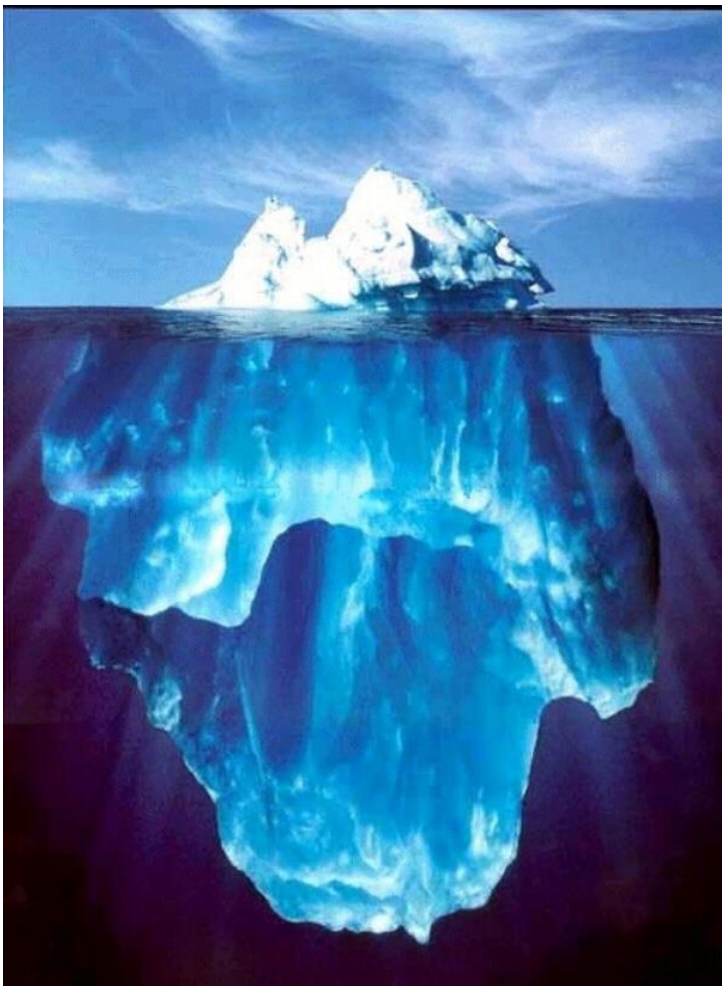
T.F.M

IMPACTO DEL BIG DATA EN EL APRENDIZAJE AUTOMÁTICO



Alfonso Campos de Padua

CIFF

Abstract

Con el advenimiento de las tecnologías Big Data, el panorama tecnológico está sufriendo profundos cambios. Estos cambios, tienen consecuencias en los entornos analíticos utilizados para realizar aprendizaje automático. Y dichas consecuencias, no solo afectan al entorno analítico desde un punto de vista meramente físico - hardware -, sino que impactan también en la forma en la que el analista entiende su relación con la propia plataforma. Es desde este ángulo, desde el que se deben evaluar las oportunidades y desafíos enfrente de los analistas contemporáneos. Este trabajo, pretende ahondar en las implicaciones de realizar analítica orientada al aprendizaje automático en entornos no distribuidos frente a entornos distribuidos. En particular, el foco se pondrá sobre la facilidad de uso, entendiendo esta como facilidad operacional - puesta en marcha de una plataforma de desarrollo -, facilidad de desarrollo - codificación -, disponibilidad de funcionalidades - funcionalidad recogida en las librerías - y rendimiento - tiempo de ejecución. Los resultados avalan la madurez de las tecnologías existentes no distribuidas y la necesidad de contar con un entorno distribuido de mayores dimensiones para poder aprovechar los beneficios del procesamiento distribuido. Sin embargo, incluso frente a la inmadurez del ecosistema de aprendizaje automático en entornos distribuidos, resulta prometedor iniciarse en estas tecnologías donde, cabe recordar que el mismo código que funciona en un entorno con una unidad de computación, funciona en un entorno con decenas de miles de estas unidades.

Palabras clave: Big Data, Aprendizaje Automático, Analista, Entornos Distribuidos

1. Introducción

1.1. Motivación

En los tiempos que corren, es difícil trabajar en el sector de las tecnologías de la información y no haber oído hablar del Big Data. Con mayor o menor acierto, te ves inmerso en conversaciones donde palabras como Cloudera, Hadoop, Hive… aparecen mencionadas como si se tratara de la quintaesencia del conocimiento informático.

Mucho se ha mencionado ya sobre el clima que se vive en torno al Big Data. Sin duda la mejor cita es la que equipara la expectación en torno al Big Data a la que se vive en torno al sexo cuando se es adolescente. Ahora mismo no hay empresa que se precie que no tenga por máximo objetivo el aunar todas las fuentes de datos de los negocios en un único punto. Se está en una fase inicial, donde no se tiene claro el qué o para qué, sino simplemente se tiene claro que hay que subirse a este tren y entrar en el mundillo. A este fin, todos los actores del mercado están contratando arquitectos, ingenieros y otros roles, con el objetivo de montar un *Data Lake*. Es la versión Big Data de lo que fue el *Data Warehouse* para el Business Intelligence.

Sin embargo, todo el ecosistema del Big Data no deja de ser una herramienta, un medio, que permita realizar una tarea. Todavía no ha llegado el tiempo - al menos en mercados como el español - donde las plataformas ya estén operativas y sea necesario tener claro el qué. Será ese el momento, no ya de la arquitectura, de la ingesta de datos, de la gobernanza, sino de la analítica.

El qué es una pregunta difícil, pues al contrario que con la plataforma - que tiene un enfoque generalista que se abstrae hasta cierto punto de la pregunta concreta de negocio que se quiere resolver - requiere de una pregunta clara, un dominio concreto, una finalidad definida. Es probablemente este el ámbito donde surge la figura del científico de datos o *Data Scientist*.

El científico de datos es la enésima iteración de analista, minero de datos… y se caracteriza por trabajar principalmente realizando modelos sobre plataformas Big Data. Es importante señalar, que el ámbito de los modelos que utiliza tiene un enfoque basado en las técnicas de aprendizaje automático, ya que son estas técnicas las que sacan mayor partido del gran volumen de datos que se consigue tener en un *Data Lake*. Además, estas técnicas permiten abordar la elaboración de modelos tomando múltiples dimensiones para los datos, dimensiones que se han visto enriquecidas con las aportaciones de fuentes de datos heterogéneas - frente a lo que teníamos antes, el *Data Warehouse*.

Y, sin embargo, en general, el científico de datos trabaja para resolver mediante estas herramientas y técnicas un problema concreto en un ámbito de negocio determinado. Esto le exige forzosamente conocer dicho ámbito de negocio, de igual forma que un analista había de conocer el contexto de aquello que pretendía modelar, lo que inevitablemente le vuelve íntimamente ligado al dato y a su significado.

Existe quizás, entre la plataforma y las miríadas de roles para establecerla, proveerla y gestionarla y los científicos de datos centrados en el dominio de negocio y en entender el dato concreto que lo sustenta, espacio para aquellos que trabajen en el punto donde el algoritmo y el conjunto de procesos en que se descompone, encuentra su más íntima relación con la plataforma y la tecnología subyacente en que acaba finalmente implementado.

Es quizás esta la frontera donde todavía no se debería hablar de ciencia más aun de ingeniería. A fin de cuentas, es el punto donde se han de buscar soluciones lo suficientemente particulares como para cubrir un problema concreto y lo suficientemente generales como para poder aplicarse a otros problemas de la misma índole. Es el punto donde se examina de cerca a los algoritmos para ver “de que pasta” están hechos. Es el dominio del *Machine Learning*

Engineer, un rol desconocido en España que alcanza el cuarto de millón por salario en los rincones más exclusivos del valle del Silicio[1].

1.2. Conceptos

1.2.1. Big Data

Pero, ¿qué es el Big Data? El Big Data es el nombre con el que se conoce a un conjunto de tecnologías que se caracterizan por conformar un ecosistema distribuido de procesamiento de datos. Este ecosistema tiene algunas virtudes sobre sus antepasados primigenios (como el Grid Computing) siendo claves al menos dos:

- Localidad del dato; es decir, ya no es el dato el que se distribuye entre distintas unidades de computación, sino que es el algoritmo de computación el que se distribuye entre distintas unidades con datos.
- Abstracción; el entorno abstrae al usuario del detalle concreto de los elementos distribuidos, presentando un API (*Application Programming Interface* o interfaz de programación de aplicaciones) unificado de interacción que generaliza las tareas sobre cualquier clúster, con independencia de los recursos concretos de este - se acabó pensar en la computación distribuida o paralela en términos de MPI (Message Passing Interface o interfaz de paso de mensajes).

Un entorno Big Data se caracteriza por permitir el procesamiento de un volumen masivo de datos desestructurados en un tiempo razonable. Es complejo definir qué es un volumen masivo, cómo de desestructurados son los datos y cuál es la velocidad razonable. Muchas veces, la definición de cada uno de estos tres conceptos (denominados las 3 V’s) depende más del ámbito o contexto de uso que de razones puramente objetivas. Además, su interpretación se ve

[1] https://recode.net/2015/07/15/ai-conspiracy-the-scientists-behind-deep-learning/

modificada con el tiempo. Sin embargo, una medida razonable del tamaño a partir del cual se puede hablar de Big Data comenzaría en unidades de GB. Igualmente, la velocidad de procesamiento en función del volumen de datos, podría establecerse en torno a 1 minuto/GB. Finalmente, la variedad en los datos puede incluir desde documentos semiestructurados como XML o JSON hasta ficheros no estructurados como información de Audio o Video.

Los ecosistemas Big Data se distribuyen tanto con licencia libre (versiones Vainilla), como con licencia comercial, siendo 3 empresas los principales referentes: Cloudera, HortonWorks y MapR. Habitualmente, estas distribuciones están disponibles como un entorno *todo en uno*, como es el caso de las distribuciones *Sandbox*. Este tipo de entornos preparados, suele ser útil para elaborar pruebas o desarrollar código. Incidentalmente, también resultan oportunos para realizar trabajos de fin de máster.

1.2.2. Entorno distribuido

Pero, ¿qué es un entorno distribuido? Un entorno distribuido es un entorno que se soporta sobre diferentes unidades de computación alojadas sobre máquinas independientes interconectadas entre sí. Esta agrupación de unidades de computación interconectadas se suele definir como clúster. Cuando la algoritmia se define para trabajar sobre este tipo de unidad lógica, permite aprovechar las diferentes unidades de computación a la vez, lo que forzosamente implica la paralelización de la computación.

Es posible aprovechar la paralelización de algoritmos en entornos no distribuidos, bien porque se realicen en paralelo tareas independientes, bien porque se realice la misma tarea. Sin embargo, la computación distribuida se caracteriza por la absoluta independencia de las unidades de procesamiento que poseen su propia CPU, RAM, disco duro… e interactúa con el resto de elementos de computación a través de una red externa y no de forma interna.

