UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática

TFG del Grado en Ingeniería Informática

Paralelización de algoritmos de selección de instancias con la arquitectura Spark

Presentado por Alejandro González Rogel
en Universidad de Burgos — 4 de febrero de 2016
Tutor: Álvar Arnaiz González
Carlos López Nozal

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática

D. Álvar Arnaiz González, profesor del departamento de Ingeniería Civil, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Alejandro González Rogel, con DNI 71311632-V, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado Paralelización de algoritmos de selección de instancias con la arquitectura Spark .

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 4 de febrero de 2016

Vº. Bº. del Tutor:	Vº. Bº. del co-tutor:
D. Álvar Arnaiz González	D. Carlos López Nozal

Resumen

La enorme cantidad de datos a los que ahora tenemos acceso han generado una serie de problemas cuando queremos extraer y estudiar la información contenida en dichos datos. Por esta razón, recientemente se ha empezado a desarrollar y mejorar nuevas técnicas que permitan resolver el problema, dos de las cuales serán analizadas y desarrolladas a lo largo de esta memoria: la computación paralela y los algoritmos de selección de instancias.

Usando Apache Spark, el objetivo principal de este proyecto será la comparación entre técnicas secuenciales y paralelas de minería de datos y la implementación y evaluación de dos algoritmos concretos de selección de instancias: *Locality sensitive hashing instance selection* y *Democratic instance selection*.

Descriptores

Minería de datos, big data, computación paralela, selección de instancias, Apache Spark, Locality sensitive hashing instance selection, Democratic instance selection.

Abstract

The huge amount of data which we have access to, has led to new problems when we try to analyze the information contained in that data. Because of that, new techniques have come out trying to solve the problem. Through this paper I will discuss about some of these new solutions: parallelization and instance selection algorithms.

Using Apache Spark engine, the main goal of this project will be the comparison between sequential and parallel data mining techniques and the implementation and evaluation of two concrete instance selection algorithms: Locality sensitive hashing instance selection and Democratic instance selection.

Keywords

Data mining, big data, parallelization, instance selection, Apache Spark, Locality sensitive hashing instance selection, Democratic instance selection.

Índice general

Índice de figuras

Índice de cuadros

Introducción

La minería de datos es un área que se ha mantenido en intensa y constante evolución desde su aparición formal en los años 80 y 90. Desde el comienzo, y debido a esta continua evolución, el área ha estado siempre sujeta a cambios que tenían por objetivo buscar una solución a los problemas que la minería de datos iba planteando. En lo que se refiere a la etapa actual, uno de los problemas más importantes a los que se está haciendo frente es la gran cantidad de los datos y el creciente número de atributos de los mismos [17], que hacen imposible seguir aplicando aproximaciones anteriores por problemas de eficiencia.

A lo largo de esta memoria vamos a trabajar sobre algunos de los paradigmas que han surgido en el área para poder hacer frente a grandes volúmenes de datos: la paralelización de las tareas de minería de datos y la preselección de instancias para reducir el tamaño inicial del conjunto de datos y mejorar su calidad.

1.0.1. Estructura de la memoria

La memoria ha mantenido la estructura definida en primera instancia por los profesores del tribunal del TFG:

- **Objetivos del proyecto:** Donde se definirán cuáles serán las metas que hemos intentado alcanzar con la realización del trabajo.
- **Conceptos teóricos:** Donde se darán a conocer todos aquellos conceptos que, sin estar incluidos dentro del conocimiento básico, son necesarios para la comprensión del proyecto.
- **Técnicas y herramientas:** Donde se explicarán las metodologías y herramientas usadas para llevar a cabo el proyecto, junto con una justificación de las causas de su elección.

- **Aspectos relevantes del desarrollo del proyecto:** Donde se explicarán todos aquellos apartados que consideremos de interés durante la evolución del proyecto.
- **Trabajos relacionados:** Donde se dejará constancia de cualquier otro trabajo que se hubiese realizado sobre el área tratada en este proyecto.
- **Conclusiones y líneas futuras de trabajo:** Donde se dejará constancia de todo lo aprendido y extraído del proyecto, así como de diferentes posibilidades para continuar trabajando.

1.0.2. Materiales entregados

Junto con la memoria se hará entrega de los siguientes archivos:

- **Máquina virtual:** Se dejará a disposición del tribunal una imagen virtual de Ubuntu 14.04 que contendrá todos los materiales necesarios para probar el funcionamiento del proyecto.
- **Anexos:** Documentos adicionales que contienen el plan de proyecto, sus requisitos, consideraciones de diseño, los manuales de usuario y de programador y un informe de pruebas con la comparación entre este proyecto y alternativas anteriores.

Objetivos del proyecto

En el escenario actual de la minería de datos (ver definición en 3.1) la creciente cantidad de datos que requieren ser analizados, así como el aumento de su complejidad, ha generado un problema a la hora de tratar esos datos con eficiencia y velocidad. El hecho de que este problema parezca ir en aumento ha dado lugar a una búsqueda de soluciones que proporcionen una alta capacidad de cálculo y una fácil escalabilidad [17].

Una de las propuestas ha sido la posibilidad de ejecutar las labores de minería de manera paralela. Por otro lado, analizando el problema desde un enfoque diferente, aunque no incompatible, han adquirido gran popularidad los algoritmos encargados de reducir el número de instancias y atributos, de manera que podamos hacer más pequeño el conjunto a estudiar o incluso reducir su nivel de ruido. Este trabajo surge bajo la motivación de explorar estas nuevas soluciones.

Será un proyecto con dos objetivos principales: por un lado la implementación de diferentes algoritmos de reducción de instancias que, además de correr en paralelo, nos permitan reducir el número de datos iniciales. Por otro, el análisis del rendimiento de implementaciones lineales frente a nuestras implementaciones paralelas.

Como objetivo adicional, surgido durante la realización del trabajo, tendremos en consideración la creación de un entorno visual y más intuitivo donde el usuario pueda ejecutar nuestro código de una manera más sencilla.

Igualmente, tendremos en consideración la posibilidad de ejecutar el resultado de este proyecto en una red de nodos real, al ser este el entorno final donde este programa está pensado para ser ejecutado.

2.1. Implementación de algoritmos de selección de instancias

Se programarán un conjunto de algoritmos que puedan aplicarse sobre grandes conjuntos de instancias con el fin de reducir su tamaño sin perjudicar en exceso la información que transmiten. Para una definición más precisa de lo que es un algoritmo de selección de instancias ver la sección 3.3.

Se ha realizado la implementación de los siguientes algoritmos:

- *Localy sensitive hashing instance selection* (LSHIS) (ver sección 3.3.1) [5].
- *Democratic instance selection* (DemoIS) (ver sección 3.3.2) [14].

2.2. Estudio del rendimiento de la minería de datos en un modelo de ejecución en paralelo

El principal objetivo de este estudio será comprobar los tiempos de ejecución según qué paradigma (secuencial o paralelo) y según qué conjunto de datos.

Utilizaremos dos herramientas diferentes: Weka (ver definición en 4.2.2) para la ejecución lineal y Spark (ver definición 4.2.1) para la ejecución en paralelo.

Las pruebas serán realizadas en una única máquina local, con la posibilidad de extenderlas a un clúster si cumpliésemos con el objetivo definido en 2.3.

2.3. Ejecución del proyecto en un clúster en la nube

A fin de asegurar que el trabajo puede funcionar sin ningún tipo de complicación en un entorno real, se ha propuesto como objetivo el despliegue de la aplicación en algún servicio que ofrezca computación en la nube.

Se ha elegido para ello el servicio Google Cloud Dataproc [19], que ofrece la posibilidad de realizar trabajos de Spark. Igualmente, se ha considerado la posibilidad de ejecutar lanzamientos en un servidor clúster propiedad de la Universidad de Córdoba y al que la la Universidad de Burgos tiene acceso. En este último caso, además, podríamos llevar a cabo un estudio sobre la velocidad de nuestra aplicación en este tipo de sistemas.

2.4. Implementación de una interfaz gráfica de usuario

Con el objetivo de facilitar el uso de la aplicación y sus algoritmos, se ha incluido el desarrollo de una interfaz gráfica que permita diseñar experimentos sin la necesidad de que el usuario tenga que acceder a su consola de comandos.

Dicha implementación no solo permitirá configurar de manera simple los algoritmos, sino que también será un entorno donde poder seleccionar algunas de las opciones de lanzamiento de Spark o incluso seleccionar los conjuntos de datos a utilizar.

Conceptos teóricos

3.1. Minería de datos

Es el proceso mediante el cual se pretende extraer conocimiento de un conjunto de datos que, sin ser tratados o analizados previamente, no nos proporcionan información útil [17].

Se trata de un término que podría generar confusión con el de KDD (*Knowledge Discovery from Data* [12]), siendo en ocasiones tratado como un mero sinónimo de este término (que apareció antes que el de minería de datos) y en otras siendo descrito como un proceso dentro del descubrimiento de información, encargado de obtener conocimiento mediante la aplicación de algoritmos sobre datos recibidos [17].

A lo largo de esta memoria trataremos a la minería de datos como sinónimo de KDD, esto es, el conjunto de procesos que comprenden desde el pre procesamiento de los datos hasta la obtención y presentación de la información útil que contienen.

3.2. Big Data

Big Data, pese a ser un término cuya definición no ha sido nunca estandarizada, podría definirse como un conjunto de datos tan grande y complejo con el que resulta imposible trabajar utilizando metodologías tradicionales [26].

