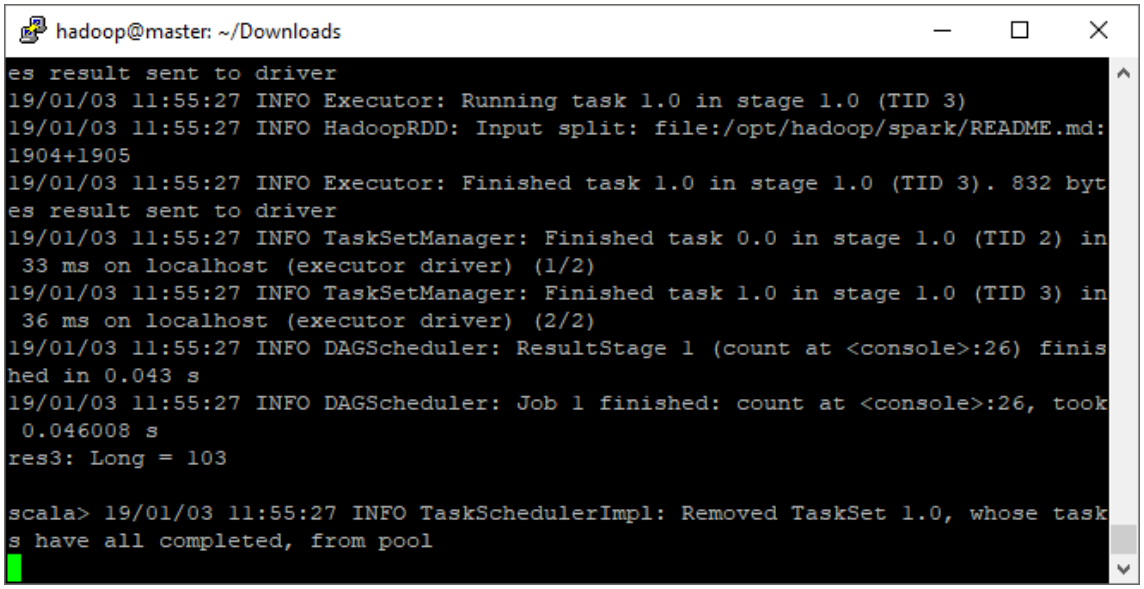

ILUSTRACIÓN 10: USO DEL MÉTODO COUNT PARA CONTAR EL NÚMERO DE PALABRAS. FUENTE: ELABORACIÓN PROPIA

2.4.1.7. Instalación de *Apache Zeppelin*

Para la instalación de *Apache Zeppelin* tenemos que descargarnos el paquete de la página oficial, transferirlo a la máquina virtual y una vez realizados esos pasos, desempaquetamos el archivo y lo movemos al directorio **"/opt"**, usando los siguientes comandos:

```
cd /home/hadoop/Downloads
tar xvf zeppelin-0.8.0-bin-all.tgz
mv zeppelin-0.8.0-bin-all zeppelin
./zeppelin/bin/install-interpreter.sh -all
mv zeppelin /opt/
```

Después de ejecutar estos comandos, creamos el fichero **"zeppelin-site.xml"** a partir de **"zeppelin-site.xml.template"**, modificando la propiedad *zeppelin.server.port* al valor 8000 para que así no haya colisión con los puertos que usa YARN.

Zeppelin contiene una versión de *Spark* que usa de manera predeterminada. Para que pueda trabajar sobre los datos del clúster, tenemos que modificar estos archivos modificar el fichero "**/opt/zeppelin/conf/zeppelin-env.sh**" y añadir la línea

```
export SPARK_HOME=/opt/hadoop/spark
```

Definiendo la variable *SPARK_HOME*, el intérprete de *Spark* usará los archivos de nuestra instalación, en vez de la que se encuentra en *Zeppelin*.

Para iniciar *Zeppelin* de forma más sencilla, crearemos un alias en el fichero "**~/.bashrc**" lo cual nos permitirá arrancar y parar el servicio de *Zeppelin*. Para ello, añadiremos la línea:

```
alias zeppelin='/opt/zeppelin/bin/zeppelin-daemon.sh'
```

Lo que nos permitirá usar los comandos:

```
zeppelin start

zeppelin stop
```

Para arrancar y parar el servicio respectivamente.

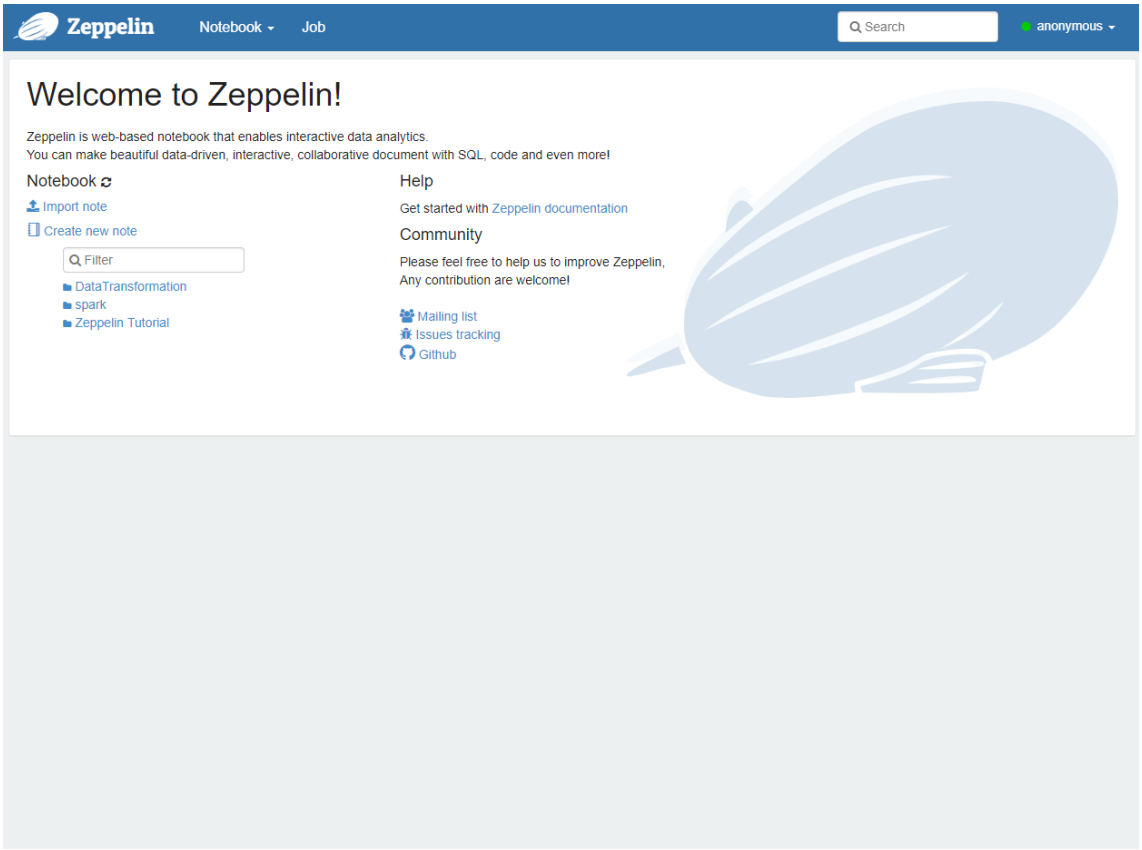

ILUSTRACIÓN 11. INTERFAZ WEB DE APACHE ZEPPELIN. FUENTE: ELABORACIÓN PROPIA

2.4.1.8. Instalación de NodeJS

En este paso se realizará la instalación del servidor *Node.js* usando el gestor de paquetes *npm*. Para esto tenemos que ejecutar los siguientes comandos en una consola con privilegios “root”.

```
sudo apt update

sudo apt install nodejs npm
```

Una vez terminada la instalación, comprobaremos que se ha instalado correctamente usando los comandos:

```
node --version

npm --version
```

Y tendría que salir como resultado:

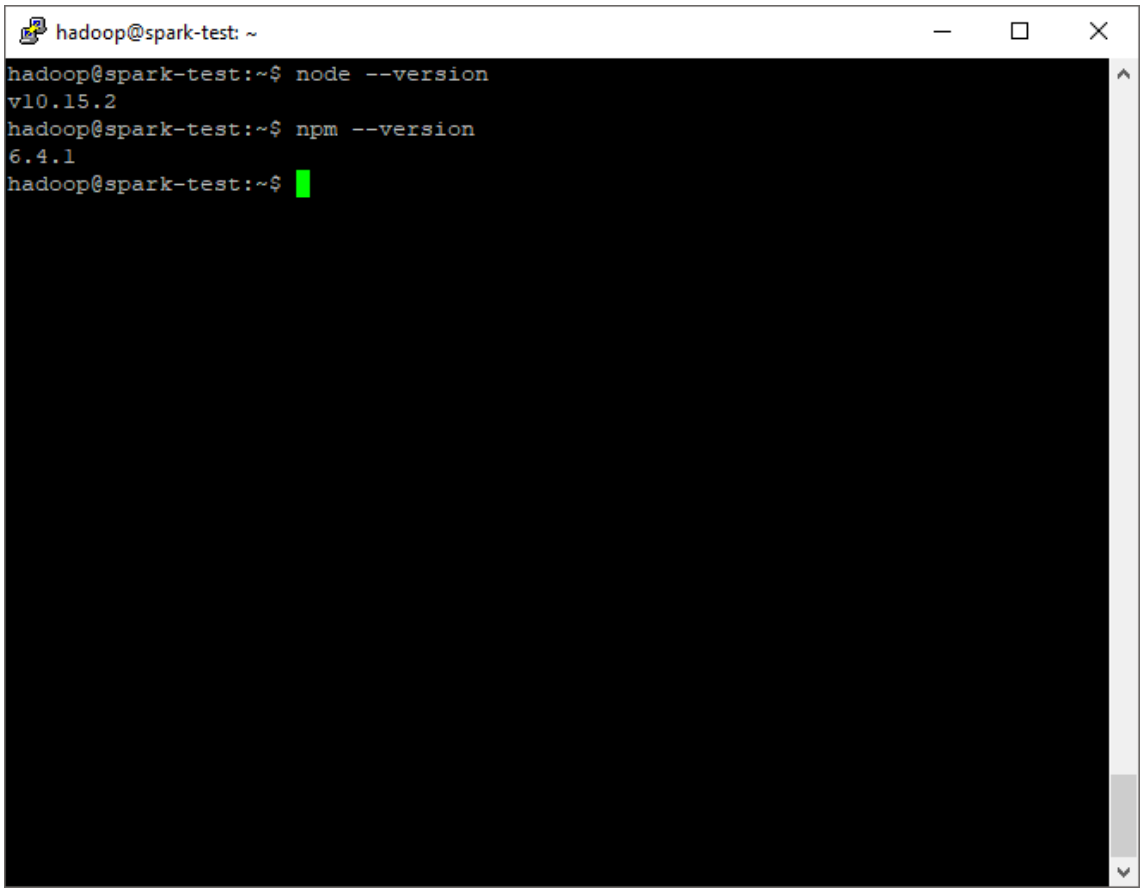

ILUSTRACIÓN 12. COPROBACIÓN DE LA INSTALACION DE NODEJS. FUENTE: ELABORACIÓN PROPIA

2.4.2. Configuración del clúster en modo multi-nodo

Para la realización de este proyecto se ha utilizado el sistema de computación MIMD proporcionado por el CICEI. El nodo maestro del clúster de computación será el nodo aton. La multicomputadora está formada por 8 nodos con las siguientes características:

- CPUs: AMD Opteron 246 a 2 GHz x2.
- RAM: 8GB de RAM a 400MHz.
- Sistema Operativo: Ubuntu 17 server.
- Disco Duro: 2 Discos Duros de 1 TB a 7200rpm.
- Interfaz de Red: 2 Interfaces de red de 1Gbit de velocidad.

Estos nodos se comunican mediante una subred privada a la que solo se tiene acceso desde la red interna del CICEI por motivos de seguridad.

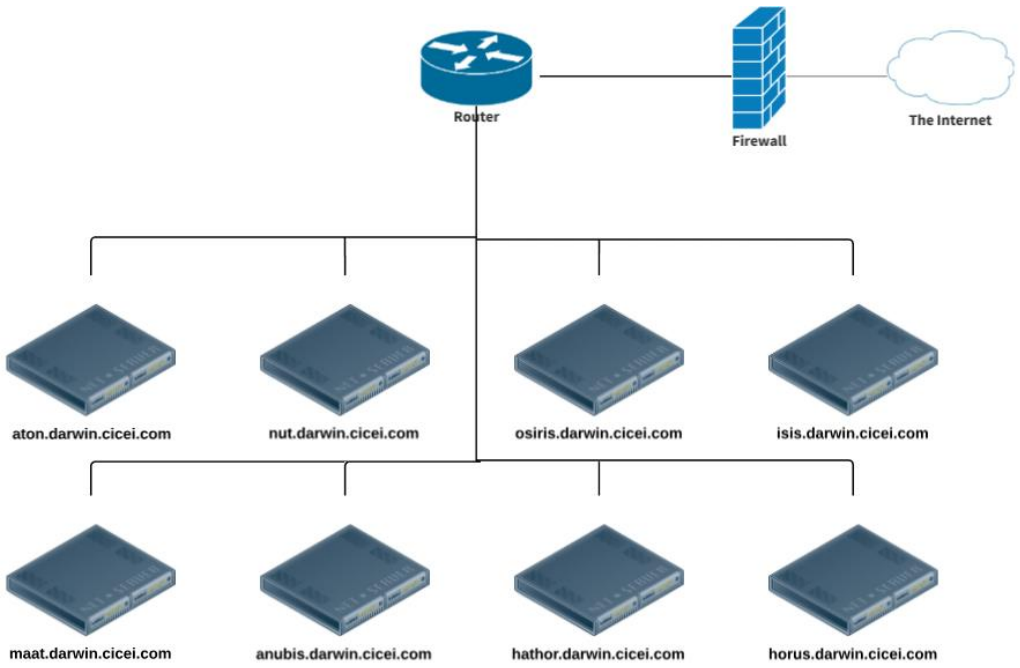

ILUSTRACIÓN 13 DIAGRAMA DEL MIMD PROPORCIONADO POR EL CICEI. FUENTE: ELABORACIÓN PROPIA.

También crearemos un usuario común en cada nodo llamado “hadoop”.

2.4.2.1. Configuración de Red

En cada nodo se debe de configurar los archivos “**/etc/hosts**” y “**/etc/network/interfaces**” como se verá en las siguientes imágenes.

```
127.0.0.1     localhost
# Nombres de los nodos
192.168.1.1   aton.darwin.cicei.com      aton
192.168.1.2   nut.darwin.cicei.com       nut
192.168.1.3   osiris.darwin.cicei.com    osiris
192.168.1.4   isis.darwin.cicei.com      isis
192.168.1.5   maat.darwin.cicei.com      maat
192.168.1.6   anubis.darwin.cicei.com    anubis
192.168.1.7   hathor.darwin.cicei.com    hathor
192.168.1.8   horus.darwin.cicei.com     horus
```

ILUSTRACIÓN 14. ARCHIVO "/ETC/HOSTS" DE CADA NODO.

Mientras que el archivo **"/etc/network/interfaces"** se configura de la siguiente manera:

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet static
       address 192.168.1.x # X es la ip de cada nodo
       netmask 255.255.255.0
       network 192.168.1.0
       broadcast 192.168.1.255

# Interfaz exclusiva del nodo maestro
auto eth1
iface eth1 inet dhcp
```

ILUSTRACIÓN 15. ARCHIVO "/ETC/NETWORK/INTERFACES" DE CADA NODO.

El último paso antes de proceder a la instalación de los servicios con los que se trabajará en el clúster consiste en la creación de pares de claves pública y privada mediante la herramienta **ssh-keygen**. Para esto necesitamos ejecutar los siguientes comandos en todos los nodos:

```
ssh-keygen
sudo systemctl enable sshd
sudo service sshd start
ssh-copy-id -i ~/.ssh/id_rsa.pub hadoop@aton
ssh-copy-id -i ~/.ssh/id_rsa.pub hadoop@nut
ssh-copy-id -i ~/.ssh/id_rsa.pub hadoop@osiris
ssh-copy-id -i ~/.ssh/id_rsa.pub hadoop@isis
ssh-copy-id -i ~/.ssh/id_rsa.pub hadoop@maat
ssh-copy-id -i ~/.ssh/id_rsa.pub hadoop@anubis
ssh-copy-id -i ~/.ssh/id_rsa.pub hadoop@hathor
ssh-copy-id -i ~/.ssh/id_rsa.pub hadoop@horus
```

2.4.2.2. Instalación de servicios y modificación de ficheros de configuración.

En esta sección se realizará la instalación de los servicios necesarios en cada nodo. En el nodo maestro se instalará *Java*, *Hadoop*, *Spark*, *Zeppelin* y *Node.js*. Mientras que en los nodos esclavos sólo se instalará *Java*, *Hadoop* y *Spark*.

Para la instalación de java necesitaremos usar los siguientes comandos:

```
sudo apt update
sudo apt install oracle-java8-installer
sudo apt install oracle-java8-set-default
```

La instalación de *Hadoop* y *Spark* la haremos de la forma mencionada en el apartado del clúster en modo *Standalone*. Lo que necesitaremos otro PC en el cual nos descargaremos los archivos de las páginas oficiales y los transferiremos mediante un programa *sftp* como puede ser *FileZilla*. Una vez hecho esto, seguiremos los pasos de instalación mencionados en el apartado de *Standalone*. Y por último instalaremos *Zeppelin* y *Node.js* en el nodo maestro de la misma forma.

Después de haber terminado con la instalación de los componentes software necesarios, pasaremos a la modificación de los ficheros de configuración en los nodos esclavos. Indicando lo siguiente:

```
nut.darwin.cicei.com
osiris.darwin.cicei.com
isis.darwin.cicei.com
maat.darwin.cicei.com
anubis.darwin.cicei.com
hathor.darwin.cicei.com
horus.darwin.cicei.com
```

ILUSTRACIÓN 16. FICHERO "/OPT/HADOOP/ETC/HADOOP/SLAVES". FUENTE: ELABORACIÓN PROPIA

```
<configuration>
	<property>
		<name>dfs.replication</name>
		<value>3</value>
	</property>
	<property>
		<name>dfs.namenode.name.dir</name>
		<value>/datos/namenode</value>
	</property>
	<property>
		<name>dfs.datanode.data.dir</name>
		<value>/datos/datanode</value>
	</property>
</configuration>
```

ILUSTRACIÓN 17. FICHERO "/OPT/HADOOP/ETC/HADOOP/HDFS-SITE.XML". FUENTE: ELABORACIÓN PROPIA

```
<configuration>
	<property>
		<name>yarn.resourcemanager.hostname</name>
		<value>aton.darwin.cicei.com</value>
	</property>
	<property>
		<name>yarn.nodemanager.aux-services</name>
		<value>mapreduce_shuffle</value>
	</property>
	<property>
		<name>yarn.nodemanager.aux-
		services.mapreduce_shuffle.class</name>
		<value>org.apache.hadoop.mapred.ShuffleHandler</value>
	</property>
</configuration>
```

ILUSTRACIÓN 18. FICHERO "/OPT/HADOOP/ETC/HADOOP/YARN-SITE.XML". FUENTE: ELABORACIÓN PROPIA

```
<configuration>
	<property>
		<name>mapreduce.framework.name</name>
		<value>yarn</value>
	</property>
</configuration>
```

ILUSTRACIÓN 19.FICHERO "/OPT/HADOOP/ETC/HADOOP/MAPRED-SITE.XML". FUENTE: ELABORACIÓN PROPIA.