Un entorno distribuido permite un escalado horizontal en número de unidades de computación órdenes de magnitud por encima del alcanzado en paralelización no distribuida.

1.2.3. Aprendizaje Automático

Pero, ¿qué es el aprendizaje automático? El Aprendizaje Automático “es una rama de la inteligencia artificial cuyo objetivo es desarrollar técnicas que permitan a las computadoras aprender. De forma más concreta, se trata de crear programas capaces de generalizar comportamientos a partir de una información no estructurada suministrada en forma de ejemplos. Es, por lo tanto, un proceso de inducción del conocimiento”[2]. El Aprendizaje Automático generaliza comportamientos construyendo modelos. Estos modelos requieren de gran número de datos para aumentar su fiabilidad, y es por esto que el aprendizaje automático encuentra su espacio natural de la mano del Big Data.

1.3. Objetivos

Este trabajo, profundiza en el beneficio real que tiene la paralelización en entornos Big Data para el analista. El beneficio no puede solo entenderse en términos puramente numéricos - como una cuestión de eficiencia en la ejecución -, sino que debe también considerar el tiempo que es necesario para que el propio analista desarrolle el código. Este tiempo, más difícil de medir, se puede expresar en términos de facilidad de uso de la plataforma, coste de adaptación al lenguaje, completitud y adecuación de las herramientas, facilidad de encontrar documentación y en última instancia, capacidad de reusabilidad y escalado.

Los pasos a dar para poder obtener resultados y conclusiones en esta línea se pueden circunscribir en las siguientes etapas.

1.3.1. Configuración de los sistemas

[2] https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico

Sin duda, el primer paso consiste en encontrar y describir el entorno *hardware* que permita llevar a cabo las pruebas. Se busca que el entorno permita probar diferentes configuraciones de recursos físicos, que sea autocontenido y que sea representativo de entornos de trabajo habituales. Se busca, en definitiva, trabajar con una imagen virtual que proporcione *out of the box* las herramientas necesarias.

Es posible no obstante que sea necesario configurar manualmente alguna herramienta para poder dotar al entorno de la flexibilidad necesaria para realizar el desarrollo multilenguaje.

1.3.2. Configuración de las herramientas de desarrollo

Una vez tengamos el entorno hardware de trabajo, es decir, la plataforma en sí, es necesario proveerla de las herramientas que vayan a ser necesarias para poder realizar el desarrollo de los algoritmos en distintas implementaciones.

Esto implica seleccionar los intérpretes con los que se va a trabajar, así como las librerías específicas de aprendizaje automático necesarias. Los intérpretes deben ser representativos de los casos de desarrollo distribuido y no distribuido y las librerías deben de presentar un API lo más homogéneo posible.

1.3.3. Selección de los algoritmos y datos

Una vez definidos los lenguajes y librerías, es necesario analizar los casos prácticos que se quieren desarrollar. La idea es que sean casos de uso típicos; una suerte de A, B, C de la modelización mediante Aprendizaje Automático.

El conjunto de datos interesa que sea fácil de manejar, minimizando el preprocesamiento necesario. Debe ser adecuado para trabajar con los diferentes algoritmos. También resulta interesante disponer de un conjunto de datos masivo para entrenamiento y un conjunto diferente para el test.

1.3.4. Implementación del código común

Para poder automatizar la obtención de los *datasets* y la correcta propagación de los datos en las rutas de trabajo necesarias, será necesario elaborar scripts.

Además, se deberán implementar en cada lenguaje los pasos necesarios para cargar en objetos los datos y realizar el preprocesamiento que, aun siendo menor, resulte necesario. Son los pasos necesarios para poder alimentar a los algoritmos con los datos.

1.3.5. Implementación del código algorítmico

Es necesario implementar las soluciones algorítmicas para los casos concretos que se elijan. Así, habrá que hacer las llamadas pertinentes para cada intérprete y lenguaje al API que exponga la implementación específica de la librería en ese contexto. En este apartado, en definitiva, se entrenan los modelos a partir de los datos de entrenamiento y se realiza la evaluación de los algoritmos con los datos de test u otras métricas pertinentes que permitan evaluar y homologar el desempeño de los algoritmos.

1.3.6. Ejecuciones y resultados

Para realizar las ejecuciones es necesario ser extremadamente cauteloso para no llenar la memoria de objetos cargados de la ejecución anterior. Además, los distintos parámetros de configuración del entorno clúster se tratarán de dejar a los valores óptimos recomendados, buscando la sencillez y el funcionamiento por defecto que un analista haría de la plataforma.

Los resultados medirán principalmente el tiempo de ejecución tanto de la construcción del modelo como de validación del mismo, si bien se hará un análisis interpretativo con las impresiones pertinentes que pongan sobre la mesa otros aspectos de relieve que expliquen o aumenten el significado y consideraciones sobre la ejecución.

1.3.7. Conclusiones

Finalmente, se recogerán unas conclusiones globales a la luz de los resultados y la experiencia obtenida que pongan en valor los resultados obtenidos frente a los resultados esperados.

1.4. Resultados esperados

A priori, cabe esperar que el rendimiento de la ejecución mediante paradigmas distribuidos supere ampliamente el rendimiento mediante paradigma no distribuido.

Además, cabe imaginar que será más sencillo desarrollar en un entorno no distribuido al poder obviar las vicisitudes que la compleja plataforma tecnológica que lo sustenta impone al desarrollador.

Parece sensato pensar que pueda ser más complejo realizar la carga de datos en el entorno distribuido, por ejemplo, que en el entorno no distribuido.

También se espera que el uso de las librerías en implementaciones distribuidas resulte más complejo y oscuro, por falta de documentación e insuficiencia de madurez, que su homólogo no distribuido.

Finalmente, se espera una curva de aprendizaje elevada a la hora de trabajar con los intérpretes distribuidos y manejarse en lenguajes complejos.

2. Entorno de pruebas

El entorno de pruebas se caracteriza por contar con una infraestructura o sistemas que posibilitan el trabajo analítico desde un punto de hardware, así como de una serie de herramientas que posibilitan el trabajo analítico desde el punto de vista del software. Resulta pertinente describir el entorno atendiendo a estos conceptos, para poder poner el foco en los aspectos que mejor definen cada uno de estos ámbitos diferenciados.

2.1. Sistemas

Desde el punto de vista de la infraestructura, se han utilizado un conjunto amplio de elementos tecnológicos que merecen una descripción detallada. Se diferencian dos ámbitos propios, el real y el virtual. El ámbito real recoge los aspectos del entorno físico de trabajo. El ámbito virtual los detalles del entorno simulado de trabajo que es el verdadero objeto de interés. Sin embargo, cabe cubrir ambos pues los recursos físicos del entorno real determinan las opciones disponibles en el entorno virtual. Además, resulta apropiado para presentar la radiografía completa de la infraestructura.

2.1.1. Descripción del Entorno Real

Las prácticas se han llevado a cabo en un ordenador personal (PC) con las siguientes características:

2.1.1.1. Procesamiento

La capacidad de procesamiento es el factor principal a la hora de operar los algoritmos, ya que en última instancia el ordenador se comporta como un gran sumador de dígitos binarios, donde millones de estas sumas deben ser procesadas cada segundo.

El entorno de pruebas posee una CPU Core i7 4770k con 4 núcleos físicos y 8 núcleos lógicos, cada uno de ellos funcionando a 3.9 GHz, lo que representa un entorno doméstico potente camino hacia la potencia típica de una *Workstation*.

2.1.1.2. Memoria

La memoria sirve de almacén temporal a lo largo de una ejecución y su tamaño determina el volumen de datos sobre el que se puede trabajar de forma efectiva. Cuando se llena, se produce la paginación, que consiste en la escritura efímera de páginas de datos a disco duro o similar, necesariamente más lento. Es importante disponer de memoria suficiente para evitar este fenómeno, de lo contrario los resultados se pueden ver alterados por este fenómeno y afectar las métricas que se quieran obtener, en particular el tiempo de ejecución de los algoritmos.

El entorno de pruebas inicialmente contaba con 16 GB de RAM, sin embargo, para poder ampliar el alcance de las pruebas se ha ampliado a 32 GB de RAM. Esta RAM es DDR3 @ 2400 MHz con latencias CL10-12-12 y trabaja en *Dual Channel*.

2.1.1.3. Almacenamiento

La unidad de almacenamiento permite alojar la información y los programas antes de que se carguen en memoria. Su rápido acceso es bastante importante pues su capacidad de acceso fácilmente puede suponer un cuello de botella en un ordenador moderno (es la pieza más lenta del ordenador).

En el caso del entorno de pruebas, su importancia se ve aumentada ya que las peticiones a disco del sistema operativo de la máquina virtual se suman a las peticiones a disco del sistema operativo en el *host* o entorno principal.

Sin embargo, en el entorno de pruebas, se dispone de una SSD o disco duro de estado sólido. Estas unidades operan con una mayor velocidad que el disco duro tradicional, siendo la

principal diferencia la reducida latencia que presentan frente al disco duro. En particular, el entorno de pruebas cuenta con una SSD Samsung 840 PRO con 250 GB de capacidad. Es sobre esta unidad donde se aloja la imagen virtual que contiene el entorno distribuido.

2.1.1.4. Software

El sistema real tiene un sistema operativo Windows 10. Además, para montar la imagen virtualizada se va a utilizar el software Virtual Box, que se distribuye de forma libre bajo licencia GPL v2[1].

2.1.2. Descripción del Entorno Virtual

Para realizar el trabajo, se ha escogido un entorno Big Data autocontenido de la mano de la imagen virtual HDP2.3 *Sandbox*. Aquí la palabra *Sandbox* hace referencia directa al concepto anglosajón de tener un entorno estanco autocontenido y replicable, ideal para poder realizar las pruebas con cierta independencia del entorno real subyacente. Dicha independencia, no obstante, no es absoluta y se asienta en los recursos físicos del entorno real descritos con anterioridad. Sin embargo, es posible arrancar este entorno estanco con una configuración particular y controlada de recursos físicos, simulando de esta forma distintos entornos reales - como si se dispusiera de ellos - siempre que los recursos de la imagen virtual sean menores que el total disponible en el entorno real.

2.1.2.1. Recursos físicos

Se ha optado por realizar las pruebas solo con una configuración determinada de CPU / RAM sin la posibilidad de poder dar cierta perspectiva del escalado de los algoritmos en función de los recursos físicos de la plataforma. Esto es así, ya que en pruebas iniciales se ha visto que es necesario modificar multitud de parámetros de configuración del clúster para que este “vea” y utilice los recursos adicionales. La configuración escogida representa una estación de trabajo o

un ordenador de sobremesa de altas prestaciones. Esta configuración hace uso de 6 núcleos lógicos y 24 GB de RAM para la máquina virtual.

2.1.2.2. Imagen Virtual

Como se adelantaba, se ha optado por una imagen virtual precargada con toda la suite Big Data de Horton, incluyendo distintos elementos que se pasan a describir.