Fue utilizado por primera vez en 1997, en referencia a un problema donde un conjunto de datos tenía dificultad para ser almacenado en memoria o incluso en disco [25, 9]. Este aspecto es vital para la comprensión y el trabajo en el área del Big Data: hemos de considerar, desde un primer momento, que no todo nuestros datos van a poder ser guardados en memoria al mismo tiempo.

Sin embargo, el volumen de información no es el único aspecto importante a considerar cuando tratamos con el Big Data. Existe una expresión denominada

"Las cuatro uves del Big Data" (*The Four V's of Big Data*) que pretende definir las cuatro características básicas de este término: volumen, variedad, velocidad y veracidad [18]. Cualquier trabajo relacionado con el Big Data deberá tener en cuenta la influencia, en mayor o menor medida según el área de trabajo, de cada uno de estos términos y actuar en consecuencia.

En lo que se refiere a este proyecto, el término tiene una importancia fundamental: nuestro objetivo final es el de implementar nuevas soluciones que permitan trabajar sobre una gran cantidad de datos, haciendo frente a todos los problemas que esto pudiese suponer.

3.3. Algoritmos de selección de instancias

El objetivo de estos algoritmos es solucionar dos problemas que afectan a la minería de datos: la cantidad cada vez mayor de datos, y su calidad. Podremos, por lo tanto, definirlos como una herramienta para eliminar, de un conjunto de instancias, aquellas que conocemos, o sospechamos, son superfluas o perjudiciales [8].

Suprimiendo una porción del conjunto de instancias durante la fase de pre procesamiento de los datos conseguimos que el tiempo de ejecución algoritmos posteriores se reduzca, dado que hay menos instancias a examinar, mientras que es posible llegar a mejorar los resultados obtenidos al finalizar el proceso de minería si conseguimos eliminar instancias ruidosas [8].

3.3.1. Locality sensitive hashing instance selection (LSHIS)

LSH, no confundir con el algoritmo de selección de instancias que definiremos a continuación, es un algoritmo que permite identificar y agrupar elementos muy semejantes. Su comportamiento se basa en un conjunto de funciones *hash* que tienen la característica fundamental la capacidad de asignar elementos similares a un mismo grupo (*bucket*) con una alta probabilidad [5].

El algoritmo LSHIS es un método de selección de instancias que se apoya en el uso de LSH. La idea es aplicar, sobre las instancias iniciales, un conjunto de funciones hash que permitan agrupar en un mismo bucket aquellas instancias con un alto grado de similitud. Posteriormente, y realizando este proceso durante varias iteraciones si es preciso, de cada uno de esos buckets seleccionaremos una instancia de cada clase para formar el conjunto de instancias final.

La ventaja de este algoritmo frente a otras alternativas es que permite realizar la selección de instancias en un tiempo de complejidad lineal, en comparación con soluciones de complejidad cuadrática o logarítmica [5].

Algoritmo 1: LSH-IS – Algoritmo de selección de instancias mediante hashing. [5]

Input: Conjunto de instancias $X = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$, conjunto $\mathcal{G}$ de familias de funciones hash
Output: Conjunto de instancias seleccionado $S \subset X$

```
1 S = ∅
2 foreach instancia x ∈ X do
3     foreach familia de funciones g ∈ G do
4         u ← cubeta asignada por la familia g a la instancia x
5         if no existen otras instancias de la misma clase que x en u then
6             Añadir x a S
7             Añadir x a u
8 return S
```

3.3.2. Democratic Instance Selection

El algoritmo DemoIS es un método de selección de instancias que consiste en aplicar, durante un número variable de rondas, otros algoritmos de selección de instancias sobre subconjuntos disjuntos del gran conjunto inicial. En cada iteración, este algoritmo asignará unos "votos" a las instancias dependiendo de si han sido seleccionadas por el algoritmo al que se han sometido. Concluidas las rondas de votación, se realizará un cálculo con una función de *fitness* para seleccionar todas aquellas instancias cuyos votos no hayan superado un determinado límite [14].

Al igual que el algoritmo LSHIS mencionado anteriormente, la motivación fundamental es la de crear un algoritmo que requiera una carga computacional menor que las soluciones tradicionales. En este caso, eso se consigue mediante la división del conjunto de datos original en otros más pequeños y, por supuesto, el correcto análisis de los resultados anteriores.

3.4. Computación paralela

La computación paralela es un tipo de computación en la que múltiples operaciones son llevadas a cabo simultáneamente [3]. Parte del principio de que algunos problemas pueden subdividirse en problemas independientes más pequeños que pueden resolverse al mismo tiempo.

Es un paradigma que desde el principio se usó para operaciones que requiriesen una gran carga computacional, pero que ha despertado mucho interés en los últimos años gracias a la fácil escalabilidad que puede ofrecer frente a otras alternativas, como el aumento de la frecuencia de los procesadores, que han alcanzado un punto donde resulta más difícil avanzar [28].

Algoritmo 2: LSH-IS – Algoritmo de selección de instancias Democratic instance selection. [14]

Input: Conjunto de instancias $T = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$, tamaño de los subconjuntos s, número de rondas r
Output: Conjunto de instancias seleccionado $S \subset T$

```
for k = 1 to r do
    Dividimos las instancias en subjuntos disjuntos t_j de tamaño s tal que ⋃_i t_j = T
    for j = 1 to s do
        Aplicamos un algoritmo de selección de instancias a t_j
        Añadimos un voto a las instancias removidas de t_j
Calculamos la mejor función fitness en base a los votos
v = Votos que producen la mejor función fitness
S = T
Eliminamos de S todas las instancias cuyo número de votos sea ≥ v
return S
```

En lo que se refiere al uso de la memoria por parte de un sistema paralelo, existe la posibilidad de que la memoria sea compartida o distribuida dependiendo de si las unidades de procesamiento poseen un espacio de memoria común o cada unidad posee su propio espacio. En lo que se refiere a la tecnología que vamos a usar (ver sección 4.2.1), la memoria siempre será distribuida [31].

3.5. Escalabilidad

Escalabilidad es la capacidad de un sistema para adaptarse y soportar una carga de trabajo cada vez mayor [7].

Durante el desarrollo de la memoria siempre usaremos el término de escalabilidad para referirnos a la escalabilidad horizontal, aquella que consigue su objetivo añadiendo más nodos a un sistema.

También es una característica que puede aplicarse a los algoritmos, para los cuales definimos escalabilidad como la capacidad de funcionar eficientemente cuando se aplican sobre situaciones que requieran una gran carga de trabajo. En lo referente al proyecto, esa situación será, obviamente, tratar con grandes conjuntos de datos.

3.6. Rendimiento

Aunque el término puede dar lugar a numerosas definiciones, en lo que se refiere a nuestro proyecto, nos referiremos al rendimiento como la velocidad a la que un algoritmo puede ofrecer el resultado esperado.

Entiéndase que, en el contexto del trabajo, el objetivo ha sido siempre el de intentar mejorar los tiempos de ejecución de nuestras implementaciones con respecto a soluciones anteriores que seguían otro tipo de paradigmas (secuencial). Nunca se ha planteado la posibilidad de mejorar otros aspectos, dejando este término con la definición indicada anteriormente.

Técnicas y herramientas

4.1. Técnicas

En esta sección detallaremos brevemente la metodología utilizada durante el desarrollo del proyecto, junto como una pequeña explicación, más concreta, de cómo se ha aplicado en este caso concreto.

4.1.1. Scrum

Scrum es una metodología ágil de desarrollo iterativo e incremental para la gestión del desarrollo de un producto [29].

Como parte de la metodología, el trabajo se ha dividido en *sprints*, intervalos de tiempo de pocas semanas que ofrecen un producto al final de los mismos. Estos, a su vez, se han dividido en hitos y estos, en tareas.

En lo que se refiere a su aplicación práctica dentro del proyecto, los sprints han tenido una duración aproximada de dos semanas, periodo tras el cual había una reunión entre alumno y tutores para hablar sobre el avance y problemas ocurridos a lo largo del sprint, así como para definir el avance del proyecto durante el próximo periodo de tiempo.

Para la gestión de los sprints, hitos y tareas nos hemos apoyado en el gestor de incidencias que la plataforma Bitbucket (ver sección 4.2.5) proporciona en el repositorio del proyecto. De esta manera, cada incidencia marcada con la etiqueta *task* corresponde con un hito a realizar, mientras que todos ellos están agrupados en sprints, llamados *milestones* en la plataforma.

Hemos utilizado este método para realizar el seguimiento del proyecto porque, pese a no ser una herramienta especialmente dedicada a esta labor, evita el uso de nuevo software y puede ofrecer un buen resultado si existe atención por parte de los coordinadores del proyecto, en este caso los tutores [11].

4.2. Herramientas

La realización de este trabajo ha dado lugar al uso de multitud de herramientas. A continuación, se van a detallar dichas herramientas, el uso que se les ha dado y la razón por la cual fueron elegidas.

4.2.1. Apache Spark

Apache Spark (http://spark.apache.org/) es un motor de interés general destinado al procesamiento distribuido de grandes conjuntos de datos. Está implementado en Scala, pero también proporciona APIs para otros lenguajes de programación (Java, Python y R) y librerías para áreas como el aprendizaje automático (ver la sección 4.2.1.1) [27].