```
nut.darwin.cicei.com
osiris.darwin.cicei.com
isis.darwin.cicei.com
maat.darwin.cicei.com
anubis.darwin.cicei.com
hathor.darwin.cicei.com
horus.darwin.cicei.com
```

ILUSTRACIÓN 20. FICHERO "/OPT/HADOOP/SPARK/CONF/SLAVES". FUENTE: ELABORACIÓN PROPIA

Al terminar de hacer todos estos pasos, se comprobará el funcionamiento del cluster ejecutando cada uno de los servicios (HDFS, YARN y *Spark*) sin encontrar errores mediante el uso de los comandos mencionados anteriormente.

2.4.3. Obtención y tratamiento del *dataset*

El *dataset* que ha sido proporcionado por el CICEI, es un *dataset* público perteneciente a la comunidad de Madrid[18]. Este contiene información sobre el tráfico en los puntos de la M-30, así como carreteras Urbanas. Los históricos de los puntos de medida se almacenan en ficheros CSV que tienen el siguiente formato:

Nombre	Tipo	Descripción
Idelem	Entero	Identificación única del Punto de Medida en los sistemas de control de tráfico.
Fecha	Fecha	Fecha y hora oficiales de Madrid con formato "YYYY-MM-DD HH:MI:SS".
identif	Texto	Identificador del Punto de medida en los Sistemas de Tráfico (Se proporciona por retrocompatibilidad).
Tipo_elem	Texto	Nombre del Tipo de Punto de Medida: Urbano o M30.
Intensidad	Entero	Intensidad del Punto de Medida en el período de 15 minutos (vehículos/hora).
Ocupacion	Entero	Tiempo de Ocupación del punto de Medida en el período de 15 Minutos (%).
Carga	Entero	Carga de los vehículos en el período de 15 Minutos. Tiene en cuenta intensidad, ocupación y capacidad de la vía. Establece el grado de uso de la vía de 0 a 100.
Vmed	Entero	Velocidad media de los vehículos en el período de 15 minutos. Sólo para los puntos de medida de M30.
Error	Texto	Indicación de si ha habido una muestra errónea o sustituida en el período de 15 minutos. N: No ha habido errores ni sustituciones. E: Los parámetros de calidad de las muestras no son óptimos. S:Alguna de las muestras recibidas era totalmente errónea.
Periodo_integracion	Entero	Número de muestras recibidas y consideradas para el período de integración.

Los campos que se han utilizado para la regresión lineal han sido: idelem, fecha y vmed. Este *dataset* se ha modificado para que cada fila de este tenga el siguiente formato:

Nombre	Tipo	Descripción
Idelem	Entero	Identificación única del Punto de Medida en los sistemas de control de tráfico.
Fecha	Fecha	Fecha y hora oficiales de Madrid con formato "YYYY-MM-DD HH:MI:SS".
Vmed-3	Real	Velocidad media de los vehículos en el período de 15 minutos anterior a Vmed-2.
Vmed-2	Real	Velocidad media de los vehículos en el período de 15 minutos anterior a Vmed-1.
Vmed-1	Real	Velocidad media de los vehículos en el período de 15 minutos anterior a Vmed.
Vmed	Real	Velocidad media de los vehículos en el período de 15 minutos. Sólo para los puntos de medida de M30.

Con esto tenemos en cada fila del *dataset* las velocidades medias que usaremos como puntos conocidos, así como el valor que se intenta predecir con la regresión lineal.

Pero nos falta una parte muy importante que es la latitud y la longitud de cada punto de medida para mostrarlo en un mapa. Para esto necesitamos otro *dataset* del portal de Madrid. Este es el *dataset* con la Ubicación de los puntos de medida del tráfico. Los atributos de los puntos de medida son los siguientes:

Nombre	Tipo	Descripción
Cod_cent	Texto	Código de centralización en los sistemas y que se corresponde con el campo códifo de otros conjuntos de datos como el de intensidad del tráfico en tiempo real.
Id	Entero	Identificador único y permanente del punto de medida.
Nombre	Texto	Denominación del punto de medida, utilizándose la siguiente nomenclatura: Para los puntos de medida de tráfico urbano se identifica con la calle y orientación del sentido de la circulación. Para los puntos de vías rápida y accesos a Madrid se identifica con el punto kilométrico, la calzada y si se trata de la vía central, via de servicio o un enlace.
Tipo_elem	Texto	Descriptor de la tipología del punto de medida según la siguiente codificación: • URB (tráfico URBANO) para dispositivos de control semafórico. • M-30 (tráfico INTERURBANO) para dispositivos de vías rápidas y accesos a Madrid.

X	Real	Coordenada X_UTM del centroide de la representación del polígono del punto de medida.
Y	Real	Coordenada Y_UTM del centroide de la representación del polígono del punto de medida.

El mayor problema con el que nos encontramos es que las coordenadas X e Y no están en forma de latitud y longitud, lo que nos dificulta a la hora de representarlas en un mapa.

El sistema de coordenadas UTM[19] (Sistema de Coordenadas Universal Transversal de Mercator) está basado en una proyección de Mercator que, en lugar de hacerla tangente al ecuador, se hace secante a un meridiano.

Se diferencia del sistema de coordenadas expresado en latitud y longitud ya que las magnitudes en el sistema UTM se expresan en metros únicamente al nivel del mar.

Las coordenadas UTM determinan el lugar dentro de un punto de una cuadrícula tomando como origen el vértice inferior izquierdo de cada recuadro. Se indica primero la abscisa (x) y, después la ordenada (y) ambas en metros.

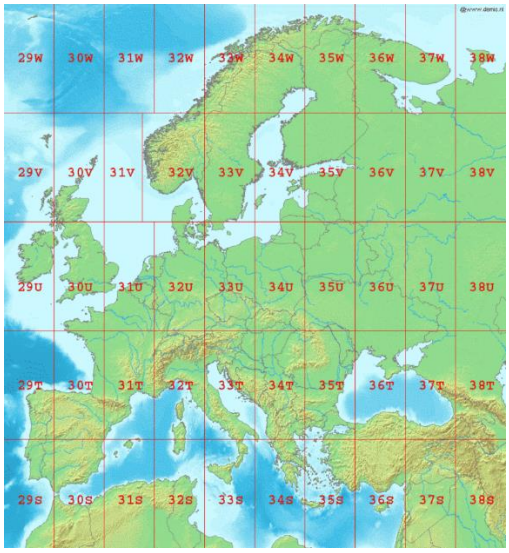

ILUSTRACIÓN 20. ZONAS UTM EUROPA. FUENTE: WIKIPEDIA

Fijándonos en el mapa de zonas UTM de Europa, podemos ver que la zona en la que se encuentra Madrid es la zona 30T. Sabiendo esto, para transformar las coordenadas del sistema de coordenadas UTM al sistema de coordenadas de latitud y longitud, usamos una herramienta web[20] para la conversión de coordenadas.

Una vez hecho esto, nos quedaría el siguiente *dataset*:

Nombre	Tipo	Descripción
Id	Entero	Identificador único y permanente del punto de medida.
Lat	Real	Coordenada latitud del punto de medida.
Lon	Real	Coordenada longitud del punto de medida.

Con estos ficheros que contienen los datos que necesitamos, usando *Apache Zeppelin* crearemos un *dataset* con el formato que nos interesa usando el siguiente código:

```
val febrero =
spark.read.format("csv").option("sep",";").option("header","true").load("hdfs://
/datos.madrid.es/2018/02-2018.csv")

val febreroLimpio = enero.filter($"tipo_elem" === "PUNTOS MEDIDA M-
30").select($"id" as "vid",$"fecha",$"vmed").sort($"fecha".asc)



val pmed_location =
spark.read.format("csv").option("sep",";").option("header","true").load("hdfs://
/pmed_location/pmed_final/*.csv")



val mes = febreroLimpio.join(pmed_location, eneroLimpio("vid") ===
pmed_location("id"))
```

En esta parte del código cargamos los datos de un mes, lo filtramos por el tipo de elemento, especificando que sea un punto de medida de la M-30. También cargamos el fichero con las coordenadas y hacemos un *JOIN* de los dos *dataframes*.

```
val real = mes.withColumn("vmed_double", $"vmed".cast(DoubleType))

val datosReal = real.select($"id",$"fecha",$"Latitude" as "lat",$"Longitude"
as "lng",$"vmed_double" as "vmed")



val window = Window.partitionBy("id").orderBy("fecha")

val vmed1 = datosReal.withColumn("vmed-1", lag($"vmed", 1, null).over(window))

val vmed2 = vmed1.withColumn("vmed-2", lag($"vmed-1", 1, null).over(window))

val vmed3 = vmed2.withColumn("vmed-3", lag($"vmed-2", 1, null).over(window))

val vmed4 = vmed3.withColumn("vmed+1", lead($"vmed", 1, null).over(window))

val dataset = vmed4.select($"id",$"fecha",$"lat",$"lng",$"vmed-3",$"vmed-
2",$"vmed-1",$"vmed", $"vmed+1").na.drop()
```

Después de haber unido los datos de velocidad con los de posición de cada punto de medida, convertimos la columna de velocidad media a formato de numero real y después creo las columnas de vmed-1, vmed-2 y vmed-3 que se van a usar como los datos de entrada del modelo de regresión lineal. Con esto, tenemos el *dataset* listo para que sea aplicado al modelo de regresión y así sacar predicciones sobre esos datos.