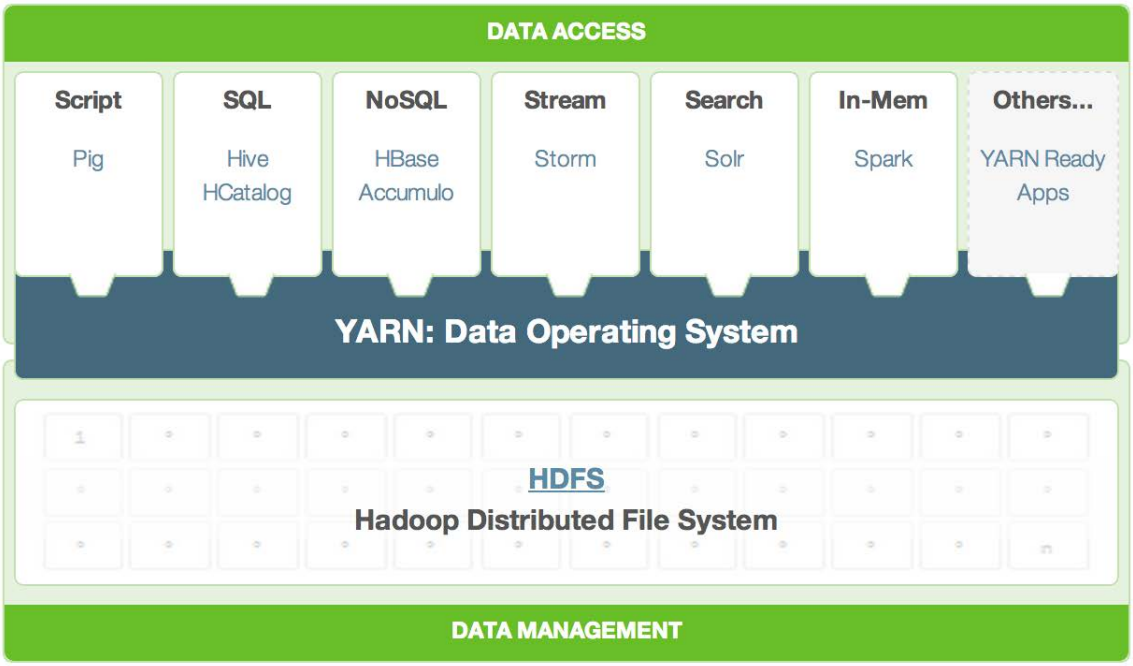
2.1.2.2.1 Entorno no distribuido

La imagen virtual simula un espacio de trabajo no distribuido que es el lugar donde se van a ejecutar los algoritmos tradicionales.

- Sistema Operativo

El entorno no distribuido opera bajo un sistema operativo Linux, en particular con una distribución ligera basada en un CentOS. Este sistema operativo nos expone como es habitual un intérprete Bash por línea de comandos que es la forma con la que interactuaremos con el sistema operativo para crear carpetas, obtener los sets de datos, mover ficheros, ejecutar programas o cargar datos en el sistema distribuido de datos HDFS.

2.1.2.2.2 Entorno distribuido



La imagen virtual simula un espacio de trabajo distribuido que es el lugar donde se van a ejecutar los algoritmos distribuidos. Este entorno distribuido es a la sazón un entorno Big Data y posee una serie de elementos o partes móviles de las cuales se han seleccionado los mínimos componentes necesarios para poder ejecutar el código Spark/PySpark desde el Notebook Zeppelin. Además, se ha utilizado una herramienta de gestión para configurar los servicios.

Todos los parámetros de configuración de los diferentes servicios se han establecido a los valores recomendados por defecto tras dotar a la imagen virtual de los recursos adicionales sobre la configuración base pensada para 4 núcleos y 8 GB de RAM que viene prefijada en la *Sandbox* de Horton.

- HDFS

Con este nombre se conoce al sistema de ficheros con el que se trabaja en mundo Big Data. HDFS es la sigla de Hadoop Distributed File System o sistema de ficheros distribuido de Hadoop. Si bien inicialmente se utilizaba solo para los procesos Map Reduce propios de Hadoop,

tras la generalización de diferentes motores de ejecución que utilizan distintas técnicas de procesamiento, HDFS ha permanecido como el sistema distribuido de almacenamiento característico en arquitecturas Big Data.

- YARN

Este componente es el motor de ejecución y su función es trabajar convirtiendo distintas implementaciones paralelizadas de los distintos algoritmos en órdenes concretos sobre los distintos nodos del entorno distribuido. La forma generalizada que tiene en su última iteración, permite que procese órdenes de trabajo - *jobs* - basados en MapReduce2, en Spark, etcétera.

- Spark

Es una de las tecnologías de paralelización de procesos más puntera en el mercado. Mediante el uso de memoria RAM en vez de disco secundario (como Hadoop) así como mediante un mayor grado de control sobre las unidades de paralelismo (Map, Reduce, Fold…) ofrece un mayor rendimiento que Hadoop, lo que se traduce en una mayor eficiencia y granularidad de los procesos distribuidos. Cabe destacar que está implementado íntegramente en Scala, un lenguaje de programación que combina paradigma orientado a objetos y funcional y trabaja sobre la máquina virtual de Java o JVM. Ofrece APIs de desarrollo en Scala, Java y Python.

- Ambari

Si bien para las pruebas que se quieren realizar este componente no es de interés, resulta adecuado destacarlo ya que es la herramienta que permite centralizar el control de las distintas tecnologías expuestas y permite mediante su interfaz gráfica tener control sobre los distintos servicios y ajustar parámetros de configuración de cada uno de ellos. Es a la vez una herramienta de configuración y de administración de todos los servicios del entorno distribuido o Clúster.

- Zeppelin

A la hora de trabajar desarrollando el código concreto que permitirá la ejecución de algoritmos distribuidos y no distribuidos, han surgido con el advenimiento del Big Data y las tecnologías de *Web Sockets* los entornos de desarrollo mediante *Notebook*. Estos entornos se caracterizan por ser accesibles mediante un navegador y exponer como una página web interactiva el entorno de desarrollo. Sin llegar a tener el nivel de funcionalidades que expone un IDE, permiten realizar un desarrollo ligero y rápido al combinar código y documentación en un único espacio, resultando muy adecuados cuando se realiza programación exploratoria, como suele ser el caso en entornos de analítica.

Dentro del elenco de distintos *Notebooks* disponibles en el mercado, Zeppelin posee la facultad de poder trabajar con distintos lenguajes (Spark, PySpark, Python, Bash, Markdown, SQL y muchos más) de forma sencilla y con una interfaz cuidada. De especial interés es el poder combinar en un mismo *Notebook* celdas de distintos lenguajes.

Cabe recalcar que ha sido necesario desinstalar y volver a instalar el servicio de Zeppelin mediante Ambari, activando un parámetro de configuración que permite disponer de algunas librerías Python que no venían activas por defecto.

2.2. Desarrollo

Para llevar a cabo el desarrollo es pertinente presentar los intérpretes que se han seleccionado como representantes de los distintos paradigmas (distribuido y no distribuido) así como introducir las librerías de aprendizaje automático seleccionadas dentro de las distintas opciones.

2.2.1. Intérpretes

Para poder llevar a cabo la codificación de los algoritmos, así como para poder configurar de forma adecuada el entorno, se han utilizado diferentes intérpretes. Se hace un breve repaso sobre las características de los mismos y por qué han sido escogidos. Un intérprete en general se asocia a un lenguaje de programación concreto, si bien depende del ámbito de ejecución del intérprete y su implementación el posibilitar que soporte más lenguajes.

2.2.1.1. No Distribuidos

Los intérpretes bajo este epígrafe no cuentan de forma nativa de mecanismos para permitir la ejecución de forma paralela. Cabe recordar que un lenguaje no es *per se* distribuido o no distribuido, sino que depende de la implementación subyacente de la librería concreta en uso en el lenguaje y su capacidad para tratar con rutinas paralelas o no.

Dicho esto, se entiende que haya intérpretes que se basan en lenguajes que no tratan de forma automática el paralelismo. Este es el caso de la mayoría de lenguajes tradicionales que conocemos (Python, R, Java…).

2.2.1.1.1 Python

Se ha utilizado este intérprete con lenguaje Python para desarrollar las versiones no distribuidas de los algoritmos. Python se ha escogido dado que se trata de un lenguaje de propósito general, con un elevado grado de uso, también en entornos analíticos o *Data Science* y con un grado de madurez suficiente como para disponer de un gran elenco de librerías para facilitar la labor analítica. El intérprete soporta a la vez programación de Python no distribuida y programación PySpark específica, que no se ha utilizado lógicamente para esta implementación.

2.2.1.1.2 Bash

Se ha utilizado este intérprete para poder codificar todos los pasos necesarios para disponer de las carpetas de trabajo, obtener los sets de datos y cualesquiera otras tareas a realizar

en el sistema de ficheros local o distribuido. Son pasos iniciales de configuración del entorno, para los que el uso de la línea de comandos es lo más adecuado.

2.2.1.1.3 Markdown

Este intérprete permite realizar un mínimo maquetado en el código. En particular separando las secciones del *Notebook*. Sus funcionalidades en general permiten el maquetado del documento, si bien en este caso, dado que los resultados se presentan en este documento, no se ha realizado la documentación en el propio *Notebook* como suele ser habitual.

2.2.1.2. Distribuido

Los intérpretes bajo este epígrafe permiten trabajar con entornos Big Data basados en Spark de forma nativa. Esto quiere decir que implementan una serie de rutinas de forma distribuida de una manera natural para el desarrollador, si bien es necesario tener algunas nociones de la tecnología subyacente para hacer un uso eficaz del paralelismo.

2.2.1.2.1 PySpark

Se ha utilizado este intérprete ya que permite desarrollar código distribuido de una forma natural al estar basado en lenguaje Python. Básicamente, permite trabajar con lenguaje Python y además exporta algunas de las funcionalidades de Spark mediante su API PySpark. No toda la funcionalidad del API nativa en Scala está expuesta, por lo que cabe esperar limitaciones en alguna funcionalidad.

2.2.1.2.2 Spark

Este intérprete expone el API original desarrollada en Scala, por lo que es necesario programar en Scala para poder trabajar con él. Como en el caso de PySpark, ofrece de manera nativa una implementación del paradigma funcional que presenta Scala pero en este caso y de la

mano de Spark, de forma distribuida. Es igualmente necesario conocer la tecnología subyacente a este paralelismo para poder utilizarlo correctamente.

2.2.2. Librerías

Los intérpretes trabajan sobre un lenguaje determinado como se ha visto, sin embargo, para poder desarrollar soluciones de aprendizaje automático resulta apropiado utilizar librerías que implementan interfaces o APIs de los algoritmos más comunes, presentando una interfaz con una serie de posibilidades de modificar los parámetros internos de ejecución de los algoritmos. Esto suele resultar suficiente a no ser que sea necesario disponer de una implementación particular del algoritmo a la que no se pueda llegar mediante el ajuste de los parámetros expuestos en su interfaz. En estos casos no queda otra opción que realizar la implementación completa.