La idea nació como proyecto en 2010, en la Universidad de California, Berkeley, y su primera versión estable apareció el 30 de mayo de 2014. La motivación inicial era la de proporcionar un nuevo modelo de computación paralela que permitiera la ejecución eficiente de modelos que debían utilizar durante múltiples iteraciones grandes conjuntos de datos. Aproximaciones anteriores basadas en el modelo de MapReduce [10] (como Hadoop), requerían cargar de nuevo todos los datos en memoria, haciendo la tarea demasiado costosa. Como beneficio adicional, Spark ha demostrado que requiere de muchas menos líneas de código a la hora de programar algoritmos destinados al manejo de *Big Data* [31].

Actualmente Spark es un proyecto de código abierto cedido a Apache, siendo uno de los más activos en cuanto a contribuciones de la comunidad [21].

Para la realización del proyecto hemos utilizado diferentes versiones de Spark. Por lo general hemos trabajado con la versión 1.5.1, pero al utilizar servicios de computación en la nube (ver sección 5.5) hemos utilizado la versión 1.5.2 en Google Cloud Dataproc y 1.6.0 en un servicio Clúster Dell de la Universidad de Córdoba.

4.2.1.1. Machine Learning Library (MLlib)

Se trata de una de las librerías incluidas en Spark. Contiene un conjunto de clases relacionadas con el campo del aprendizaje automático, como tipos de datos o funcionalidades estadísticas.

4.2.1.2. Resilient Distributed Datasets (RDD)

Es una de las características esenciales de Spark. Consiste en una colección de objetos accesible en modo solo lectura y distribuida a lo largo de un conjunto de máquinas, que pueden reconstruir una de sus particiones si esta llegase a perderse [31]. En el caso concreto de nuestro proyecto, los objetos

que formen las RDD serán, en la mayoría de los casos, las instancias utilizadas durante las labores de minería.

Estas estructuras soportan dos tipos de operaciones:

- **Transformaciones:** Actúan sobre una estructura RDD produciendo como salida una nueva RDD resultado de una modificación de la anterior. Por defecto, todas las transformaciones sobre una RDD son perezosas (*lazy*), lo que quiere decir que no se ejecutarán hasta que una acción solicite un valor concreto que requiera de la transformación [31].
- **Acciones:** Operaciones sobre las RDD que devuelven un resultado que depende del tipo de acción aplicada.

Otra de las grandes diferencias que caracterizan a las RDD de otro tipo de estructuras es la posibilidad de definir fácilmente el nivel de memoria en el que queremos alojar los datos, algo de lo que muchos *frameworks* anteriores carecían [30]. En un principio todas las RDD son efímeras, esto es, serán eliminadas de memoria si no se indica lo contrario, pero pueden mantenerse para obtener mejores resultados de rendimiento si los datos van a usarse con frecuencia. A esta acción de mantener en memoria una RDD se le llama cachear (*caching*).

4.2.2. Weka

Weka [16] es un software desarrollado para llevar a cabo labores de minería de datos. Se trata de un proyecto de software libre, realizado en Java y desarrollado por la Universidad de Waikato, Nueva Zelanda.

Contiene, no solo algoritmos de aprendizaje automático para la minería de datos, sino también algoritmos de pre procesamiento de los datos o de visualización.

Al contrario que otras tecnologías que vamos a usar, esta librería no está pensada para la ejecución en paralelo, lo que la convierte en una buena herramienta para comparar el rendimiento que aplicaciones como Spark (véase 4.2.1) pueden ofrecernos.

Se ha usado en su versión 3.6.13 durante el principio del proyecto y, posteriormente, en su versión 3.7.13 por problemas de compatibilidad con una de las librerías proporcionadas por el tutor.

4.2.3. Scala

Scala (`http://www.scala-lang.org/`) es un lenguaje de programación orientado a objetos y a la programación funcional. Es también fuertemente tipado.

Es un lenguaje compilado, produciendo como salida ficheros .class que han de ser ejecutados en una máquina virtual de Java (JVM). Esto permite que librerías de Java puedan ser utilizadas directamente en Scala y viceversa. Por la misma razón, Scala posee la misma portabilidad que Java, pudiendo ejecutarse en cualquier sistema operativo siempre y cuando cuente con una máquina virtual de Java.

Los motivos de su elección como lenguaje de programación han sido mencionados en la sección 5.1

Se han utilizado dos versiones de Scala diferentes. Para la ejecución de Spark en nuestra máquina local se ha usado Scala en su versión 2.11.7, mientras que, cuando hemos querido desplegar el proyecto en otro servicio, hemos necesitado compilar nuestro trabajo usando Scala 2.10.6.

4.2.4. Java

Java (https://java.com/) es un lenguaje de programación orientado a objetos de propósito general diseñado para producir programas multiplataforma.

Necesitamos realizar la instalación de Java porque, aunque no trabajemos directamente sobre este lenguaje, si vamos a necesitar de su máquina virtual para poder ejecutar nuestros programas. Además, también hemos utilizado algunas de sus clases.

Hemos usado Java 8 u60 para la realización del proyecto.

4.2.4.1. JConsole y JvisualVM

JConsole y JvisualVM son una serie de herramientas gráficas de monitorización para aplicaciones Java. Ambas están incluidas dentro del Java Development Kit (JDK).

En el proyecto, se han utilizado para evaluar y medir el rendimiento de aplicaciones en Java a nivel local. En el caso de JvisualVM, se ha utilizado específicamente para ver el estado de los hilos que componen la aplicación Java, algo que no permite hacer JConsole.

La elección de utilizar estas herramientas frente a cualquier otra ha sido su facilidad de uso y la posibilidad, en el caso de JConsole, de poder exportar a CSV las mediciones realizadas sobre el uso de memoria o CPU.

4.2.5. Bitbucket

Bitbucket (https://bitbucket.org/) es un repositorio de código que permite la creación, control y mantenimiento de proyectos, que podrán ser públi-

cos o privados. Puede trabajar con los sistemas de control de versiones Git y Mercurial.

Bitbucket fue propuesto como gestor del proyecto durante el primer sprint del proyecto y, al no tener preferencia por ningún otro repositorio, se aceptó como herramienta a utilizar.

4.2.5.1. Git

Git (https://git-scm.com/) es un sistema de control de versiones gratuito y de código abierto.

La elección de Git vino motivada por ser un sistema que ya había sido utilizado antes a lo largo de la carrera, por lo que no ha sido necesario aprender su funcionamiento.

Se ha utilizado la versión 2.6.2.

4.2.6. Eclipse

Eclipse (https://eclipse.org/) es un entorno de desarrollo integrado (IDE de sus siglas en inglés) gratuito y de código abierto. Aunque su principal uso se basa en el desarrollo de aplicaciones en Java, también puede ser adaptado mediante el uso de plugins para ser utilizado en el desarrollo de otros lenguajes.

Un dato importante de este entorno de desarrollo es que está puramente basado en plugins, esto es, a excepción de un pequeño kernel, cualquier otra funcionalidad está incluida como un plugin, lo que le proporciona una gran facilidad para ser escalado o adaptado a las necesidades del usuario concreto.

Hemos trabajado con la versión 4.4.2.

4.2.6.1. ScalaIDE for Eclipse

Se trata de un plugin que puede añadirse al entorno de desarrollo Eclipse para poder desarrollar en Scala [13].

Esta herramienta consigue imitar la mayoría de los aspectos que Eclipse proporciona para Java para permitir un desarrollo más cómodo, esto es, el autocompletado de código, resaltado de texto, definiciones e hipervínculos a clases, marcadores de errores u opción *debug*.

La versión utilizada de este plugin es la 4.2.0.

4.2.7. ScalaStyle

ScalaStyle (http://www.scalastyle.org/) es una plugin enfocado a la detección de medidas estáticas de calidad en el código de Scala.

Su funcionamiento se basa en la definición de una serie de reglas en un fichero .xml cuyo cumplimiento será revisado en todo el código del proyecto, indicando aquellos puntos donde existe una irregularidad con respecto a dichas reglas.

Se ha incorporado este plugin en Eclipse para detectar y corregir errores de calidad en el código.

4.2.8. Apache Maven

Apache Maven (https://maven.apache.org/) es una herramienta para la gestión y construcción de proyectos software en Java. Nace con la intención de definir una manera estándar para la construcción de proyectos.

Se basa en el concepto de *Project Object Model* (*POM*), un archivo en formato XML que describe el proyecto a construir, la manera de construirlo y las dependencias con otros componentes.

El lenguaje utilizado para la elaboración del proyecto ha sido Scala y, como se ha mencionado, Maven fue pensado para trabajar sobre proyectos Java. Es por ello que necesitaremos de añadir un plugin adicional (*Scala-Maven-Plugin*) a nuestro fichero POM.

Hemos utilizado esta herramienta para empaquetar nuestro trabajo, además de para construir el propio Spark a partir del código fuente. Se ha seleccionado esta herramienta frente a otras opciones propias para Scala por conocerse con anterioridad su funcionamiento y por no requerir demasiado esfuerzo para ser utilizada en el lenguaje de programación que deseamos.

4.2.9. Zotero

Zotero (https://www.zotero.org/) es un gestor de referencias bibliográficas gratuito. Es por esto que la función de esta aplicación ha sido la de recompilar y organizar todos los enlaces que pudiesen ser de interés para la realización del trabajo y la memoria.

Para su uso se ha utilizado el plugin en su versión 4.0 para el navegador Mozilla Firefox.

4.2.10. TeX Live

TeX Live (https://www.tug.org/texlive/) es una distribución gratuita de LaTeX creada en 1996 y mantenida actualizada hasta la fecha. LaTeX por su parte, es un sistema de creación documentos en los que se requiera una alta calidad tipográfica.