```
dataset.show
```

```
+----+-------------------+---------+---------+------+------+------+----+
|  id|              fecha|      lat|      lng|vmed-3|vmed-2|vmed-1|vmed|
+----+-------------------+---------+---------+------+------+------+----+
|6731|2018-01-01 00:45:00|40.442519|-3.659219|  61.0|  43.0|  82.0|80.0|
|6731|2018-01-01 01:00:00|40.442519|-3.659219|  43.0|  82.0|  80.0|75.0|
|6731|2018-01-01 01:15:00|40.442519|-3.659219|  82.0|  80.0|  75.0|72.0|
|6731|2018-01-01 01:30:00|40.442519|-3.659219|  80.0|  75.0|  72.0|69.0|
|6731|2018-01-01 01:45:00|40.442519|-3.659219|  75.0|  72.0|  69.0|68.0|
|6731|2018-01-01 02:00:00|40.442519|-3.659219|  72.0|  69.0|  68.0|71.0|
|6731|2018-01-01 02:15:00|40.442519|-3.659219|  69.0|  68.0|  71.0|73.0|
|6731|2018-01-01 02:30:00|40.442519|-3.659219|  68.0|  71.0|  73.0|75.0|
|6731|2018-01-01 02:45:00|40.442519|-3.659219|  71.0|  73.0|  75.0|75.0|
|6731|2018-01-01 03:00:00|40.442519|-3.659219|  73.0|  75.0|  75.0|79.0|
|6731|2018-01-01 03:15:00|40.442519|-3.659219|  75.0|  75.0|  79.0|76.0|
|6731|2018-01-01 03:30:00|40.442519|-3.659219|  75.0|  79.0|  76.0|82.0|
|6731|2018-01-01 03:45:00|40.442519|-3.659219|  79.0|  76.0|  82.0|79.0|
|6731|2018-01-01 04:00:00|40.442519|-3.659219|  76.0|  82.0|  79.0|80.0|
|6731|2018-01-01 04:15:00|40.442519|-3.659219|  82.0|  79.0|  80.0|82.0|
```

ILUSTRACIÓN 21. DATASET FINAL PARA EL MODELO DE REGRESIÓN LINEAL. FUENTE: ELABORACIÓN PROPIA.

Este dataset se guardará en la carpeta "**/Datasets/2018/**" del sistema de ficheros HDFS del clúster para su posterior uso mediante la línea:

```
dataset.coalesce(1).write.mode("overwrite").format("json").option("multiLine",
"true").save("hdfs:///Datasets/2018/01-2018.json")
```

2.4.4. Creación del Modelo de Regresión Lineal

Partiendo de los *Datasets* creados usando los pasos del apartado anterior, procedemos a crear el modelo de regresión lineal en *Spark*. Empezamos dividiendo el *dataset* de forma aleatoria en dos: uno que usaremos de entrenamiento del modelo y otro para comprobar la veracidad de este.

```
val (entrenamiento, test) = {
    val split = dataset.randomSplit(Array(0.8,0.2))
    (split(0),split(1))
}
```

Una vez dividido, creamos un objeto del tipo *VectorAssembler*, que se va a encargar de que la columna de "*features*", que a su vez es una columna del tipo vector (real, real, real), es la columna que contiene los datos de entrada de nuestro modelo. Al mismo tiempo iniciamos un objeto del tipo *LinearRegression* y le indicamos unos parámetros como son el número máximo de iteraciones y cuál es la columna que tendrá las predicciones del modelo. Una vez creados estos objetos, se crea un pipeline cuyas etapas son el *VectorAssembler* y *LinearRegression*, en ese orden.

```
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.regression.LinearRegression


val ensamblador = new VectorAssembler().setInputCols(Array("vmed-
3","vmed-2","vmed-1")).setOutputCol("features")
val rl = new
LinearRegression().setMaxIter(10).setLabelCol("label").setFeaturesCo
l("features")
val pipeline = new Pipeline().setStages(Array(ensamblador,rl))
```

En el código que se muestra a continuación es un set de parámetros que se van a usar a la hora del afinamiento de hiper-parámetros. En el se encuentran todas las combinaciones posibles de los valores de los parámetros[21] *regParam* y *elasticNetParam* pertenecientes a las regresiones lineales. El parámetro de *elasticNet* combina los errores L1 y L2 de los métodos de *Lasso* y del modelo de regresión de arista respectivamente. En el caso de que el valor de *elasticNet* sea 1 sería equivalente a un modelo *Lasso*, mientras que, si fuese 0, el modelo entrenado equivaldría a un modelo de regresión de arista. Y *regParam* es el parámetro de regularización del modelo. Esto nos permite elegir el modelo más preciso que se puede hacer combinando esos parámetros.

```
import org.apache.spark.ml.tuning.ParamGridBuilder


val paramGrid = new ParamGridBuilder().
    addGrid(rl.regParam, Array(0.1, 0.01)).
    addGrid(rl.elasticNetParam, Array(0.0, 0.5, 1.0)).
    build()
```

Creamos un objeto del tipo *TrainValidationSplit*, lo que nos ofrece el mejor modelo posible entrenando el estimador, en este caso el *pipeline* creado anteriormente, el set de parámetros *paramGrid*. Esto lo hace dividiendo el *dataset* de entrenamiento en 2 partes, indicado por el *TrainRatio*, en este caso del 0.8, lo que indica que va a usar un 80% del *dataset* para entrenar y el otro 20% para probar los resultados. Una vez hallado el mejor *paramMap*, ajusta el estimador, devolviendo el modelo de regresión lineal.

```
import
org.apache.spark.ml.tuning.TrainValidationSplit
import
org.apache.spark.ml.evaluation.RegressionEvaluator


val tvs = new TrainValidationSplit().
    setEstimator(pipeline).
    setEvaluator(new RegressionEvaluator()).
    setEstimatorParamMaps(paramGrid).
    setTrainRatio(0.8).
    setParallelism(2)
```

Empieza el proceso de entrenamiento del modelo.

```
val modelo = tvs.fit(entrenamiento)
```

Cuando se ha terminado de entrenar el modelo, se pasa a la fase de comprobación. En esta fase, se usa el modelo que acabamos de entrenar para transformar el set de prueba que se había creado al principio de este apartado. Una vez hecho esto, obtenemos las métricas que nos permiten saber si el modelo que hemos creado es deficiente o no.

```
import org.apache.spark.mllib.evaluation.RegressionMetrics

val prediccion = modelo.transform(test)

val sumario = prediccion.select("prediction","label")

val metricas = new RegressionMetrics(sumario.rdd.map(x =>
(x(0).asInstanceOf[Double], x(1).asInstanceOf[Double])))


println("Error Cuadrático Medio: " +
metricas.rootMeanSquaredError)

println("Coeficiente de determinación: " + metricas.r2)
```

```
import org.apache.spark.mllib.evaluation.RegressionMetrics
prediccion: org.apache.spark.sql.DataFrame = [id: string, fecha:
string ... 8 more fields]

sumario: org.apache.spark.sql.DataFrame = [prediction: double,
label: double]

metricas: org.apache.spark.mllib.evaluation.RegressionMetrics =
org.apache.spark.mllib.evaluation.RegressionMetrics@6745ed30

Error Cuadrático Medio: 8.527854622149873

Coeficiente de determinación: 0.8562931253602593
```

Consideramos que se ha creado un modelo que se ajusta a la realidad cuando vemos un Error Cuadrático Medio bajo y un coeficiente de determinación alto. En caso de que consideremos que el modelo es aceptable, lo podemos guardar para su posterior uso en el sistema de ficheros HDFS.