2.2.2.1. No Distribuidas

Las librerías no distribuidas no soportan paralelización en entornos distribuidos, si bien pueden hacer uso de paralelismo local. No obstante, en el caso que nos ocupa y para sencillez de las pruebas, no se ha activado esta funcionalidad, asegurando que la ejecución no distribuida es *de facto* una ejecución en un solo núcleo.

2.2.2.1.1 Scikit-Learn

Es una de las librerías de aprendizaje automático más conocida del entorno Python junto con Statsmodels. Se ha escogido de forma preferente, ya que se orienta más a aprendizaje automático contemporáneo, mientras que Statsmodels tiene un enfoque más orientado a la modelización estadística y contraste de hipótesis.

La librería expone un elevado número de APIs para trabajar con diferentes algoritmos de aprendizaje automático y goza de elevado grado de madurez.

2.2.2.2. Distribuidas

Las librerías distribuidas son librerías que proporcionan una interfaz conocida al desarrollador, que recoge los parámetros habituales de modificación que exponen las interfaces no distribuidas de los mismos algoritmos, si bien la implementación que realizan por debajo queda oblicua al propio desarrollador, que no necesita preocuparse por los detalles de la implementación y de cómo gestiona la librería el paralelismo. Lógicamente, estas librerías están pensadas para trabajar en un entorno concreto con un paradigma de paralelización o un entorno distribuido definido.

2.2.2.2.1 MLlib

Es la librería por excelencia para trabajar con algoritmos de aprendizaje automático sobre tecnología Spark. Permite el desarrollo sobre PySpark, (Scala) Spark y JavaSpark, si bien presenta APIs en distinto grado de madurez para cada uno de los intérpretes, siendo la versión Scala la más completa.

3. Aprendizaje Automático: Actores

Para llevar a cabo el conjunto de pruebas de las que es objeto este trabajo, una vez contamos con un entorno provisto de componentes hardware y de herramientas software, conviene dar un repaso a los protagonistas de la implementación de técnicas de aprendizaje automático visto por los 2 actores principales; los algoritmos y los datos a utilizar.

3.1. Algoritmos

Un factor importante en la elección de las librerías comentadas para el caso distribuido y el no distribuido, es que en ambos casos exponen con APIs muy parecidas acceso a la implementación de los algoritmos más habituales del aprendizaje automático. Estamos hablando de algoritmos de aprendizaje supervisado (clasificación, regresión), no supervisado (*clustering*), filtros colaborativos o reducción de dimensionalidad, entre otros. En el caso concreto de este trabajo se han seleccionado tres algoritmos particulares, buscando representar estos grupos mencionados de la mejor forma posible. Estos grupos mencionados son representativos de distintas técnicas de aprendizaje automático, y frecuentemente se utilizan para comparar el rendimiento de distintas tecnologías de procesamiento.

3.1.1. Regresión Logística

La regresión logística es un algoritmo de aprendizaje supervisado que busca determinar a partir de un conjunto de características o *features* el valor de una variable concreta que toma un valor binario o perteneciente a un conjunto. Se trata por tanto de un algoritmo utilizado para clasificar los datos y determinar si pertenece a una categoría determinada o no (o a una entre varias).

Para utilizar este algoritmo, se suelen identificar dos etapas diferenciadas. En la primera, se entrena un modelo a partir de un conjunto etiquetado (que contiene la variable objetivo) de

datos. Esto permite construir un modelo conforme a estos datos. El modelo debe ser lo bastante general como para no sobreajustar los datos de entrenamiento y lo bastante particular como para ofrecer un modelo que representa la curva característica de comportamiento con la que predecir resultados.

En la segunda etapa, se alimenta al modelo de datos nuevos que no conoce y se ve cómo de fieles son las predicciones frente al valor real etiquetado de la categoría en este conjunto de datos.

3.1.2. Clustering (KMeans)

KMeans es un algoritmo de aprendizaje no supervisado que permite inferir a partir de alguna métrica de distancia agrupaciones en los datos. Para ello, se escoge un número de agrupaciones que orbitan en torno a un centroide y el algoritmo va calculando el mejor valor para cada centroide, tomando como objetivo el minimizar la distancia de todos los puntos de un mismo clúster al centroide y maximizar la distancia entre diferentes centroides.

Para utilizar este algoritmo, solo es necesario presentar datos con características sin etiquetar, ya que no es un algoritmo supervisado. En cualquier caso, se construye un modelo a partir del conjunto de datos usado en entrenamiento y se puede medir la bondad de los centroides para este mismo conjunto. Esta evaluación se expresa en términos de la suma de errores cuadráticos de predicción o SSE y permite evaluar cuan bueno es el modelo construido sobre los propios datos de entrenamiento (se podrían usar igualmente datos diferentes para generalizar el modelo).

3.1.3. Análisis de Componentes Principales (PCA)

Este algoritmo permite abordar la reducción de dimensionalidad del número de características o *features* de un modelo. El objetivo es buscar un modelo que retenga el máximo de información, pero sea más sencillo de entender.

Para utilizar este algoritmo es necesario construir un modelo a partir de las características de los datos de entrenamiento. Luego se procesan las características, tanto de los datos de entrenamiento como los datos de test, de forma que se puedan construir representaciones simplificadas de estos sets de datos.

Finalmente, para poder evaluar el algoritmo, se pueden construir sendos modelos de regresión logística con la versión original de los datos y esta versión simplificada y evaluar ambos modelos con el conjunto de datos del test original y el simplificado. Se debería apreciar una pérdida de precisión, si bien debería ser lo menor posible si el número de características escogido es suficiente.

3.2. Datos

Si bien los algoritmos representan el qué queremos hacer, es extremadamente importante escoger con cuidado el conjunto de datos sobre el que aplicar estos algoritmos. Hay varios parámetros a los que hay prestar atención.

3.2.1. Tamaño

A la hora de poder medir el rendimiento de los algoritmos paralelizados en entornos Big Data, es necesario aproximarse a un tamaño de datos grande. Sin embargo, hay un límite al volumen de datos que es posible tratar de forma razonable con algoritmos no distribuidos, ya que el comportamiento de estos algoritmos mantiene los datos en memoria, lo que supone agotar rápidamente la RAM del ordenador y depender de la paginación, lo que invalida rápidamente los

resultados al pasar a ser determinante este acceso a disco por encima de las capacidades del procesador.

Por ello, se ha seleccionado un volumen que, siendo grande, no sobrepase las capacidades de memoria existentes. Este volumen orbita en torno a 1 GB de datos en formato CSV. Al tratar estos datos el espacio que ocupan en memoria es mayor, pues se representan en objetos que añaden complejidad para minimizar el tiempo de procesamiento.

3.2.2. Dimensiones

El conjunto de datos seleccionado debe tener unas dimensiones que aporten sentido a las técnicas PCA. Por ello, se ha buscado un conjunto de datos con alrededor de 40 características o *features*. Además, se busca fiabilidad en los modelos de clasificación, por lo que el conjunto de datos de entrenamiento debe tener en torno al millón de registros, un valor a priori suficiente para hablar de Big Data.

3.2.3. Idoneidad

Resulta esencial disponer de conjuntos de datos apropiados para cada uno de las técnicas que se quieren aplicar. Esto implica disponer de un conjunto de datos etiquetado con una variable objetivo (para poder realizar aprendizaje supervisado). Además, esta variable objetivo debe poder ser tratada para poder aplicar un clasificador (regresión logística) sobre los datos. También es necesario que los datos sean proclives a técnicas simples de clusterización utilizando una distancia euclídea.

Finalmente, resulta conveniente que los datos requieran del menor preprocesamiento posible. El objetivo del trabajo no es realmente buscar el mejor modelo sobre los datos, sino aplicar las técnicas algorítmicas descritas con diferentes implementaciones basadas en paradigmas diferentes y comparar los resultados en términos de tiempo de ejecución entre otros.

Los resultados deben ser comparables, por ello sí resulta adecuado tener una medida de evaluación del modelo obtenido para considerar homologable el esfuerzo invertido en cada implementación.

Con todo lo expuesto, la decisión final ha sido escoger el conjunto de datos público KDD 99.

Este conjunto de datos, utilizado para la tercera edición de la célebre competición internacional KDD, presenta datos de tramas de red etiquetadas indicando si se trata de mensajes normales o mensajes de ataque (y el tipo de ataque).

Lo interesante de cara al trabajo es que se presentan dos *datasets* para entrenamiento - uno reducido de 75 MB, otro completo de 743 MB - y un *dataset* adicional de test. Cada uno de ellos cuenta con 40 características, siendo todas las características menos tres numéricas (lo que reduce el preprocesamiento ya que ignoraremos estas tres variables). Todos los conjuntos incluyen la variable objetivo que es categórica, si bien puede ser tratada con facilidad para convertirla en binaria (separando tramas normales o de ataque, ya que ignoramos el tipo de ataque). En cuanto a volumetría, el conjunto menor de entrenamiento cuenta con ~500 k entradas, el mayor con ~ 5 m de entradas y el de test unas ~ 300 k entradas.

Cumple por tanto el conjunto de datos con los requisitos necesarios para poder servir de combustible a los algoritmos.

4. Codificación

Llegados a este punto, tenemos el entorno listo para el trabajo y una hoja de ruta concreta de lo que queremos hacer y sobre qué datos tenemos que hacerlo. Es necesario ahora dar los pasos para codificar el código del que es en realidad objeto todo el trabajo.

Sin embargo, debemos orientar este código diferenciando dos ámbitos diferentes. Por un lado, tenemos una parte común que preprocesa los datos y por el otro, el código de las implementaciones algorítmicas en sí.

Gracias a la flexibilidad que proporciona Zeppelin, podemos tener en un mismo documento o *Notebook* el código de estos distintos ámbitos separado en diferentes celdas debidamente etiquetadas e independientes entre sí.

4.1. Código común

Bajo este epígrafe es pertinente incluir dos ámbitos de código diferentes, el que prepara el entorno desde un punto de vista de ficheros y carpetas y el que carga los datos desde los ficheros adecuados en los objetos utilizados para alimentar a los algoritmos.

4.1.1. Preparación del entorno

Es necesario realizar un conjunto de operaciones a nivel de línea de comandos de Linux para poder obtener los conjuntos de datos comprimidos y descomprimirlos en local. Adicionalmente, se suben comprimidos al espacio de ficheros HDFS, ya que se puede operar con ellos directamente comprimidos sin problema en el entorno distribuido (en el entorno local ha generado problemas tratarlos comprimidos de forma directa y por eso se ha optado por descomprimirlos).

Por el camino, se crean una serie de carpetas en FS y HDFS poniendo especial énfasis en que aniden de las carpetas padre. Todas las rutas y sentencias se han construido en torno a unas constantes que faciliten modificar en un único punto el código si es necesario.

Como se mencionó previamente, el conjunto de datos se descompone en tres *datasets* diferentes. El *dataset* de entrenamiento pequeño utilizado para las pruebas iniciales del sistema, pero que a todos los efectos podemos ignorar, pues no será el utilizado para las pruebas reales sobre algoritmos. El *dataset* de entrenamiento grande que utilizaremos como conjunto de datos de entrenamiento. Y el *dataset* de test, de menor tamaño, utilizado para validar algunos de los modelos implementados.