En el proyecto se ha utilizado LaTeX para la realización de la memoria, así como los documentos anexos.

Se ha utilizado su versión más reciente hasta la fecha, TeX Live 2015.

4.2.11. TexMaker

TexMaker (http://www.xm1math.net/texmaker/) es un editor multiplataforma pensado para el desarrollo de documentos escritos en LaTeX. Al igual que otros muchos editores, esta plataforma presenta diferentes herramientas para hacer la creación de los documentos mucho más sencilla, tales como el autocompletado de etiquetas, la detección de errores ortográficos o el coloreado de texto.

Hemos usado la versión 4.4.1.

4.2.12. Pencil Project

Pencil Project (http://pencil.evolus.vn/) es una herramienta de prototipado de interfaces gráficas.

Permite diseñar la apariencia de una aplicación arrastrando los componentes desde un menú de selección hasta una pantalla de dibujo, donde podremos modificar aspectos, como tamaño o texto, hasta conseguir el resultado que queremos. Es por ello que la función de esta aplicación es solo la de crear una imagen visual de nuestro prototipo, en ningún momento dicho prototipo tendrá funcionalidad ninguna.

He aplicado esta herramienta durante la realización de la interfaz gráfica del programa, para poder organizar los componentes en el espacio de una manera más sencilla.

Aspectos relevantes del desarrollo del proyecto

5.1. Elección del lenguaje de programación

Spark es un sistema que proporciona soporte a diferentes lenguajes de programación: Java, Scala, Python y, recientemente, R [27]. Eliminando este último como posible elección, por el desconocimiento del lenguaje y la poca documentación que hay sobre su uso con Spark, se ha realizado una comparativa entre las opciones restantes para ver cuál de ellas reportaría más ventajas en la realización del proyecto.

Para elegir los aspectos a tener en cuenta durante esta comparación me he basado en el estándar ISO 9126 para la calidad del software [20]. Los puntos a analizar han sido finalmente los siguientes:

- **Experiencia previa:** Se tendrá en cuenta el contacto que se haya tenido con los lenguajes anteriormente.
- **Eficiencia de ejecución:** Valoraremos el rendimiento de cada lenguaje en función del tiempo que requieren para la ejecución de programas.
- **Facilidad de mantenimiento:** Las ventajas y facilidades del lenguaje en el caso de que se requiriese corregir o mantener un algoritmo o aplicación.
- **Adecuación:** Beneficios aportados por el lenguaje para su uso concreto en el desarrollo de algoritmos para Spark.
- **Documentación disponible:** Facilidad para encontrar información actualizada sobre el uso del lenguaje en Spark.

Nótese que la tabla 5.1 es una comparativa entre las características de los diferentes lenguajes para su uso en Spark, no una comparativa entre las características propias de cada uno a nivel general. Por lo tanto, aspectos que no tengan influencia en Spark o que sean iguales para todos los lenguajes no serán incluidos en la comparativa. Así mismo, si dos lenguajes compartiesen una característica muy parecida o idéntica, esta será incluida una sola vez en la comparativa, fusionando las dos celdas que correspondan.

Criterio	Java	Scala	Python
Experiencia previa	Se ha trabajado en Java múltiples veces durante el grado.	Es un lenguaje sobre el que nunca se ha trabajado.	Se han aprendido las nociones básicas durante la carrera.
Eficiencia de ejecución	Compila los ficheros generando archivos .class y los ejecuta sobre una máquina virtual de Java (JVM). Esto conlleva que la ejecución sea considerablemente más rápida que la del intérprete de Python.		Es un lenguaje interpretado, lo que afecta negativamente a su rendimiento. Sin embargo, su rendimiento en comparación con Scala mejora considerablemente si contamos con muchos procesadores [23].
Facilidad de mantenimiento	Código más extenso, aunque la aparición Java 8, con elementos como las funciones lambda han mejorado este aspecto.	Menos líneas de código y una sintaxis más fácilmente legible.	
Adecuación	Trabajar sobre Java nos obliga, a la hora de programar, a transformar estructuras y clases de Spark solo soportadas al trabajar en Scala o Python. Además, no cuenta con un intérprete interactivo.	Spark está pensado para trabajar sobre Scala o Python. De hecho, Spark ha sido creado en Scala, por lo que su conocimiento puede ayudar a lo largo del proyecto. Ambos lenguajes cuentan con un intérprete interactivo.	
Documentación disponible	Existe buena documentación en la página oficial de Spark, aunque es más escasa en otras fuentes. Además, muchas veces la documentación no trabaja sobre Java 8.	Existe una amplia documentación sobre el uso de ambos lenguajes en Spark.	

Cuadro 5.1: Comparativa entre características de Java, Scala y Python para trabajar sobre Spark.

Como elección final, decidimos que el lenguaje a utilizar será Scala, argumentando lo siguiente sobre los puntos que hemos comparado:

- **Sobre la experiencia previa:** No se considera un problema aprender el lenguaje. Además, los archivos generados tras la compilación y, por consiguiente, la manera de ejecutarlos o monitorizar su rendimiento, es similar a Java, un lenguaje ya conocido.
- **Sobre la eficiencia de ejecución:** El rendimiento, en lo que a tiempo de ejecución se refiere, es muy similar a Java, por lo que se considera una ventaja frente Python.
- **Sobre la facilidad de mantenimiento:** En el momento de la elección del lenguaje no contamos con experiencia en el mantenimiento de un gran código de minería de datos, pero parece lógico que a menos cantidad de líneas que mantener, más fácil puede resultar la tarea.
- **Sobre la adecuación:** Tras probar Java y Scala con Spark se ha llegado a la conclusión de que el primero implica, no solo más código, sino también operaciones de conversión de estructuras que funcionan en Scala o Python pero son diferentes para Java. Además, frente a Java, Scala cuenta con un intérprete de comandos que nos permite hacer pruebas sin la necesidad de tener que generar y compilar el código cada vez que queramos probar algo.
- **Sobre la documentación disponible:** Scala cuenta, al igual que Python, con una extensa documentación actualizada. Al realizar pruebas con Java en Spark se ha notado un pequeño problema de falta de documentación, pero sobretodo, un problema para encontrar documentación actualizada para aspectos nuevos de Java 8 y que serán frecuentemente usados, concretamente las funciones lambda.

5.2. Comparativa de rendimiento en la ejecución de clasificaciones entre Weka y Spark

Como primer aspecto a evaluar durante la realización del proyecto, se llevó a cabo una comparativa entre las ejecuciones que ofrecen Weka y Spark.

La intención de esta prueba era doble: por un lado, suponía un primer acercamiento a la librería Spark y al modelo de funcionamiento en este tipo de entornos. Por otro, se pretendía medir el tiempo de ejecución de Spark frente a alternativas anteriores que, como principal diferencia, no están pensadas para ser ejecutadas en paralelo.

Para realizar las mediciones utilizamos conjuntos de datos de diferentes tamaños, pero siempre un mismo algoritmo: el Naive Bayes. Naive Bayes es

un algoritmo de clasificación probabilístico que ya se encontraba implementado tanto en la librería de Weka [22] como en la de Spark, razón por la cual ha sido elegido.

Los resultados, así como una explicación más detallada del experimento, pueden encontrarse en el material adjunto a la memoria. Sin embargo, como pequeña aproximación, se muestra en la gráfica 5.1 la evolución del tiempo de ejecución del algoritmo según el número de instancias y según qué configuración de lanzamiento (Weka y Spark con diferente número de ejecutores). Sin entrar a detalle, puede apreciarse claramente una reducción del tiempo de ejecución en Spark a medida que añadimos nuevos ejecutores, lo que se traduce en añadir más unidades de procesamiento a la ejecución.

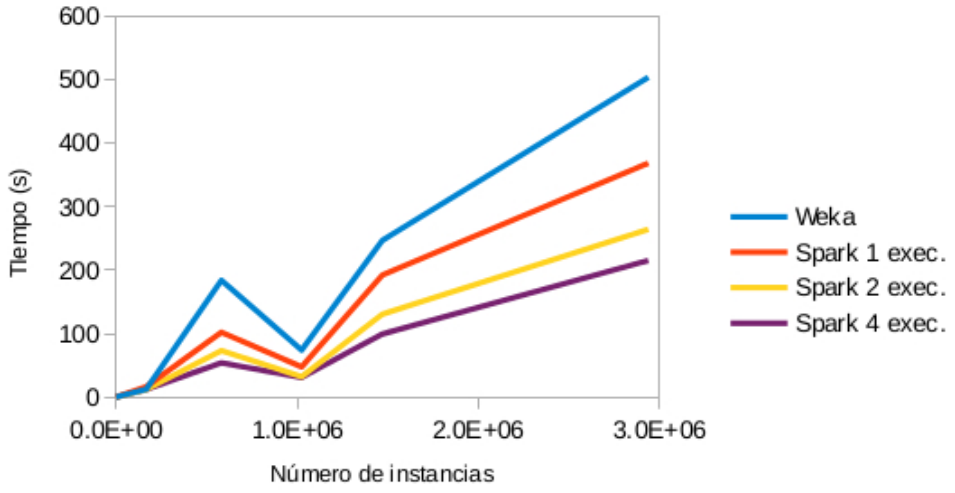


Figura 5.1: Evolución del tiempo de ejecución según el tamaño del conjunto de datos usando Naive Bayes.

5.3. Implementación de algoritmos de selección de instancias

Marcado como el objetivo fundamental de este proyecto, la realización de dos algoritmos de selección de instancias ha ocupado la mayor parte de tiempo dentro del mismo. A continuación se expondrán algunos de los aspectos que más influencia han tenido durante esta fase de implementación y una pequeña explicación más concreta de la propia implementación de los algoritmos.