```
modelo.write.overwrite().save("hdfs:///models/LinearRegression")
```

2.4.5. Desarrollo de Aplicación MLlib

Se ha usado el programa *IntelliJ* para la creación de la aplicación MLlib, para esto necesitamos instalar el plugin de Scala. Esto lo hacemos pulsando “ctrl+alt+S”, lo cual abre el menú de propiedades de *IntelliJ*. Una vez allí, abrimos la pestaña de *Plugins*, y seleccionamos la pestaña que pone “Instalar un plugin *JetBrains*”. Una vez hecho esto, escribimos *Scala* en el buscador e instalamos el plugin de *Scala*.

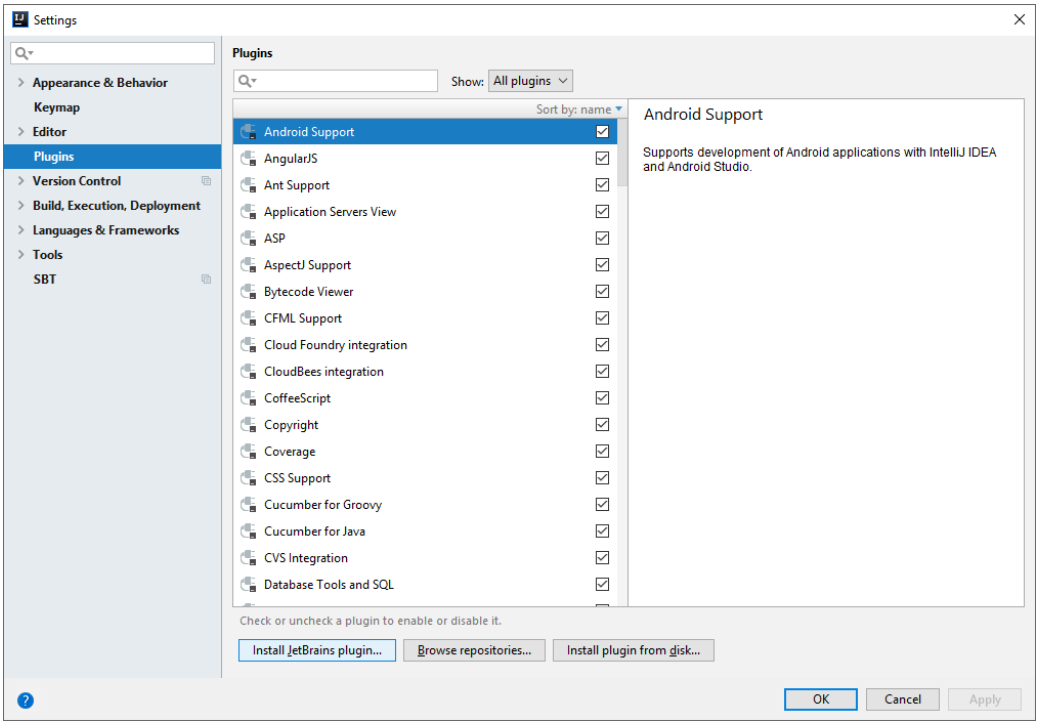

ILUSTRACIÓN 22. INSTALACIÓN DE PLUGIN JETBRAINS. FUENTE: ELABORACIÓN PROPIA.

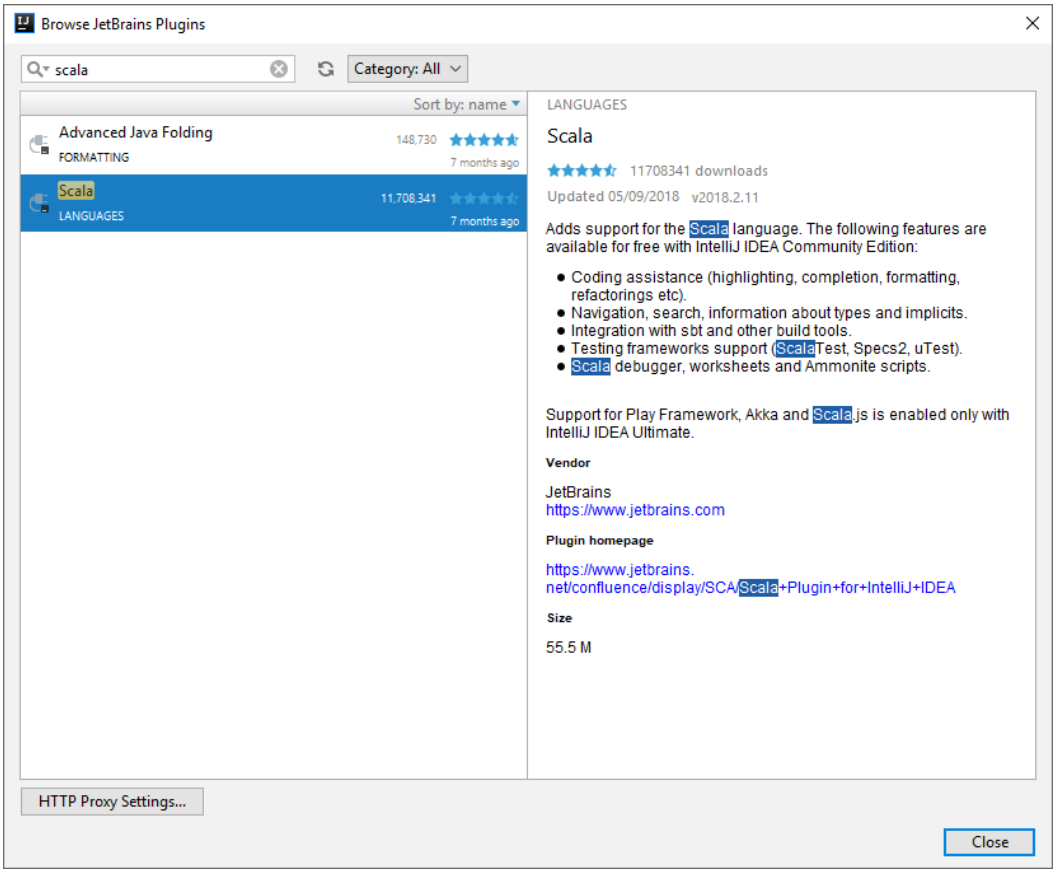

ILUSTRACIÓN 23. INSTALACIÓN DE PLUGIN SCALA JETBRAINS. FUENTE: ELABORACIÓN PROPIA

Una vez hecho esto se procede a la creación de un proyecto SBT.

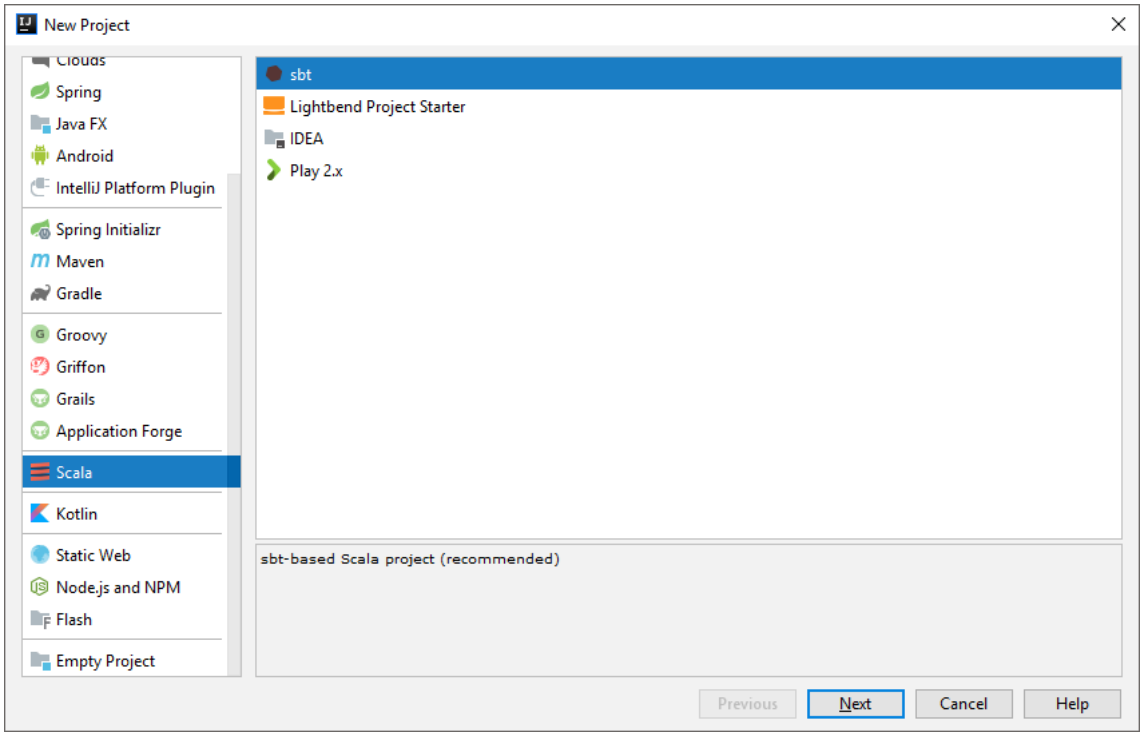

ILUSTRACIÓN 24. CREACIÓN DE PROYECTO SCALA. FUENTE: ELABORACIÓN PROPIA.

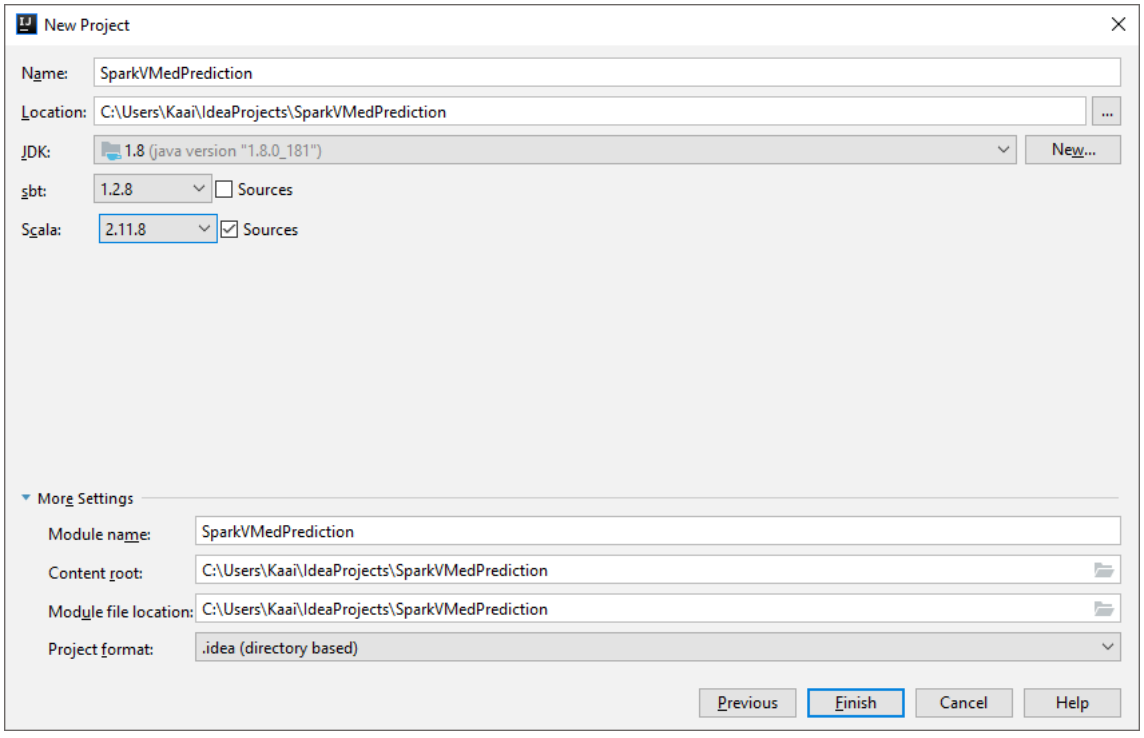

ILUSTRACIÓN 25. VERSIÓN DE SCALA. FUENTE: ELABORACIÓN PROPIA.

Una vez creado el proyecto hay que importar las librerías. Esto se hace en el fichero "**build.sbt**" que lo podemos encontrar en la raíz del proyecto.

```
name := "SparkVMedPrediction"

version := "0.6"

scalaVersion := "2.11.8"

// https://mvnrepository.com/artifact/org.apache.spark/spark-core
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.3.1"
// https://mvnrepository.com/artifact/org.apache.spark/spark-sql
libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.3.1"
// https://mvnrepository.com/artifact/org.apache.spark/spark-mllib
libraryDependencies += "org.apache.spark" %% "spark-mllib" % "2.3.1"
```

Una vez importadas las librerías de Spark creamos un nuevo fichero en "**/src/main/scala**" con el nombre de Prediction.scala.

Lo que se hace en esta aplicación es recuperar un dataset usando los parámetros que se le pasan, los cuales son una fecha y una hora. La fecha con formato "DD-MM-AAAA" y la hora con formato "HH:MM:SS". Para esto debemos inicializar la aplicación accediendo al contexto Spark.