4.1.1.1. Bash

Bash FINISHED

```
%sh

# Constants
LOCAL_FOLDER="/home/zeppelin/tfm"
HDFS_FOLDER="/user/zeppelin/tfm"

SMALL_TRAIN_DATASET_URL="http://kdd.ics.uci.edu/databases/kddcup99"
BIG_TRAIN_DATASET_URL="http://kdd.ics.uci.edu/databases/kddcup99"
TEST_DATASET_URL="http://kdd.ics.uci.edu/databases/kddcup99"

SMALL_TRAIN_DATASET_FILE="kddcup.data_10_percent"
BIG_TRAIN_DATASET_FILE="kddcup.data"
TEST_DATASET_FILE="corrected"

GZIP=".gz"

# Folder Setup
mkdir -p $LOCAL_FOLDER
hdfs dfs -mkdir -p $HDFS_FOLDER

# Dataset Gathering
## Small Train Dataset
#rm -f $LOCAL_FOLDER"/"$SMALL_TRAIN_DATASET_FILE
wget -P $LOCAL_FOLDER -q $SMALL_TRAIN_DATASET_URL"/"$SMALL_TRAIN_DATASET_FILE$GZIP
hdfs dfs -put -f $LOCAL_FOLDER"/"$SMALL_TRAIN_DATASET_FILE$GZIP $HDFS_FOLDER
gzip -fd $LOCAL_FOLDER"/"$SMALL_TRAIN_DATASET_FILE$GZIP

## Big Train Dataset
#rm -f $LOCAL_FOLDER"/"$BIG_TRAIN_DATASET_FILE
wget -P $LOCAL_FOLDER -q $BIG_TRAIN_DATASET_URL"/"$BIG_TRAIN_DATASET_FILE$GZIP
hdfs dfs -put -f $LOCAL_FOLDER"/"$BIG_TRAIN_DATASET_FILE$GZIP $HDFS_FOLDER
gzip -fd $LOCAL_FOLDER"/"$BIG_TRAIN_DATASET_FILE$GZIP

## Test Dataset
#rm -f $LOCAL_FOLDER"/"$TEST_DATASET_FILE
wget -P $LOCAL_FOLDER -q $TEST_DATASET_URL"/"$TEST_DATASET_FILE$GZIP
hdfs dfs -put -f $LOCAL_FOLDER"/"$TEST_DATASET_FILE$GZIP $HDFS_FOLDER
gzip -fd $LOCAL_FOLDER"/"$TEST_DATASET_FILE$GZIP
```

Took 24 seconds. (outdated)

4.1.2. Preparación de los datos

Una vez que disponemos de los conjuntos de datos cargados en el espacio de ficheros tanto local como en el clúster (en una estructura de carpetas adecuada), vamos a cargar los datos en variables en memoria para cada lenguaje. En el caso de Python se utilizarán *dataframes* Pandas para almacenar los objetos de trabajo. En el caso de PySpark y Spark, se cargarán en un objeto conocido como una unidad distribuida de datos RDD o *Resilient Distributed Dataset* (conjunto de datos persistente distribuido).

Adicionalmente, se va a realizar un preprocesado de los datos para prepararlos para su interacción con los motores analíticos. Por un lado, se eliminan del conjunto de características aquellas que no son numéricas (posiciones 1,2 y 3). Además, se trata la variable objetivo (41) para transformarla de categórica a binaria, siendo 0.0 el valor escogido para representar las tramas normales y 1.0 las que representan un ataque.

Como medida de control, se imprime en la salida el número de registros de cada *dataset* cargado en memoria para poder cerciorarnos de que es idéntico en todos los escenarios.

4.1.2.1. Python

Python FINISHED

```
%pyspark
import pandas as pd

#Constants
LOCAL_FOLDER="/home/zeppelin/tfm"

SMALL_TRAIN_DATASET_FILE="kddcup.data_10_percent"
BIG_TRAIN_DATASET_FILE="kddcup.data"
TEST_DATASET_FILE="corrected"

# Variables
all = range(0, 42)
used = all[0:1]+all[4:41]

# Dataset Loading
## Small Train Dataset
smallTrainFeatures = pd.read_csv(LOCAL_FOLDER+"/"+SMALL_TRAIN_DATASET_FILE, header=None, usecols = used,  dtype = 'float64')
smallTrainLabel = pd.read_csv(LOCAL_FOLDER+"/"+SMALL_TRAIN_DATASET_FILE, header=None, usecols = [41])
print "Small Train data size is {0}".format(smallTrainLabel.size)

## Big Train Dataset
bigTrainFeatures = pd.read_csv(LOCAL_FOLDER+"/"+BIG_TRAIN_DATASET_FILE, header=None, usecols = used,  dtype = 'float64')
bigTrainLabel = pd.read_csv(LOCAL_FOLDER+"/"+BIG_TRAIN_DATASET_FILE, header=None, usecols = [41])
print "Big Train data size is {0}".format(bigTrainLabel.size)

## Test Dataset
testFeatures = pd.read_csv(LOCAL_FOLDER+"/"+TEST_DATASET_FILE, header=None, usecols = used,  dtype = 'float64')
testLabel = pd.read_csv(LOCAL_FOLDER+"/"+TEST_DATASET_FILE, header=None, usecols = [41])
print "Test data size is {0}".format(testLabel.size)

# Dataset Refining
## Small Train Dataset
smallTrainLabel[41] = smallTrainLabel[41].apply(lambda x: 0.0 if x == 'normal.' else 1.0).astype(float)

## Big Train Dataset
bigTrainLabel[41] = bigTrainLabel[41].apply(lambda x: 0.0 if x == 'normal.' else 1.0).astype(float)

## Test Dataset
testLabel[41] = testLabel[41].apply(lambda x: 0.0 if x == 'normal.' else 1.0).astype(float)
```

```
Small Train data size is 494021
Big Train data size is 4898431
Test data size is 311029
```

Took 24 seconds.

En el caso de la implementación pandas, se ha optado por separar en dos objetos pandas diferentes las características (Xs) y la variable objetivo (y). Se puede ver que las características se rotulan con el postfijo Features y la variable objetivo con el postfijo Label.

4.1.2.2. PySpark

PySpark FINISHED

```
%pyspark

#Constants
HDFS_FOLDER="/user/zeppelin/tfm"

SMALL_TRAIN_DATASET_FILE="kddcup.data_10_percent"
BIG_TRAIN_DATASET_FILE="kddcup.data"
TEST_DATASET_FILE="corrected"

GZIP=".gz"

# Dataset Loading
## Small Train Dataset
smallTrain = sc.textFile(HDFS_FOLDER+"/"+SMALL_TRAIN_DATASET_FILE+GZIP)
print "Small Train data size is {0}".format(smallTrain.count())

## Big Train Dataset
bigTrain = sc.textFile(HDFS_FOLDER+"/"+BIG_TRAIN_DATASET_FILE+GZIP)
print "Big Train data size is {0}".format(bigTrain.count())

## Test Dataset
test = sc.textFile(HDFS_FOLDER+"/"+TEST_DATASET_FILE+GZIP)
print "Test data size is {0}".format(test.count())

# Dataset Refining
from pyspark.mllib.regression import LabeledPoint
from numpy import array

def parse_interaction(line):
    line_split = line.split(",")
    # leave_out = [1,2,3,41]
    clean_line_split = line_split[0:1]+line_split[4:41]
    attack = 1.0
    if line_split[41]=='normal.':
        attack = 0.0
    return LabeledPoint(attack, array([float(x) for x in clean_line_split]))

def parse_interaction_cluster(line):
    line_split = line.split(",")
    # leave_out = [1,2,3,41]
    clean_line_split = line_split[0:1]+line_split[4:41]
    return array([float(x) for x in clean_line_split])

## Small Train Dataset
smallTrainRefined = smallTrain.map(parse_interaction)
smallTrainRefinedCluster = smallTrain.map(parse_interaction_cluster)

## Big Train Dataset
bigTrainRefined = bigTrain.map(parse_interaction)
bigTrainRefinedCluster = bigTrain.map(parse_interaction_cluster)

## Test Dataset
testRefined = test.map(parse_interaction)
testRefinedCluster = test.map(parse_interaction_cluster)
```

```
Small Train data size is 494021
Big Train data size is 4898431
Test data size is 311029
```

Took 16 seconds. (outdated)

En el caso de PySpark, seguimos utilizando la familiar sintaxis de Python como lenguaje, si bien el tratamiento exige la codificación de código adicional. En este caso, los algoritmos esperan que construyamos un LabeledPoint para la regresión logística y un array de puntos para la clusterización con KMeans. El LabeledPoint no deja de ser una dupla de float y array de float

(la “y” y las “Xs”). El array de puntos son solo las Xs (recordemos que KMeans es un algoritmo no supervisado).

4.1.2.3. Spark

Spark

FINISHED

```
{
//Constants
val HDFS_FOLDER="/user/zeppelin/tfm"

val SMALL_TRAIN_DATASET_FILE="kddcup.data_10_percent"
val BIG_TRAIN_DATASET_FILE="kddcup.data"
val TEST_DATASET_FILE="corrected"

val GZIP=".gz"

//Dataset Loading
////Small Train Dataset
val smallTrain = sc.textFile(HDFS_FOLDER+"/"+SMALL_TRAIN_DATASET_FILE+GZIP)
println("Small Train data size is " + smallTrain.count())

////Big Train Dataset
val bigTrain = sc.textFile(HDFS_FOLDER+"/"+BIG_TRAIN_DATASET_FILE+GZIP)
println("Big Train data size is " + bigTrain.count())

////Test Dataset
val test = sc.textFile(HDFS_FOLDER+"/"+TEST_DATASET_FILE+GZIP)
println("Test data size is " + test.count())

//Dataset Refining
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint

def labelAttack(v:String):Double={
    if (v == "normal.")
        return 0.0
    1.0
}

def parse_interaction(line:String):LabeledPoint={
    val parts = line.split(",")
    LabeledPoint( labelAttack(parts(41)), Vectors.dense (parts.zipWithIndex.filter(_._2 != 1).filter(_._2 != 2).filter(_._2 != 3
).filter(_._2 != 41).map(_._1).map(_.toDouble)) )
}

////Small Train Dataset
val smallTrainRefined = smallTrain.map(parse_interaction)
val smallTrainRefinedCluster = smallTrain.map(s => Vectors.dense(s.split(',').zipWithIndex.filter(_._2 != 1).filter(_._2 != 2
).filter(_._2 != 3).filter(_._2 != 41).map(_._1).map(_.toDouble))).cache()

////Big Train Dataset
val bigTrainRefined = bigTrain.map(parse_interaction)
val bigTrainRefinedCluster = bigTrain.map(s => Vectors.dense(s.split(',').zipWithIndex.filter(_._2 != 1).filter(_._2 != 2
).filter(_._2 != 3).filter(_._2 != 41).map(_._1).map(_.toDouble))).cache()
```

```
////Test Dataset
val testRefined = test.map(parse_interaction)
val testRefinedCluster = test.map(s => Vectors.dense(s.split(',').zipWithIndex.filter(_._2 != 1).filter(_._2 != 2).filter(_._2 !
= 3).filter(_._2 != 41).map(_._1).map(_.toDouble))).cache()
}
```

```
Small Train data size is 494021
Big Train data size is 4898431
Test data size is 311029
```

Took 4 seconds. (outdated)

Con Spark lo primero que llama la atención es la sintaxis Scala, muy compacta y difícil de seguir en algunos puntos. Para la interacción con los algoritmos, se utiliza un LabeledPoint que en este caso es una dupla de float y Vector, un tipo de dato que define distinta

implementación en función de si se trata de un Vector *dense* o *sparse*. Para el caso de KMeans se accederá directamente al segundo miembro de la dupla que son las Xs.