5.3.1. Indeterminismo, comunicación limitada, grandes conjuntos de datos

Por la propia naturaleza de la librería, la programación en Spark se ha diferenciado en muchos aspectos del tipo de programación secuencial que se ha aplicado hasta ahora. El hecho de que nuestros algoritmos estén pensados para ejecutarse en paralelo y distribuidos en una gran red de nodos nos proporciona numerosas ventajas, a las que ya nos referimos en las secciones 3.4 y 3.5, pero también plantea problemas que es necesario tener muy en cuenta para la correcta ejecución del programa.

En primer lugar, una ejecución como la nuestra pierde la capacidad de predecir el orden en el que se ejecutarán las operaciones o el propio orden de las instancias cuando son distribuidas o intercambiadas por la red de trabajadores. Este problema ha obligado a modificar el planteamiento inicial de los algoritmos propuestos, pensados para ser ejecutados secuencialmente, y adaptarlos al nuevo escenario.

Así mismo, y con la misma consecuencia, se nos plantea el problema de la comunicación entre nodos. Suponiendo una amplia red de nodos con un gran poder de computación en cada uno de ellos, es sencillo pensar que la comunicación entre ellos puede ser complicada si no queremos que esto afecte de manera muy negativa al rendimiento. Es por ello que Spark no proporciona demasiadas posibilidades en este aspecto o, aquellas que ofrece, son bastante específicas o realmente costosas, como las operaciones de unión (*join*). Todo esto ha sido tenido en cuenta durante la fase de implementación, afectando a las operaciones o incluso a la manera en la que almacenamos los conjuntos de datos, cuyas instancias han sido a menudo ligadas a diferentes valores que permiten controlar el flujo del programa.

Por último, en un intento de optimización por parte de Spark, nos encontramos dentro de la librería la postura de evitar la aparición de operaciones costosas o, por lo menos, realizarlas de manera tal que no tengan un efecto tan perjudicial en el rendimiento. Esto tiene influencia en varios aspectos, siendo el más fácil de entender el de la distribución de las instancias entre los nodos. Algo tan aparentemente sencillo como la división de un conjunto de datos en particiones de igual tamaño, es algo que no ha sido implementado en Spark por el altísimo coste y complejidad que supondría ejecutarlo eficientemente sobre un gran conjunto de datos. En cambio, se proporcionan estrategias basadas en tablas hash o el análisis de pequeños subconjuntos de prueba que permitan realizar la división en particiones aproximadamente iguales. Este tipo de limitaciones han tenido que considerarse cuando hemos necesitado distribuir los nodos de una manera específica.

5.3.2. Lazy evaluation

Un aspecto particular de Spark, y muy importante de cara a la implementación, es el hecho de que Spark utiliza una estrategia de evaluación perezosa (*Lazy Evaluation*), como ya ha sido definido en la sección 4.2.1.

Esto, si bien se hace en un intento de conseguir optimizar las operaciones, ha de tenerse en cuenta si no se quiere obtener la situación contraria. Hay que comprender que las RDD solo se crean cuando se necesitan y que, por ello, no todas las operaciones se realizan estrictamente en el orden en el que se definen en el código.

En el siguiente ejemplo (5.1), mostramos un fragmento de código que genera un conjunto de pares "entrenamiento-test" y realiza sobre cada uno de ellos una operación cualquiera:

```
// Lectura
val originalData = readDataset(readerArgs)
// Creación de conjuntos para la validación
   cruzada
val cvfolds = createCVFolds(originalData)
// Por cada conjunto entrenamiento-test
cvfolds.map {
    case (train, test) => {
      // Realiza operaciones
    }
  }
```

Código 5.1: Código de ejecución de una validación cruzada en Scala

Una primera impresión del código 5.1 puede hacernos suponer que primero se generan todos los subconjuntos y, posteriormente, sobre cada uno de ellos se realiza la operación que se le pide. Sin embargo, el comportamiento real es muy distinto y es que, hasta que no necesitamos los conjuntos "train" y "test" dentro del bucle, no calculamos dichos conjuntos. De esta manera, cada vez que realicemos una iteración del bucle calcularemos solo los conjuntos que vayamos a necesitar para esa iteración.

Tener esto en cuenta es importante a la hora de programar, pues puede provocar, entre otros aspectos negativos, cuellos de botella, excesivo uso de almacenamiento con datos que no van a usarse en un futuro cercano o necesidad de recalcular grandes estructuras RDD.

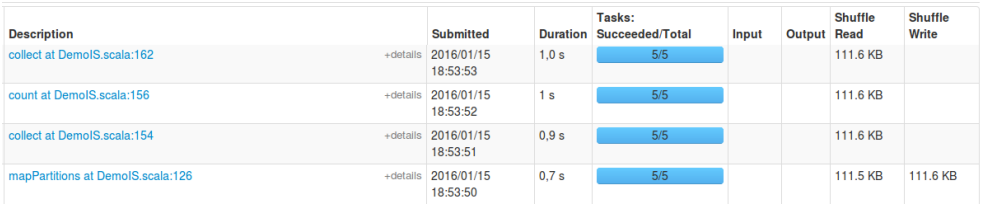
5.3.3. Correcta gestión de los recursos

Como se ha mencionado anteriormente en esta memoria (ver sección 4.2.1), una de las ventajas de Spark frente a otras alternativas es, no solo la posibilidad

de usar diferentes niveles de almacenamiento, sino también la posibilidad de definir nosotros qué almacenar y dónde hacerlo.

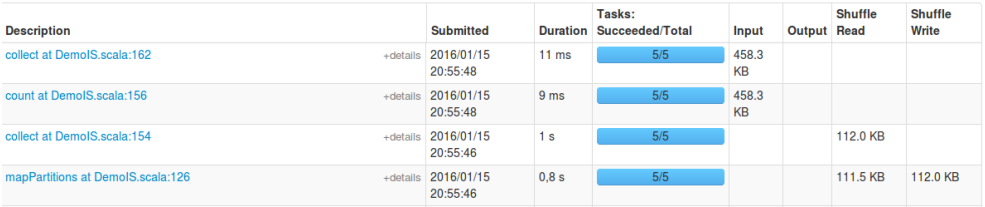
Aunque parezca un aspecto con importancia menor, solo útil cuando se busca optimizar al máximo una operación, lo cierto es que es un aspecto sumamente importante que tiene una altísima influencia en el tiempo de ejecución. Tener en mente qué estructuras RDD almacenar, dónde hacerlo o si realmente merece la pena ocupar espacio ante una operación que podría recalcularse es algo que se ha visto completamente necesario desarrollando el proyecto.

Tomemos como ejemplo el algoritmo DemoIS, donde se decidió persistir una estructura RDD generada tras la fase de votaciones. Podemos ver como el tiempo de ejecución de algunas de las operaciones ha sido reducido del orden de segundos (en la imagen 5.2) al tiempo de milisegundos (en la imagen 5.3).



Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
collect at DemoIS.scala:162 +details	2016/01/15 18:53:53	1,0 s	5/5			111.6 KB	
count at DemoIS.scala:156 +details	2016/01/15 18:53:52	1 s	5/5			111.6 KB	
collect at DemoIS.scala:154 +details	2016/01/15 18:53:51	0,9 s	5/5			111.6 KB	
mapPartitions at DemoIS.scala:126 +details	2016/01/15 18:53:50	0,7 s	5/5			111.5 KB	111.6 KB

Figura 5.2: Ejecución de operaciones en el algoritmo DemoIS sin persistir ninguna RDD.



Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
collect at DemoIS.scala:162 +details	2016/01/15 20:55:48	11 ms	5/5	458.3 KB			
count at DemoIS.scala:156 +details	2016/01/15 20:55:48	9 ms	5/5	458.3 KB			
collect at DemoIS.scala:154 +details	2016/01/15 20:55:46	1 s	5/5			112.0 KB	
mapPartitions at DemoIS.scala:126 +details	2016/01/15 20:55:46	0,8 s	5/5			111.5 KB	112.0 KB

Figura 5.3: Ejecución de operaciones en el algoritmo DemoIS persistiendo el conjunto de datos tras las votaciones.

Así mismo, también es necesario tener en cuenta la manera en la que se realizan las operaciones. Un punto importante de Spark es que necesita escribir los datos a disco cuando quiere redistribuirlos por la red de nodos, por si necesitase recuperarse de algún fallo. Escribir a disco es una operación muy costosa, por lo que se ha de intentar reducir al mínimo. El orden de las operaciones o el uso de diferentes operaciones para llegar a un mismo resultado, han sido considerados para aumentar el rendimiento de nuestro programa.

Un ejemplo lo encontramos entre el uso de dos funciones que las RDD proporcionan: *reduceByKey* y *groupByKey*. Aunque a menudo los resultados alcanzados por una de las funciones puede ser conseguido mediante la otra, *groupByKey* obliga a escribir a disco la estructura RDD completa, mientras que *reduceByKey* reduce considerablemente esta operación [15]. Problemas similares ocurren con funciones como los *join* o *subtractByKey*, utilizada también en este proyecto.