```
import org.apache.spark.sql.{SQLContext, SparkSession}
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.ml.tuning.TrainValidationSplitModel


    val spark = SparkSession.builder().appName("Predicción Vmed").getOrCreate()
    val conf = new SparkConf();
    conf.setMaster("spark://spark-test:7077")
    val sc = new SparkContext(conf)
    implicit val sqlContext = new SQLContext(sc)
    import sqlContext.implicits._
```

Luego de haber cargado el contexto *Spark*, filtramos los parámetros por fecha y hora y cargamos el dataset que contiene los datos pertinentes, así como el modelo que se va a usar para predecir la velocidad media:

```
var fecha = args(0)
    var horas = args(1)
    val split = fecha.split("-")
    val split2 = horas.split(":")
    val año = split(0)
    val mes = split(1)
    val dia = split(2)
    val hora = split2(0)
    val min = split2(1)
    val seg = split2(2)
val data = sqlContext.read.format("json").load("hdfs:///Datasets/" + año
+ "/" + mes + "-" + año + ".json/*.json")
    val timestamp = año + "-" + mes + "-" + dia + " " + hora + ":" + min
+ ":" + seg
    val dataset = data.filter($"fecha" ===
timestamp).select($"id",$"fecha",$"lat" ,$"lng",$"vmed-2" as "vmed-
3",$"vmed-1" as "vmed-2",$"vmed" as "vmed-1", $"vmed+1")
    val modelo =
TrainValidationSplitModel.load("hdfs:///models/LinearRegression")
```

Finalmente, utilizamos el modelo que acabamos de cargar en memoria para transformar los datos y así obtener las predicciones. Al terminar guarda estos resultados en el sistema de ficheros HDFS.

```
    val prediccion = modelo.transform(dataset)
    val resultado =
prediccion.select($"id",$"lat",$"lng",$"vmed-1" as "vmed",
$"prediction", $"vmed+1" as "value")
    val path = "hdfs:///results/result"+ año + mes + dia +
hora + min + seg

resultado.coalesce(1).write.mode("overwrite").format("json").o
ption("header","true").save(path)
    spark.stop()
```

Para ejecutar este programa en el clúster tenemos que ejecutar el comando:

```
spark-submit –class Prediction sparkvmedprediction_1.11-0.6.jar
“12-01-2018” “13:00:00”
```

2.4.6. Desarrollo de Aplicación Web[22]

En primer lugar, se creará una carpeta donde se almacenarán todos los ficheros necesarios para la aplicación web. El nombre de esta carpeta será el nombre del proyecto, que en este caso es *SparkHereMap*. Después de haber creado la carpeta del proyecto, inicializaremos el proyecto de *node* usando el comando:

```
npm init
```

Al terminar la inicialización del proyecto, instalaremos todos los módulos necesarios para el desarrollo de nuestra aplicación. Esto se hará usando el comando:

```
npm install express body-parser jsonfile nodemon shelljs ejs --save
```

Una vez hecho esto, pasaremos a explicar o que hace cada una de las subcarpetas del proyecto:

- **Data**: En esta carpeta se encuentra el fichero "**result.json**" que contiene los resultados de la predicción hecha por la aplicación MLlib.
- **Jars**: Contiene el paquete *jar* que se ha creado en el apartado anterior.
- **Node_modules**: Contiene los distintos módulos necesarios para el funcionamiento de *node.js*.
- **Routes**: Contiene los ficheros controladores de las rutas que a las se puede acceder desde la aplicación.
- **Scripts**: Contiene los scripts *javascript* y *bash* que son necesarios para el funcionamiento de la aplicación.
- **Stylesheets**: Contiene los ficheros *css* que proporcionan el estilo de la aplicación.
- **View**: Contiene los ficheros *ejs* en los cuales se encuentra código *javascript* empotrado con HTML.

Para iniciar la aplicación necesitamos usar el comando

```
nodemon
```

Este plugin permite el refrescado automático de la aplicación cuando se detecta algún cambio en el código.

A continuación, mostraremos un ejemplo de ejecución de la aplicación creada.

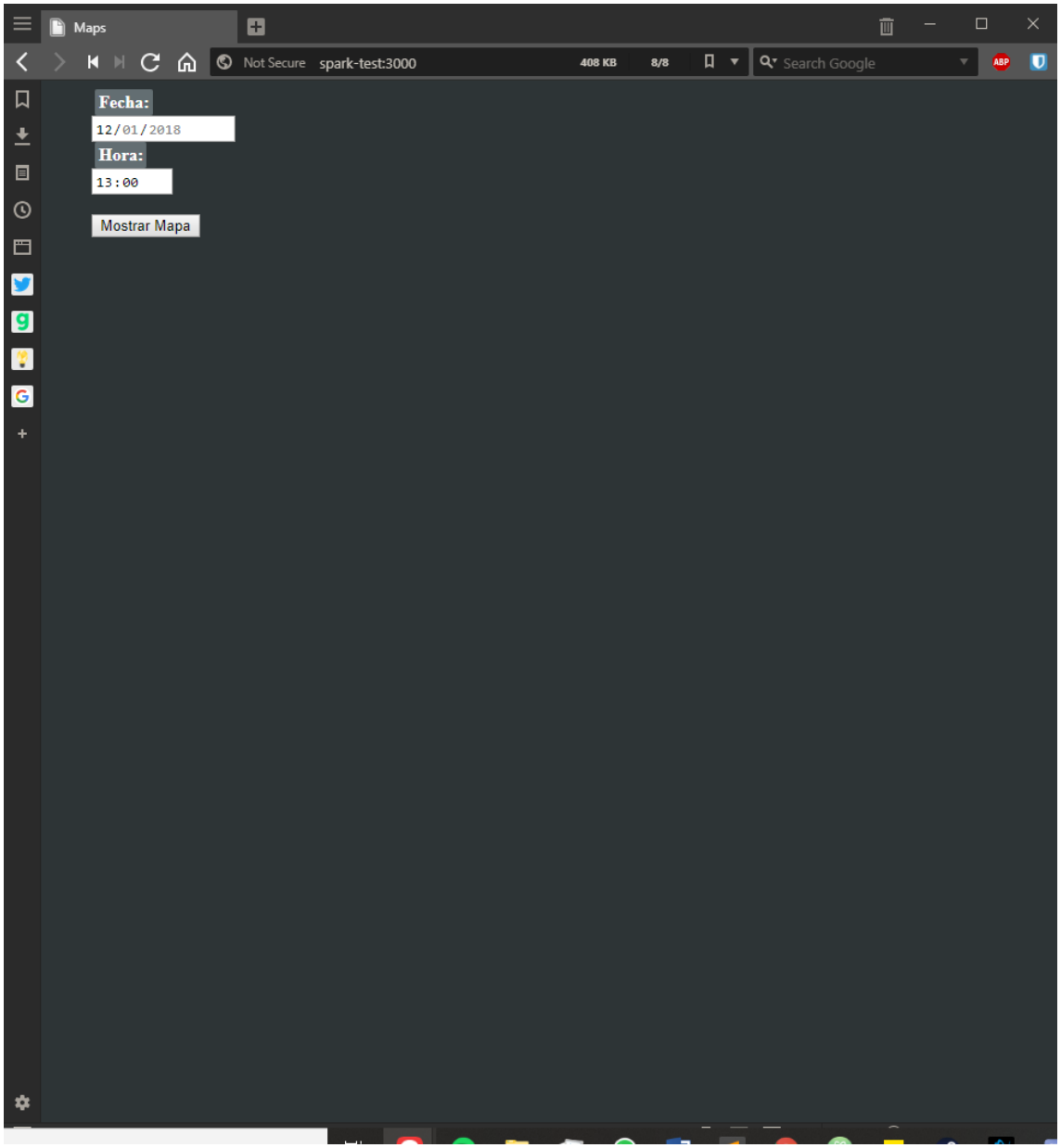

ILUSTRACIÓN 26. PÁGINA INICIAL. FUENTE: ELABORACIÓN PROPIA

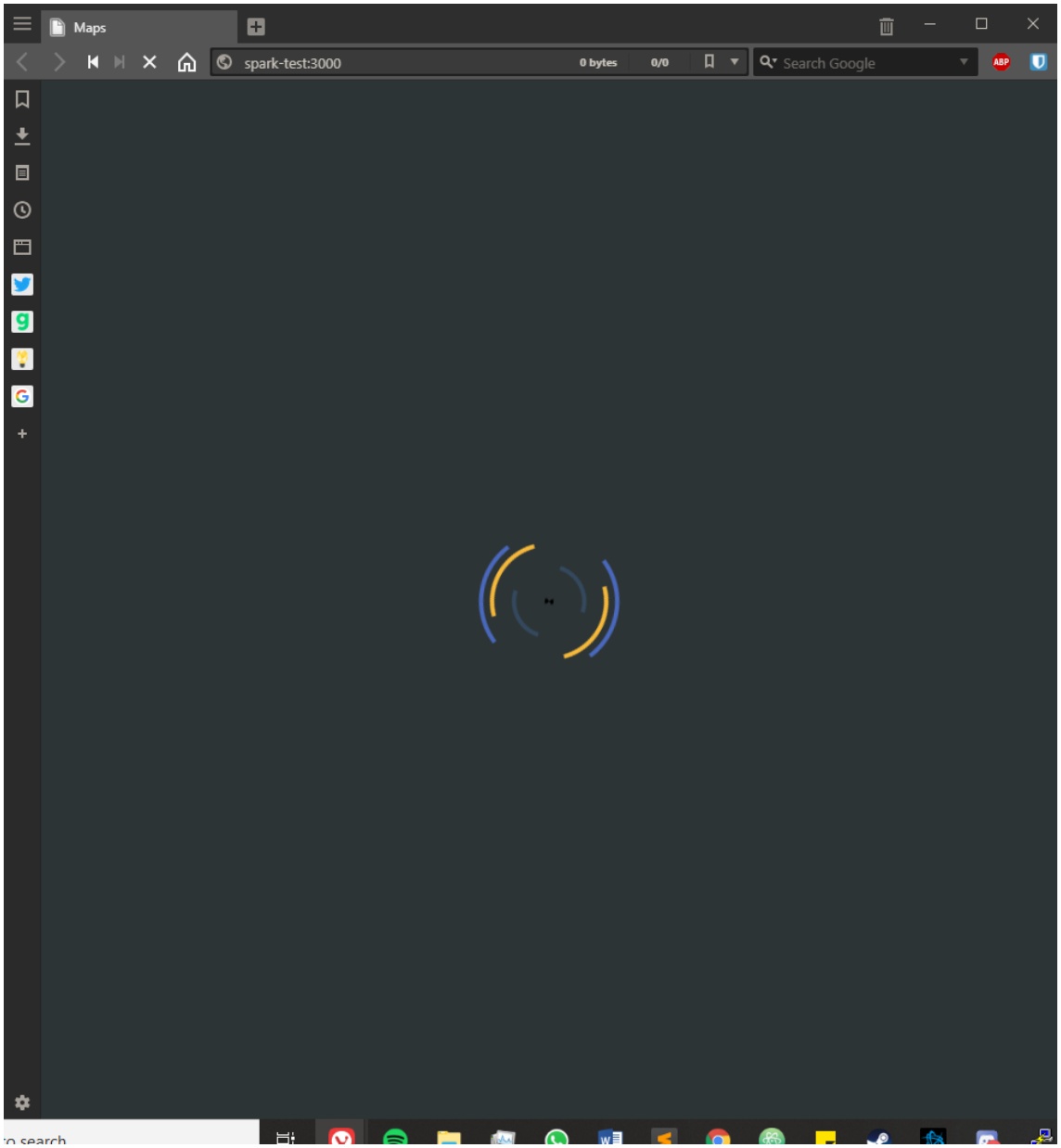

ILUSTRACIÓN 27. PANTALLA DE CARGA MIENTRAS SE ESPERA POR LOS RESULTADOS. FUENTE: ELABORACIÓN PROPIA

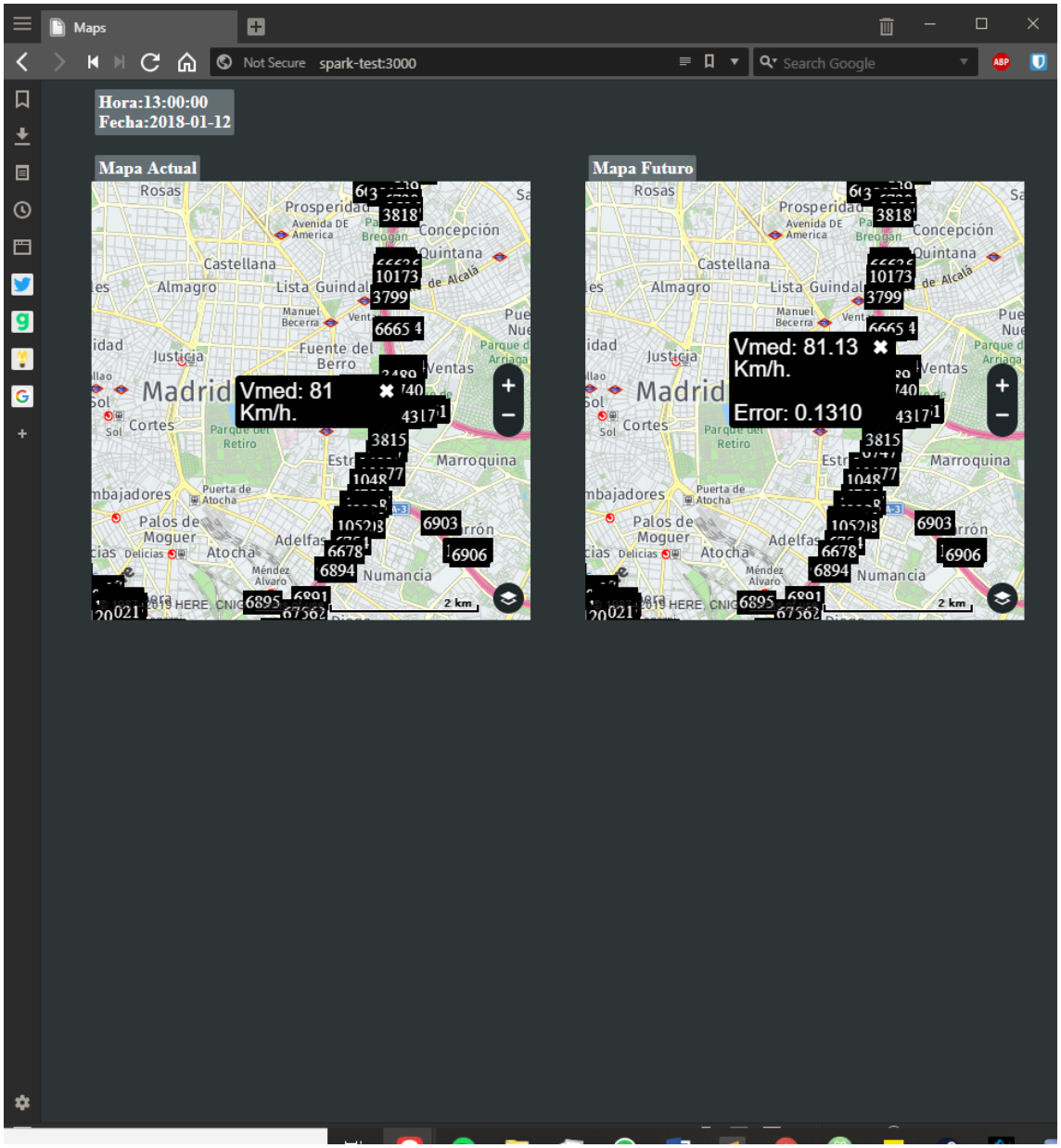

ILUSTRACIÓN 28. RESULTADO DE LA EJECUCIÓN. FUENTE: ELABORACIÓN PROPIA

2.5. Algoritmo de *SpeedUp*

Se ha usado el siguiente programa para la evaluación del informe *SpeedUp.* La base consiste en cargar el modelo, que se ha entrenado con un algoritmo de regresión lineal, y los datos, que en este caso muestra el dataset que contiene los datos de todos los meses de 2017. Después de haber cargado estos 2 elementos, se aplica el modelo a los datos y se sacan unas predicciones. A partir de estas predicciones, también se sabrá el grado de fiabilidad de ellas, mediante el uso del coeficiente de determinación y del error cuadrático medio.

Se han sacado los datos que se muestran a continuación ejecutando el programa mostrado anteriormente mediante el uso del siguiente comando:

```
spark-submit --class SpeedUp –master yarn --deploy-mode cluster SpeedUp.jar
```

El cual se ha ejecutado 10 veces para cada tipo de cluster (mono-nodo o multi-nodo) y usando tanto el dataset reducido como el completo.

```
monoNodoDatasetReducido <- c(40,33,30,36,37,31,38,31,30,32)
multiNodoDatasetReducido <- c(34,27,28,28,27,29,28,28,29,30)

monoNodoDatasetCompleto <- c(43,30,40,33,36,30,36,41,36,35)
multiNodoDatasetCompleto <- c(34,29,29,31,32,29,33,30,35,34)

mediaMonoNodoReducido = mean(monoNodoDatasetReducido)
mediaMultiNodoReducido = mean(multiNodoDatasetReducido)

mediaMonoNodoCompleto = mean(monoNodoDatasetCompleto)
mediaMultiNodoCompleto = mean(multiNodoDatasetCompleto)


datosCompletos <- c(mediaMonoNodoCompleto, mediaMultiNodoCompleto)

datosReducidos <- c(mediaMonoNodoReducido, mediaMultiNodoReducido)



barplot(
  datosCompletos,
  main = "Test SpeedUp con los datos completos de 2017.",
  ylab = "Media de los tiempos de ejecución.",
  names.arg = c("Mono Nodo", "Multi Nodo"),
  beside = TRUE
)

barplot(
  datosReducidos,
  main = "Test SpeedUp con los datos reducidos (enero - junio) de 2017.",
  ylab = "Media de los tiempos de ejecución.",
  names.arg = c("Mono Nodo", "Multi Nodo"),
  beside = TRUE
)
```

2.5.1. Prueba *SpeedUp* con diferentes configuraciones del clúster

Se recuerda que unos de los objetivos del trabajo era ver las limitaciones del clúster. Aunque se debe tener en cuenta que las máquinas que se disponen del CICEI no son modernas.

A continuación, se va a mostrar una serie de gráficas en las cuales se muestra la comparativa del rendimiento entre distintas configuraciones del clúster, dependiendo de si el cluster trabaja con uno o tres nodos. También se mostrarán comparativas usando dos *datasets*, el primero contendrá los datos de los meses comprendidos entre enero y junio. Primero se mostrarán las gráficas de comparación entre el clúster mono-nodo contra el multi-nodo usando el dataset completo.

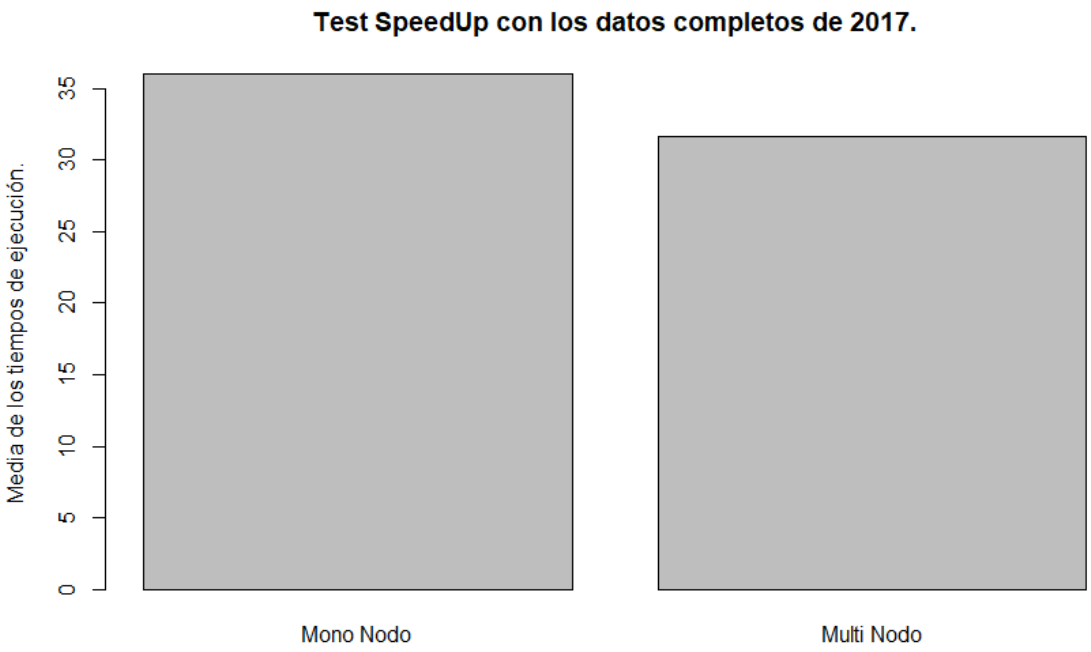

ILUSTRACIÓN 29. TEST SPEEDUP CON EL DATASET COMPLETO DE 2017. FUENTE: ELABORACIÓN PROPIA

En esta gráfica se ve la comparación entre las medias de los tiempos de ejecución del clúster en mono-nodo contra multi-nodo usando el *dataset* completo. Donde se tarda de media en la ejecución del programa unos 36 segundos usando el clúster en modo mono-nodo, cuando usándolo en modo multi-nodo, tarda 31.6 segundos de media. Con esto apreciamos una bajada del tiempo de ejecución de 4.4 segundos, lo que se traduce en una subida del rendimiento de un 11.37%.

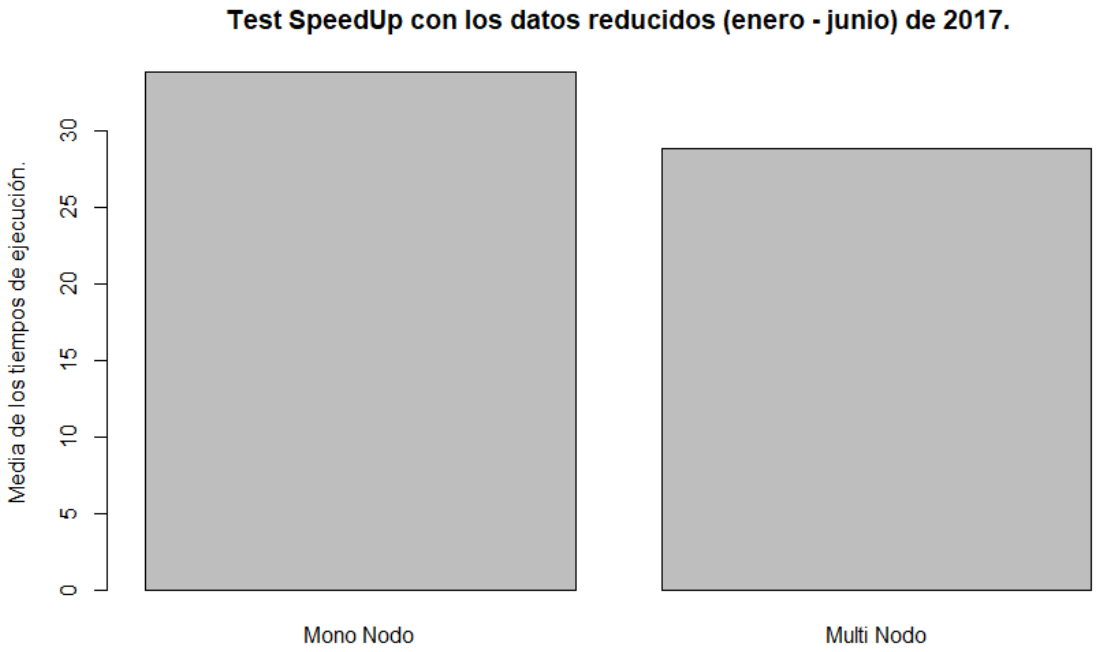

ILUSTRACIÓN 30.TEST SPEEDUP CON EL DATASET REDUCIDO DE 2017. FUENTE: ELABORACIÓN PROPIA

Mientras que usando el dataset reducido, podemos apreciar que se pasa de una media de 33.8 segundos en modo mono-nodo a una media de 28.8 segundos en modo multi-nodo. Lo que nos indica un aumento de rendimiento del 9.73%.

Al comparar el aumento de rendimiento experimentado por el clúster según el tamaño del dataset con el que trabaja. Podemos sacar como conclusión que se produce un aumento del rendimiento experimentado por el clúster mayor en casos de que se trabaje con una mayor cantidad de datos.

3. Conclusiones y Trabajos Futuros

La finalidad de este Trabajo de Fin de Grado era el aprendizaje sobre las tecnologías emergentes en el campo de la ciencia de datos, las cuales son *el Big Data* y el *Machine Learning*. Para alcanzar estos objetivos se han de *Hadoop* y *Spark* para la creación de un sistema de computación distribuida que permita al usuario interactuar con él de forma sencilla. En este caso, se ha terminado con un clúster ya configurado que se puede usar para futuros proyectos o tareas de investigación.

Al haber realizado el diseño e implementación del sistema, se puede afirmar la obtención de las competencias definidas al principio de este Trabajo. Se ha obtenido la capacidad para diseñar, desplegar, gestionar y mantener sistemas complejos multicomputador. Así como la gestión de la información para llevar a cabo proyectos basados en Inteligencia Artificial.

Debido a que este tipo de proyecto toca distintas disciplinas en la informática, se puede ampliar de formas muy diversas.

Aunque no se ha mencionado mucho, una de las características más importantes de este proyecto es la seguridad, ya que en este tipo de clúster puede ser usado para el procesamiento de datos imprescindibles para una organización. Debido a esto, se puede mejorar la seguridad implantando una serie de medidas como son el aislamiento de las redes, implantación de medidas de seguridad en red como pueden ser firewall, etc.

Otra de las posibles mejoras de este trabajo, podría ser la creación de una aplicación que usando uno de los modelos disponibles en MLlib, muestre las predicciones de velocidad a una hora concreta en cualquier punto de la M30.

Se podría realizar el diseño de una red de computación inter-universitaria para el desarrollo en materias de I+D. Esta idea, en caso de que se llevase a cabo, podría dar pie a la creación de empleo.

4. Fuentes de información

1. World Internet Usage and Population Statistics (En Inglés). (2019 Marzo). Lugar de publicación: https://www.internetworldstats.com/stats.htm

2. APIs and SDKs for maps and location-aware web and mobile apps (En Inglés). Lugar de publicación: https://developer.here.com

3. Apasoft Training. Curso Monta un cluster Hadoop Big Data desde cero. Lugar de publicación: https://www.udemy.com

4. VNI Forecast Highlights Tool. Lugar de publicación: https://www.cisco.com/c/m/en_us/solutions/service-provider/vni-forecast-highlights.html

5. Macrodatos. Lugar de publicación: https://es.wikipedia.org/wiki/Macrodatos

6. Juan, C. (2016 Noviembre 03). ¿Cuáles son las 5 V's del Big data?. Lugar de publicación: https://www.iebschool.com/blogs/5-vs-del-big-data/

7. Aprendizaje Automático o Machine Learning. Lugar de publicación: https://es.wikipedia.org/wiki/Aprendizaje_automático

8. Apache Hadoop. Lugar de publicación: https://hadoop.apache.org

9. MapReduce y Yarn. Lugar de publicación: https://es.wikipedia.org/wiki/MapReduce

10. Apache Spark. Lugar de publicación: https://spark.apache.org

11. Machine Learning Library (MLlib) Guide (En Inglés). Lugar de publicación: https://spark.apache.org/docs/2.3.1/ml-guide.html

12. Apache Zeppelin (En Inglés). Lugar de publicación: https://zeppelin.apache.org

13. Scala (Lenguaje de programación, en inglés). Lugar de publicación: https://www.scala-lang.org

14. SBT (Empaquetador de aplicaciones Scala, en inglés). Lugar de publicación: https://www.scala-sbt.org

15. Node.js (Framework para web basado en javascript, en inglés). Lugar de publicación: https://nodejs.org/en/

16. Github. Lugar de publicación: https://github.com

17. R (Lenguaje de programación dedicado a la estadística, en inglés). Lugar de publicación: https://www.r-project.org

18. Datos recogidos en los puntos de M-30. Lugar de publicación: https://datos.madrid.es/portal/site/egob/menuitem.c05c1f754a33a9fbe4b2e4b284f1a5a0/?vgnextoid=33cb30c367e78410VgnVCM1000000b205a0aRCRD&vgnextchannel=374512b9ace9f310VgnVCM100000171f5a0aRCRD

19. Sistema de coordenadas Universal transversal de Mercator. Lugar de publicación: https://es.wikipedia.org/wiki/Sistema_de_coordenadas_universal_transversal_de_Mercator

20. Conversor de coordenadas a UTM. Lugar de publicación: http://www.zonums.com/online/coords/cotrans.php?module=14

21. Linear Methods (Parámetros regParam y elasticNetParam, en inglés). Lugar de publicación: https://spark.apache.org/docs/1.5.2/ml-linear-methods.html

22. Cruz Martín, F A. Repositorio que contiene el código fuente de la aplicación web. Lugar de publicación: https://github.com/Cyre/SparkHereMap