4.2. Código algorítmico

En este apartado se tratan los diferentes algoritmos implementados para cada uno de los 3 lenguajes/intérpretes. Se va a estructurar por cada algoritmo, haciendo una breve introducción del mismo en consonancia con el código implementado. Luego, como en el caso anterior se harán apreciaciones de la implementación en cada lenguaje/intérprete.

Se ha tratado de etiquetar de forma diferenciada mediante comentarios en el código los dos bloques característicos, construcción del modelo y evaluación del modelo. Se van a medir tiempos en ambos casos, complementando esta información con el grado de precisión del modelo alcanzado a fin de poder extrapolar los valores y concluir que los modelos construidos son semejantes y por tanto comparables.

El entrenamiento de los modelos siempre se hará a partir del gran conjunto de datos de entrenamiento. La evaluación se hará con los datos de test, salvo para el algoritmo de KMeans, donde se usa una medida interna de error sobre los propios datos de entrenamiento.

4.2.1. Regresión logística

Para construir un modelo basado en regresión logística primero debemos entrenar el modelo a partir de los datos de entrenamiento. Sobre este modelo, realizaremos la evaluación tanto con el conjunto de entrenamiento como con el de test. Cabe esperar un valor de precisión elevado, mayor para los datos de entrenamiento que para los datos de test.

4.2.1.1. Python

Python FINISHED

```
%pyspark
from sklearn import linear_model
from time import time

#Model Building
t0 = time()
logreg = linear_model.LogisticRegression(solver='lbfgs')
logreg.fit(bigTrainFeatures, bigTrainLabel[41].values)
tt = time() - t0

print "Classifier trained in {0} seconds".format(round(tt,3))

#Model Testing
t0 = time()
train_accuracy = logreg.score(bigTrainFeatures, bigTrainLabel[41].values)
test_accuracy = logreg.score(testFeatures, testLabel[41].values)
tt = time() - t0

print "Prediction made in {0} seconds. Train accuracy is {1}. Test accuracy is {2}"\
.format(round(tt,3), round(train_accuracy,4), round(test_accuracy,4))
```

```
Classifier trained in 108.861 seconds
Prediction made in 0.499 seconds. Train accuracy is 0.9898. Test accuracy is 0.8823
```

Took 109 seconds.

Para la implementación Python, cabe destacar que hemos usado la clase LogisticRegression del módulo linear_model de scikit-learn. Se ha realizado la ejecución con los parámetros por defecto, salvo para indicar que se usará ‘lbfgs’. Esto entre otras cosas, implica que se realiza regularización de las características de forma automática. Se ha utilizado dado que la implementación distribuida la emplea por defecto, y al no usarse se apreciaban valores inferiores en la precisión sobre los datos de test.

4.2.1.2. PySpark

PySpark FINISHED

```
%pyspark
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from time import time

#Model Building
t0 = time()
logit_model = LogisticRegressionWithLBFGS.train(bigTrainRefined)
tt = time() - t0

print "Classifier trained in {0} seconds".format(round(tt,3))

#Model Testing
t0 = time()
labels_and_preds_train = bigTrainRefined.map(lambda p: (p.label, logit_model.predict(p.features)))
train_accuracy = labels_and_preds_train.filter(lambda (v, p): v == p).count() / float(bigTrainRefined.count())

labels_and_preds_test = testRefined.map(lambda p: (p.label, logit_model.predict(p.features)))
test_accuracy = labels_and_preds_test.filter(lambda (v, p): v == p).count() / float(testRefined.count())
tt = time() - t0

print "Prediction made in {0} seconds. Train accuracy is {1}. Test accuracy is {2}"\
.format(round(tt,3), round(train_accuracy,4), round(test_accuracy,4))
```

```
Classifier trained in 192.695 seconds
Prediction made in 288.556 seconds. Train accuracy is 0.9942. Test accuracy is 0.9164
```

Took 482 seconds. (outdated)

La implementación PySpark hace uso de la clase LogisticRegressionWithLBFGS, que como indicábamos anteriormente aplica regularización por defecto. Esta clase se localiza dentro de los algoritmos de clasificación de la librería MLlib.

La construcción del modelo es equivalente a la implementación mediante sklearn. Sin embargo, se puede apreciar como para evaluar el modelo es necesario implementar un cálculo complejo, símbolo de que la librería no posee de una funcionalidad análoga al método score de sklearn.

4.2.1.3. Spark

Spark FINISHED

```
{
import org.apache.spark.mllib.classification.{LogisticRegressionWithLBFGS, LogisticRegressionModel}

//Model Building
val t0 = System.nanoTime()
val model = new LogisticRegressionWithLBFGS().run(bigTrainRefined)
val tt = System.nanoTime() - t0

println("Classifier trained in " + BigDecimal(tt/1000000000.0).setScale(3, BigDecimal.RoundingMode.HALF_UP).toDouble + " seconds")

//Model Testing
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.evaluation.MulticlassMetrics

val tt0 = System.nanoTime()
val trainPredictionAndLabels = bigTrainRefined.map { case LabeledPoint(label, features) =>
  val prediction = model.predict(features)
  (prediction, label)
}

val train_metrics = new MulticlassMetrics(trainPredictionAndLabels)
val train_accuracy = train_metrics.precision


val testPredictionAndLabels = testRefined.map { case LabeledPoint(label, features) =>
  val prediction = model.predict(features)
  (prediction, label)
}

val test_metrics = new MulticlassMetrics(testPredictionAndLabels)
val test_accuracy = test_metrics.precision
val ttt = System.nanoTime() - tt0

println("Prediction made in " + BigDecimal(ttt/1000000000.0).setScale(3, BigDecimal.RoundingMode.HALF_UP).toDouble
    + " seconds. Train accuracy is " + BigDecimal(train_accuracy).setScale(4, BigDecimal.RoundingMode.HALF_UP).toDouble
    + ". Test accuracy is " + BigDecimal(test_accuracy).setScale(4, BigDecimal.RoundingMode.HALF_UP).toDouble)
}
```

```
Classifier trained in 170.476 seconds
Prediction made in 68.238 seconds. Train accuracy is 0.9965. Test accuracy is 0.8702
```

Took 239 seconds.

La implementación Spark es bastante diferente visualmente al estar codificada en lenguaje Scala. Sin embargo, tiene bastantes parecidos con la versión PySpark. El entrenamiento del modelo se realiza con la clase LogisticRegressionWithLBFGS, de forma análoga a PySpark. La principal diferencia es que para la evaluación se utiliza la clase MulticlassMetrics, perteneciente al paquete de *evaluation* dentro de MLlib. El uso de esta clase, si bien no llega a ser tan sencillo como en sklearn, por lo menos resulta más limpio.

Para mostrar los tiempos por pantalla es necesario elaborar un código bastante complejo para los redondeos en decimales, lo cual no deja de ser llamativo.

4.2.2. KMeans

Para el caso de clustering con KMeans, vamos a construir un modelo a partir de los datos de entrenamiento. A este modelo le vamos a especificar una serie de parámetros como el número de clústeres, el número de iteraciones, cuántas ejecuciones realizar o aleatoriedad al escoger la posición inicial de los centroides. Esto lo hacemos para asegurarnos de que estamos construyendo el mismo tipo de clúster, ya que la evaluación es algo menos intuitiva en este caso.

En efecto, para evaluar KMeans, calculamos la suma de cuadrados residual total entre cada punto y el centroide mas próximo, que es el valor que el algoritmo trata de minimizar. Esta métrica nos da una idea del trabajo que ha hecho el algoritmo en términos de encontrar los mejores centroides. Cabe esperar que el valor sea parecido en las distintas implementaciones (dentro del mismo orden de magnitud).

4.2.2.1. Python

Python

FINISHED

```
%pyspark
from sklearn.cluster import KMeans
from sklearn import metrics

#Model Building
t0 = time()
random_K_means = KMeans(init='random', n_clusters = 2, max_iter = 100, n_init = 10)
random_K_means.fit(bigTrainFeatures)
tt = time() - t0

print "Classifier trained in {0} seconds".format(round(tt,3))

#Model Evaluation
t0 = time()
wssse = random_K_means.inertia_
tt = time() - t0

print "Evaluation made in {0} seconds. WSSSE is {1}.".format(round(tt,3), round(wssse,4))
```

```
Classifier trained in 57.976 seconds
Evaluation made in 0.0 seconds. WSSSE is 3.05254895755e+18.
```

Took 58 seconds.

La implementación Python es bastante simple. Se ha utilizado la clase KMeans del módulo clúster de sklearn para construir y entrenar un modelo y posteriormente se obtiene el valor de error de forma directa, pues al construir el modelo ya se ha calculado.

4.2.2.2. PySpark

PySpark

FINISHED

```
%pyspark
from pyspark.mllib.clustering import KMeans
from time import time
from math import sqrt

#Model Building
t0 = time()
clusters = KMeans.train(bigTrainRefinedCluster, 2, maxIterations=100, runs=10, initializationMode="random")
tt = time() - t0

print "Classifier trained in {0} seconds".format(round(tt,3))

#Model Evaluation
def error(point):
    center = clusters.centers[clusters.predict(point)]
    return sum([x**2 for x in (point - center)])

t0 = time()
wssse = bigTrainRefinedCluster.map(lambda point: error(point)).reduce(lambda x, y: x + y)
tt = time() - t0

print "Evaluation made in {0} seconds. WSSSE is {1}.".format(round(tt,3), round(wssse,4))
```

```
Classifier trained in 258.317 seconds
Evaluation made in 246.779 seconds. WSSSE is 3.05254895751e+18.
```

Took 505 seconds.

Para la implementación PySpark, se hace uso de la clase KMeans del módulo clustering de MLlib. La construcción del modelo es simple, sin embargo, para la evaluación es necesario implementar un método que realice los cálculos a mano. El valor que se obtiene para el error es idéntico al obtenido en el caso de Python.