5.3.4. Implementación de resursos necesarios

Como ya se ha comentado anteriormente, la librería MLlib de Spark con la que hemos estado trabajando aún se encuentra en una fase donde no contamos con una amplia colección de clases. Es por ello que han tenido que implementarse algunos algoritmos adicionales a los propuestos como objetivo. Cabe destacar:

- **Algoritmo Condensed Nearest Neighbour (CNN):** Un algoritmo de selección de instancias simple y cuya ejecución es secuencial. Ha sido incluido de manera obligatoria para poder ejecutar correctamente nuestro algoritmo Democratic instance selection (ver algoritmo en la sección 3.3.2).
- **Algoritmo *k*-Nearest Neighbours (*k*NN):** Un clasificador simple, necesario tanto para la implementación de Democratic instance selection como para comparar el funcionamiento de nuestros selectores de instancias. Está programado de manera secuencial, de manera que requiere que todos los datos sean recogidos en una sola máquina para poder aplicar el algoritmo. En un primer momento se creyó que podríamos contar con una implementación en Spark del kNN gracias al material presentado en la Conferencia de la Asociación Española para la Inteligencia Artificial 2015 (CAEPIA 2015) [24], pero finalmente tuvo que implementarse una versión menos ambiciosa del clasificador.

5.3.5. Implementación concreta de LSHIS

Dadas las restricciones ya mencionadas en esta misma sección, la implementación del algoritmo LSHIS ha sufrido modificaciones con respecto a su propuesta original, cuyo pseudocódigo puede verse en el algoritmo 1.

Podemos ver una nueva versión del método en el pseudocódigo 3. El cambio fundamental puede apreciarse cuando existen dos o más familias de funciones hash. Por culpa del indeterminismo que genera la ejecución en paralelo nos vemos obligados a ejecutar una serie de operaciones sobre conjuntos que no eran necesarias en la ejecución secuencial, donde se solucionaba el problema mediante el uso de un nuevo bucle *for*.

Cabe destacar que estas operaciones realizadas cuando hay dos o más familias de funciones hash no usan en ningún momento el conjunto de instancias completo, sino que usan el conjunto solución generado por iteraciones anteriores y el conjunto solución generado por esta nueva iteración, ambos suponiéndose más pequeños que el conjunto inicial. Esta es una consideración muy importante en comparación con otras alternativas que surgieron, porque implica que la carga de trabajo va a ser menor que si tuviésemos que operar con todo el conjunto de instancias inicial.

Algoritmo 3: LSH-IS – Implementación paralela en Spark

Input: Conjunto de instancias $X = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$, conjunto $\mathcal{G}$ de familias de funciones hash
Output: Conjunto de instancias seleccionado $S \subset X$

```
1  S = ∅
2  foreach familia de funciones hash en G do
3      foreach instancia x en X do
4          {u, c} ← tupla formada por la cubeta asignada a x y su clase
5          Asociar x a su tupla {u, c}
6      sel ← Seleccionar una instancia aleatoria por cada par llave {u, c}
7      if es la primera iteración then
8          S = sel
9      else
10         foreach instancia x en S do
11             {u, c} ← tupla formada por la cubeta asignada a x y su clase
12             Asociar x a su tupla {u, c}
13         Añadir a S aquellas instancias de sel que no seleccionadas en S
14 return S
```

Como del pseudocódigo anterior no puede deducirse con claridad cómo están estructuradas las operaciones dentro del entorno paralelo, se incluyen los siguientes diagramas (ver imágenes 5.4 y 5.5) que permiten identificar como se realizan las operaciones dentro de un grupo de nodos. Compruébese que todas las operaciones se ejecutan en paralelo. Los bloques de "Conjunto de datos inicial" y "Solución" simplemente han sido añadidos para aclarar el diagrama, pero tanto los datos iniciales como el resultado podrían perfectamente ser datos que ya se encontrasen distribuidos.

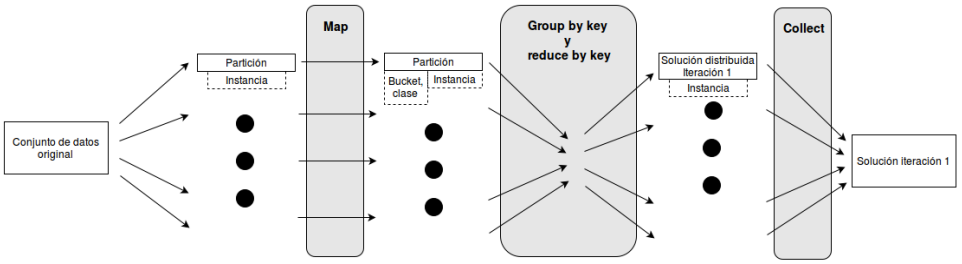


Figura 5.4: Diagrama de la ejecución del algoritmo LSHIS durante su primera iteración.

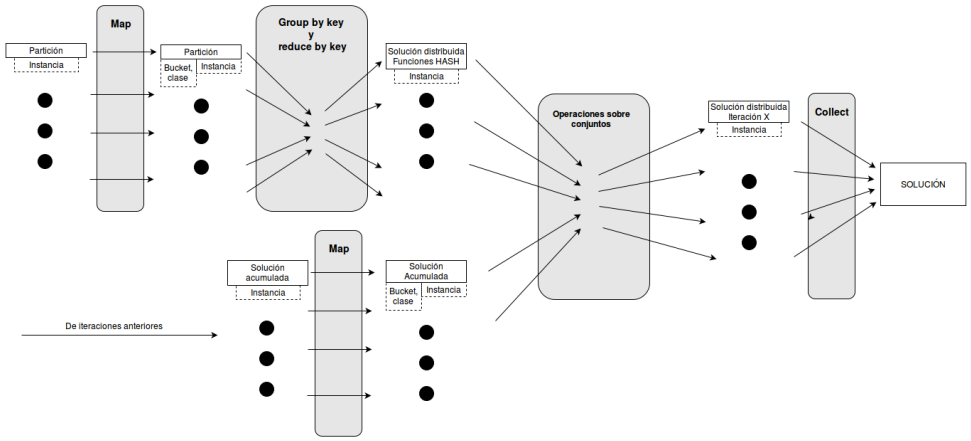


Figura 5.5: Diagrama de la ejecución del algoritmo LSHIS durante su segunda iteración y sucesivas.

5.3.6. Implementación concreta de DemoIS

La implementación de este algoritmo no ha requerido modificar el pseudocódigo del mismo, que puede verse en la sección 3.3.2). Sin embargo, sí existen varios detalles que conviene destacar sobre la actual implementación.

En primer lugar, como puede apreciarse en la figura 5.6, no todo el algoritmo se ejecuta en paralelo. A la hora de calcular el *fitness* óptimo lo hacemos de manera secuencial. Esto es así porque no contamos con una implementación paralela del algoritmo kNN, necesario para este proceso. Aún con esto, durante esa etapa del proceso solo seleccionamos un pequeño grupo de instancias en comparación con el conjunto inicial, por lo que operar con tal número de datos no supone un problema de rendimiento tan grande.

Existen otras consideraciones de cara a la implementación, como el uso de un particionador aleatorio que no genera particiones del mismo tamaño, por lo mencionado anteriormente, o la creación de una clase individual que evite la serialización completa del algoritmo al realizar la primera operación *map*. Esto último, será tratado con más detenimiento en el anexo de diseño incluido junto con la memoria.

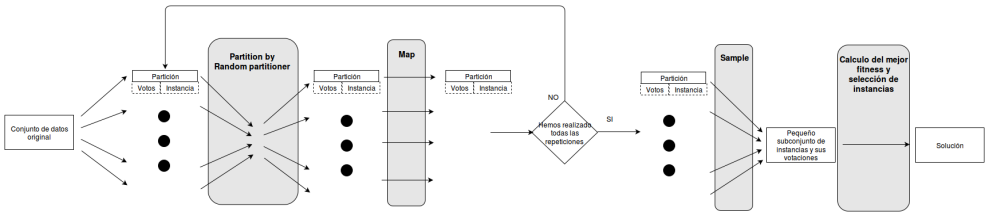


Figura 5.6: Diagrama de la ejecución del algoritmo DemoIS.

5.4. Comparativa entre las implementaciones de Weka y Spark

Como otro de los objetivos principales del proyecto, se procedió a realizar una comparación entre nuestras implementaciones de los algoritmos de selección de instancias con las ya existentes para su uso en Weka.

De nuevo, la finalidad de esta comparación tenía varios objetivos: evaluar el rendimiento entre ambas aproximaciones, comprobar el correcto funcionamiento de los algoritmos y evaluar como los cambios de implementación podían haber afectado a su funcionalidad.

Aunque se proporciona más información en los documentos anexos, podemos observar en las gráficas 5.7 y 5.7 una evolución del tiempo de ejecución según el algoritmo y la cantidad de datos analizados. Para ello se proporcionan varios tipos de ejecuciones, pudiéndose diferenciar uno para Weka y otros para Spark con diferente número de ejecutores.

Puede verse claramente que, pese a su ejecución en paralelo, el algoritmo LSHIS continúa siendo más rápido en su ejecución secuencial. Esto es debido en parte a que es un algoritmo muy rápido incluso cuando se aplica sobre un número de instancias grande. Por otro lado, DemoIS proporciona, desde un primer momento y con un conjunto de datos relativamente pequeño, mejores tiempos que la implementación anterior.

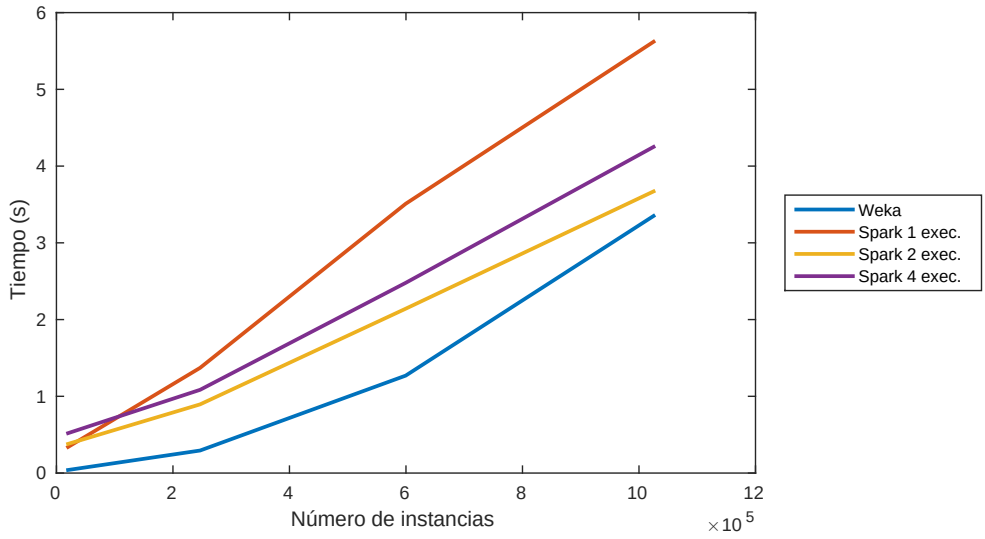


Figura 5.7: Evolución del tiempo de ejecución según el tamaño del conjunto de datos usando LSHIS.

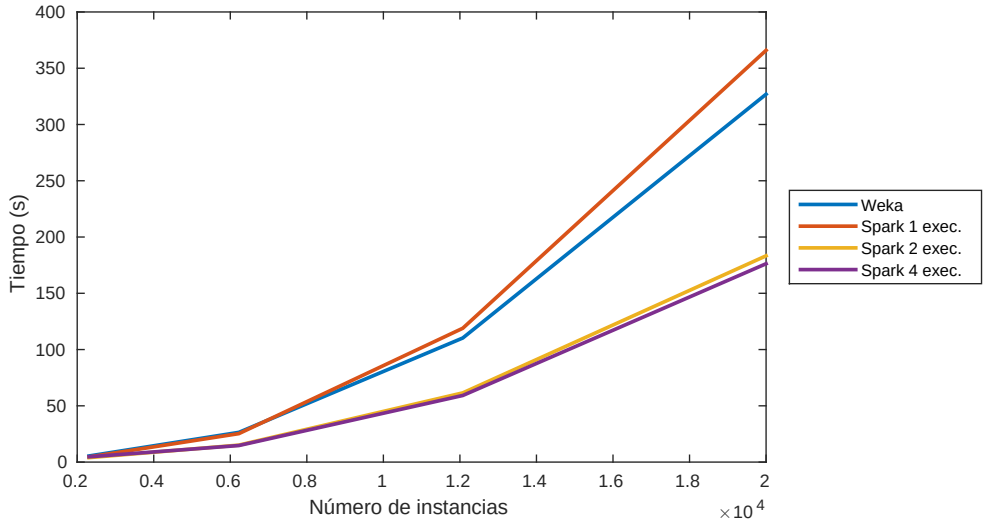


Figura 5.8: Evolución del tiempo de ejecución según el tamaño del conjunto de datos usando DemoIS.

5.5. Ejecución en la nube

Una vez se comprobó el buen funcionamiento del proyecto, se procedió a desplegar el trabajo en el entorno en el que está destinado a ser ejecutado: un

clúster real.

Este despliegue se ha realizado en dos plataformas diferentes:

- Google Cloud Dataproc [19]: Servicio, actualmente en estado *beta*, que Google ofrece para lanzar tareas de minería de datos en un clúster propiedad de la compañía. Utilizando una versión de prueba, los recursos han sido limitados (máximo de 8 procesadores por cluster), pero eso no ha impedido realizar lanzamientos correctamente.
- Servidor Clúster Dell: Un servicio al que la Universidad de Burgos tenía acceso y en el cual se ha permitido lanzar algunas ejecuciones. Este clúster está configurado con un sistema de colas de trabajo PBS (*Portable Batch System*), por lo que se ha necesitado de un script propuesto por la Universidad de Ohio [6] para ejecutar nuestras pruebas. Aunque se han tomado algunas mediciones sobre el comportamiento de nuestra aplicación en este entorno, se han encontrado dificultades al lanzar Spark sobre este tipo de sistema de gestión, donde algunos nodos quedaban bloqueados durante la ejecución presumiblemente por la falta de recursos.

Por las características de los servicios en los que se ha desplegado el programa, se necesitó realizar pequeños cambios en la implementación y definir una versión anterior de Scala a la hora de compilar el código.

Puede encontrarse información más concreta sobre cómo realizar experimentos sobre Google Cloud Dataproc en el manual del usuario anexo a esta memoria. Igualmente, resultados sobre las pruebas realizadas en estos servicios o pequeñas consideraciones a tener en cuenta en la etapa de implementación pueden encontrarse en los documentos “Pruebas del sistema” y “Manual del programador” respectivamente.

5.6. Implementación de un entorno gráfico

En la etapa final, se vio como el proyecto había alcanzado una complejidad considerable en cuando a las opciones de lanzamiento se refiere. Este hecho hacía de la ejecución por línea de comandos un método de lanzamiento mucho más tedioso de lo que a un usuario normal podría resultarle ya de principio. Es por esta razón que, con la intención de facilitar el uso de la biblioteca, se decidió la implementación de una pequeña interfaz gráfica que hiciese más intuitivo el uso del proyecto.

Así pues, se ha realizado una interfaz gráfica que permite introducir, mediante campos de texto, todas las opciones necesarias para la ejecución de los algoritmos. Además, posibilita la acción de crear baterías de ejecuciones al dar

la opción de indicar más de una configuración de Spark, conjunto de datos y/o filtro a la vez.

Esta interfaz ofrece dos modos de ejecución:

- Permite ejecutar los algoritmos directamente desde la interfaz gráfica. Es una opción no recomendada si lo que se busca es eficiencia en los tiempos de ejecución pero es una manera sencilla de realizar pruebas en modo local.
- Permite la compresión, en un archivo de extensión .zip, de todos los conjuntos de datos necesarios para lanzar las ejecuciones definidas. Además, también proporciona un script .sh para realizar las ejecuciones indicadas. Este tipo de operación facilita que el usuario pueda definir todas las operaciones desde su máquina y, posteriormente, pueda trasladarlas de manera sencilla a un servidor más potente donde serán ejecutadas.

Después de la creación y revisión de algunos prototipos no funcionales, la apariencia final del producto evolucionó hasta alcanzar el aspecto que muestra la figura 5.9.

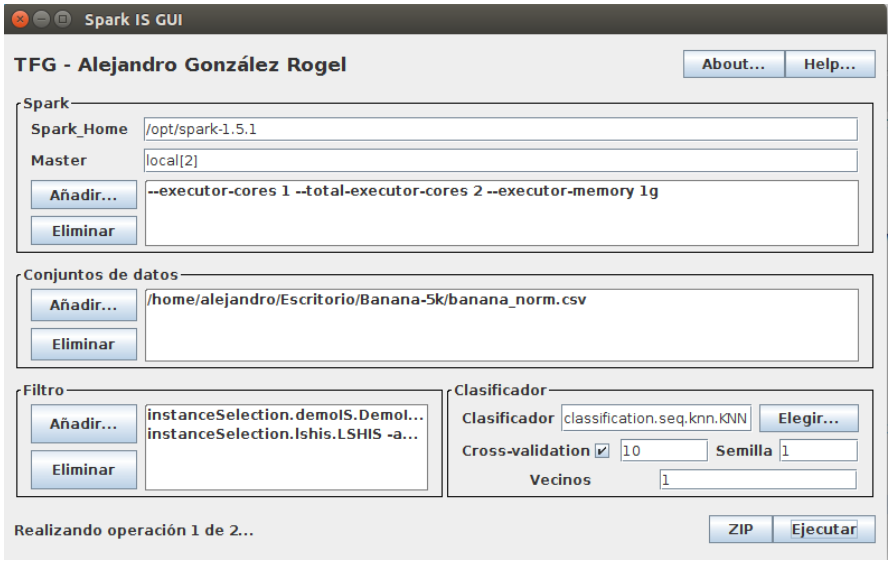


Figura 5.9: Apariencia final de la interfaz gráfica del proyecto.

Trabajos relacionados

6.1. Weka

Aunque ya ha sido mencionado y definido con anterioridad en esta memoria (ver sección 4.2.2), parece obligatoria una mención a dicho programa en este apartado.

Cabe destacar que, aunque tanto Weka como este proyecto comparten un objetivo común como es el de presentar una biblioteca con algoritmos de minería de datos y hacerlo de manera que sea fácil su utilización, el ámbito de aplicación de ambos programas es completamente distinto. Mientras que Weka parece pensado enfocado a pequeños/medianos conjuntos de datos y una ejecución secuencial de sus algoritmos, este proyecto utiliza Spark para hacer factible la idea de aplicar algoritmos a situaciones demasiado grandes y complejas como para ser tratadas en una sola máquina en un tiempo prudencial.

6.2. Knowledge Extraction based on Evolutionary Learning (KEEL)

KEEL [2, 1] es una herramienta de minería de datos basada en Java que permite la aplicación, sobre conjuntos de datos, de algoritmos de extracción de conocimiento. Al igual que Weka, también posibilita la opción de aplicar métodos de pre procesamiento sobre los datos originales antes de que sean utilizados en las labores de minería.