4.2.2.3. Spark

Spark

FINISHED

```
{
import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}

//Model Building
val k = 2
val maxIterations = 100
val runs = 10
val initializationMode = "random"

val t0 = System.nanoTime()
val clusters = KMeans.train(bigTrainRefinedCluster, k, maxIterations, runs, initializationMode)
val tt = System.nanoTime() - t0

println("Classifier trained in " + BigDecimal(tt/1000000000.0).setScale(3, BigDecimal.RoundingMode.HALF_UP).toDouble + " seconds")

//Model Testing
val tt0 = System.nanoTime()
val wssse = clusters.computeCost(bigTrainRefinedCluster)
val ttt = System.nanoTime() - tt0

println("Evaluation made in " + BigDecimal(ttt/1000000000.0).setScale(3, BigDecimal.RoundingMode.HALF_UP).toDouble
    + " seconds. WSSSE is " + BigDecimal(wssse).setScale(4, BigDecimal.RoundingMode.HALF_UP).toDouble)
}
```

```
Classifier trained in 67.423 seconds
Evaluation made in 1.04 seconds. WSSSE is 3.0525489575100616E18.
```

Took 68 seconds. (outdated)

La implementación Spark hace uso de las clases KMeans y KMeans Model del módulo clustering de MLlib. Podemos ver que la construcción del Cluster es análoga, sin embargo, la evaluación es mucho más sencilla que en el caso de PySpark gracias a la funcionalidad computeCost de la clase KMeansModel.

El valor de error obtenido es idéntico a los casos de Python y de PySpark.

4.2.3. PCA

Para la implementación de PCA se ha seguido una receta algo más compleja que en los casos anteriores. Primero se ha entrenado un modelo mediante los datos de entrenamiento. Después se ha aplicado este modelo al conjunto de entrenamiento y al de test para obtener dos nuevos conjuntos de dimensionalidad reducida. Se han construido dos modelos de regresión logística similares a los del primer apartado, uno entrenado con los datos originales y el otro con los datos de dimensionalidad reducida calculados sobre los datos de entrenamiento. Finalmente, se ha realizado la evaluación de estos modelos de nuevo uno con los datos de test originales y el otro con los datos de dimensionalidad reducida calculada sobre los datos de test.

De esta forma, se puede comprobar la precisión sobre los datos de test que es la métrica que utilizaremos para juzgar la homologabilidad de los resultados.

Cabe destacar que dada la madurez de la implementación de Spark utilizada a lo largo de las pruebas (versión 1.4.1), no se dispone de posibilidad de realizar esta implementación en PySpark, por lo que solo se cuenta con la versión Python y la Spark, que, sin embargo, deberían de ser suficiente objeto de estudio al ser la primera no distribuida y la segunda distribuida.

Se ha escogido un valor de 10 características como dimensión a utilizar por PCA. Recordemos que el conjunto de datos tras procesarlo tiene 38.

4.2.3.1. Python

Python FINISHED

```
%pyspark
from sklearn.decomposition import PCA

#Model Building
t0 = time()
pca = PCA(10).fit(bigTrainFeatures)
tt = time() - t0

print "Classifier trained in {0} seconds".format(round(tt,3))

#Model Appliance
t0 = time()
trainPcaFeatures = pca.transform(bigTrainFeatures)
testPcaFeatures = pca.transform(testFeatures)
tt = time() - t0

print "Classifier applied in {0} seconds".format(round(tt,3))

#Model Integration
from sklearn import linear_model

logreg = linear_model.LogisticRegression(solver='lbfgs')
logreg.fit(bigTrainFeatures, bigTrainLabel[41].values)

logreg_pca = linear_model.LogisticRegression(solver='lbfgs')
logreg_pca.fit(trainPcaFeatures, bigTrainLabel[41].values)

#Model Evaluation
t0 = time()
original_accuracy = logreg.score(testFeatures, testLabel[41].values)
pca_accuracy = logreg_pca.score(testPcaFeatures, testLabel[41].values)
tt = time() - t0

print "Evaluation made in {0} seconds. Original accuracy is {1}. PCA(10) accuracy is {2}".\
format(round(tt,3), round(original_accuracy,4), round(pca_accuracy,4))
```

```
Classifier trained in 12.965 seconds
Classifier applied in 1.443 seconds
Evaluation made in 0.042 seconds. Original accuracy is 0.8823. PCA(10) accuracy is 0.8069
```

Took 249 seconds. (outdated)

La implementación Python hace uso de la clase PCA de la librería *decomposition* de sklearn. El entrenamiento del modelo resulta bastante sencillo de codificar. Posteriormente, se utiliza la llamada a la función transform para reducir la dimensionalidad de los conjuntos de datos utilizando el modelo construido.

El resto del código ya es conocido al ser análogo al utilizado para el primer apartado.

4.2.3.2. Spark

Spark FINISHED

```
{
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature.PCA

//Model Building
val t0 = System.nanoTime()
val pca = new PCA(10).fit(bigTrainRefined.map(_.features))
val tt = System.nanoTime() - t0

println("Classifier trained in " + BigDecimal(tt/1000000000.0).setScale(3, BigDecimal.RoundingMode.HALF_UP).toDouble + " seconds")

//Model Appliance
val tt0 = System.nanoTime()
val trainPca = bigTrainRefined.map(p => p.copy(features = pca.transform(p.features)))
val testPca = testRefined.map(p => p.copy(features = pca.transform(p.features)))
val ttt = System.nanoTime() - tt0

println("Classifier applied in " + BigDecimal(ttt/1000000000.0).setScale(3, BigDecimal.RoundingMode.HALF_UP).toDouble + " seconds")

//Model Integration
import org.apache.spark.mllib.classification.{LogisticRegressionWithLBFGS, LogisticRegressionModel}

val model = new LogisticRegressionWithLBFGS().run(bigTrainRefined)
val model_pca = new LogisticRegressionWithLBFGS().run(trainPca)

//Model Evaluation
import org.apache.spark.mllib.evaluation.MulticlassMetrics

val ttt0 = System.nanoTime()

val originalValuesAndPreds = testRefined.map { point =>
  val score = model.predict(point.features)
  (score, point.label)
}
val originalMetrics = new MulticlassMetrics(originalValuesAndPreds)
val original_accuracy = originalMetrics.precision


val pcaValuesAndPreds = testPca.map { point =>
  val score = model_pca.predict(point.features)
  (score, point.label)
}
val pcaMetrics = new MulticlassMetrics(pcaValuesAndPreds)
val pca_accuracy = pcaMetrics.precision

val tttt = System.nanoTime() - ttt0

println("Prediction made in " + BigDecimal(tttt/1000000000.0).setScale(3, BigDecimal.RoundingMode.HALF_UP).toDouble
    + " seconds. Original accuracy is " + BigDecimal(original_accuracy).setScale(4, BigDecimal.RoundingMode.HALF_UP).toDouble
    + ". PCA(10) accuracy is " + BigDecimal(pca_accuracy).setScale(4, BigDecimal.RoundingMode.HALF_UP).toDouble)
}
```

```
Classifier trained in 70.482 seconds
Classifier applied in 0.003 seconds
Prediction made in 8.472 seconds. Original accuracy is 0.8702. PCA(10) accuracy is 0.8078
```

Took 453 seconds.

La implementación Spark hace uso de la clase PCA perteneciente al módulo *feature* de MLlib. El entrenamiento del modelo es igual de sencillo que en el caso Python, si bien para transformar el conjunto de datos mediante este es necesario lidiar con algunos aspectos del objeto LabeledPoint para solo convertir las Xs. Como en el caso Python se proporciona una función transform.

Los resultados obtenidos para la precisión son prácticamente idénticos a los obtenidos con Python, lo que avala que los modelos generados son correctos. Además, se aprecia una pérdida de precisión sobre los datos de test de dimensionalidad reducida, algo lógica al contar con solo 10 *features* y por tanto ser un modelo más sencillo.

5. Resultados

Si bien se han realizado algunos comentarios sobre la implementación en el apartado anterior, es hora de presentar los tiempos medidos en cada configuración, así como de comentar localmente los resultados de cada algoritmo y razonar algunos de los factores observados que expliquen y amplíen los valores recogidos.

La metodología para medir los tiempos ha consistido en realizar el reinicio completo de la máquina virtual para cada nuevo lenguaje y cada nuevo algoritmo. De esta forma se evitaba el tener demasiados objetos cargados en menoría con el peligro de paginar a memoria secundaria.

Además, es pertinente recordar que las pruebas se han lanzado desde Zeppelin, indicando en la configuración del intérprete que se hiciera uso de 2 instancias o contenedores de 2 núcleos cada una para el entorno distribuido. La memoria por ejecutor se ha establecido en 4 GB. Se pretende con esta configuración aproximar el comportamiento distribuido al haber más de un contenedor (unidad de ejecución) en funcionamiento. Los trabajos son lanzados por Zeppelin mediante spark-client, siendo esta la forma más óptima para procesos interactivos como el código del Notebook.

Los tiempos se han medido en el propio código de ejecución, si bien Zeppelin de manera nativa muestra el código de ejecución de los contenidos de una celda. Esta funcionalidad se ha utilizado como una segunda comprobación, pero los resultados transcritos son los recogidos en el código explícitamente (y cuya implementación y salida se pudo ver en el apartado anterior).

5.1. Código común

Tabla 1	Bash	DS
Python	26,000	46,000
PySpark		45,000
Spark		34,000

En la Tabla 1 se pueden apreciar los tiempos medidos en segundos con tres decimales de precisión para la ejecución del código común. La ejecución del código Bash que permite la obtención de los datasets y la preparación del entorno solo se realiza una vez.

El tiempo de carga de los datos en memoria y preprocesamiento se mide de forma independiente para cada lenguaje, encontrándose que es más rápido para el caso de Scala a pesar de ser distribuido que en el caso no distribuido de Python. PySpark se mueve en el entorno de tiempos de Python.

En cualquier caso, las diferencias no son exageradas, pero parecen desmentir la premisa de la que se partía de pensar que la carga de datos en una variable distribuida fuera necesariamente más lenta que en el caso no distribuido.

Cabe destacar, que los objetos pandas no fueron pensados para gestionar datasets tan grandes, y quizás no hagan uso de posibles optimizaciones orientadas a este tipo de escenarios.

5.2. Código algorítmico

Para comentar los resultados de la ejecución de los diferentes algoritmos, vamos a detenernos en cada uno de ellos con los datos medidos de tiempo de ejecución y la medida de evaluación del algoritmo (precisión, error…).

Para la regresión lineal (LR) se ha medido el tiempo de construcción del modelo (MB) y el de evaluación (MT). Para la clusterización con KMeans (KM) se ha medido el tiempo de ejecución para construir el modelo (MB) y el tiempo para calcular la suma de errores cuadrática (MT). Para la reducción de la dimensionalidad con PCA (PCA) se ha medido el tiempo para construir el modelo (MB), para transformar los conjuntos de datos (MA) y para evaluar la regresión logística construida con los distintos conjuntos (ME).

Todos los resultados de tiempos se expresan en segundos con 3 decimales de precisión. Las medidas de precisión se expresan con 4 decimales. El error en la cluterizacion se representa con 2 decimales.

5.2.1. Regresión Logística

Tabla 2	LR MB	LR MT	LR Train	LR Test
Python	109,262	0,480	0,9898	0,8823
PySpark	175,468	272,023	0,9942	0,9164
Spark	158,125	67,596	0,9965	0,8702

La tabla 2 recoge los resultados obtenidos para los tiempos medidos y la precisión obtenida. La precisión medida sobre los datos de entrenamiento y los datos de test, es comparable en los tres lenguajes, siendo algo mayor para el caso de PySpark en el conjunto de test. Esto nos permite validar que el trabajo realizado por los algoritmos en cada lenguaje para construir el modelo es similar.

A la hora de evaluar los tiempos que ha empleado cada implementación en construir el modelo, se constata con sorpresa que los tiempos de Python son menores al resto de implementaciones por un buen margen. Además, PySpark resulta más lento que Spark.

Lo verdaderamente llamativo, es que la evaluación del modelo es instantánea en el caso de Python (sin duda ya ha calculado este valor al construir el modelo), ¡mientras que para PySpark (que recordemos no contaba con una función nativa para evaluación) resulta un trabajo de unos 270 segundos! Spark emplea unos 70 segundos en realizar la evaluación, resultando mucho más rápido que su homólogo distribuido PySpark pero sin embargo notablemente lento comparado con el virtual tiempo de 0 segundos de Python.

5.2.2. KMeans

Tabla 3	KM MB	KM MT	KM SSE
Python	57,976	0,000	3,05E+18

PySpark	175,838	238,545	3,05E+18
Spark	97,530	0,899	3,05E+18

La tabla 3 recoge los resultados obtenidos para el caso de KMenas. Se muestran los tiempos de construcción del modelo y evaluación, así como el error obtenido como suma de errores cuadráticos sobre el conjunto de entrenamiento.

A la luz de los valores de error, podemos llegar a la certeza de que KMeans ha utilizado los mismos centroides en todas las implementaciones, obteniendo los mismos valores de error.

A la hora de comparar los tiempos de ejecución para construir el modelo, es notable de nuevo como Python es más rápido que PySpark o Spark. En particular, PySpark es bastante más lento que su homólogo Spark, lo que confirma la tónica observada en el apartado anterior.

Con respecto a los tiempos de evaluación, de nuevo se aprecia como Python y en este caso también Spark, deben estar procesando esta información en la propia construcción del modelo, mientras que PySpark debe realizar todos los cálculos “a mano” mediante una implementación que quizás no sea la más óptima.

5.2.3. PCA

Tabla 4	PCA MB	PCA MA	PCA ME	Prec Orig	Prec PCA
Python	10,788	0,686	0,045	0,8823	0,8069
Spark	70,390	0,006	9,228	0,8702	0,8078

La tabla 4 recoge los tiempos medidos para construir un modelo PCA, transformar los *datasets* mediante este modelo, y evaluar la precisión de los modelos de regresión logística construidos con y sin PCA. Además, se puede ver la precisión alcanzada sobre los datos de test con los datos originales o con los datos reducidos a 10 características.

En este caso, cabe recordar que no se ha podido implementar una versión PySpark pues no se ofrece en esta iteración de Spark (1.4.1). Sí es posible utilizar Spark pues sí se ofrece la versión Scala.

Podemos ver, que los valores de precisión obtenidos son comparables tanto antes como después de aplicar PCA, por lo que se valida que el modelo se ha construido de forma correcta. En concreto, la precisión desciende, lo cual es lógico ya que el modelo tiene menos Xs.

En cuanto a los tiempos, la construcción del modelo es mucho más rápida (7x) en Python frente a Spark. Sin embargo, la conversión o transformación, siendo casi instantánea, es más rápida en Spark, lo cual da una primera idea de cuál es su fortaleza. Finalmente, la evaluación es extremadamente rápida en Python sin ser lenta en Spark.

Parece que, una vez más, Python sorprendentemente se impone en velocidad a Spark a pesar de que Python no haga uso de un entorno distribuido y Spark sí.

6. Conclusiones

Existe la premisa cuando trabajamos en entornos distribuidos, de asumir que el procesamiento paralelo es necesariamente más rápido que el secuencial. Es un punto de partida sensato el pensar que una tarea que se fragmente en subtareas más pequeñas ejecutadas concurrentemente llevará menos tiempo. Sin embargo, es igualmente cierto que esto obvia el propio mecanismo en sí que permite dicha paralización y el coste computacional y operacional que del mismo derive.

Un planteamiento similar sucede cuando abordamos el trabajo en un entorno Big Data. Se da por supuesto que los beneficios de la paralelización serán plenamente latentes desde un primer momento en cuanto se disponga de la mínima concurrencia posible. A la luz de los resultados obtenidos, esta premisa debe ser revisada.

6.1. Sobre entornos distribuidos y no distribuidos

La primera conclusión que se puede sacar de los resultados, es que los entornos distribuidos están pensados para escalar a decenas, cientos o miles de nodos. Además, los nodos suelen contener por sí mismos decenas de núcleos. Cuando hablamos de Big Data, hablamos de entornos clúster con una dimensión elevada donde todos estos nodos puedan ser aprovechados.

Sin lugar a dudas, un entorno distribuido “de salón” como el que se ha utilizado, pone de manifiesto que la distribución tiene un coste computacional elevado, máxime, cuando se trata de simular entornos distribuidos en un entorno físicamente no distribuido.

A la luz de los resultados, el sobrecoste de la gestión y mantenimiento de la miríada de servicios que acaban haciendo posible el trabajo en un entorno distribuido, da réditos en volúmenes de nodos mucho mayor del disponible para estas pruebas, un sobrecoste que acaba penalizando el rendimiento más de lo que lo ayuda en entornos pequeños.

Daría pie a una extensión de este trabajo el adentrarse en explorar el rendimiento de estos algoritmos en función de la dimensión del entorno. Sin embargo, esta exploración transciende la motivación original del trabajo de evaluar el impacto del Big Data en el analista de a pie.

6.2. Sobre la madurez de las librerías

Como ocurre en otros ámbitos de la informática y la vida en general, la experiencia es un grado. No todo se resuelve con fuerza bruta, sino que las optimizaciones de que se disponga hacen un gran recorrido en acelerar las ejecuciones. Los datos ponen de manifiesto que varios años de optimizaciones han dado como resultado scikit-learn como una librería madura, repleta de funcionalidad y muy optimizada. Si bien, es cierto que Python nunca fue desarrollado para trabajar con estos datos, y la gestión que hace Pandas al cargar los datos en memoria o al realizar transformaciones, ponen de manifiesto la naturaleza no funcional de Python.

Igual que scikit-learn se demuestra una herramienta madura, repleta de optimizaciones y funcionalidad, se vislumbra PySpark como una implementación inmadura, donde además el rendimiento es netamente inferior que en el caso de Spark. Spark recoge un buen número de funcionalidad frente a scikit-learn, pero está todavía un paso por detrás en funcionalidad.

6.3. Sobre PySpark y Spark

Es inevitable fijarse en como dos intérpretes basados en la misma tecnología pueden ofrecer unas diferencias tan grandes en el rendimiento. Sin lugar a dudas, PySpark paga tanto una implementación Spark más ineficiente, como la falta de métodos específicos para algunas funcionalidades que Spark sí tiene implementados. Aun así, en cálculos sobre librerías y métodos similares, se demuestra que PySpark es más lento que Spark.

6.4. Sobre Python y Scala

También resulta llamativo comprobar cómo la transición de Python a PySpark allí donde existen APIs similares resulta bastante natural y sencilla. Scala por el contrario, expone con facilidad la necesidad de integrar el paradigma funcional. Esto resulta cuando se domina en un código muy compacto y natural, pero la curva de aprendizaje es elevada. Además, algunas funcionalidades (como mostrar los números con determinada precisión), exigen de complejas conversiones, signo de que, como lenguaje, Scala también tiene todavía que recorrer bastante camino.

6.5. Sobre la interacción del desarrollo y la tecnología

Hay una parte importante de trabajo de investigación - prueba y error - que nunca consigue llegar al documento final. En este caso, el gran desconocido es el propio entorno distribuido en sí. Aunque se ha hablado del entorno y de la tecnología en líneas generales, el trabajo de customización necesario para poder por ejemplo cambiar los recursos disponibles en ejecución, no es ni de lejos sencillo. Hay una miríada de parámetros repartidos por distintas tecnologías, y todo tiene algo que opinar sobre la próxima ejecución.

Esto que podría ser un hecho aislado, cobra su máximo exponente a la hora de desarrollar el código. Es necesario conocer y profundizar en la forma en la que Spark trabaja, para obtener los máximos beneficios en la distribución de las tareas. Como ha ocurrido con otras tecnologías, quizás con el paso del tiempo, esta necesidad disminuya y sea el propio Spark el que decida la mejor manera de gestionar el paralelismo, pero en este punto en el desarrollo, lo cierto es que el analista no puede llegar y sentarse a desarrollar sin más; debe tener como mínimo algunas ideas de cómo funciona la tecnología que permite la distribución y cómo le impacta como desarrollador.

6.6. Sobre lo que queda pendiente

Este trabajo, ha permitido extraer algunas conclusiones útiles sobre entornos distribuidos, madurez de herramientas o las diferencias entre PySpark y Spark. Sin embargo, quedan algunas dudas sin despejar que podrían dar ciertamente para continuarlo.

Queda pendiente el poder establecer cómo escalan en tiempo los algoritmos cuando se aumenta los recursos físicos del clúster. Esto permitiría observar qué volumetría de nodos es necesaria para superar el rendimiento de Python. También sería interesante ver qué tipo de curva de escalabilidad se genera (lineal, logarítmica…) y a que valores.

También queda pendiente probar estos mismos algoritmos con *datasets* de mayor tamaño. Resultaría interesante explorar el efecto del tamaño en el rendimiento de los algoritmos distribuidos y no distribuidos, para constatar si Python finalmente se derrumba con *datasets* muy grandes.

Finalmente, sería interesante confeccionar una guía que recogiera los diferentes parámetros de configuración del clúster y permitiera establecer su valor de forma conjunta en función de los recursos físicos disponibles, de forma que el entorno distribuido aprovechara todos los recursos de forma eficiente. Esto ahora no ocurre, una lección en configuración y administración a través de todo un elenco de capas y tecnologías que además ha resultado fallida, imposibilitando el poder simular diferentes configuraciones físicas de clúster, aunque sigan siendo “de salón”.

6.7. Sobre el analista

El analista tiene una difícil decisión entre manos. Por un lado, tiene un conjunto de tecnología madura y probada que ya conoce y funciona. Y además funciona muy bien. Por el otro, una tecnología inmadura y ciertamente compleja, que implica además que se moleste en tratar con la infraestructura y en cómo esta funciona.

Sin embargo, esta nueva tecnología tiene una ventaja definitiva: escala. El código que programas para un nodo es el mismo que programas para mil. Por ello y teniendo muy presente las dificultades y que estamos todavía en una etapa temprana, quizás vaya siendo hora de que el analista desempolve ese viejo libro de texto que reza: “Entornos Distribuidos, Introducción”.