La diferencia fundamental de este software con respecto al presentado a lo largo de la memoria, vuelve a ser, como en el caso anterior, la ejecución secuencial de algoritmos de KEEL con respecto a la ejecución paralela de Spark.

6.3. Documentos científicos de LSHIS y DemoIS

A lo largo del proyecto, y en especial todo lo relacionado con la implementación y prueba de los algoritmos, se ha contado con los documentos científicos que definían la implementación y uso de LSHIS y DemoIS. Dichos documentos, ya mencionados anteriormente en la memoria, han sido "LSH-IS: Un nuevo algoritmo de selección de instancias de complejidad lineal para grandes conjuntos de datos"[5] y "Democratic instance selection: A linear complexity instance selection algorithm based on classifier ensemble concepts"[14].

Ambos documentos, además de contener información sobre los algoritmos implementados, también contienen comparaciones de rendimiento con otras alternativas de selección de instancias. En lo que se refiere a este proyecto, las pruebas realizadas han tenido como objetivo la comparativa entre tiempos de ejecución de las implementaciones secuencial y paralela, por lo que podemos defender que se han evaluado aspectos diferentes durante la etapa de experimentación.

Además, se ha contado con la implementación de los algoritmos LSHIS y DemoIS para su ejecución en Weka [4], lo que ha posibilitado realizar las experimentaciones defendidas en esta memoria y los documentos anexos.

Conclusiones y Líneas de trabajo futuras

La realización del proyecto ha permitido comprobar la complejidad de la programación paralela y aplicada a escenarios donde el rendimiento, la comunicación entre unidades de procesamiento o la falta de recursos son aspectos a tener muy en cuenta.

Los objetivos propuestos al comienzo del trabajo se han cumplido, logrando la implementación paralela de varios algoritmos de selección de instancias y su posterior comparación con las implementaciones anteriores. Igualmente, otras metas definidas *a posteriori*, como la implementación de una interfaz gráfica y la ejecución en la nube, han sido correctamente cumplidas.

Finalmente, y centrándonos en el ámbito de las comparaciones, este proyecto nos ha ayudado a desmentir la hipótesis inicial con la que se comenzó a trabajar: que todos los algoritmos propuestos conseguirían reducir el tiempo de ejecución con respecto a sus antiguas versiones. Se ha visto como el algoritmo LSHIS, debido a su rapidez, es difícil de mejorar cuando intentamos paralelizarlo. Por el contrario, el algoritmo DemoIS muestra resultados muy favorables incluso con conjuntos de instancias pequeños.

7.0.1. Lineas de trabajo futuras

La propia naturaleza del proyecto ofrece una gran variedad de alternativas para seguir trabajando:

- **Continuar trabajando sobre los algoritmos ya implementados:** Aunque es posible que el algoritmo LSHIS pueda ser abordado utilizando un paradigma distinto (*streaming*), DemoIS todavía puede ofrecer un gran margen de mejora. Una implementación paralela del algoritmo kNN o una nueva estrategia para la creación de subconjuntos podrían suponer,

no solo una mejora en el comportamiento del algoritmo, sino también una todavía más rápida ejecución. Además, existen algunas variaciones de los algoritmos que no han sido implementadas en su versión paralela.

- **Continuar las mediciones en la nube:** Aunque se han realizado numerosos experimentos para comprobar el rendimiento de los algoritmos propuestos, la mayoría de ellos se han realizado en una única máquina, limitando nuestros recursos o el tamaño de los conjuntos de datos a utilizar. Es cierto que se han realizado algunas pruebas en otro tipo de entornos, pero se han visto limitadas por problemas técnicos. Realizar más mediciones en un auténtico clúster permitiría realizar comparaciones con conjuntos de datos mucho más grandes, bastantes más recursos, y probablemente ayudase a definir más claramente el alcance del paradigma de programación en paralelo.

- **Mejorar métodos de lectura/escritura:** Un aspecto que no ha sido tratado durante este proyecto ha sido la manera en la que leer o almacenar los conjuntos de datos. Técnicas como la lectura de datos de un sistema de ficheros distribuido o de una base de datos podrían ser de mucha utilidad en este área y, por lo tanto, podrían ser abordados en futuras mejoras.

- **Seguir ampliando el número de algoritmos de minería de datos:** La estructura del proyecto y el campo en el que se desarrolla, dejan la puerta abierta a la implementación de multitud de algoritmos que se ejecuten en paralelo, pues no existe actualmente una gran librería que cubra este ámbito.

Bibliografía

[1] J. Alcalá Fdez, A. Fernandez, J. Luengo, J. Derrac, S García, L. Sánchez, and F. Herrera. KEEL Data-Mining Software Tool: Data Set Repository, Integration of Algorithms and Experimental Analysis Framework. *Journal of Multiple-Valued Logic and Soft Computing*, 17:255–287, 2011.

[2] J. Alcalá Fdez, L. Sánchez, M.J García, S. del Jesus, S. Ventura, J.M Garrell, J. Otero, C. Romero, J. Bacardit, M.V Rivas, J.C. Fernández, and F. Herrera. KEEL: A Software Tool to Assess Evolutionary Algorithms to Data Mining Problems. *Soft Computing*, 13:307–318, 2009.

[3] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[4] Álvar Arnáiz González, José Francisco Díez Pastor, César García Osorio, and Juan José Rodríguez Díez. Herramienta de apoyo a la docencia de algoritmos de selección de instancias. In *Jornadas de Enseñanza de la Informática*. Universidad de Castilla-La Mancha, 2012.

[5] Álvar Arnaiz-González, José F. Diez Pastor, César García Osorio, and Juan J. Rodrígez. LSH-IS: Un nuevo algoritmo de selección de instancias de complejidad lineal para grandes conjuntos de datos. In *Actas de la XVI Conferencia de la Asociación Española para la Inteligencia Artificial CAEPIA 2015*. Asociación Española para la Inteligencia Artificial, 2015.

[6] Troy Baer, Paul Peltz, Junqi Yin, and Edmon Begoli. Integrating apache spark into pbs-based hpc environments. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, page 34. ACM, 2015.

[7] André B. Bondi. *Characteristics of Scalability and Their Impact on Performance*. ACM, 2000. http://doi.acm.org/10.1145/350391.350432.

[8] Henry Brighton and Chris Mellish. Advances in instance selection for instance-based learning algorithms. *Data mining and knowledge discovery*, 2002. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.294.6862&rep=rep1&type=pdf.

[9] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th conference on Visualization'97*, pages 235–ff. IEEE Computer Society Press, 1997.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[11] Cedric Dugas. Agile workflow with GitHub issues, 2014. http://www.position-absolute.com/articles/agile-workflow-with-github-issues/.

[12] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37, 1996.

[13] Scala IDE for Eclipse. ScalaIDE for Eclipse. http://scala-ide.org.

[14] César García-Osorio, Aida de Haro-García, and Nicolás García-Pedrajas. Democratic instance selection: A linear complexity instance selection algorithm based on classifier ensemble concepts. *Artificial Intelligence*, 174(5–6):410 – 441, 2010. http://www.sciencedirect.com/science/article/pii/S0004370210000123.

[15] Vida Ha and Karau Holden. Everyday I'm Shuffling - Tips for Writing Better Spark Programs. https://docs.google.com/presentation/d/1efr89yUhiVXZtaJ7yQDjKpseQB7OEl_wGYm5Q5IwOo8/edit?usp=sharing.

[16] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Petter Reutemann, and Ian H. The weka data mining software: An update. *SIGKDD Explorations*, 11, 2009.

[17] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[18] IBM. The four v's of big data. http://www.ibmbigdatahub.com/infographic/four-vs-big-data.

[19] Google Inc. Google Cloud Dataproc. https://cloud.google.com/dataproc/.

[20] ISO. Software engineering – product quality. ISO 9126-1:2003, International Organization for Standardization, Geneva, Switzerland, 2008.

[21] Jim Jagielski. Apache Software Foundation-Organization summary, 2015. https://www.openhub.net/orgs/apache.

[22] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.

[23] Peter Kerpedjiev. Python vs. Scala vs. Spark, 2015. http://emptypipes.org/2015/01/17/python-vs-scala-vs-spark/,visited 2015-10-15.

[24] Jesús Maillo, Isaac Triguero, and Francisco Herrera. Un enfoque mapreduce del algoritmo k-vecinos más cercanos para big data. In *Actas de la XVI Conferencia de la Asociación Española para la Inteligencia Artificial CAEPIA 2015*. Asociación Española para la Inteligencia Artificial, 2015.

[25] Gil Press. A very short history of big data. http://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data/.

[26] Chris Snijders, Uwe Matzat, and Ulf-Dietrich Reips. "big data": Big gaps of knowledge in the field of internet science. *International Journal of INternet Science*, 7:1–5, 1997. http://www.ijis.net/ijis7_1/ijis7_1_editorial.pdf.

[27] Apache Spark. Spark 1.5.1 documentation. http://spark.apache.org/docs/1.5.1/.

[28] Wikipedia. Parallel computing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Parallel_computing&oldid=688312615,visited 2015-11-3.

[29] Wikipedia. Scrum (software development) — Wikipedia, The Free Encyclopedia. "https://en.wikipedia.org/w/index.php?title=Scrum_(software_development)&oldid=688230392", visited 2015-10-19.

[30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *University of California, Berkeley*, 2012. http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf.

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *University of California, Berkeley*, 2010. http://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf.