



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
Documentación Técnica



Presentado por Alejandro González Rogel
en Universidad de Burgos — 4 de febrero de 2016

Tutor: Álgvar Arnaiz González
Carlos López Nozal

Índice general

Índice general	I
Índice de figuras	III
Apéndice A Plan de Proyecto	1
A.1. Introducción	1
A.2. Planificación temporal	1
A.3. Estudio de viabilidad	5
Apéndice B Especificación de Requisitos	9
B.1. Introducción	9
B.2. Objetivos generales	9
B.3. Catalogo de requisitos	10
B.4. Especificación de requisitos	12
Apéndice C Especificación de diseño	21
C.1. Introducción	21
C.2. Diseño de datos	21
C.3. Diseño procedimental	22
C.4. Diseño arquitectónico	24
Apéndice D Documentación técnica de programación	33
D.1. Introducción	33
D.2. Estructura de directorios	33
D.3. Manual del programador	34
D.4. Compilación, instalación y ejecución	37
D.5. Otros aspectos relevantes a la hora de implementar	46
Apéndice E Pruebas del sistema	51
E.1. Introducción	51

E.2. Conjuntos de datos	52
E.3. Entorno de las pruebas	53
E.4. Comparativa entre la ejecución de clasificadores en Weka y Spark	53
E.5. Comparativa entre la ejecución secuencial y paralela, en una sola máquina, de LSHIS y DemoIS	58
E.6. Medición del tiempo de filtrado, en un clúster, de LSHIS y DemoIS	65
Apéndice F Documentación de usuario	67
F.1. Introducción	67
F.2. Requisitos de usuarios	67
F.3. Instalación	68
F.4. Instalaciones Opcionales	70
F.5. Manual del usuario	72
Bibliografía	91

Índice de figuras

A.1. Evolución del interés por Apache Spark en la plataforma Stack Overflow [7].	6
A.2. Interés de las empresas por apostar por el Big Data (2014) [8]. . .	7
B.1. Diagrama de casos de uso.	12
C.1. Diagrama del flujo principal del programa.	23
C.2. Diagrama de paquetes.	24
C.3. Diagrama de clases de la aplicación.	26
C.4. Diagrama de clases mostrando el patrón de diseño Estrategia. . . .	27
C.5. Diagrama de clases mostrando dos patrón de diseño Estrategia. . .	28
C.6. Diagrama de clases del paquete <i>instanceSelection.lshis</i>	30
C.7. Diagrama de clases del paquete <i>instanceSelection.demoIS</i>	31
D.1. Ejemplo de monitorización de una aplicación en marcha.	36
D.2. Pantalla de inicio de un clúster Standalone.	36
D.3. Ejemplo de la interfaz del servidor de históricos.	37
D.4. Diálogo para seleccionar un método de importación en Eclipse IDE. .	39
D.5. Diálogo para importar un proyecto al entorno Eclipse IDE.	40
D.6. Paquete importado en el menú lateral.	41
D.7. Ventana de configuración de ejecuciones en Scala IDE.	42
D.8. Ventana de configuración de ejecuciones en Scala IDE - Argumentos. .	43
D.9. Fragmento del menú superior de Scala IDE.	43
D.10. Ventana de configuración de Maven.	44
D.11. Página principal de la documentación con Scaladoc.	46
E.1. Evolución del tiempo de clasificación según el número de instancias clasificadas.	58
E.2. Hilos de ejecución de Spark en el uso del algoritmo Naive Bayes sobre “Poker” con 4 procesadores asignados.	58

E.3. Evolución del tiempo de ejecución según el tamaño del conjunto de datos usando LSHIS.	64
E.4. Evolución del tiempo de ejecución según el tamaño del conjunto de datos usando DemoIS.	64
F.1. Ejemplo del fichero resultante de una ejecución.	76
F.2. Ventana principal de la interfaz gráfica.	77
F.3. Diálogo para la selección de opciones de Spark.	78
F.4. Diálogo para la selección de un conjunto de datos.	79
F.5. Diálogo para la selección de filtro.	79
F.6. Diálogo para la selección de filtro después de seleccionar un algoritmo.	79
F.7. Panel para la configuración del clasificador y la validación cruzada.	80
F.8. Ventana principal de la interfaz gráfica rellena.	81
F.9. Creación de un nuevo proyecto en Google Cloud.	82
F.10. Página de inicio de un proyecto en Google Cloud.	83
F.11. Menú de selección de servicios de Google Cloud.	84
F.12. Pantalla principal de Google Cloud Dataproc totalmente vacía.	85
F.13. Pantalla principal de Google Cloud Dataproc tras la creación de un clúster.	86
F.14. Ejemplo de una tarea configurada.	87
F.15. Tarea en ejecución en Dataproc.	88
F.16. Pantalla principal de nuestro clúster en Dataproc.	88

Apéndice A

Plan de Proyecto

A.1. Introducción

A lo largo de este apéndice se va a presentar tanto la evolución temporal que ha seguido el proyecto durante su realización como aquellos aspectos de viabilidad que puedan afectar al trabajo si se deseara continuar con él en un futuro.

A.2. Planificación temporal

En esta sección vamos a presentar un resumen de lo que ha sido la realización del proyecto.

El desarrollo del proyecto se ha llevado a cabo mediante el uso de metodologías ágiles, realizando reuniones periódicas con los tutores para debatir sobre el desarrollo del trabajo y definir nuevos objetivos. Estas reuniones se han realizado, salvo excepciones, de manera bisemanal, y suponían el fin de un periodo de trabajo (*sprint*) y el comienzo del siguiente. Durante estos encuentros, además, se presentaba el trabajo realizado durante la última iteración.

Con el fin de no expandir más de lo necesario este anexo, solo se presentará una breve descripción del trabajo realizado en cada sprint del proyecto. Información más amplia sobre cada uno de los hitos y/o tareas puede encontrarse en la sección *Issues* del repositorio del proyecto, donde los hitos formarán un *issue* y las tareas, así como conversaciones entre los miembros del proyecto, pueden verse recogidas dentro de dicho *issue*.

Sprint 1 [29/09/15-13/10/15]

Partiendo de un estado de desconocimiento de la tecnología a utilizar durante el desarrollo, se ha invertido gran parte del tiempo en aprender el fun-

cionamiento de Spark y la terminología del área de trabajo.

Igualmente, se realiza el despliegue de todos los materiales necesarios para el desarrollo. Algunos de ellos no fueron necesarios para iteraciones siguientes, como Python o Hadoop, y han sido eliminados más adelante. Además, el despliegue se realizó por duplicado, al encontrar problemas con la ejecución de Spark en el sistema operativo Windows.

Finalmente, se presentan dos pequeños programas (uno en Java y otro en Scala) para comprobar el funcionamiento de Spark y poder decidir el lenguaje a usar durante la etapa de desarrollo.

Sprint 2 [13/10/15-29/10/15]

Se selecciona Scala como lenguaje de programación y comenzamos a aprender las bases de dicho programa.

Como objetivo principal de este sprint se plantea realizar una comparativa entre el tiempo de ejecución entre Weka y Spark con el algoritmo Naive Bayes. Todo ello genera una serie de hitos, definidos en las líneas sucesivas.

Se busca una serie de conjuntos de datos con diferentes características de tamaño/número de atributos, que además cumplan una serie de condiciones, como la ausencia de valores nulos. A la mayoría de los conjuntos seleccionados, pese a todo, hubo que pre procesarlos para que pudiesen funcionar sin problemas con Spark.

Se genera una pequeña clase lanzadora en Spark para poder lanzar y medir nuestros experimentos. Del mismo modo se realizan pequeñas modificaciones en el código de Weka para poder realizar correctamente mediciones.

Además, el comienzo de la documentación fuerza al aprendizaje de L^AT_EX.

Finalmente, se termina el sprint con la entrega del algoritmo en Scala utilizado para las mediciones y un informe con la explicación, resultados y conclusiones de la comparación entre Weka y Spark.

Sprint 3 [29/10/15-19/11/15]

A partir de este momento comenzamos a centrarnos en otro de los objetivos principales del proyecto: la implementación de algoritmos de selección de instancias.

Se implementa una primera versión del LSHIS y se generan diferentes métodos para realizar labores como la lectura de datos. Es la primera implementación con cierta complicación que realizamos tanto en Scala como para Spark, por lo que este hito ya se plantea con una carga de trabajo importante.

Al final del sprint contamos con una implementación de una primera versión funcional del algoritmo LSHIS.

Sprint 4 [19/11/15-10/12/15]

Se reestructura todo el código de la iteración anterior, creando nuevos paquetes, clases y relaciones de herencia. Además, se empieza a utilizar herramientas para el control estático de la calidad del código, lo que fuerza algunos cambios para adaptarnos a los estándares fijados.

El propio algoritmo LSHIS sufre una mejora, permitiendo ahora ejecutarlo con uno o varios componentes OR.

Se comienza la implementación del algoritmo DemoIS, pero queda suspendida a la espera de contar con una implementación paralela del algoritmo k -Nearest Neighbors (k NN).

Igualmente frenada por esta espera del algoritmo k NN, se plantea una comparación entre nuestra implementación LSHIS y la ya realizada en Weka, que nunca llega a producirse dentro de este sprint.

Se crea una máquina virtual con todos los materiales necesarios para el despliegue y distribución sencilla de nuestra aplicación.

Al finalizar este sprint contamos con una versión mejorada del algoritmo LSHIS, una mejor estructuración de nuestro código y un entorno para la fácil distribución de nuestro trabajo.

Sprint 5 [10/12/15-22/12/15]

Se implementa de manera completa el algoritmo DemoIS. Esto obliga a la creación de componentes como el algoritmo k NN, cuya implementación paralela no pudimos conseguir.

Se realizan pruebas sobre nuestro algoritmo LSHIS. Dichas pruebas generan un número considerable de problemas y, finalmente, cuando conseguimos realizar las mediciones, éstas arrojan malos resultados.

La máquina virtual sufre modificaciones para facilitar la ejecución del proyecto dentro de la misma.

Tras el sprint, podemos mostrar un nuevo algoritmo de selección de instancias y una máquina virtual mejor preparada para permitir la ejecución del proyecto.

Sprint 6 [22/12/15-08/01/16]

Se realiza una primera implementación de la interfaz gráfica que acaba por afectar a prácticamente la totalidad de la biblioteca, creando nuevas clases, métodos y ocasionando algún cambio en el flujo del programa.

Las pruebas sobre nuestros algoritmos siguen siendo problemáticas y continuamos trabajando sobre ello. Durante este periodo se encuentran una serie de

errores que afectaban a la manera en la que medíamos el tiempo de ejecución, así como unos errores en los algoritmos de Weka con los que intentábamos comparar nuestros resultados.

Finalmente, comenzamos a estudiar como lanzar nuestra ejecución en algún servicio de computación en la nube, pero no llegamos a finalizar la tarea al considerar más prioritario acabar con los problemas descritos anteriormente.

Por lo tanto, al terminar esta etapa contamos con una nueva manera de ejecutar nuestra aplicación (mediante la interfaz gráfica) y una serie de errores importantes corregidos, tanto en nuestra aplicación como en la librería de Weka facilitada.

Sprint 7 [08/01/16-25/01/16]

Se realiza una nueva iteración sobre nuestra implementación de los algoritmos, esta vez centrada en mejorar el tiempo de ejecución de los mismos, que continua siendo muy elevado con respecto a Weka. Es algo que se resolverá en este sprint.

Habiendo solventado todos los problemas, comenzamos, de nuevo, a realizar pruebas sobre el rendimiento de nuestro trabajo.

Igualmente, iniciamos otra tarea que había quedado suspendida, el despliegue del proyecto en un clúster real, lanzándolo finalmente en dos servicios diferentes: Google Cloud Dataproc [12] y un servicio clúster al que tiene acceso la Universidad de Burgos.

Durante este periodo también se trabaja en una gran refactorización del código, cuya calidad había disminuido considerablemente con todos los cambios introducidos en el sprint anterior.

Al finalizar esta iteración contamos con unos algoritmos que han demostrado funcionar correctamente y los resultados de una serie de comparativas que hemos realizado entre nuestras implementaciones y las proporcionadas para la librería Weka.

Sprint 8 [25/01/16-05/02/16]

Entre este periodo y el anterior, se mantienen una serie de ejecuciones de nuestros algoritmos en el servicio clúster al que tenía acceso la universidad, pero acaban cancelándose por problemas con los nodos.

Se realizan modificaciones en el código en vista a mejorar su calidad o corregir algún comportamiento indeseado ocasionado por algunas modificaciones en la iteración anterior.

Para terminar, se prepara la versión final de todos los materiales necesarios para la presentación del proyecto al finalizar este último sprint.

A.3. Estudio de viabilidad

El trabajo presentado a lo largo de este proyecto podría enfocarse de dos maneras distintas en esta sección: por un lado, podemos hablar de la viabilidad de los algoritmos implementados. Por otro, de las bases que este proyecto podría sentar para construir una librería con algoritmos de selección de instancias que corran en paralelo.

Sobre el primer punto, la viabilidad de los algoritmos, podemos basarnos en los estudios realizados (presentados en el anexo E) para concluir que, aunque una de nuestras implementaciones (LSHIS) no aporta una gran mejora en cuanto a tiempo de ejecución se refiere, la otra (DemoIS) puede llegar fácilmente a mejorar los tiempos de respuesta de su predecesora. Además, ambos algoritmos están pensados para su ejecución sobre grandes conjuntos de datos, algo que no pueden soportar las versiones anteriores de estos algoritmos y que nos proporciona una ventaja fundamental en un ambiente donde cada vez existen un mayor número de datos que analizar.

Sobre el proyecto como una base para formar una librería, supondría, no solo trabajar en un área en continua expansión, sino comenzar en un momento en el que no existen muchas otras aplicaciones similares que trabajen con Spark. Todo esto nos proporciona una gran flexibilidad para orientar el proyecto a donde se estime conveniente.

Finalmente, es interesante referirse también al posible futuro de la tecnología en la que se basa el proyecto: Spark.

Aunque Apache Spark es una librería cuya versión inicial se presentó a mediados de 2014, ha gozado de buena acogida desde entonces, siendo actualmente uno de los proyectos mejor valorados de la fundación Apache Software Foundation y uno de los más activos [13]. Igualmente, podemos observar en la gráfica A.1 como el interés por esta tecnología ha crecido en un corto periodo de tiempo hasta equipararse al nivel de muchas de las soluciones ya existentes en el mercado. Nos encontramos, por lo tanto, operando con una tecnología que promete seguir desarrollándose gracias a la aceptación con la que ha sido recibida.

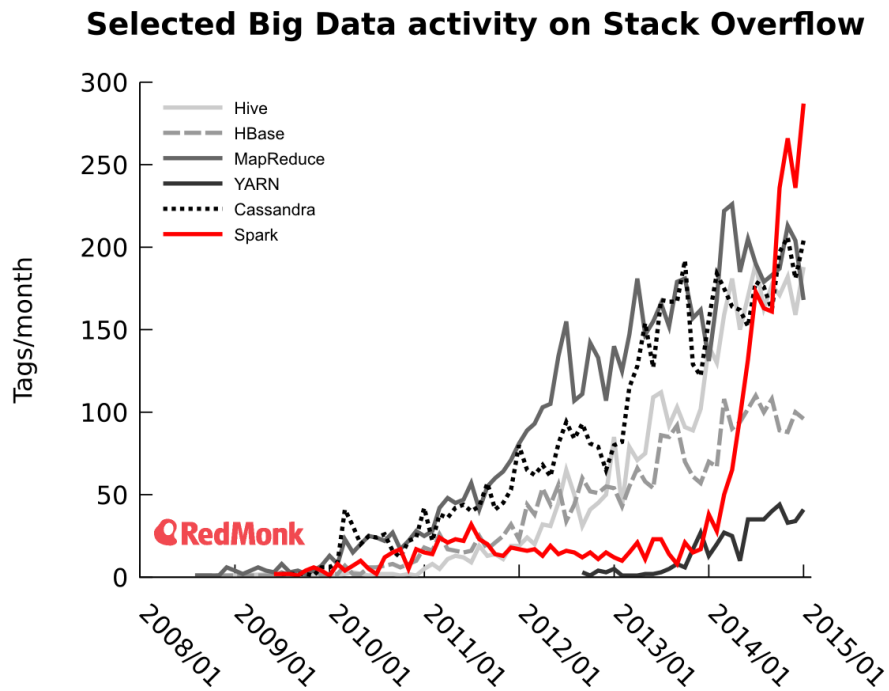


Figura A.1: Evolución del interés por Apache Spark en la plataforma Stack Overflow [7].

Viabilidad económica

Aunque este proyecto nunca ha sido enfocado a un aspecto de comercialización inmediata, podemos ver un escenario favorable en el sentido económico.

La minería de datos es un ámbito del que se espera rápida y fuerte expansión. Podemos observar en la gráfica A.2 como la intención de muchas de las empresas es la de continuar apostando por la inversión en minería de datos, que ha llegado a convertirse en una prioridad para una gran cantidad de empresas de sectores muy diversos [8].

Figure 1: Investments in Big Data analytics are strong

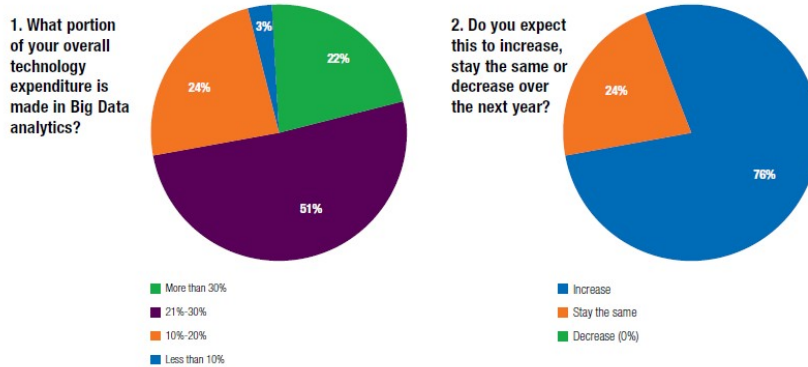


Figura A.2: Interés de las empresas por apostar por el Big Data (2014) [8].

Un campo en expansión puede proporcionarnos un buen terreno para obtener la rentabilidad económica, pero para ello necesitamos contar con alguna ventaja competitiva que nos permita diferenciarnos del competidor. Podemos destacar la modernidad de la tecnología utilizada, que, al estar enfocada al llamado *big data*, consigue realizar tareas que soluciones anteriores no podían resolver en un tiempo prudencial. Además, dentro del tipo de software que oferta esta posibilidad, podemos destacar de Spark la capacidad para gestionar la memoria y agilizar con ello las operaciones realizadas.

En referencia a los servicios sobre los que ejecutar nuestra aplicación, nos encontramos en un punto en el que no es necesario contar con una enorme inversión inicial para hacerse con todo el hardware necesario (clúster con varios nodos, gran capacidad de memoria RAM, etc.). Actualmente, estos servicios pueden contratarse a empresas que ya cuentan con todos los recursos (tales como Google y su servicio Google Cloud), pagando solo por el uso que hacemos de ellos, lo que abre todavía más el mercado de empresas que podrían permitirse el uso de un clúster para ejecutar tareas de minería de datos.

Clientes objetivo y licencias de distribución

Aunque, como se ha mencionado, nunca se ha tenido en mente enfocar este producto a ningún aspecto de comercialización concreto, podemos distinguir dos grupos de clientes sobre los que podría trabajarse:

- **Educación e investigación:** Al igual que otras alternativas existentes en el mercado (como Weka [10] o KEEL [2, 1]), nuestro proyecto podría abordar el área de la educación desde una nueva perspectiva: la creación de una librería de algoritmos de minería de datos paralelos.

En esta situación, se distribuiría el producto utilizando la licencia GNU GPL (*GNU General Public License*), por la ventaja que esto podría proporcionar para estudiar, modificar o distribuir nuevos materiales a partir de nuestro producto. Además, este tipo de licencia garantizaría que cualquier material generado como una iteración del proyecto siga manteniendo la misma licencia y, por lo tanto, pueda seguir siendo útil en el área de la educación.

- **Empresas que manejen grandes volúmenes de datos:** Aunque la minería de datos es algo que parece estar cogiendo fuerza en muchas áreas, nuestra ventaja competitiva reside en la capacidad para tratar problemas de *big data* (grandes conjuntos de datos) de manera eficiente. Es por ello que nuestro objetivo deberían ser aquellas organizaciones, por lo general de gran tamaño, que tengan el problema de tratar con tal cantidad de datos.

En este supuesto, el proyecto podría orientarse para actuar como una capa intermedia entre la empresa y un servicio de computación de la nube. Por las características ya mencionadas de estos servicios de computación, nuestro producto debería ofrecerse de manera que el usuario adquiriese derecho a uso por un periodo de tiempo limitado, pudiendo renovar o cancelar la suscripción pasado ese tiempo y haciendo más asequible el precio final. Cabe destacar que esta es una situación que el trabajo presentado, más orientado a la investigación, no está preparado para afrontar en la etapa actual, por lo que no se teorizará más sobre el posible desarrollo económico.

Viabilidad legal

En lo que se refiere a aspectos legales, este trabajo no se encuentra en problemática con ningún aspecto que pudiese considerarse de dudosa o nula legalidad, dado que se ha centrado en la implementación y pruebas de diferentes algoritmos.

Con respecto a la minería de datos, el problema más cuestionable es la protección de los datos analizados y la privacidad que se puede ofrecer si dichos datos hacen referencia a una persona o entidad concreta. Sin embargo, en su estado actual, nuestra aplicación no gestiona ni tienen interés en gestionar ningún aspecto de este terreno.

Especificación de Requisitos

B.1. Introducción

A lo largo de este apéndice, vamos a hacer referencia a todo lo relacionado con los objetivos del proyecto, indicando apropiadamente las metas y cuáles han sido los requisitos que han guiado el desarrollo de nuestro trabajo.

B.2. Objetivos generales

Podemos definir dos claros objetivos con los que se comenzó el proyecto:

- La implementación de una serie de algoritmos de selección de instancias que puedan ser ejecutados utilizando el nuevo paradigma de programación paralela que nos ofrece Spark. Estos algoritmos son: *Locally sensitive hashing instance selection* (LSHIS) [4] y *Democratic instance selection* (DemoIS) [9]
- La comparativa entre el rendimiento de las soluciones secuenciales de minería de datos frente a las nuevas soluciones de ejecución paralela que han surgido recientemente. Esta tarea se llevará a cabo utilizando una librería ya existente (Weka) para realizar ejecuciones que sigan el modelo secuencial, y Spark para realizar ejecuciones paralelas.

Las mediciones deberán estar enfocadas en dos sentidos: una comparación general del funcionamiento entre Spark y Weka, y la comparación concreta de nuestras implementaciones, definidas en el punto anterior, con sus homólogas para la librería Weka, que serán proporcionadas por los tutores del proyecto.

Igualmente, y conforme el proyecto iba avanzando, se consideró añadir una nueva serie de objetivos, de importancia menor, a nuestra aplicación:

- La creación de una interfaz gráfica más amigable y visual que no solo facilite el uso de todo el material generado sino que permita, además, definir baterías de experimentos para ser ejecutados al instante o en una máquina diferente.
- El despliegue del proyecto en el entorno donde está pensado para ser ejecutado: un clúster con diferentes nodos y múltiples unidades de procesamiento.

B.3. Catalogo de requisitos

A continuación se va a proceder a listar todas aquellas características exigidas al software que se ha implementado.

Se podrán distinguir entre requisitos funcionales, que son aquellos que definen un comportamiento específico esperado, y requisitos no funcionales, que son aquellos que, sin traducción directa en un comportamiento concreto, definen aspectos influyentes de la aplicación.

Requisitos funcionales

- **RF-1** La aplicación ha de permitir el lanzamiento, mediante línea de comandos, de ejecuciones de minería de datos que consten de un algoritmo de selección de instancias y un clasificador.
 - **RF-1.1** Se podrá indicar si se desea o no realizar la medición del tiempo de filtrado.
 - **RF-1.2** Se podrán indicar opciones para la correcta lectura del conjunto de datos. Esto implica indicar donde estará el atributo de clase y si el fichero contiene o no cabecera.
 - **RF-1.3** Se deberá indicar un algoritmo de selección de instancias, junto con su configuración, al iniciar la tarea.
 - **RF-1.4** Se deberá indicar un algoritmo de clasificación, junto con su configuración, al iniciar la tarea.
 - **RF-1.5** Se podrá indicar si se desea realizar validación cruzada en las ejecuciones. En caso de no indicarse nada, existirá una opción por defecto de usar el 10 % del conjunto de datos inicial como test para nuestras pruebas.
 - **RF-1.6** Se generará un fichero con el resumen de la ejecución: reducción del algoritmo de selección de instancias, precisión del clasificador y, si ha sido requerido, el tiempo de filtrado.

- **RF-2** La aplicación ha de permitir, vía interfaz gráfica, el lanzamiento de ejecuciones o baterías de ejecuciones que consten de un algoritmo de selección de instancias y un clasificador.
 - **RF-2.1** Se podrán seleccionar algunas de las opciones de lanzamiento de Spark (ruta al directorio de instalación de Spark, ruta al nodo maestro, número de núcleos, número de ejecutores y memoria de cada ejecutor).
 - **RF-2.2** Se podrán elegir uno o más conjuntos de datos, aunque en cada ejecución solo se use uno.
 - **RF-2.3** Se podrá elegir y configurar uno o más algoritmos de preprocesamiento, aunque en cada ejecución solo se use uno.
 - **RF-2.4** Se deberá seleccionar y configurar un algoritmo de clasificación.
 - **RF-2.5** Se podrán configurar las opciones para realizar una validación cruzada.
 - **RF-2.6** Se almacenará un fichero por cada lanzamiento con los datos de reducción y precisión de la ejecución.
- **RF-3** La aplicación ha de permitir, mediante la interfaz gráfica, definir ejecuciones o baterías de ejecuciones y exportar las definiciones de estos experimentos en un archivo .zip con los materiales necesarios.
 - **RF-3.1** Se podrán seleccionar algunas de las opciones de lanzamiento de Spark (ruta al directorio de instalación de Spark, ruta al nodo maestro, número de núcleos, número de ejecutores y memoria de cada ejecutor).
 - **RF-3.2** Se podrán elegir uno o más conjuntos de datos, junto con sus opciones para una correcta lectura de los mismos. Se utilizará un conjunto de datos por ejecución
 - **RF-3.3** Se podrá elegir y configurar uno o más algoritmos de preprocesamiento, usando uno por ejecución.
 - **RF-3.4** Se podrá seleccionar y configurar un algoritmo de clasificación.
 - **RF-2.5** Se podrán configurar las opciones para realizar una validación cruzada.
 - **RF-3.6** Se deberá generar un archivo .zip que contenga un script para ejecutar la batería de ejecuciones, junto con todos los conjuntos de datos necesarios para las ejecuciones.
- **RF-4** Debe existir una sección destinada a explicar el funcionamiento de la interfaz gráfica.

Requisitos no funcionales

- **RNF-1** La implementación no ha de tener problemas para correr en paralelo o en sistemas distribuidos formados por varios nodos.
- **RNF-2** El código ha de cumplir con una serie de medidas estáticas de calidad que faciliten el futuro mantenimiento del proyecto. Estas medidas son las especificadas por defecto por el programa Scalastyle (<http://www.scalastyle.org/>).
- **RNF-3** Los resultados de reducción y precisión generados por nuestros algoritmos de selección de instancias han de ser similares a los generados por la implementación secuencial de dichos algoritmos.
- **RNF-4** La aplicación ha de poder lanzarse en algún de los servicios de computación que ofrecen la posibilidad de correr Spark, concretamente Google Cloud Dataproc.
- **RNF-5** Ha de proporcionarse un sistema que facilite el uso de nuestra aplicación y la generación de baterías de ejecuciones sin necesidad de recurrir a la consola de comandos.

B.4. Especificación de requisitos

Diagrama de casos de uso

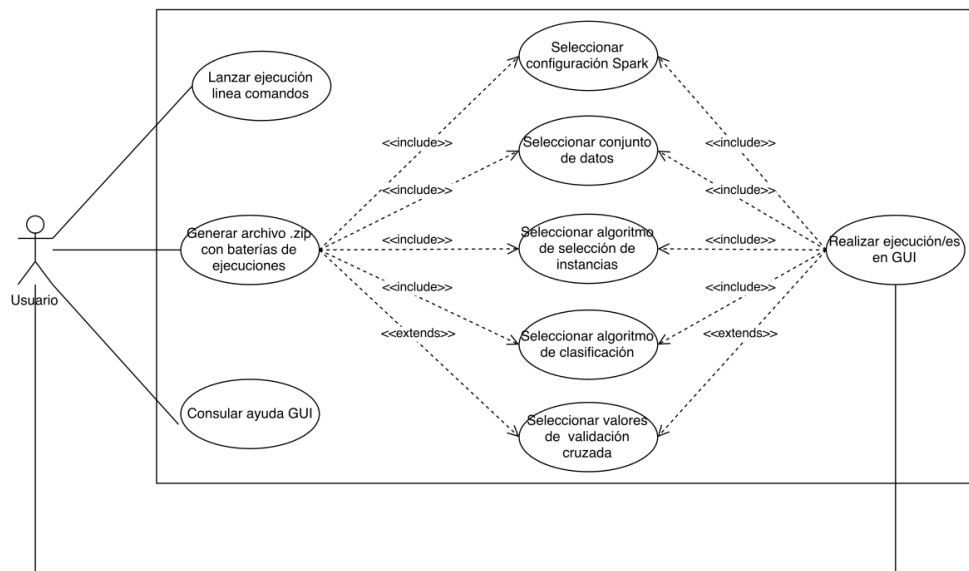


Figura B.1: Diagrama de casos de uso.

Caso de uso	Lanzar ejecución - Línea de comandos
Versión	1.0
Autor	Alejandro González Rogel
Requisitos	RF-1 RF-1.1 RF-1.2 RF-1.3 RF-1.4 RF-1.5 RF-1.6
Descripción	El usuario puede realizar la invocación, por línea de comandos, de un nueva tarea que implique un selector de instancias y un clasificador. Para ello necesitará aportar, además, una lista de parámetros que permitan la configuración de todos los componentes involucrados.
Precondiciones	Ninguna
Acciones	<ol style="list-style-type: none"> 1. El usuario realiza el lanzamiento de la aplicación mediante consola de comandos usando el script “spark-submit” proporcionado por Spark. 2. La aplicación realiza la ejecución. <ol style="list-style-type: none"> 2.1 Se muestra periódicamente, en mensajes por consola, el punto de la ejecución en el que nos encontramos. 2.2 La aplicación escribe los resultados de la ejecución en un fichero. 2.3 La aplicación muestra un último mensaje indicando la correcta ejecución.
Postcondiciones	Ninguna
Excepciones	Los parámetros introducidos en la invocación del programa son erróneos. Se paralizará la ejecución y se informará mediante un mensaje por consola de comandos si este fuese el caso.
Importancia	Alta

Cuadro B.1: Caso de uso “Lanzar ejecución - Línea de comandos”.

Casos de uso

Caso de uso	Realizar ejecución/es GUI
Versión	1.0
Autor	Alejandro González Rogel
Requisitos	RF-2 RF-2.1 RF-2.2 RF-2.3 RF-2.4 RF-2.5 RF-2.6
Descripción	Mediante el uso de la interfaz gráfica, el usuario puede programar una o varias ejecuciones y lanzarlas en el momento.
Precondiciones	El usuario ha iniciado la interfaz gráfica.
Acciones	1. El usuario completa los campos de opciones comunes de Spark. 2. El usuario indica una o varias configuraciones de Spark 3. El usuario indica uno o varios conjuntos de datos 4. El usuario selecciona uno o varios selectores de instancias. 5. El usuario selecciona un clasificador. 6. El usuario puede indicar opciones para la validación cruzada (opcional). 7. El usuario presiona un botón para ejecutar las tareas programadas. 8. La aplicación informa cuando se finalice la operación.
Postcondiciones	Los resultados de la ejecución han sido almacenados en un fichero de texto en la ruta correspondiente.
Excepciones	Alguno de los parámetros introducidos por el usuario son erróneos. En ese caso se cancela la ejecución o ejecuciones que se vean afectadas y se informa al usuario de lo ocurrido.
Importancia	Baja

Cuadro B.2: Caso de uso “Realizar ejecución/es GUI”.

Caso de uso	Generar archivo .zip con baterías de ejecuciones
Versión	1.0
Autor	Alejandro González Rogel
Requisitos	RF-3 RF-3.1 RF-3.2 RF-3.3 RF-3.4 RF-3.5 RF-3.6
Descripción	Mediante el uso de la interfaz gráfica, el usuario puede programar una o varias ejecuciones y generar un archivo de extensión .zip que contendrá un script de lanzamiento y todos los conjuntos de datos necesarios para la ejecución.
Precondiciones	El usuario ha iniciado la interfaz gráfica.
Acciones	1. El usuario completa los campos de opciones generales de Spark. 2. El usuario indica una o varias configuraciones de Spark 3. El usuario indica uno o varios conjuntos de datos 4. El usuario selecciona uno o varios selectores de instancias. 5. El usuario selecciona un clasificador. 6. El usuario puede indicar opciones para la validación cruzada (opcional). 7. El usuario presiona un botón para generar el archivo zip. 8. La aplicación informa cuando se finalice la operación.
Postcondiciones	Se ha generado un archivo de extensión .zip
Excepciones	Ninguna
Importancia	Media

Cuadro B.3: Caso de uso “Generar archivo .zip con baterías de ejecuciones”.

Caso de uso	Seleccionar configuración Spark
Versión	1.0
Autor	Alejandro González Rogel
Requisitos	RF-2 RF-2.1 RF-3 RF-3.1
Descripción	El usuario puede indicar una configuración de Spark rellenando un conjunto de campos de texto.
Precondiciones	El usuario ha iniciado la interfaz gráfica.
Acciones	1. El usuario completa los campos comunes para todas las configuraciones de Spark 2. El usuario presiona un botón para añadir una nueva configuración de Spark. 3. La aplicación muestra un nuevo diálogo. 4. El usuario rellena todos los campos mostrados en el nuevo diálogo. 5. El usuario presiona un botón para aceptar la nueva configuración.
Postcondiciones	Una nueva configuración ha de haberse añadido a la lista de configuraciones.
Excepciones	Ninguna
Importancia	Baja

Cuadro B.4: Caso de uso “Seleccionar configuración Spark”.

Caso de uso	Seleccionar conjunto de datos
Versión	1.0
Autor	Alejandro González Rogel
Requisitos	RF-2 RF-2.2 RF-3 RF-3.2
Descripción	El usuario puede indicar un fichero que contenga un conjunto de datos así como algunos parámetros que permitan su lectura.
Precondiciones	El usuario ha iniciado la interfaz gráfica.
Acciones	1. El usuario ha presionado un botón para añadir un nuevo conjunto de datos. 2. La aplicación muestra un nuevo diálogo. 3. El usuario selecciona el botón para buscar en los archivos del sistema el fichero deseado. Alternativamente, puede escribirlo manualmente. 4. El usuario cumplimenta, si lo ve necesario, todos los campos restantes mostrados en el diálogo. 5. El usuario presiona un botón para aceptar la nueva configuración.
Postcondiciones	Un nuevo conjunto de datos, junto con sus opciones de lectura, ha de haberse añadido a la lista de conjuntos de datos.
Excepciones	Ninguna
Importancia	Baja

Cuadro B.5: Caso de uso “Seleccionar conjunto de datos”.

Caso de uso	Seleccionar algoritmo de selección de instancias
Versión	1.0
Autor	Alejandro González Rogel
Requisitos	RF-2 RF-2.3 RF-3 RF-3.3
Descripción	El usuario puede seleccionar un algoritmo de selección de instancias de entre las posibles opciones y configurarlo para su ejecución.
Precondiciones	El usuario ha iniciado la interfaz gráfica.
Acciones	1.El usuario ha presionado el botón para añadir un nuevo selector de instancias. 2. La aplicación muestra un nuevo diálogo. 3. El usuario selecciona, tras pulsar el botón de selección, el filtro que desea. 4. La aplicación añade nuevos campos al diálogo. 5. El usuario cumplimenta los nuevos campos mostrados en el diálogo. 6. El usuario presiona un botón para aceptar la nueva configuración.
Postcondiciones	Un nuevo selector de instancias, junto con su configuración, ha de haberse añadido a la lista de conjuntos de datos.
Excepciones	Ninguna
Importancia	Baja

Cuadro B.6: Caso de uso “Seleccionar algoritmo de selección de instancias”.

Caso de uso	Seleccionar algoritmo de clasificación.
Versión	1.0
Autor	Alejandro González Rogel
Requisitos	RF-2 RF-2.4 RF-3 RF-3.4
Descripción	El usuario puede seleccionar un algoritmo de clasificación y configurarlo para su ejecución.
Precondiciones	El usuario ha iniciado la interfaz gráfica.
Acciones	1. El usuario presiona el botón que permite seleccionar entre los diferentes clasificadores posibles. 2. El usuario selecciona un clasificador. 3. La aplicación genera nuevos campos con las opciones de configuración del clasificador. 4. El usuario rellena los campos con las opciones de configuración.
Postcondiciones	Ninguna
Excepciones	Ninguna
Importancia	Baja

Cuadro B.7: Caso de uso “Seleccionar algoritmo de clasificación”.

Caso de uso	Seleccionar valores de validación cruzada.
Versión	1.0
Autor	Alejandro González Rogel
Requisitos	RF-2 RF-2.5 RF-3 RF-3.5
Descripción	El usuario puede indicar una configuración de validación cruzada con la que realizar las ejecuciones.
Precondiciones	El usuario ha iniciado la interfaz gráfica.
Acciones	1. El usuario marca la opción que habilita la validación cruzada. 2. El usuario cumplimenta los campos de configuración de la validación cruzada.
Postcondiciones	Ninguna
Excepciones	Ninguna
Importancia	Baja

Cuadro B.8: Caso de uso “Seleccionar valores de validación cruzada”.

Caso de uso	Consultar ayuda GUI
Versión	1.0
Autor	Alejandro González Rogel
Requisitos	RF-4
Descripción	El usuario puede consultar el funcionamiento de la interfaz gráfica mediante un botón de ayuda proporcionado en dicha interfaz.
Precondiciones	Se ha iniciado la interfaz gráfica.
Acciones	1. El usuario presiona sobre el botón de ayuda en la interfaz. 2. La aplicación muestra una nueva ventana con la ayuda.
Postcondiciones	Ninguna
Excepciones	Ninguna
Importancia	Baja

Cuadro B.9: Caso de uso “Consultar ayuda GUI”.

Especificación de diseño

C.1. Introducción

En este documento se especificarán todas aquellas cuestiones que tengan relación con la manera en la que han organizado los componentes que forman el proyecto, así como la debida justificación de dicha organización.

C.2. Diseño de datos

Cuando en este proyecto hablamos de datos, es obligatorio referirse a los conjuntos de datos que estamos indicados a analizar. Es por ello que se ve necesario definir cuáles han de ser las características necesarias para que los mencionados conjuntos sean analizados correctamente.

A lo largo de esta sección vamos a tratar diferentes aspectos referentes al formato que han de cumplir los datos para que nuestro proyecto funcione sin problemas.

Estructura de los conjuntos de datos

En primer lugar, hemos de hablar del almacenamiento. En *Big Data* es habitual contar con grandes conjuntos de datos almacenados en un sistema de ficheros distribuido al que tienen acceso una gran cantidad de nodos. En nuestro caso, el conjunto de datos inicial puede estar en un sistema de ficheros local (pero replicado en todos los nodos) o ser leído de un sistema de ficheros HDFS (Hadoop Distributed File System) o cualquier otro sistema de ficheros distribuidos soportado por Hadoop. A nivel local se ha trabajado con la primera opción, pues solo contamos con un único nodo. Cuando se ha utilizado la computación en la nube de Google Cloud Dataproc contábamos con el sistema de ficheros distribuidos que proporciona el servicio Google Cloud Storage.

En segundo lugar, es necesario mencionar el formato del fichero destinado a ser leído por el programa. Dicho fichero podrá contener una cabecera, que deberá ser eliminada por el programa, y un conjunto de instancias a razón de una instancia por línea. Los atributos de cada instancia estarán separados por comas, pudiendo estar el atributo de clase en primera o última posición.

Preprocesamiento de los datos

Spark, y en particular MLlib, es una librería relativamente moderna que todavía no proporciona soporte para muchos tipos de operaciones. En lo que a este proyecto se refiere, se ha notado la falta de posibilidades para preprocesar los datos de entrada que utilizamos para nuestros algoritmos.

Dado que implementar este tipo de funcionalidades podría suponer una carga de trabajo aún mayor, los conjuntos de datos utilizados necesitan cumplir algunos requisitos para ser correctamente tratados:

- El conjunto de datos ha de contener solamente atributos numéricos, y esto incluye el atributo de clase.
- El conjunto de datos debe haber sido normalizado con anterioridad para una correcta ejecución.
- No han de existir atributos nulos.

C.3. Diseño procedimental

Esta sección estará dedicada a definir y detallar el flujo general de nuestro programa.

No se va a dedicar especial atención al funcionamiento de los algoritmos de selección de instancias implementados (LSHIS y DemoIS). La razón a esto es que su pseudocódigo, junto con un esquema de su funcionamiento en paralelo, ya fueron mostrados en la memoria principal del trabajo.

Diagrama de flujo de una ejecución

En la imagen [C.1](#) podemos ver el diagrama de flujo que define cómo se gestiona el lanzamiento de una tarea de minería. Independientemente del método seguido para definir un lanzamiento, el diagrama va a ser siempre el mismo, simplemente hay diferentes maneras de llegar al mismo primer punto de inicio (consola o interfaz).

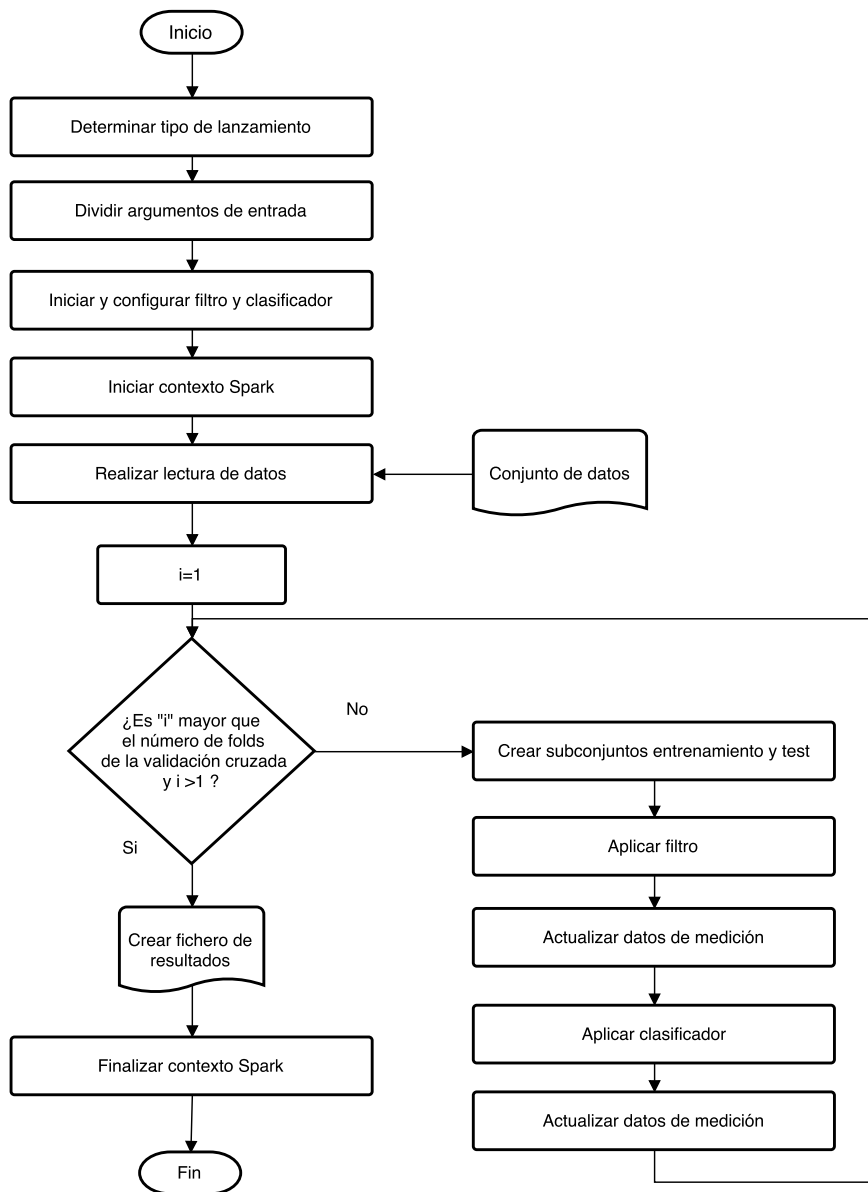


Figura C.1: Diagrama del flujo principal del programa.

Merece la pena recordar dos aspectos de lo que se habló en los aspectos relevantes de la memoria y que tienen influencia en el diagrama de flujo mostrado:

- Spark utiliza el paradigma de la “llamada por necesidad” o *lazy evaluation*. Esto significa que no todas las operaciones se realizarán tal y como el código especifica, sino que algunas operaciones no serán ejecutadas

hasta que sea estrictamente necesario hacerlo.

- Spark no almacena ninguna estructura RDD en ningún nivel de memoria, si no se indica explícitamente en el código. Esto puede dar lugar a que algunos datos deban recalcularse si no se persistieron en su momento o si, pese a ser persistidos, fueron parcial o totalmente borrados por falta de espacio. Igualmente, aunque indiquemos que deseamos persistir los datos hay múltiples maneras de hacerlo y múltiples escenarios posibles según qué opción. Como este aspecto depende fuertemente del tipo de persistencia configurado en Spark, del conjunto de datos o hasta del algoritmo usado, no se menciona explícitamente en el diagrama C.1.
- Es necesario indicar que existen actualmente dos modos de lanzamiento, ambos realizando la misma función. Su única diferencia es que uno de los dos modos realiza la medición del tiempo del filtrado, lo que le obliga a ejecutar una serie de operaciones enfocadas de manera ligeramente diferente y algunas otras operaciones “vacías” (con una carga de trabajo ínfima y sin propósito ninguno más que el de forzar la ejecución del programa hasta cierto punto). Este aspecto tampoco se recoge en el diagrama de flujo, pero puede ser consultado en la sección D.5 si se desea obtener más información.

C.4. Diseño arquitectónico

A lo largo de esta sección describiremos la estructura interna de nuestro programa, así como la comunicación o dependencia entre diferentes elementos.

Paquetes

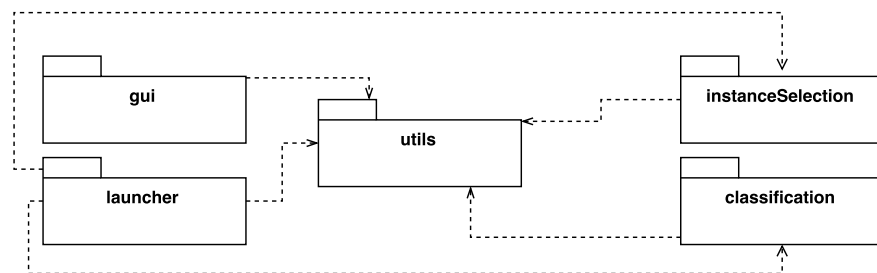


Figura C.2: Diagrama de paquetes.

- **gui:** Contiene todas aquellas clases relacionadas con la interfaz gráfica.

- **launcher:** Paquete que almacena todas las clases relacionadas con el lanzamiento de las ejecuciones. Por su propia finalidad, requiere estar relacionada con el paquete *utils*, *instanceSelection* y *classification*.
- **instanceSelection:** Contiene las implementaciones de algoritmos de selección de instancias. Uno de sus subpaquetes proporciona un espacio donde alojar aquellos algoritmos que funcionen de manera secuencial.
- **classification:** Contiene las implementaciones de los diferentes clasificadores. De nuevo, contiene un paquete donde poder almacenar algoritmos secuenciales.

Diagrama de clases

En la imagen C.3 podemos observar cómo se relacionan todas las clases que intervienen en la ejecución de las tareas de minería. Esto implica que aquellas clases relacionadas con la interfaz gráfica han sido omitidas.

Patrones de diseño

A continuación se van a analizar algunos patrones de diseño que pueden encontrarse en las clases que intervienen en lanzamiento de las tareas de minería.

Patrón de diseño Singleton

En el lenguaje de programación Scala existen dos tipos de clases, aquellas definidas mediante la palabra reservada “class”, que funcionan como, por ejemplo, en el lenguaje Java, y aquellas definidas mediante la palabra reservada “object”.

Estas clases “object” de Scala corresponden a un tipo especial de clase con una serie de restricciones que la hacen actuar como un *singleton*: No puede haber más de una instancia por clase, su constructor no puede tomar parámetros y la clase es accesible para cualquier otro elemento del programa del programa.

No existe intención de definir el funcionamiento del lenguaje de programación Scala, pero es necesario saber que cualquier clase que contenga un método *main* o extienda de *scala.App*, deberá ser una clase de tipo “object”.

Por lo dicho, encontramos este patrón de diseño en las clases dedicadas a iniciar la ejecución: `launcher.ExperimentLauncher` y `gui.SparkISGUI`.

Patrón de diseño Estrategia

La intención de aplicar este patrón es poder seleccionar, en tiempo de ejecución, una determinada manera de realizar una acción (estrategia).

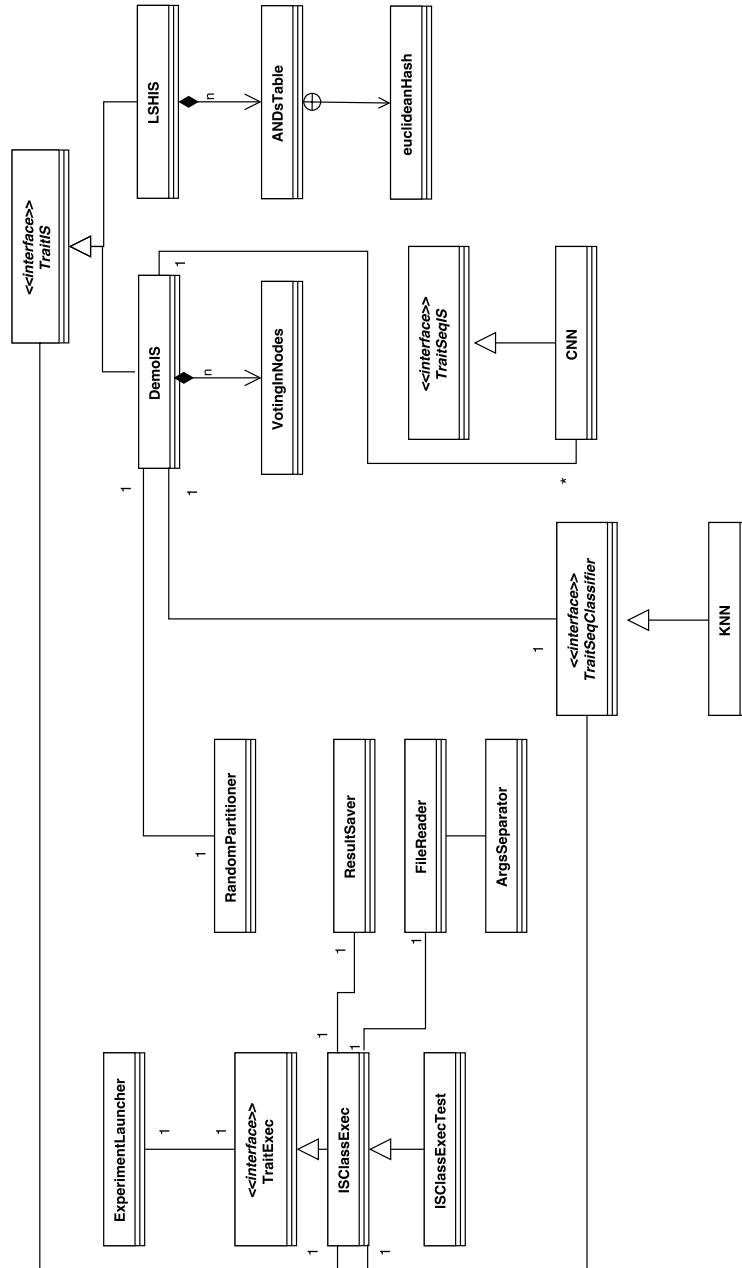


Figura C.3: Diagrama de clases de la aplicación.

Se puede apreciar este patrón de diseño en las clases relacionadas con el lanzamiento de una ejecución, donde el usuario puede haber indicado por parámetro qué estrategia desea utilizar (ver diagrama en C.4). Como puede deducirse del diagrama, actualmente existen dos posibles estrategias sobre las que elegir, aunque si la librería continuase creciendo podrían implementarse tantas estrategias como sean necesarias.

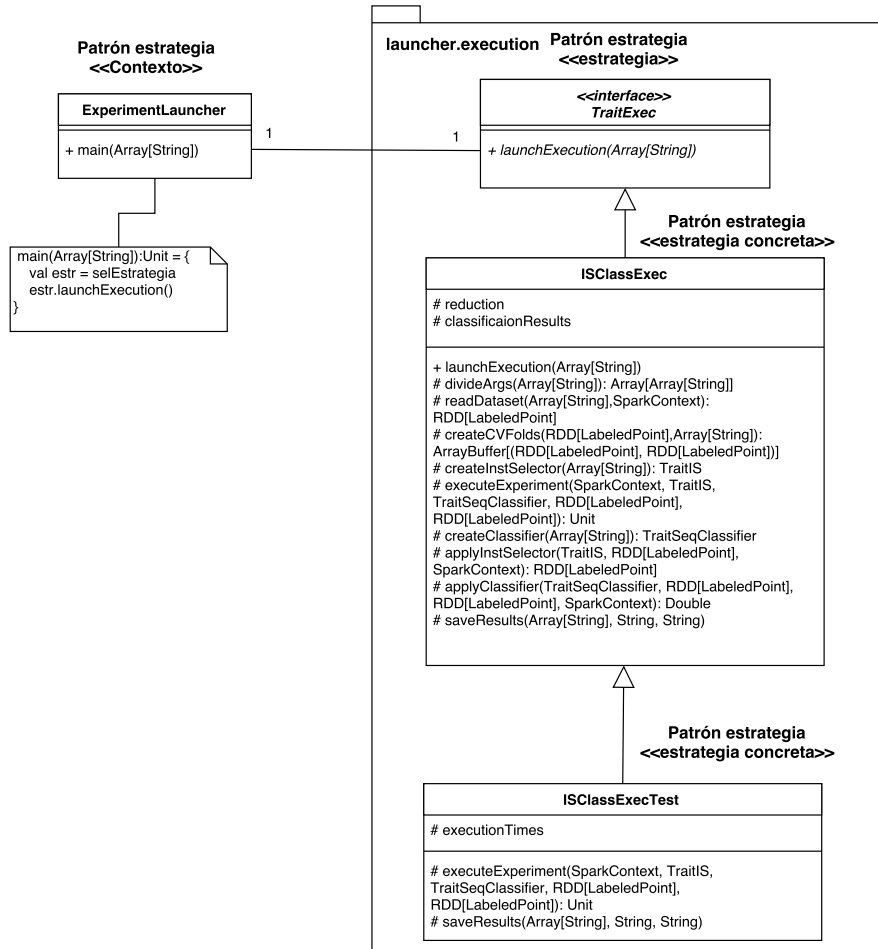


Figura C.4: Diagrama de clases mostrando el patrón de diseño Estrategia.

Del mismo modo, y con la misma finalidad, podremos encontrar un patrón de diseño estrategia entre las clases que realizan las labores de minería de datos y los algoritmos de selección de instancias y clasificación, tal y como muestra la imagen C.5. Esto nos posibilita, de nuevo, poder seleccionar un tipo de algoritmo en tiempo de ejecución. En este caso, el diagrama solo muestra aquellos métodos que tienen relación con el patrón concreto, por considerarse este diagrama con más elementos que el anterior.

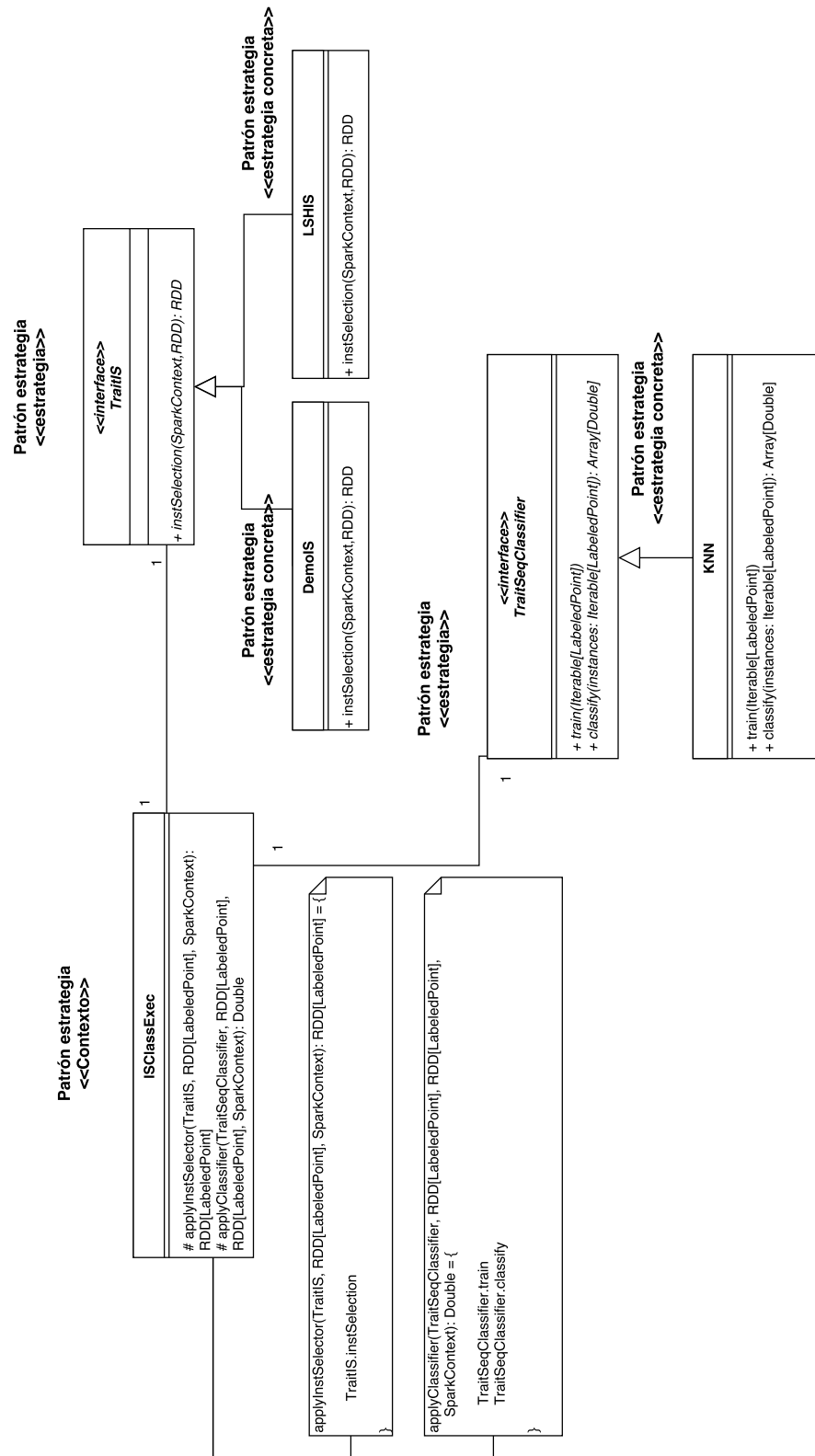


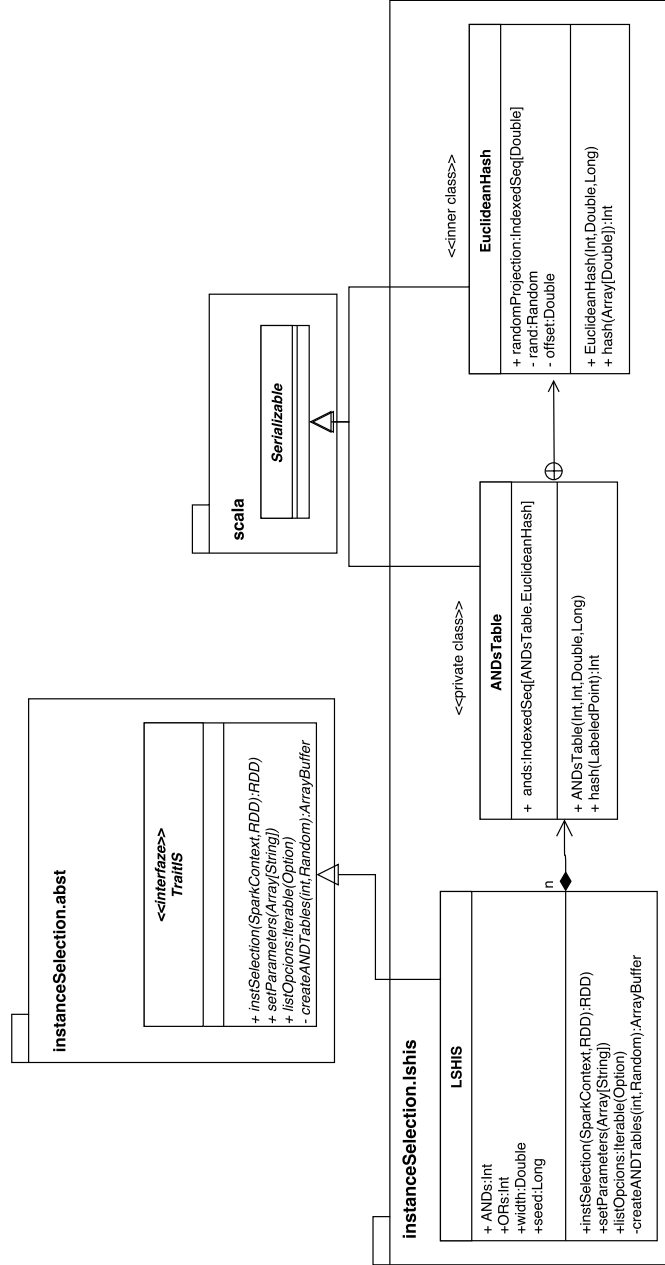
Figura C.5: Diagrama de clases mostrando dos patrón de diseño Estrategia.

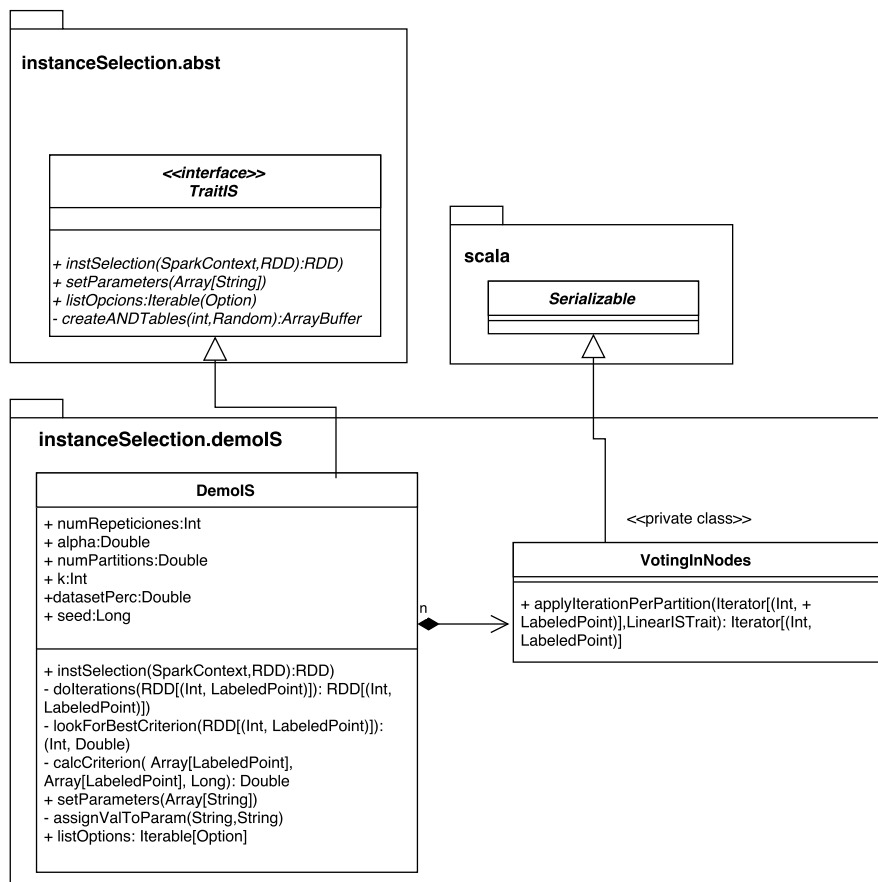
Serialización únicamente lo necesario

Uno de los problemas de diseño a los que nos hemos enfrentado durante la fase de implementación es el hecho de distribuir algunas clases por nuestra red de nodos.

Por norma general, hemos intentado serializar la menor cantidad de objetos posibles, algo que ha tenido una influencia en el diseño de los dos algoritmos de selección implementados: LSHIS y DemoIS. Podemos ver el diagrama de clases de los paquetes *instanceSelection.lshis* y *instanceSelection.demoIS* en las figuras C.6 y C.7 respectivamente.

Puede apreciarse, en los mencionados diagramas, que existe una clase principal no serializable que contiene el algoritmo, y una serie de clases anexas que heredan de la interfaz *scala.Serializable*. Estas últimas clases serán las que contendrán información que verdaderamente queremos distribuir. De esta manera, conseguimos dividir aquellos elementos que deseamos mantener en una única máquina de todos aquellos que son necesarios en la red de nodos para realizar las operaciones pertinentes.

Figura C.6: Diagrama de clases del paquete `instanceSelection.lshis`.

Figura C.7: Diagrama de clases del paquete *instanceSelection.demoIS*.

Documentación técnica de programación

D.1. Introducción

En este apéndice se van a describir todos aquellos aspectos que se consideren de interés para una persona que desee continuar el desarrollo del proyecto o quiera obtener un conocimiento más avanzado del mismo.

D.2. Estructura de directorios

A continuación se va a hacer mención a la funcionalidad de cada una de las carpetas que conforman el árbol de directorios de nuestro proyecto:

- **src:** Carpeta destinada a almacenar el código fuente la aplicación.
 - **main/scala:** Este directorio contendrá todos los ficheros fuente que estén escritos en Scala. Al no haberse usado ningún otro lenguaje ni contar con clases para realizar pruebas, este es el único directorio de la carpeta “src”.
- **resources:** Carpeta destinada a almacenar todos aquellos elementos que, sin ser código, también son necesarios para la ejecución del programa. Aquí podríamos incluir fichero .xml, html, imágenes o cualquier otro fichero multimedia, etc.
 - **gui:** Contiene todos los recursos utilizados por la interfaz gráfica.
 - **loggerStrings:** Almacena ficheros de texto con sentencias que pueden ser utilizadas por los diferentes *logger* en diferentes clases de nuestro programa.

- **target:** Contiene los archivos resultantes de aplicar alguna operación de Apache Maven sobre el proyecto (compilación, empaquetado o generación de documentación). Para más información al respecto visitar la sección [D.4](#), situada en este mismo anexo.
- **site/scaladocs:** Ruta que contiene la documentación, generada por Scaladoc, de las clases del proyecto.

D.3. Manual del programador

A lo largo de esta sección vamos a describir diferentes tareas que, si bien no se refieren directamente a la ejecución del programa, pueden conducir a un manejo más avanzado del mismo.

Antes de proceder es necesario comprender y haber realizado el proceso de instalación de todos los componentes mencionados en la sección [F.3](#), independientemente de si esos componentes han sido marcados como opcionales o no para el usuario normal.

Configurando Spark para guardar información sobre ejecuciones

Con la configuración por defecto, Spark apenas almacena información sobre las ejecuciones realizadas, pudiendo acceder a parámetros sobre la ejecución solo mientras el programa está corriendo. A lo largo de esta sección explicaremos que es posible modificar este comportamiento y la manera de lograrlo.

Lo primero que necesitamos es modificar algunas propiedades de Spark. En su carpeta de configuración (`$SPARK_HOME/conf`) debemos buscar un fichero por nombre “spark-defaults.conf”. Si es la primera vez que realizamos este tipo de operaciones es posible que este fichero no exista, haciendo necesario crearlo nosotros mismos o renombrar el fichero “spark-defaults.conf.template” que actúa de ejemplo y está situado en la misma carpeta.

Sea cual sea la opción elegida, en nuestro fichero de configuración “spark-defaults.conf” escribiremos las siguientes líneas:

```
spark.eventLog.enabled    true
spark.eventLog.dir        /opt/spark-1.5.1/logs/exec-logs
spark.history.fs.logDirectory /opt/spark-1.5.1/logs/exec-logs
```

La primera línea es obligatoria y no admite cambios. Las dos siguientes sirven para indicar el fichero donde deseamos almacenar la información de las ejecuciones. La ruta seleccionada es indiferente, pero es necesario destacar dos aspectos:

- Las dos rutas indicadas han de ser iguales.
- Las rutas han de apuntar a un directorio existente. Si indicamos una ruta inexistente Spark no generará los directorios que falten, sino que emitirá un error cuando intentemos ejecutar una aplicación y esta quiera guardar sus datos de ejecución. Es por ello que, en el caso concreto de la configuración ofrecida como ejemplo, deberíamos crear la carpeta “exec-logs” en `/opt/spark-1.5.1/logs` antes de proceder.

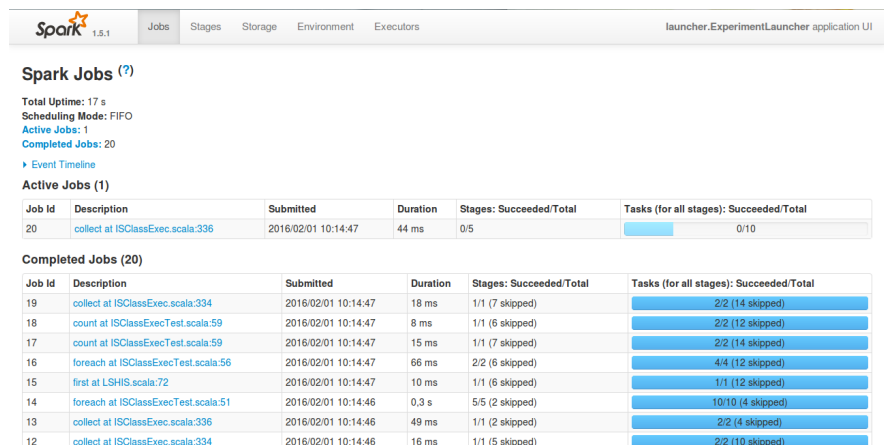
Con esto, deberíamos contar con toda la información de nuestras ejecuciones aun después de que estas hayan acabado. Será en la siguiente sección cuando hablaremos de cómo acceder a todos estos datos para su posterior análisis.

Monitorización y análisis

Un aspecto clave en el desarrollo de aplicaciones para *big data* es el de controlar y analizar el rendimiento de los algoritmos. Para ello, vamos a explicar en la siguiente sección como acceder a este tipo de información. No es objetivo de este anexo, sin embargo, explicar toda la información analizable ni la manera en la que ésta se organiza.

Antes de continuar, es necesario indicar que toda esta información proporcionada aquí corresponde a la manera de analizar una aplicación ejecutada en el modo *Standalone* de Spark (ver sección F.5 para más información). Si Spark se ejecutase usando algún administrador de clústeres el modo de acceder a la información podría ser distinto dependiendo del software usado. Igualmente, si ejecutamos en modo local no existe una manera de acceder a los datos de análisis una vez la aplicación ha finalizado su ejecución.

Con todo, comenzaremos contemplando la posibilidad de observar la ejecución de un programa mientras éste está todavía en ejecución. Esto puede hacerse accediendo, mediante el navegador, a la ruta <http://localhost:4040> (suponemos que nuestra máquina es el nodo maestro). Podemos ver un ejemplo de la página inicial en la imagen D.1. Esta información solo estará disponible mientras la aplicación lanzada siga ejecutando, y toda la información que podemos ver será borrada si no hemos seguido los pasos descritos en la anterior subsección.



Spark Jobs (?)

Total Uptime: 17 s
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 20

▶ Event Timeline

Active Jobs (1)

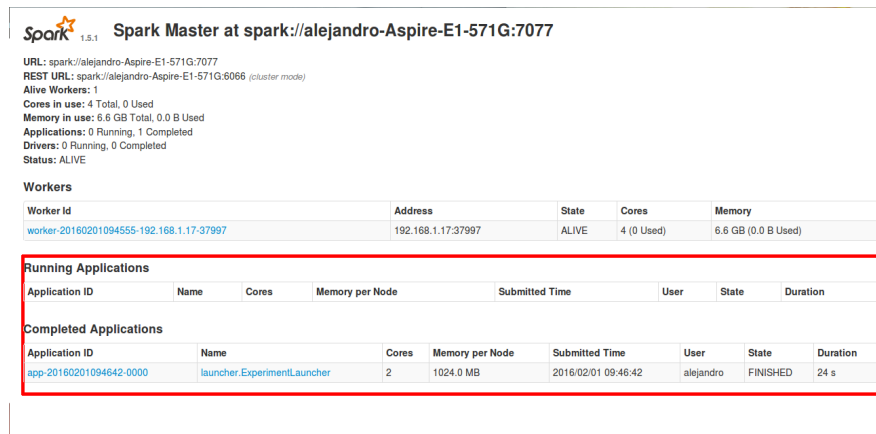
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
20	collect at ISClassExec.scala:336	2016/02/01 10:14:47	44 ms	0/5	0/10

Completed Jobs (20)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
19	collect at ISClassExec.scala:334	2016/02/01 10:14:47	18 ms	1/1 (7 skipped)	2/2 (14 skipped)
18	count at ISClassExecTest.scala:59	2016/02/01 10:14:47	8 ms	1/1 (6 skipped)	2/2 (12 skipped)
17	count at ISClassExecTest.scala:59	2016/02/01 10:14:47	15 ms	1/1 (7 skipped)	2/2 (14 skipped)
16	foreach at ISClassExecTest.scala:56	2016/02/01 10:14:47	66 ms	2/2 (6 skipped)	4/4 (12 skipped)
15	first at LSHIS.scala:72	2016/02/01 10:14:47	10 ms	1/1 (6 skipped)	1/1 (12 skipped)
14	foreach at ISClassExecTest.scala:51	2016/02/01 10:14:46	0.3 s	5/5 (2 skipped)	10/10 (4 skipped)
13	collect at ISClassExec.scala:336	2016/02/01 10:14:46	49 ms	1/1 (2 skipped)	2/2 (4 skipped)
12	collect at ISClassExec.scala:334	2016/02/01 10:14:46	16 ms	1/1 (5 skipped)	2/2 (10 skipped)

Figura D.1: Ejemplo de monitorización de una aplicación en marcha.

Suponiendo que queremos realizar un análisis de todas las aplicaciones que han sido lanzadas en la red de nodos en modo *Standalone*, podemos hacerlo consultado la dirección <http://localhost:8080>. Además de información referente a nuestro clúster, podemos observar un listado de aplicaciones ejecutadas o en ejecución, a cuya información se puede acceder pulsando sobre cualquiera de ellas (ver imagen D.2).



Spark Master at spark://alejandro-Aspire-E1-571G:7077

URL: spark://alejandro-Aspire-E1-571G:7077
REST URL: spark://alejandro-Aspire-E1-571G:8086 (cluster mode)
Alive Workers: 1
Cores in use: 4 Total, 0 Used
Memory in use: 6.6 GB Total, 0.0 B Used
Applications: 0 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160201094555-192.168.1.17-37997	192.168.1.17:37997	ALIVE	4 (0 Used)	6.6 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160201094642-0000	launcher.ExperimentLauncher	2	1024.0 MB	2016/02/01 09:46:42	alejandra	FINISHED	24 s

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160201094642-0000	launcher.ExperimentLauncher	2	1024.0 MB	2016/02/01 09:46:42	alejandra	FINISHED	24 s

Figura D.2: Pantalla de inicio de un clúster Standalone.

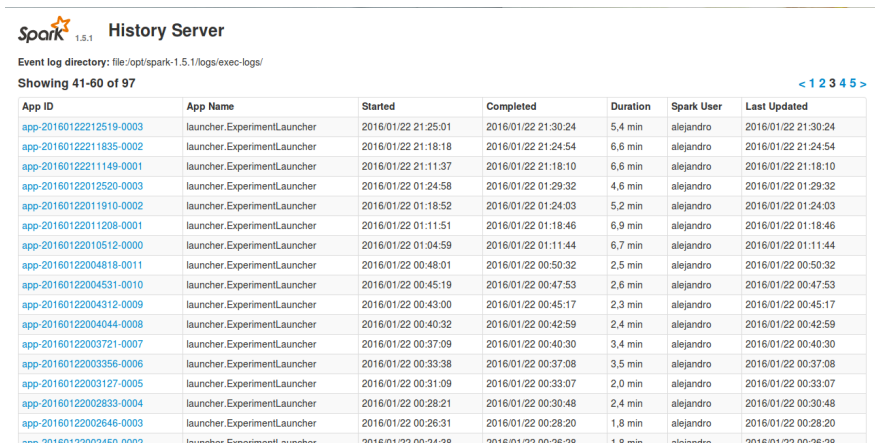
De nuevo, la ruta indicada es válida si únicamente estamos trabajando en la máquina local o somos el nodo maestro de la red. Igualmente, volvemos a recordar que hemos tenido que habilitar la opción de Spark para guardar información de las ejecuciones.

Por último, si deseamos acceder a toda la información de ejecuciones guardada en nuestra máquina, independiente de si continuamos formando parte

de una red de nodos o hemos sido el nodo maestro de varias redes, podemos accederlo iniciando el servicio “History Server” y accediendo a la ruta (<http://localhost:18080/>), donde se nos mostrará un listado con todas las aplicaciones ejecutadas de las que se tiene información. Para iniciar el servicio mencionado, desde consola de comandos, debemos escribir el comando:

```
$ $SPARK_HOME/sbin/start-history-server.sh
```

Podemos ver un ejemplo del histórico de ejecuciones en la imagen D.3.



The screenshot shows the Spark History Server web interface. At the top, it says "Spark 1.5.1 History Server". Below that, it indicates the "Event log directory: file:/opt/spark-1.5.1/logs/exec-logs/" and "Showing 41-60 of 97". There is a pagination control "< 1 2 3 4 5 >". The main content is a table with the following columns: App ID, App Name, Started, Completed, Duration, Spark User, and Last Updated. The table lists various applications, all with "App Name" as "launcher.ExperimentLauncher" and "Spark User" as "alejandro". The "Started" and "Completed" times are in YYYY-MM-DD HH:MM:SS format. The "Duration" is in minutes. The "Last Updated" time is also in YYYY-MM-DD HH:MM:SS format.

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
app-20160122212519-0003	launcher.ExperimentLauncher	2016/01/22 21:25:01	2016/01/22 21:30:24	5.4 min	alejandro	2016/01/22 21:30:24
app-20160122211835-0002	launcher.ExperimentLauncher	2016/01/22 21:18:18	2016/01/22 21:24:54	6.6 min	alejandro	2016/01/22 21:24:54
app-20160122211149-0001	launcher.ExperimentLauncher	2016/01/22 21:11:37	2016/01/22 21:18:10	6.6 min	alejandro	2016/01/22 21:18:10
app-20160122210520-0003	launcher.ExperimentLauncher	2016/01/22 01:24:58	2016/01/22 01:29:32	4.6 min	alejandro	2016/01/22 01:29:32
app-2016012221011910-0002	launcher.ExperimentLauncher	2016/01/22 01:18:52	2016/01/22 01:24:03	5.2 min	alejandro	2016/01/22 01:24:03
app-2016012221011208-0001	launcher.ExperimentLauncher	2016/01/22 01:11:51	2016/01/22 01:18:46	6.9 min	alejandro	2016/01/22 01:18:46
app-2016012221010512-0000	launcher.ExperimentLauncher	2016/01/22 01:04:59	2016/01/22 01:11:44	6.7 min	alejandro	2016/01/22 01:11:44
app-201601222004818-0011	launcher.ExperimentLauncher	2016/01/22 00:48:01	2016/01/22 00:50:32	2.5 min	alejandro	2016/01/22 00:50:32
app-201601222004531-0010	launcher.ExperimentLauncher	2016/01/22 00:45:19	2016/01/22 00:47:53	2.6 min	alejandro	2016/01/22 00:47:53
app-201601222004312-0009	launcher.ExperimentLauncher	2016/01/22 00:43:00	2016/01/22 00:45:17	2.3 min	alejandro	2016/01/22 00:45:17
app-201601222004044-0008	launcher.ExperimentLauncher	2016/01/22 00:40:32	2016/01/22 00:42:59	2.4 min	alejandro	2016/01/22 00:42:59
app-201601222003721-0007	launcher.ExperimentLauncher	2016/01/22 00:37:09	2016/01/22 00:40:30	3.4 min	alejandro	2016/01/22 00:40:30
app-201601222003356-0006	launcher.ExperimentLauncher	2016/01/22 00:33:38	2016/01/22 00:37:08	3.5 min	alejandro	2016/01/22 00:37:08
app-201601222003127-0005	launcher.ExperimentLauncher	2016/01/22 00:31:09	2016/01/22 00:33:07	2.0 min	alejandro	2016/01/22 00:33:07
app-201601222002833-0004	launcher.ExperimentLauncher	2016/01/22 00:28:21	2016/01/22 00:30:48	2.4 min	alejandro	2016/01/22 00:30:48
app-201601222002646-0003	launcher.ExperimentLauncher	2016/01/22 00:26:31	2016/01/22 00:28:20	1.8 min	alejandro	2016/01/22 00:28:20

Figura D.3: Ejemplo de la interfaz del servidor de históricos.

D.4. Compilación, instalación y ejecución

A lo largo de esta sección vamos a hablar de todos los pasos necesarios para poder trabajar sobre el código del proyecto. Esto incluye la instalación de diferentes componentes, la configuración del entorno de desarrollo y la comprensión de la herramienta Apache Maven como administrador en labores como la compilación.

Instalación

Si deseamos trabajar sobre el código del proyecto, además de necesitar todos los materiales definidos en la sección F.3 vamos a necesitar los definidos en este apartado.

Scala 2.11.7

Aunque podemos descargar la última versión de Scala desde la página oficial (<http://www.scala-lang.org/>), la descarga que se nos ofrece por defecto es un archivo .tar.gz que nos obligaría a realizar toda la instalación ma-

38 APÉNDICE D. DOCUMENTACIÓN TÉCNICA DE PROGRAMACIÓN

nualmente (suponiendo que estamos en un sistema Ubuntu como el utilizado durante el proyecto).

Por ello, vamos a realizar la instalación mediante un archivo `.deb` (paquete Debian) que también puede ser utilizado por nuestro sistema y que nos ahorrará realizar la instalación manualmente. Podemos acceder al repositorio que guarda este paquete desde un navegador (<http://www.scala-lang.org/files/archive/>) o descargarlo mediante el siguiente comando:

```
$ wget
  www.scala-lang.org/files/archive/scala-2.11.7.deb
```

Independientemente del método seguido, una vez tengamos el archivo en nuestro ordenador ejecutamos el siguiente comando desde la carpeta que contenga el paquete `.deb`:

```
$ sudo dpkg -i scala-2.11.7.deb
```

Si todo ha salido correctamente, una vez termine de ejecutarse la orden anterior podemos ejecutar el comando `scala -version` y esperar una salida similar a esta:

```
$ scala -version
Scala code runner version 2.11.7 -- Copyright
2002-2013
```

Recordar que, en el caso de utilizar cualquier otra versión de Scala, esta tiene que ser compatible con la versión Java que se encuentre instalada. Por ejemplo, Java 8 solo puede ser utilizado a partir de versiones 2.11 y será obligatorio a partir de la versión de Scala 2.12. [24]

Scala IDE

Scala IDE es el entorno de desarrollo utilizado durante el proyecto.

Descargaremos la última versión disponible del producto desde su página oficial: <http://scala-ide.org/>

Dentro de las posibles descargas que podemos seleccionar, elegiremos aquella que esté destinada a nuestro sistema operativo, en nuestro caso concreto, “Linux - 64 bits”. Al descargar esta herramienta, en realidad estamos descargando el entorno de desarrollo Eclipse (<https://eclipse.org/>) con una serie de *plugins* añadidos, entre los que destaca el llamado Scala IDE.

Una vez descargado, descomprimiremos el archivo en la ruta `/opt`. Mediante línea de comandos podemos hacerlo utilizando la siguiente sentencia:

```
$ cd /opt/ && sudo tar -zxvf \
~/Downloads/scala-SDK-4.2.0-vfinal-2.11-\
linux.gtk.x86_64.tar.gz
```

Recuerde que el código puede cambiar dependiendo del nombre del archivo que sea descargado o del directorio donde se encuentre.

Importando el proyecto en Scala IDE

Suponiendo que se ha seguido la instalación del entorno de desarrollo mencionado (Scala IDE) seguiremos los siguientes pasos para importar y ejecutar nuestro proyecto desde el propio programa.

Primero de todo, para importar el código a nuestro lugar de trabajo debemos dirigirnos al menú superior, opción “File” y, de entre las alternativas del menú desplegable, seleccionar “Import”. Alternativamente también podemos abrir el menú desplegable haciendo clic derecho sobre el panel de nombre “Package explorer”, situado generalmente a la izquierda de la pantalla.

Veremos un nuevo diálogo similar al mostrado en la figura D.4, donde seleccionaremos, entre todas las opciones, aquella que diga “Existing Maven Projects”. Posteriormente, presionamos sobre el botón “Next” para continuar.

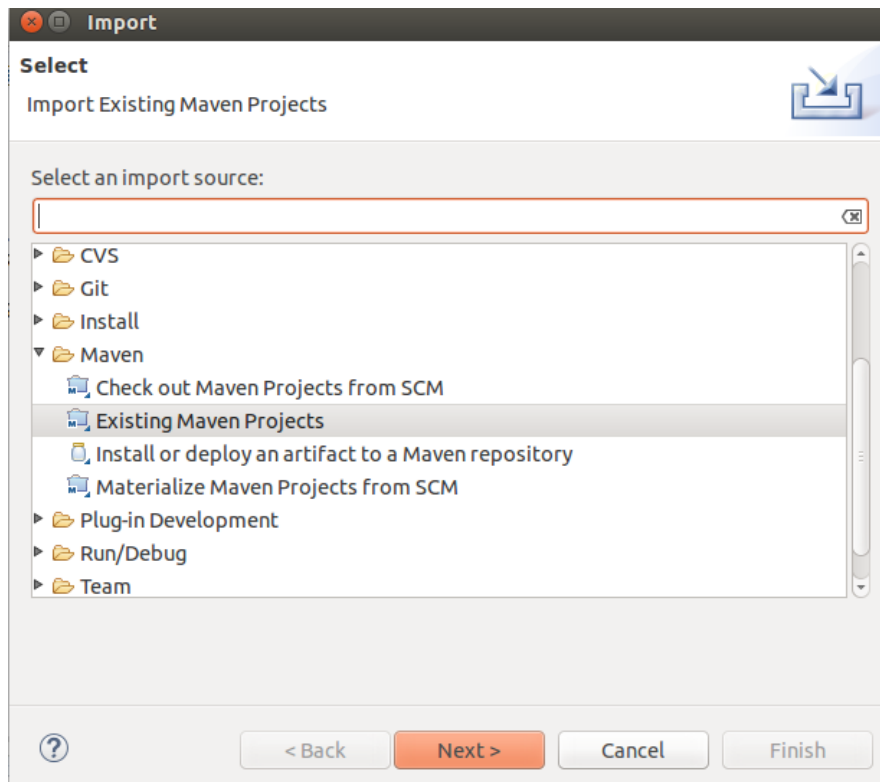


Figura D.4: Diálogo para seleccionar un método de importación en Eclipse IDE.

En el nuevo diálogo presionaremos sobre el botón “Browse” para indicar el directorio principal de nuestro proyecto. Si todo es correcto, una vez seleccionemos la carpeta que deseemos el resultado debería ser similar al mostrado en la imagen D.5.

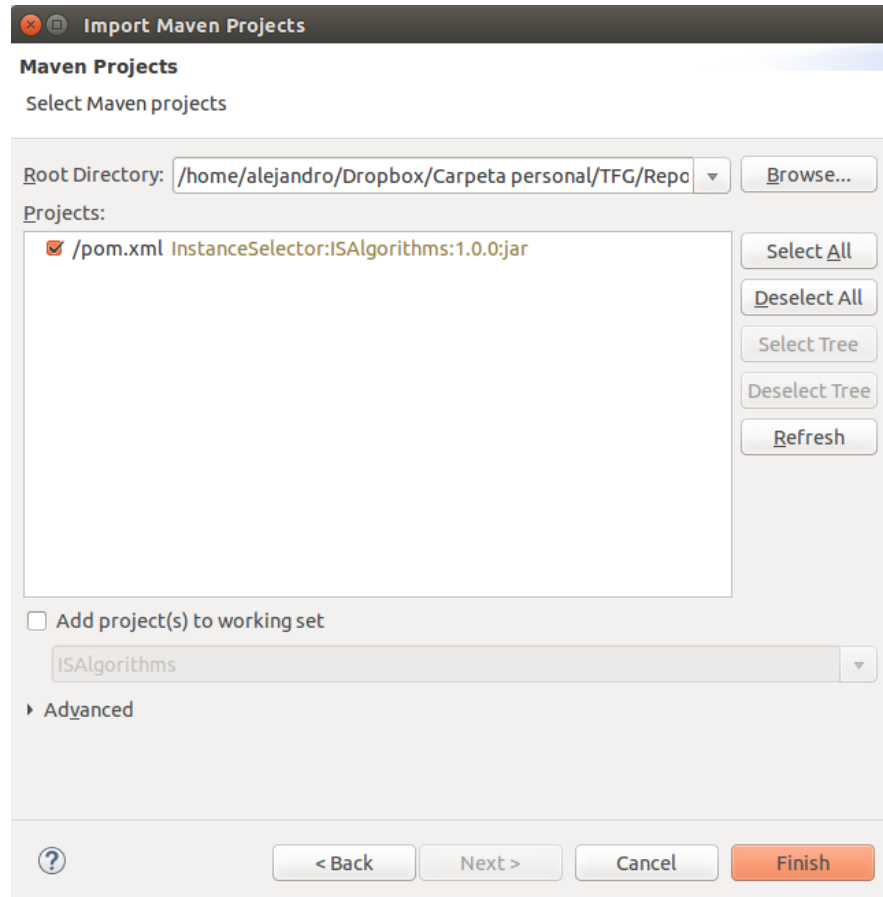


Figura D.5: Diálogo para importar un proyecto al entorno Eclipse IDE.

Pulsamos sobre el botón de finalizar y, si todo ha salido correctamente, deberíamos poder ver nuestro proyecto en el menú lateral izquierdo, tal como muestra D.6

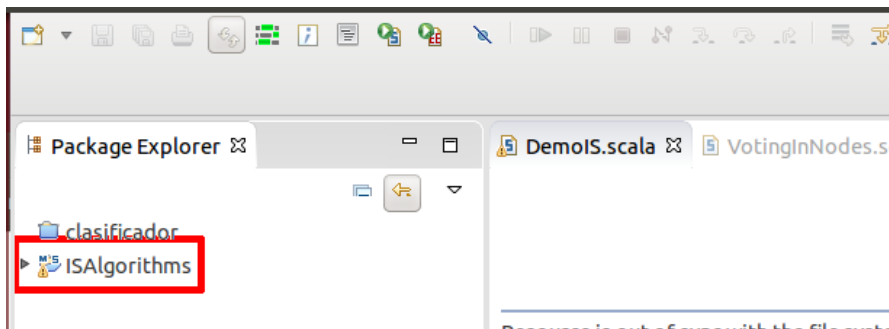


Figura D.6: Paquete importado en el menú lateral.

Ejecutando el proyecto en Scala IDE

Como puede observarse en el manual del usuario (F.5) el lanzamiento de una aplicación en Spark se realiza mediante un script que proporciona el propio Spark y en el cual se pueden indicar ciertos parámetros de ejecución. Por ello, para el lanzamiento del programa desde Eclipse hemos de modificar añadir algún fragmento de código que genere una configuración de Spark que, en cualquier otro método de lanzamiento, es creada por el script utilizado.

Para ello hemos de modificar un fragmento de código en la clase *ISClassExec* del paquete *launcher.execution*, tal y como puede verse en D.1 y D.2

```
val sc = new SparkContext()
```

Código D.1: Código para la ejecución del programa mediante el script “spark-submit”.

```
val master = "local[2]"
val sparkConf =
  new SparkConf().setMaster(master)
  .setAppName("Prueba_Eclipse")
val sc = new SparkContext(sparkConf)
```

Código D.2: Ejemplo de código para la ejecución del programa desde Eclipse.

Ahora, debemos indicar, de manera manual, cuál será la clase principal que deseamos lanzar. Es posible que esto cambie de acuerdo al tipo de ejecución que queremos realizar (lanzar la interfaz gráfica o un experimento de Spark), pero los pasos a seguir son idénticos.

Desde la pantalla de Scala IDE nos dirigiremos al menú superior, presionamos sobre “Run” y seleccionamos la opción “Run configurations”. Esto abrirá una nueva ventana similar a la mostrada en la figura D.7.

Presionando dos veces sobre el texto “Scala application”, en el menú izquierdo, podremos generar una nueva configuración para el proyecto. Allí rellenaremos los campos requeridos, con especial interés al campo “Main Class”, que deberá escribirse manualmente porque el entorno no detecta ninguna de las clases principales de los proyectos. En nuestro caso, ese campo siempre será *launcher.ExperimentLauncher* para lanzar experimentos y *gui.SparkISGUI* para lanzar la interfaz gráfica.

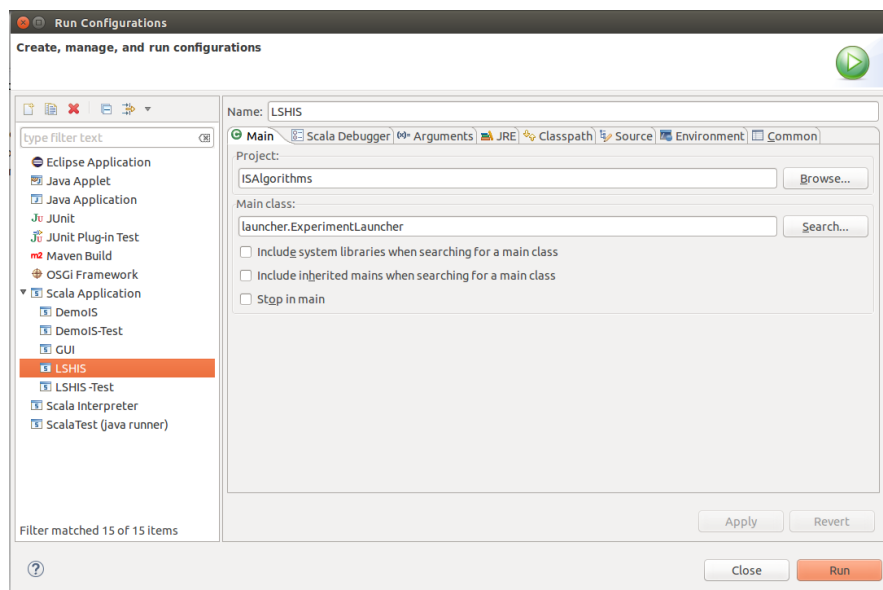


Figura D.7: Ventana de configuración de ejecuciones en Scala IDE.

Finalmente, si deseamos introducir argumentos al iniciar una ejecución, podemos hacerlo desde la pestaña “Arguments”, en la ventana anterior, rellenando el campo “Program arguments” con la información que deseemos (ver ejemplo en [D.8](#))

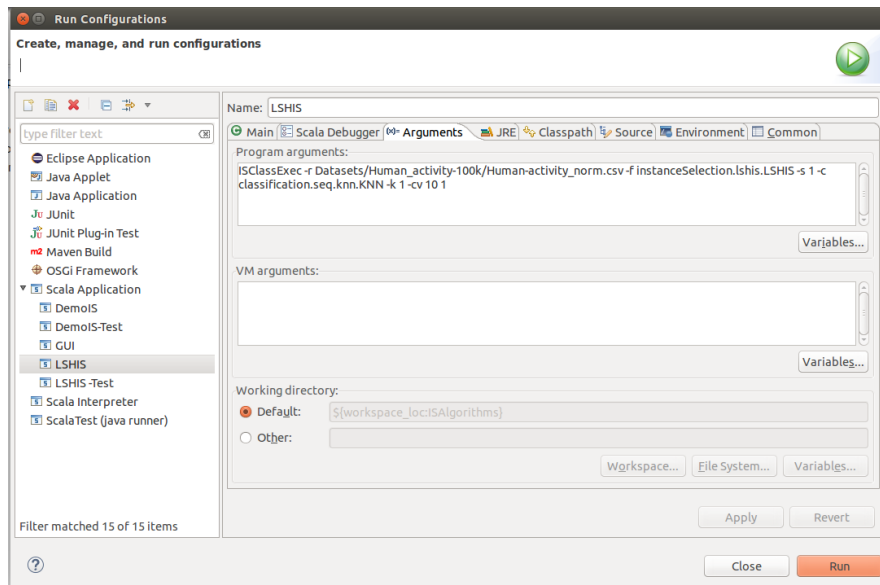


Figura D.8: Ventana de configuración de ejecuciones en Scala IDE - Argumentos.

Con todo definido, podemos presionar sobre el botón “Run” para comenzar a ejecutar nuestro programa.

Para posteriores ejecuciones no necesitaremos volver a definir de nuevo todo el proceso, sino que la opción será guardada y podrá ser accedida de manera sencilla seleccionándola desde el menú desplegable que proporciona el botón de ejecutar en el menú superior de Eclipse (ver figura D.9).

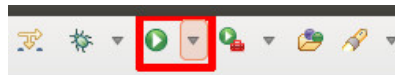


Figura D.9: Fragmento del menú superior de Scala IDE.

Si tenemos configurado el entorno para no realizar la compilación automática de las clases o hemos modificado algún recurso, debe realizarse una acción anterior al lanzamiento de cualquier ejecución. Desde la ventana principal del entorno de desarrollo, haremos clic con el botón derecho sobre la carpeta de nuestro proyecto, “Run as”, “Maven build”. Se nos abrirá una ventana similar a la mostrada en la figura D.10.

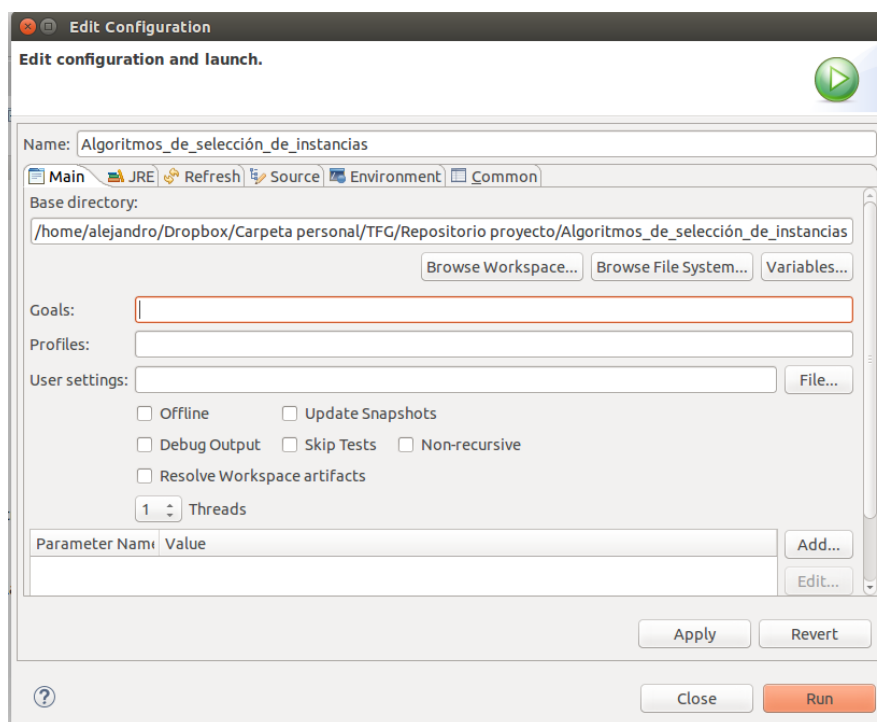


Figura D.10: Ventana de configuración de Maven.

Simplemente, necesitamos escribir en el campo de texto “Goals” los objetivos “clean package” y presionar sobre el botón “Run”. Esto debería realizar la compilación de todo el proyecto desde el entorno de desarrollo.

Compilación

Para realizar la labor de compilación vamos a necesitar del software Apache Maven, cuya instalación ha sido indicada en F.4. Una vez contemos con este requisito la operación será prácticamente automática.

Desde consola nos dirigimos al directorio raíz donde esté situado el proyecto. Una vez allí, ejecutamos el comando siguiente:

```
$ mvn clean package
```

Si es la primera vez que se realiza esta operación sobre el proyecto, o incluso la primera vez que se utiliza Maven, es muy probable que se requiera de algo de tiempo adicional mientras se descargan todos los materiales necesarios para poder llevar a cabo la labor de compilación.

Toda la información referente a la administración del proyecto (dependencias, fases, etc.) debe encontrarse en el documento situado en la carpeta raíz.

El nombre de este documento suele ser “pom.xml” pero, dado que en este proyecto existen diferentes distribuciones finales y un fichero de este tipo solo debería generar una única distribución, si deseamos utilizar cualquier otro fichero de los proporcionados para administrar el proceso deberemos ejecutar el comando:

```
$ mvn -f otro_pom.cml clean package
```

Actualmente se cuentan con los siguientes ficheros para generar distribuciones:

- **pom.xml:** Realiza una compilación y empaquetado de todo el material del proyecto. Debe encontrarse instalada la versión de Scala 2.11 para realizar la compilación.
- **pom_gui.xml:** Genera una distribución que contará únicamente con los componentes gráficos del sistema. Vuelve a ser requerido Scala 2.11 para la fase de compilación.
- **pom_cluster.xml:** Crea una distribución que contendrá todos los elementos menos los relacionados con la interfaz gráfica. La compilación ha de realizarse esta vez con Scala 2.10, por ser la versión que necesitan los servicios que hemos utilizado.

Generando documentación

De nuevo mediante el uso de Apache Maven, podemos obtener la documentación de las clases del proyecto utilizando el siguiente comando desde el directorio raíz del proyecto:

```
$ mvn clean scala:doc@Scaladoc
```

La documentación será generada en la carpeta *target/sites/scaladocs* utilizando el software “Scaladoc”.

Es necesario indicar que este generador de documentación tiene algunos inconvenientes:

- El programa ignora cualquier variable, método o clase privada, sin la posibilidad de poder revertir esta opción. Es por ello que cualquier material marcado como privado no aparecerá en los documentos generados.
- Por defecto, los ficheros generados mostrarán todos los métodos de una clase: los definidos y los heredados de cualquier otro componente, aunque no pertenezca a nuestra librería. Esto puede llegar a resultar molesto en algunas clases, como las pertenecientes a la interfaz gráfica. En este caso, podemos presionar sobre el botón “Hide all” que encontraremos en cada

fichero html generado y que permite ocultar todos aquellos métodos o atributos heredados de componentes que, por lo general, no suelen interesarnos.



Figura D.11: Página principal de la documentación con Scaladoc.

D.5. Otros aspectos relevantes a la hora de implementar

Por parte del programador, es necesario conocer una serie de aspectos que han influido en la implementación actual del código y que podrían seguir siendo influyentes si se continuase el trabajo.

Comprobar medidas estáticas de calidad

A la hora de trabajar sobre el proyecto, se ha utilizado una herramienta llamada Scalastyle (<http://www.scalastyle.org/>) para buscar aspectos del código que afecten negativamente al programa y, sobretudo, a las futuras labores de mantenimiento. Su funcionamiento se basa en comprobar una serie de reglas a lo largo de los ficheros fuente del proyecto. Dichas reglas estarán definidas en un fichero de nombre “scalastyle.config.xml” en el directorio raíz del proyecto.

El material proporcionado en el repositorio ofrece un archivo con las reglas utilizadas en este proyecto y se ha dejado la puerta abierta a poder chequear el código mediante la siguiente secuencia en consola:

```
$ mvn verify
```

Esta operación generará una salida bajo el nombre de “scalastyle-output.xml” que contendrá un informe con todos los casos en los que las reglas definidas no se estén cumpliendo.

Versión de Scala

Uno de los problemas a los que se enfrenta este proyecto se encuentra con la versión de Scala utilizada para compilar las clases. Scala 2.10 y Scala 2.11 no producen binarios compatibles, lo que impide que programas escritos en la versión anterior puedan ser leídos cuando esperamos clases compiladas para Scala 2.11 y viceversa.

Se comenzó el proyecto utilizando la versión de Scala 2.11 y, por lo tanto, se configuró Spark para trabajar con esa misma versión. Sin embargo, nos hemos encontrado con la necesidad de ejecutar el proyecto en sistemas donde Spark continua utilizando Scala 2.10.

Este cambio de versiones, además de afectar a la manera en la que debemos compilar el proyecto, tiene influencia a la hora de programar, pues podemos encontrar clases que cumplan su función en una versión y resulten en algún error en la siguiente.

Pensemos, por ejemplo, en la clase *scala.util.Random*, cuya versión en Scala 2.10 no es serializable, mientras que en su versión siguiente sí lo es. Esto hace que, en caso de querer distribuir dicha clase entre un grupo de nodos, la acción resulte en error con versiones anteriores a Scala 2.11. En este caso concreto, se ha suprimido el uso de la clase *scala.util.Random* para ser sustituido por *java.util.Random*, que no presenta este tipo de problemas.

No se han encontrado más clases que ocasionen errores de compatibilidad pero se advierte este problema como medida de precaución para futuras iteraciones.

Incluir nuevos algoritmos

Un aspecto que puede resultar de interés para el programador es conocer como añadir nuevos algoritmos a la librería. Esto es sencillo y únicamente consta de dos requisitos:

- En primer lugar, sea cual sea el tipo de algoritmo que deseamos implementar, ha de heredar de la clase “Trait”¹ que le corresponda, implementando todos los métodos que se reciben de esa clase.
- Los algoritmos de selección de instancias paralelos deberán heredar de *instanceSelection.abstr.TraitIS*.

¹Las clases “Trait” en Scala son similares a las clases “Interfaze” en Java

- Los algoritmos de selección de instancias secuenciales deberán heredar de *instanceSelection.seq.abstr.TraitSeqIS*.
 - Al no existir ningún algoritmo de clasificación paralelo, no existe interfaz definida de la que heredar.
 - Los algoritmos de clasificación secuenciales deben heredar de *classification.seq.abstr.TraitSeqClassifier*.
 - No existen otro tipo de elementos, como por ejemplo filtros de atributos, y por lo tanto tampoco no cuentan con una interfaz definida.
- Para que los algoritmos sean reconocidos por la interfaz gráfica han de ser definidos en unos ficheros .xml situados en la carpeta “resources” bajo el nombre de “availableFilters.xml” o “availableClassifiers.xml”. En estos ficheros habrá que indicar la ruta a nuestro algoritmo, así como alguna información sobre sus parámetros configurables. Podemos ver un ejemplo de la estructura de los ficheros .xml en el siguiente código:

```
<algorithms>
...
<algorithm>
  <name>paquete.sub1.sub2.Clase</name>
  <options>
    <option>
      <name>Opción 1</name>
      <description>Def</description>
      <command>-com</command>
      <default>1</default>
      <optionType>1</optionType>
    </option>
  </options>
</algorithm>
...
</algorithms>
```

La utilidad de cada etiqueta viene definida al inicio de los propios ficheros.

***k*-folds en la validación cruzada**

Uno de los problemas que ya hemos mencionado anteriormente tanto en otros anexos como en la propia memoria, es que Spark todavía se encuentra en un estado inicial donde algunos de los recursos que proporciona son escasos.

Quiero destacar el caso de la función *kfolds* que podemos encontrarnos en la clase *org.apache.spark.mllib.util.MLUtils* de la librería de Spark. Esta

función, en la versión que hemos trabajado con Spark, se encuentra en estado “experimental” según la propia documentación de la clase [18].

Se ha probado y demostrado durante este proyecto que la implementación actual de este método dista mucho de la implementación en Weka, la alternativa secuencial a Spark que hemos utilizado en este proyecto. Los k -folds generados por Spark dividen el subconjunto inicial de tal manera que el resultado de clasificación final de nuestros experimentos se ve gravemente perjudicado. Se atribuye esto a que la función en Spark no realiza divisiones estratificadas, esto es, divisiones donde la proporción de un determinado tipo de instancias en el subconjunto final es similar a su proporción en el conjunto completo.

Como la finalidad de este proyecto no ha sido la de profundizar sobre este aspecto, se ha implementado una manera de crear los k -folds utilizando otras operaciones de Spark (función “sample” de la estructuras RDD) y algunas operaciones sobre conjuntos. Si bien la solución aportada parece haber resuelto el problema, es necesario indicar que genera problemas en conjuntos de datos donde las instancias recogidas están repetidas muchas veces dentro del conjunto de datos.

Por todo esto, se anima al futuro programador a comprobar el funcionamiento de la función *kfolds* que proporcione Spark o a tener en cuenta las limitaciones de la implementación proporcionada.

Medición de tiempos de filtrado usando ISClassExecTest

Puede ser de interés para el programador conocer la razón por la cual la clase *launcher.execution.ISClassExecTest*, destinada a medir el tiempo de ejecución de la selección de instancias, posee una serie de operaciones que no se encuentran incluidas en la versión principal (*launcher.execution.ISClassExec*).

Consideremos el siguiente fragmento de código, encargado de realizar la medición del tiempo que tardamos en filtrar un conjunto de datos de nombre “train”

Pueden observarse varias cosas que no vemos en una ejecución normal:

- El conjunto de datos “train” se persiste antes de iniciar la medición. Esto se realiza para estar seguros de que, a la hora de medir los tiempos de ejecución, nuestro conjunto de datos estará distribuido completamente entre nuestros nodos.
- Se realizan una serie de operaciones sin aparente sentido, pues vemos hasta en dos ocasiones la sentencia *foreachPartition { x => None }* que, teóricamente, manda hacer “nada” a cada partición. En realidad, esta sentencia tiene como finalidad forzar a Spark a realizar todas las

```

train.persist()
train.name = "TrainData"
train.foreachPartition { x => None }

// Instanciamos y utilizamos el selector de
// instancias
val start = System.currentTimeMillis
val resultInstSelector =
    applyInstSelector(instSelector, train,
        sc).persist
resultInstSelector.foreachPartition { x => None }
executionTimes += System.currentTimeMillis -
    start

```

Código D.3: Código de medición de tiempo de filtrado

operaciones necesarias para calcular la estructura RDD sobre la que se está aplicando antes de poder continuar. Puede parecer un tema trivial pero eliminar la sentencia puede influir en las mediciones realizadas, pues en ocasiones medirían operaciones que no deseamos tener en consideración y que deseamos que se ejecuten antes de comenzar a medir el tiempo.

Es necesario comprender que la clase *ISClassExecTest* fue creada con la intención de medir de la manera más precisa posible los tiempos de ejecución del filtrado en Spark. Sin embargo, en tareas de *big data*, esto no es lo habitual, pues una precisión medida en milisegundos no tiene sentido cuando las ejecuciones pueden durar días. Es por ello que, para mediciones que no requieran gran precisión, se aconseja utilizar la información que presentan las herramientas de monitorización de Spark (ver subsección [D.3](#)).

Pruebas del sistema

E.1. Introducción

A lo largo de este proyecto, se han realizado una serie de mediciones cuyo objetivo principal consistió en comparar el comportamiento entre ejecuciones secuenciales y paralelas en tareas de minería. A lo largo de esta sección, vamos a mostrar todos los experimentos realizados, junto con una descripción de los elementos involucrados y las conclusiones extraídas.

Podemos diferenciar entre tres grandes comparaciones:

- **Comparativa entre las herramientas utilizadas:** Utilizando material ya incluido en las bibliotecas de Weka y Spark, se ha realizado una comparación entre ambas tecnologías. Tendremos en cuenta, no solo el tiempo de ejecución, sino otros factores tales como el uso de memoria o el funcionamiento de los hilos de ejecución (con especial interés en su comportamiento con Spark)
- **Comparativa entre las ejecuciones, en una sola máquina, de los algoritmos LSHIS y DemoIS según sea su implementación secuencial o paralela:** En este caso, nuestro objetivo principal será comparar tiempos de ejecución del algoritmo de filtrado, aunque también tendremos en cuenta parámetros otros parámetros que puedan demostrar el buen funcionamiento de nuestra implementación.
- **Comparativa entre las ejecuciones en un clúster de los algoritmos LSHIS y DemoIS según su implementación secuencial o paralela:** Al igual que en el caso anterior, el objetivo principal en este caso es medir el tiempo de ejecución de la operación de selección de instancias.

Nombre del conjunto	Instancias	Atributos	Clases
Iris	150	4	3
Image Segmentation [16]	2,310	19	7
Banana shaped [11]	5,300	2	2
Pen-Based Recognition of Handwritten Digits [16]	10,992	16	10
Letter Recognition [16]	20,000	16	26
Human Activity Recognition [22]	165,632	17	5
Coverttype [16]	581,012	54	6
Poker [16]	1,025,010	10	10
HIGGS [16] [6]	11,000,000 ¹	28	2

Cuadro E.1: Conjuntos de datos utilizados para la comparación entre Weka y Spark.

E.2. Conjuntos de datos

Los conjuntos de datos (*datasets*), ordenados de menor a mayor según el número total de instancias de cada uno, pueden encontrarse en la tabla E.1. No todos los conjuntos han sido utilizados en todas las comparaciones, puesto que algunas ejecuciones podrían no acabar en un tiempo prudencial. Por ello, en cada prueba vendrá especificado qué datasets han sido usados.

Indicar que, a la hora de seleccionar los conjuntos, se han elegido aquellos que compartan algunas características comunes:

- No existen campos de tipo texto.
- No existen campos vacíos en ninguna de las instancias de los atributos.

Además, es importante indicar que todos los atributos han sido normalizados antes de realizar las mediciones.

¹El conjunto original consta de 11,000,000 instancias, pero por lo general se he reducido su tamaño original para acercarlo más al tamaño de los otros conjuntos. Cualquier cambio será especificado en la medición a la que afecte.

E.3. Entorno de las pruebas

Aunque cada medición tiene una serie de características únicas que vendrán definidas en la subsección correspondiente, existen algunas circunstancias que son comunes a todas ellas:

- El formato utilizado en los ficheros que contienen datos ha sido `.arff` para Weka y `.csv` para Spark. La razón por la que no se han utilizado ficheros `.csv` en Weka ha sido por la posibilidad de que esto produzca errores a la hora de leer el archivo [23].
- Para todas las pruebas hemos usado una validación cruzada 10x2.
- Como ya ha sido mencionado anteriormente, todos los atributos de los conjuntos de datos utilizados han sido normalizados previamente.
- En ejecuciones locales (ver experimento E.4 y E.5) las pruebas se han realizado en un ordenador con capacidad para soportar hasta cuatro hilos simultáneamente y una memoria RAM de 8GB.
- En todas las ejecuciones hemos contado con memoria suficiente como para poder incluir en ella todo el conjunto de datos. Es necesario destacar que, pese a todo, este supuesto no es común dentro del *big data*.

E.4. Comparativa entre las ejecución de clasificadores en Weka y Spark

El objetivo final de esta comparación es observar el diferente funcionamiento de las librerías, centrando nuestro interés en cómo Spark puede mejorar el tiempo de respuesta a medida que añadimos nuevas unidades de ejecución.

Para realizar las mediciones hemos seleccionado una serie de conjuntos de datos y aplicado sobre ellos un algoritmo de clasificación: Naive Bayes.

Naive Bayes es un algoritmo de clasificación probabilístico y relativamente simple que ya se encontraba implementado tanto en la librería de Weka [14] como en la de Spark, razón por la cual ha sido elegido. En un principio hemos supuesto que no habría grandes diferencias en cuanto a tiempo de ejecución o recursos entre ambas implementaciones.

La ejecución del algoritmo se analizará tanto en Weka como en Spark, siendo en este último ejecutado con diferente número de hilos: 1, 2 y 4.

Criterios comparados

Los aspectos que hemos tenido en cuenta a la hora de recoger datos han sido:

- **Tiempo de ejecución:** El periodo analizado es aquel que incluye la lectura de datos, la preparación, la ejecución de la validación cruzada y el cálculo del resultado final.
- **Memoria:** Espacio medio de memoria RAM que consume la ejecución del algoritmo.
- **Porcentaje de CPU:** Media del porcentaje de CPU utilizado con respecto a la potencia total. Estas pruebas han sido realizadas sobre una máquina con capacidad para soportar 4 hilos al mismo tiempo, por lo que el uso completo de uno de los hilos supondría un porcentaje de carga de la CPU del 25 % con respecto al total, el uso exclusivo de dos hilos sería un 50 % y así sucesivamente.

Otros aspectos relevantes

- En lo que se refiere a Spark, se ha creado una clase en Scala que es capaz de leer el conjunto de datos, crear diferentes *folds* de dicho conjunto, entrenar y probar el clasificador y mostrar un resultado final. Weka ya proporciona herramientas de este tipo y por lo tanto no ha sido necesario generar ninguna otra clase.
- Para las ejecuciones en Spark, se ha ejecutado en modo local, lo que genera en Spark una única unidad (*driver*) a la que se la asignarán todos los recursos que le proporcionemos y que será la encargada de realizar la clasificación.
- Los conjuntos de datos utilizados para esta comparación han sido, ordenados de menor a mayor: “Iris”, “Human activity Recognition”, “Covertype”, “Poker” y “HIGGS”. Más información sobre los mismos en la sección [E.2](#).

Resultados

A continuación se muestran los resultados ordenados de menor a mayor según el tamaño del conjunto de datos:

Criterio	Weka	Spark 1 hilo	Spark 2 hilos	Spark 4 hilos
Tiempo (s)	0.1	1.97	2.13	2.15
Memoria (MB)	-	-	-	-
CPU (%)	-	-	-	-

¹ Los datos de las mediciones sobre el uso de memoria y CPU en el conjunto Iris no son concluyentes debido al corto periodo de tiempo que tarda en ejecutarse.

Cuadro E.2: Rendimiento sobre el conjunto de datos Iris.

Criterio	Weka	Spark 1 hilo	Spark 2 hilos	Spark 4 hilos
Tiempo (s)	12.37	16.13	11.75	11.71
Memoria (MB)	242.01	173.80	219.75	219.17
CPU (%)	25.5	36.02	54.03	59.43

Cuadro E.3: Rendimiento sobre el conjunto de datos Human Activity Recognition.

Criterio	Weka	Spark 1 hilo	Spark 2 hilos	Spark 4 hilos
Tiempo (s)	183.95	101.91	73.23	53.78
Memoria (MB)	576.78	177.37	213.7	211.7
CPU (%)	28.17	28.26	43.10	71.94

Cuadro E.4: Rendimiento sobre el conjunto de datos Coverttype.

Criterio	Weka	Spark 1 hilo	Spark 2 hilos	Spark 4 hilos
Tiempo (s)	73.90	47.44	31.84	30.77
Memoria (MB)	424.65	162.93	243.76	255.11
CPU (%)	30.05	29.41	51.22	55.68

Cuadro E.5: Rendimiento sobre el conjunto de datos Poker.

Criterio	Weka	Spark 1 hilo	Spark 2 hilos	Spark 4 hilos
Tiempo (s)	246.97	191.9	130.37	99.2
Memoria (MB)	832.88	872.56	864.92	921.49
CPU (%)	28.60	26.64	49.91	89.24

Cuadro E.6: Rendimiento sobre el conjunto de datos HIGGS (1.469.873 instancias).

Criterio	Weka	Spark 1 hilo	Spark 2 hilos	Spark 4 hilos
Tiempo (s)	503.93	368.72	264.02	215.21
Memoria (MB)	1747.22	883.28	911.77	853.58
CPU (%)	29	26.37	50.74	91.58

Cuadro E.7: Rendimiento sobre el conjunto de datos HIGGS (2.939.746 instancias).

Conclusiones

- Puede observarse claramente que Weka es considerablemente más rápido que Spark cuando utilizamos conjuntos de datos pequeños, como observamos en la tabla E.2 o incluso en la tabla E.3, pero su tiempo de ejecución se ve reducido por Spark cuando el conjunto de datos empieza a sobrepasar las 100.000. Una evolución de los tiempos de ejecución puede verse en la gráfica E.1
- Nótese en la tabla E.4 que el hecho de que su ejecución sea mayor que la de otros conjuntos de datos más grandes no tiene nada que ver con el comportamiento anómalo de las librerías, sino que tiene un número de atributos mucho mayor que otros datasets mayores y, en el caso de este algoritmo, eso tiene un gran peso sobre la ejecución final. Para más información sobre los conjuntos de datos ver E.2.
- Por lo general, y a la vista de la gráfica E.1, podemos decir que el tiempo de ejecución del algoritmo Naive Bayes de Weka es mayor si lo comparamos con las ejecuciones sobre un solo hilo de Naive Bayes en Spark. Es posible que una de las causas sea la diferente implementación del algoritmo en Weka y en Spark.
- Como era de esperar, doblar el número de hilos no implica reducir a la

mitad el tiempo de procesamiento, sino que genera un beneficio menor que, en algún momento y dependiendo del tamaño del conjunto de datos analizado, dejará de ser significativo aunque sigamos añadiendo nuevos hilos.

- Generalmente Spark utiliza menos memoria que Weka para ejecutar el programa. Es probable que esto se deba dos factores importantes de Spark: que las estructuras de datos solo son calculadas cuando se necesitan y que Spark da una gran importancia al correcto uso de los recursos, evitando almacenar nada a no ser que sea especificado en el código.
- Parece que el porcentaje de RAM requerido por Spark aumenta ligeramente cuantos más hilos tengamos en ejecución, algo que se aprecia bien en los conjuntos de datos pequeños y medianos. Ver, por ejemplo, la tabla E.5.
- El porcentaje de uso de la CPU en Weka se sitúa siempre entorno a los valores 25-30 %. Esto es así porque la ejecución de todas las tareas es secuencial, consumiendo únicamente un hilo de los 4 que posee la máquina en la que se están realizando las pruebas.
- Vemos que el porcentaje de uso de la CPU en las diferentes pruebas con Spark suele corresponder al número de hilos con los que se lanza la aplicación: 25 % para un hilo, 50 % para dos y, teóricamente, 90-100 % para cuatro. Sin embargo, y como podemos apreciar en las tablas E.5 o E.4, la ejecución con cuatro hilos no aprovecha al máximo las capacidades del procesador cuando el conjunto de datos es pequeño. Observando más de cerca el evento, vemos que, independientemente del número de hilos asignados a Spark, en estos conjuntos de datos únicamente se lanzan dos hilos como máximo. Atribuimos esto a un comportamiento propio de Spark cuando actúa en modo local, que evalúa que no existe necesidad de manejar tantos hilos de ejecución.

Un ejemplo más ilustrativo de lo anteriormente citado lo encontramos en la imagen E.2, donde se muestran algunos de los hilos de ejecución del lanzamiento de Spark con el algoritmo Naive Bayes sobre el conjunto de datos “Poker” con 4 procesadores asignados. Aunque deberían existir 4 hilos bajo el nombre de *Executor*, solamente se generan dos.

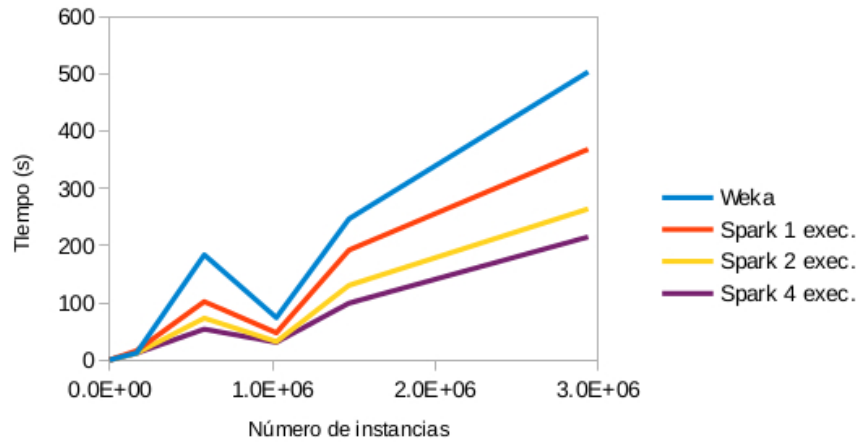


Figura E.1: Evolución del tiempo de clasificación según el número de instancias clasificadas.

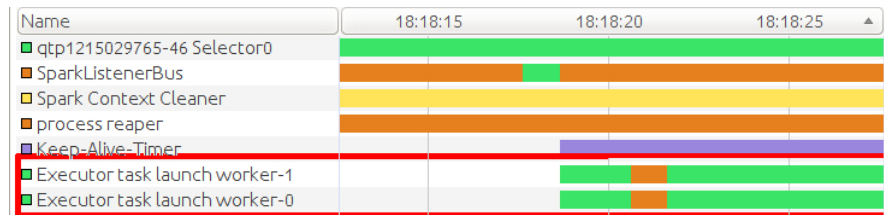


Figura E.2: Hilos de ejecución de Spark en el uso del algoritmo Naive Bayes sobre “Poker” con 4 procesadores asignados.

E.5. Comparativa entre la ejecución secuencial y paralela, en una sola máquina, de LSHIS y DemoIS

Se procederá a medir diferentes aspectos de la ejecución secuencial y paralela de los algoritmos mencionados. Para esta comparación, se ha vuelto a utilizar Weka y Spark. En el caso de Weka, ya contábamos con la implementación de los algoritmos [3]. En el caso de Spark, el código utilizado ha sido el resultado de este proyecto.

Mediante estas nuevas pruebas, pretendemos conseguir dos objetivos: Por un lado, queremos evaluar que los algoritmos implementados funcionen como se espera de ellos, es decir, que proporcionen resultados similares a los que proporcionaban los algoritmos ya implementados en Weka. Por otro, queremos realizar un estudio de que índice de mejora hemos conseguido en lo que a

tiempo de ejecución del algoritmo se refiere.

Para todo esto, vamos a realizar lanzamientos de ejecuciones que consten de un algoritmo de selección de instancias (LSHIS o DemoIS) y un clasificador k NN que realice una clasificación con los datos obtenidos del filtrado actuando como conjunto de entrenamiento.

De nuevo, la ejecución del algoritmo se analizará tanto en Weka como en Spark, siendo en este último ejecutado con diferente número de ejecutores: 1, 2 y 4. Avanzamos, por lo que se dirá en la sección E.5, que cada ejecutor lleva asignado un hilo de ejecución.

Criterios comparados

Se ha realizado la comparación teniendo en mente la medición de los siguientes parámetros:

- **Reducción del conjunto de datos:** Relación entre el tamaño del conjunto de datos original y el obtenido tras aplicar el algoritmo. Este valor resulta de la siguiente ecuación:

$$1 - \left(\frac{nInstanciasFinales}{nInstanciasIniciales} \right)$$

- **Tiempo:** Tiempo real que ha requerido la ejecución del algoritmo de selección de instancias.
- **Precisión de un clasificador con el nuevo conjunto:** Porcentaje de acierto en test al aplicar el algoritmo del vecino más cercano (k -Nearest Neighbor) sobre el nuevo conjunto de datos.

Adicionalmente, en muchos experimentos realizados (no todos) se presenta el porcentaje de acierto en test del clasificador k -NN cuando no se ha aplicado ninguna operación de selección de instancias sobre el conjunto de entrenamiento. De esta manera, el lector puede comprobar el efecto que tiene realizar un algoritmo de selección sobre el resultado de clasificación, pero no será un aspecto estudiado en este anexo.

Otros aspectos relevantes

- En lo que se refiere únicamente a Spark, se ha corrido en modo *Standalone* [20]. Dentro de este modo, destacar lo siguiente:
 - Solo existe un nodo trabajador (*worker node*) con un solo trabajador (*worker*).
 - Cada ejecutor (*executor*) tendrá asignado un hilo de ejecución.

- Las semillas utilizadas en Spark para generar números aleatorios han sido las mismas en todas sus configuraciones (1, 2 y 4 ejecutores.)
- Para las pruebas realizadas con el algoritmo LSHIS se han utilizado los conjuntos de datos, ordenados de menor a mayor: “Letter recognition”, “Human activity recognition”, “Covertime”, “Poker” y “HIGGS”.
- Para las pruebas realizadas con el algoritmo DemoIS se han utilizado los conjuntos de datos, ordenados de menor a mayor: “Image segmentation”, “Banana shaped”, “Pen-Based recognition of handwritten digits” y “Letter Recognition”.
- Para más información sobre los conjuntos de datos ver la sección [E.2](#)

Configuración del algoritmo LSHIS

Durante las mediciones, se ha utilizado la siguiente configuración del algoritmo LSHIS:

- **Funciones AND:** 10
- **Funciones OR:** 1
- **Anchura de los bucket:** 1

Configuración del algoritmo DemoIS

Durante las mediciones, se ha utilizado la siguiente configuración del algoritmo DemoIS:

- **Número de votaciones:** 10
- **Alpha:** 0.75
- **Número de instancias por subconjunto de votación:** Aprox. 1000 instancias.
- **Porcentaje de datos para calcular el límite de votos:** 10 %
- **Vecinos cercanos para calcular el límite de votos:** 1

Resultados de las mediciones

Criterio	Weka	Spark 1 Ejecutor	Spark 2 Ejecutores	Spark 4 Ejecutores	Conjunto original
Reducción (%)	65.86	67.03	67.2	67.27	-
Tiempo (s)	0.04	0.34	0.38	0.52	-
Precisión (%)	92.7	92.8	93.04	92.84	95.97

Cuadro E.8: Comparativa de LSHIS sobre el conjunto de datos “Letter Recognition”.

Criterio	Weka	Spark 1 Ejecutor	Spark 2 Ejecutores	Spark 4 Ejecutores	Conjunto original
Reducción (%)	99.47	99.46	99.46	99.45	-
Tiempo (s)	0.06	0.44	0.38	0.52	-
Precisión (%)	85.61	84.39	84.94	84.68	99,57

Cuadro E.9: Comparativa de LSHIS sobre el conjunto de datos “Human Activity Recognition”.

Criterio	Weka	Spark 1 Ejecutor	Spark 2 Ejecutores	Spark 4 Ejecutores
Reducción (%)	95.8	95.61	95.6	95.6
Tiempo (s)	0.59	3.9	2.15	2.44
Precisión (%)*	-	-	-	-

Cuadro E.10: Comparativa de LSHIS sobre el conjunto de datos “Coverttype”.

Criterio	Weka	Spark 1 Ejecutor	Spark 2 Ejecutores	Spark 4 Ejecutores
Reducción (%)	63.73	61.27	61.25	61.45
Tiempo (s)	3.35	5.62	3.67	4.25
Precisión (%)*	-	-	-	-

Cuadro E.11: Comparativa de LSHIS sobre el conjunto de datos “Poker”.

Criterio	Weka	Spark 1 Ejecutor	Spark 2 Ejecutores	Spark 4 Ejecutores**
Reducción (%)	60.91	54.46	54.47	-
Tiempo (s)	5.34	13.9	9.75	-
Precisión (%)*	-	-	-	-

Cuadro E.12: Comparativa de LSHIS sobre el conjunto de datos “HIGGS” (1.469.873 instancias).

Criterio	Weka	Spark 1 Ejecutor	Spark 2 Ejecutores	Spark 4 Ejecutores	Conjunto original
Reducción (%)	71.22	77.9	78.01	78.19	-
Tiempo (s)	5.43	4.59	3.97	4.7	-
Precisión (%)	96.32	95.9	95.46	95.51	96.76

Cuadro E.13: Comparativa de DemoIS sobre el conjunto de datos “Image Segmentation”.

Criterio	Weka	Spark 1 Ejecutor	Spark 2 Ejecutores	Spark 4 Ejecutores	Conjunto original
Reducción (%)	63.63	63.97	64.1	63.35	-
Tiempo (s)	18.24	15.61	10.82	10.2	-
Precisión (%)	85.75	85.93	85.98	86.1	87.2

Cuadro E.14: Comparativa de DemoIS sobre el conjunto de datos “Banana shape”.

Criterio	Weka	Spark 1 Ejecutor	Spark 2 Ejecutores	Spark 4 Ejecutores	Conjunto original
Reducción (%)	77.12	82.58	82.05	81.3	-
Tiempo (s)	20.6	15.48	10.43	10.18	-
Precisión (%)	98.18	98.1	97.85	98.18	99.39

Cuadro E.15: Comparativa de DemoIS sobre el conjunto de datos “Pen-Based Recognition of Handwritten Digits”.

Criterio	Weka	Spark 1 Ejecutor	Spark 2 Ejecutores	Spark 4 Ejecutores	Conjunto original
Reducción (%)	56.9	56.99	56.15	58.28	-
Tiempo (s)	326.98	365.9	183.3	176.26	-
Precisión (%)	92.61	92.48	92.7	92.4	95.97

Cuadro E.16: Comparativa de DemoIS sobre el conjunto de datos “Letter Recognition”.

Aclaraciones sobre las mediciones

*: El cálculo de la precisión no ha sido medido por la imposibilidad de hacerlo en un tiempo razonable.

** : Por falta de recursos en la máquina local no se ha podido realizar la medición adecuada con esta configuración.

Conclusiones

- En general, puede apreciarse que los valores de reducción y precisión son similares en Weka que en cualquier otra ejecución de Spark. Podemos deducir, por lo tanto, que el comportamiento de estas nuevas implementaciones es bueno, dado que se comportan de la manera que se espera.
- En lo que se refiere al tiempo empleado, podemos apreciar dos comportamientos diferentes dependiendo del algoritmo medido.

Por su rapidez incluso en una ejecución secuencial, la implementación en paralelo del algoritmo LSHIS no aporta ninguna ventaja con respecto a su versión secuencial.

La implementación paralela del algoritmo DemoIS, sin embargo, muestra un mejor rendimiento desde conjuntos de datos muy pequeños, con 2.000 instancias en la tabla E.13. Durante el resto de mediciones hemos conseguido ejecuciones hasta un 50 % más rápidas con Spark (ver E.16)

Puede observarse la evolución de los tiempos de ejecución de acuerdo al número de instancias analizadas en las gráficas E.3 y E.4.

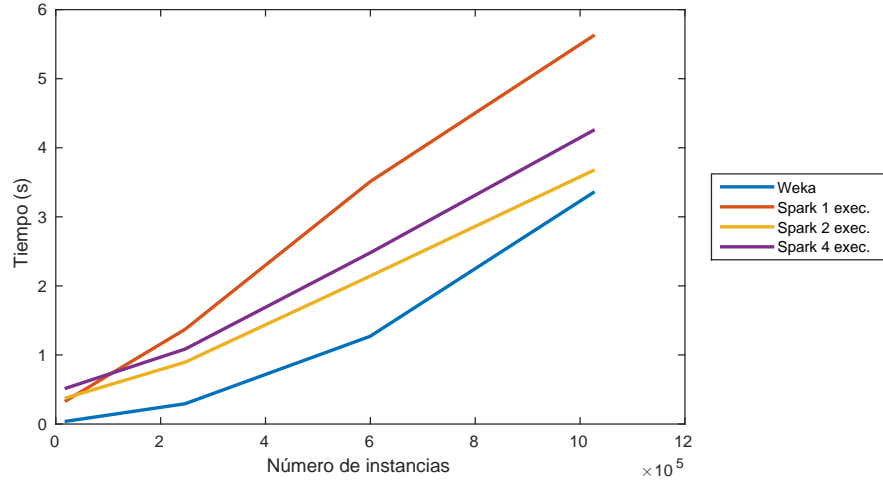


Figura E.3: Evolución del tiempo de ejecución según el tamaño del conjunto de datos usando LSHIS.

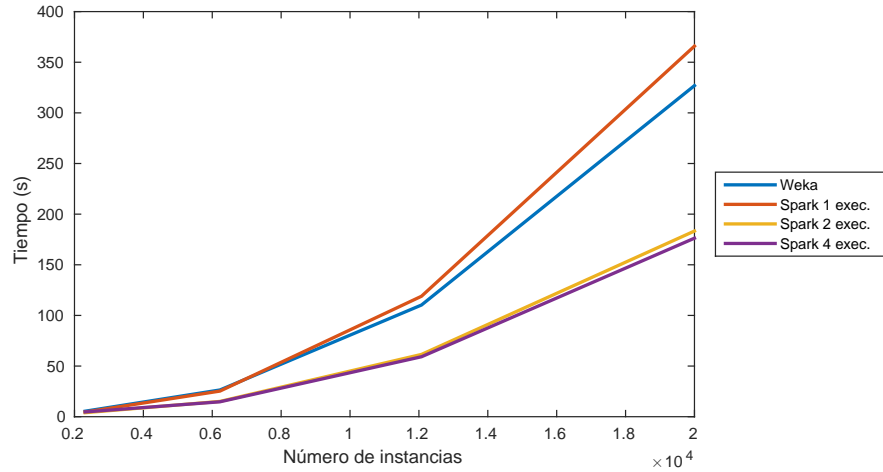


Figura E.4: Evolución del tiempo de ejecución según el tamaño del conjunto de datos usando DemoIS.

- Aunque los resultados de ejecución del algoritmo LSHIS no son mejores que los de su implementación secuencial, dado el supuesto de *Big Data*

de que los conjuntos de datos no suelen caber en memoria, la implementación en Spark permite la ejecución del algoritmo cuando la anterior implementación de Weka no lo podría conseguir.

- Puede observarse en cualquiera de las mediciones que, pese a utilizar las mismas semillas para generar números aleatorios, los resultados de los algoritmos en Spark son ligeramente diferentes según qué configuración usemos. Esto se debe a la ejecución paralela no garantiza un orden en el que los datos pueden ser computados o distribuidos, por lo que la ejecución se ve ligeramente afectada.
- Con los conjuntos de datos utilizados, el lanzamiento con cuatro ejecutores no parece que aporte una gran ventaja. En el caso del algoritmo LSHIS, probablemente debido al exceso de comunicación entre ejecutores, añadir más de dos ejecutores resulta incluso perjudicial.
- Aunque no puede apreciarse fácilmente en los resultados mostrados, cuando solo existe una función OR, el porcentaje de reducción del algoritmo LSHIS depende fuertemente de la semilla utilizada para generar las funciones hash. De hecho, en conjuntos de datos como HIGGS (ver E.2) han llegado a apreciarse ejecuciones cuyo resultado variaba en hasta 200.000 instancias con tan solo modificar la semilla. Es por ello que en algunas mediciones, como en tabla E.8 y tabla E.12, pueden apreciarse porcentajes de reducción diferentes entre las mediciones realizadas con Weka y con Spark.
- Puede observarse como en algunas mediciones del algoritmo DemoIS (tabla E.13 y tabla E.15) el porcentaje de reducción es mayor con Spark que con Weka, sin afectar esto a la precisión. Esto se debe a una ligera variación en el cálculo del *fitness* de la nueva implementaciones, donde utilizamos un subconjunto de test más parecido al original (muestra estratificada). Un ligero cambio en los parámetros de lanzamiento de Weka (parámetro *alpha*) conduciría al mismo resultado obtenido con Spark.

E.6. Medición del tiempo de filtrado, en un clúster, de LSHIS y DemoIS

En un intento de aumentar los recursos disponibles y de probar el funcionamiento del proyecto en un sistema distribuido, se han lanzado pruebas sobre un servicio clúster. Se intenta, no solo realizar ejecuciones que necesiten de más recursos, sino también ver el funcionamiento de nuestro programa en un entorno donde encontramos problemas adicionales, como puede ser un aumento del tiempo de comunicación entre los diferentes nodos (hasta ahora solo existía un único nodo)

El servicio utilizado, perteneciente a la Universidad de Córdoba y lanzado desde la Universidad de Burgos, es un servidor Cluster Dell con 63 nodos 4xQuadCore Xeon @ 2 GHz y un total de 252 núcleos. Está configurado con un sistema de gestión de colas de trabajo PBS (*Portable Batch System*), por lo que para la ejecución de Spark se ha necesitado de un script de lanzamiento proporcionado por la Universidad de Ohio [5].

En esta ocasión, se han lanzado ejecuciones con Weka y en Spark con hasta 16 procesadores.

Sin embargo, las ejecuciones tuvieron que ser suspendidas por problemas entre el script mencionado y el sistema de gestión de colas, que generaban que ciertos núcleos dejaran de trabajar y paralizaran la ejecución hasta que volvían a ser iniciados. Es por ello que no contamos con todas las mediciones que se plantearon en un primer momento y que no podemos asegurar que los tiempos medidos sean completamente veraces, por lo que no han sido incluidos en la memoria ni realizaremos un análisis sobre ellos.

Si puede decirse, sin embargo, que nuestros algoritmos han sido ejecutados en otro servicio (Google Cloud Dataproc) con conjuntos de datos mayores que los que ocasionaban problemas en el servidor Clúster Dell y no se han detectado problemas en la ejecución, por lo que se descarta que pueda haber un error en el código de la aplicación.

Documentación de usuario

F.1. Introducción

A lo largo de este apéndice vamos a indicar el material necesario para poder ejecutar el proyecto, así como la manera de instalarlo y hacerlo funcionar una vez contemos con todos los elementos necesarios.

F.2. Requisitos de usuarios

Es objetivo de esta sección destacar los requisitos mínimos para la ejecución del proyecto, tanto a nivel software como hardware.

Requisitos software

A continuación se muestra una lista de todos los requisitos mínimos para la ejecución del proyecto.

Podemos advertir diferentes distribuciones de nuestro proyecto y diferentes maneras de ejecutarlo, por lo que no todos los componentes son necesarios en todos los casos (ver sección [F.5](#) para obtener más información).

- Java 8.
- Apache Maven - Solo necesario si necesitamos Apache Spark.
- Apache Spark - Solo necesario si deseamos lanzar las ejecuciones en nuestra máquina.

Requisitos mínimos del sistema

Dado que existen diferentes modos de uso del proyecto, es necesario indicar aquí varios casos:

- Únicamente deseamos generar un archivo .zip para nuestro proyecto:
 - **Procesador:** Pentium 2 @ 266 MHz [17]
 - **Memoria RAM:** 128MB [17]
 - **Espacio libre en disco:** aprox. 180 MB + Espacio adicional para los conjuntos de datos.
- Deseamos ejecutar los algoritmos en nuestro proyecto:
 - **Procesador:** Procesador multihilo con capacidad de soportar, al menos, 2 hilos simultáneamente.
 - **Memoria RAM:** 4 GB.
 - **Espacio libre en disco:** aprox. 2.1GB más el espacio adicional para los conjuntos de datos que deseemos almacenar.

Quede constancia, además de los requisitos mencionados, que hay que tener en cuenta dos aspectos más:

- La memoria mínima que un ejecutor de Spark puede tener es 1 GB. Por lo tanto, tal vez sea necesario ajustar el número de ejecutores si no tenemos memoria suficiente.
- Los requisitos definidos para la ejecución de Spark en ningún caso se van a parecer a los usados en una ejecución “real” en un clúster, cuyas recomendaciones pueden encontrarse en la página oficial de Spark [19].

F.3. Instalación

A continuación se van a definir los métodos de instalación de todos los componentes necesarios para ejecutar el proyecto, de cara a facilitar el mantenimiento en futuras versiones de la biblioteca y permitir la instalación al usuario.

Estos métodos de instalación hacen referencia a la instalación en máquinas con un sistema operativo basado en alguna distribución de Debian. En nuestro caso, esta distribución ha sido Ubuntu 14.04.

Oracle Java 8

Aunque existen distribuciones libres de Java, usaremos la que proporciona Oracle (<http://www.java.com>) por el hecho de que incluye una serie de programas que hemos usado para medir el rendimiento de aplicaciones Java (JConsole y JVisualVM). Cualquier otra distribución será igualmente válida, pero es posible que no incluya estas herramientas. Podemos acceder tanto

al JRE como al JDK desde la siguiente dirección: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

Sin embargo, Oracle no proporciona una instalación automática de Java para Linux. Es por esta razón vamos a utilizar un PPA (*Personal Package Archive*) que proporciona un instalador para diferentes versiones de Java [21]. Este instalador no contiene ningún archivo Java, pero será el encargado de descargarlos en nuestra máquina (de igual manera que podríamos hacerlo manualmente) y realizar la instalación automáticamente, facilitando el proceso de instalación.

Una vez entendido esto, ejecutaremos los siguientes comandos:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
$ sudo apt-get install oracle-java8-set-default
```

Podemos comprobar si la instalación ha sido correcta ejecutando el comando `java -version` en la terminal. Si todo ha salido bien deberíamos recibir una salida similar a la siguiente:

```
$ java -version
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b17,
mixed mode)
```

ISAlgorithms

Puede descargarse desde el repositorio del proyecto en <https://bitbucket.org/agr00095/tfg-alg.-seleccion-instancias-spark>.

Aunque para algunas acciones no será necesario que nuestro software se encuentre en un directorio concreto, se recomienda situar el fichero en el directorio `$SPARK_HOME/lib`, si es que existe. Esto será obligatorio en el lanzamiento de baterías de ejecuciones directamente desde la interfaz gráfica y el lanzamiento de baterías predefinidas anteriormente usando, de nuevo, la interfaz.)

Nombrar que la variable `$SPARK_HOME` hace referencia a la ruta de instalación de Spark. Se mencionará su uso, así como la manera de definirla, en la subsección destinada a la instalación del propio Spark en F.4.

Igualmente se recomienda utilizar el nombre “ISAlgorithms.jar” como nombre del fichero .jar, independientemente de su distribución. Esto es de nuevo obligatorio al realizar acciones que involucren la interfaz gráfica.

Para más información sobre las posibles distribuciones ver la sección F.5.

F.4. Instalaciones Opcionales

Existen una serie de componentes que no van a ser necesarios en todas las distribuciones del producto (ver sección F.5). Los componentes definidos a continuación son necesarios para permitir la ejecución de tareas de minería en la máquina local.

Apache Maven

Este elemento, si bien no es necesario para la ejecución del proyecto, es necesario para la construcción de Apache Spark, por lo que es incluido en este listado.

Lo primero que debemos hacer es descargarnos el paquete que contiene la herramienta. Esto podemos hacerlo desde el navegador en la página oficial de Apache Maven (<https://maven.apache.org/>) o mediante el siguiente comando en consola:

```
$ wget http://apache.rediris.es/maven/maven-3/3.3.3
/binaries/apache-maven-3.3.3-bin.tar.gz
```

Recordar que el código de arriba es orientativo, pudiéndose seleccionar otro lugar desde donde realizar la descarga u otra versión del producto.

Descomprimos el paquete descargado y lo movemos a la carpeta */usr/local*:

```
$ tar -zxf apache-maven-3.3.3-bin.tar.gz
$ sudo mv apache-maven-3.3.3 /usr/local
$ sudo ln -s /usr/local/apache-maven-3.3.3/bin/mvn
/usr/bin/mvn
```

Podemos comprobar que la instalación ha sido realizada correctamente si al escribir el comando *mvn --version* recibimos una salida parecida a la siguiente:

```
$ mvn --version
Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0df;
 2015-04-22T13:57:37+02:00)
Maven home: /usr/local/apache-maven-3.3.3
Java version: 1.8.0_66, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-8-oracle/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "3.19.0-25-generic", arch:
"amd64", family: "unix"
```

Apache Spark

Nos dirigiremos a la página oficial de Apache Spark (<http://spark.apache.org/>) y, una vez allí, a su sección de descargas. Elegiremos la versión 1.5.1 por ser la utilizada a lo largo de la práctica, y descargaremos el código fuente. Igualmente, y como hemos mencionado en otras instalaciones, podemos ejecutar el siguiente comando para hacernos con el paquete:

```
$ wget http://apache.rediris.es/spark/spark-1.5.1
/spark-1.5.1.tgz
```

Una vez tengamos el archivo en nuestra máquina lo descomprimos y movemos a la carpeta que deseemos lanzando estos comandos desde el directorio que contenga el paquete descargado:

```
$ tar -xvf spark-1.5.1.tgz
$ sudo mv spark-1.5.1 /opt
```

Finalmente vamos a construir Spark utilizando los siguientes comandos desde la carpeta donde lo hemos ubicado. En nuestro caso, estará en */opt/spark-1.5.1*:

```
$ sudo ./dev/change-scala-version.sh 2.11
$ sudo mvn -Dscala-2.11 -Pnetlib-lgpl \
-DskipTests clean package
```

Destacar que esta invocación no vincula a Spark con ninguna versión de Hadoop o un administrador de clústeres concreto, por lo que si se requiere alguno de estos servicios deberán añadirse nuevos parámetros al comando, algo que no se discutirá aquí.

Esta última operación llevará un tiempo.

Finalmente, si deseamos comprobar que Spark se ha desplegado correctamente podemos ejecutar, por ejemplo, el intérprete de comandos de Scala para Spark:

```
$ ./bin/spark-shell
```

Aunque recibiremos una serie de avisos (*warnings*) debido a que no hemos configurado ciertos aspectos de Spark, el intérprete debería poder lanzarse y funcionar sin problemas.

Finalmente, sería recomendable que, por comodidad, se añadiese la variable `$SPARK_HOME` al sistema. Para ello, podemos modificar el fichero *~/.profile* y añadirle la siguiente línea:

```
export SPARK_HOME=/opt/spark-1.5.1/
```

Para que este cambio tenga efecto necesitaremos, sin embargo, reiniciar la sesión de usuario.

F.5. Manual del usuario

Existen multitud de formas de lanzar una ejecución con nuestra aplicación, por lo que se dedicará esta sección a hablar y definir cada una de ellas.

Brevemente, vamos a considerar 4 formas de lanzamiento:

- Lanzamiento desde línea de comandos.
- Lanzamiento desde la interfaz gráfica.
- Definición de una batería de ejecución en la interfaz gráfica y lanzamiento por separado.
- Lanzamiento en el servicio de computación en la nube Google Cloud Dataproc.

Recordemos, antes de empezar, que nuestra aplicación consta de varias distribuciones y algunas cuentan con limitaciones en cuanto a su funcionalidad. Igualmente, existen diferentes modos de despliegue de Spark, algo que también influye en nuestra ejecución. Todo esto será explicado en esta sección.

Distribuciones

Hemos de entender que existen tres distribuciones diferentes de nuestra aplicación, cada una de ellas con unas características diferentes en función de la finalidad del producto, por lo que es necesario elegir aquella que se adapte mejor al uso que pensemos dar al proyecto:

- **ISAlgorithms:** Presenta todo el material del proyecto, desde la interfaz gráfica hasta los algoritmos. Es un tipo de distribución general que puede realizar todas las operaciones posibles. La versión de Scala utilizada para compilar las clases ha sido 2.11.7.
- **ISAlgorithms.gui:** Contiene únicamente los componentes de la interfaz gráfica, por lo que no puede lanzar ejecuciones. Aun así, permite definir experimentos y generar un archivo .zip para exportarlos a cualquier otra máquina que contenga una distribución del programa que permita el lanzamiento. La versión de Scala utilizada durante la compilación ha sido 2.11.7.
- **ISAlgorithms.cluster:** Contiene todo el material a excepción de la interfaz gráfica, que se encuentra innecesaria al ser el objetivo de esta distribución la ejecución de tareas en una red de nodos remota. La versión de Scala utilizada para compilar ha sido 2.10.6.

Cabe destacar que la versión de Scala utilizada en la compilación es importante, pues las versiones 2.10 y 2.11 de Scala no generan código compatible y Spark ha de estar configurado previamente para poder correr código de una u otra versión.

Modos de despliegue de Spark

Spark proporciona diferentes modos de lanzamiento, que vamos a diferenciar en tres grupos:

- **Modo local:** El más simple de todos. Como su propio nombre puede dejar ver, en este modo solo contaremos con un único trabajador que tendrá asignados todos los recursos. A nivel interno, las aplicaciones locales no distribuyen el trabajo a unidades llamadas *executors*, como se hace en cualquier otro tipo de despliegue, sino que todo el trabajo es manejado por una sola unidad denominada *driver*.

No necesita ningún tipo de gestión previa a ser desplegado, solo ha de indicarse que se desea correr en este modo cuando lanzamos una aplicación con Spark.

Es una buena opción cuando se están realizando pruebas pero está limitado en muchos aspectos.

- **Modo Standalone:** Proporciona un tipo de despliegue sin tener que depender de software de administración de clústeres, aunque requiere una versión compilada de Spark en cada nodo.

Puede resultar eficiente usarlo en un grupo de nodos pequeño, pero no es buena opción cuando contamos con una red más amplia [15].

Este es el modo de despliegue que hemos utilizado en gran parte del proyecto. Una explicación más detallada de como iniciarlo en una sola máquina (aquella en el que se esté trabajando) puede encontrarse más adelante en esta misma sección.

- **Otros modos de despliegue:** Existen numerosos administradores de clústeres que pueden funcionar con Spark, tales como Mesos o YARN, pero en ningún caso hemos necesitado programar un servicio similar, por lo que no entraremos a detalle. Cabe destacar, sin embargo, que el gestor de clústeres YARN es el que usa Google Cloud Dataproc cuando hemos lanzado ejecuciones sobre él.

Iniciar el modo Standalone en una máquina (Opcional)

Es posible que, intentando evitar el modo de ejecución de Spark local, queramos ejecutar nuestros programas como si funcionasen en un auténtico

clúster, aunque éste solo tenga un nodo. Para ello son necesarios algunos pasos previos.

Desde el directorio de instalación de Spark podemos ejecutar el siguiente comando para iniciar el nodo maestro:

```
./sbin/start-master.sh
```

Esta operación, que podría tardar varios segundos, debería abrir un nuevo puerto (<http://localhost:8080/>), desde donde poder acceder a la información de nuestra red de nodos que, de momento, debería estar vacía.

Ahora, vamos a iniciar un nodo trabajador (*worker*) y asociarlo al master con el siguiente comando:

```
./sbin/start-slave.sh <URL_master>
```

Si hemos los pasos correctamente, y no hemos modificado la configuración por defecto, la dirección URL del nodo master debería ser algo similar a *spark://ruta:puerto*, donde “ruta” es el atributo `$HOSTNAME` de la máquina donde hemos iniciado el nodo maestro y “puerto”, por defecto, es 7077. En caso de duda, siempre podemos visitar <http://localhost:8080/>, donde, entre otra información, podemos ver la URL del master.

Lanzamiento de ejecución desde línea de comandos

Es el método de ejecución más rápido pero requiere un mínimo conocimiento sobre los algoritmos y parámetros que podemos configurar.

Se presenta un ejemplo de ejecución en el código F.1. Sobre este ejemplo, vamos a intentar comprender cada uno de sus componentes de la llamada para poder, en el futuro, configurar nuestras propias ejecuciones.

```
$SPARK_HOME/bin/spark-submit \
--master spark://alejandro:7077 \
--class "launcher.ExperimentLauncher" \
$SPARK_HOME/lib/ISAlgorithms.jar ISClassExec \
-r ./Datasets/Banana-5k/banana_norm.csv \
-f instanceSelection.demoIS.DemoIS -np 5 \
-c classification.seq.knn.KNN -k 1 \
-cv 10 1
```

Código F.1: Ejemplo de ejecución desde línea de comandos

- **Script de lanzamiento:** `$SPARK_HOME/bin/spark-submit` es un script que proporciona Spark para permitir el lanzamiento de aplicaciones. Siempre vamos a necesitar esta sentencia al lanzar una aplicación.

- **Opciones de Spark:** Spark cuenta con multitud de parámetros para personalizar su funcionamiento, pero solo mencionaremos algunos que nos son de especial interés (y que aparecen en el ejemplo):
 - **Nodo master:** `--master spark://alejandro:7077` Indica la ruta al nodo maestro de nuestro clúster. Esta opción puede tomar como parámetro la sentencia `local[n]`, para indicar una ejecución a nivel local con *n* número de núcleos o la sentencia `spark://ruta:puerto` si la ejecución se realiza en otro modo de despliegue de Spark.
Puede definirse una ruta por defecto en los ficheros de configuración de Spark (concretamente en `$SPARK_HOME/conf/spark-defaults.conf`), pero se le hace una mención especial porque, sin este argumento definido en algún sitio, la ejecución no podría tener lugar.
 - **Clase principal:** `--class "launcher.ExperimentLauncher"` señala la clase principal a cargar dentro del .jar que deseamos lanzar y que mencionaremos más adelante.

Todas las diferentes opciones pueden consultarse mediante el comando `$SPARK_HOME/bin/spark-submit --help`. También es posible definir valores para estas opciones en los ficheros de configuración de Spark (`$SPARK_HOME/conf`), de manera que no sea necesario indicar el parámetro en cada ejecución.

- **Archivo .jar:** `$SPARK_HOME/lib/ISAlgorithms.jar` define el archivo .jar que deseamos ejecutar. Si se han seguido los pasos de instalación previos, debería encontrarse en la misma ruta que la del ejemplo.
- **Argumentos para nuestro programa:** El resto de la sentencia consta de valores para la configuración del lanzamiento de nuestra aplicación. Podemos encontrar un listado con todos los parámetros disponibles en la sección F.5. Aunque esta parte del programa es la que más variaciones puede sufrir de una ejecución a otra, se puede definir una estructura general:
 - **Tipo de ejecución (obligatorio):** El primer argumento indica la tarea de minería de datos que deseamos ejecutar.
 - **Argumentos para el lector de conjuntos de datos (obligatorio):** Comienza con la sentencia “-r” y debe indicar la ruta al fichero que contenga el conjunto de datos seguido de las posibles necesarias para la correcta lectura del fichero.
 - **Argumentos para la configuración del algoritmo de selección de instancias (obligatorio):** Comienza con la sentencia “-f”

y debe indicar la ruta, dentro del fichero .jar, a la clase que contenga el algoritmo que deseamos lanzar y, seguidamente, una lista con las opciones de configuración de dicho algoritmo en el caso de que queramos cambiar las opciones por defecto.

- **Argumentos para la configuración del algoritmo de clasificación (obligatorio):** Comienza con la sentencia “-c” seguida de la ruta, dentro del fichero .jar, a la clase que contenga el algoritmo que deseamos lanzar y, si se desea, una lista con opciones para configurar los parámetros por defecto del algoritmo.
- **Argumentos para la configuración de la validación cruzada (opcional):** Empieza con la sentencia “-cv” seguida de uno o dos números, siendo el primero el que indica el número de *folds* que deseamos crear y el segundo una semilla para la división aleatoria del conjunto de datos inicial. Si no indicamos ningún parámetro, se ejecutará un experimento donde el conjunto de entrenamiento supondrá un 90 % del total de instancias y, el de test, un 10 %.

Si todo ha salido correctamente, la ejecución empezará a procesarse. Es posible que veamos aparecer algunos mensajes informativos por consola, dependiendo de si hemos habilitado esa opción en Spark o hemos mantenido la que se ofrece por defecto.

Cuando la ejecución haya finalizado correctamente, se habrá generado una nueva carpeta de nombre “results” en el directorio desde el cual lanzamos la ejecución. Dentro del mismo, podremos encontrar un archivo con el nombre del clasificador usado, seguido de la fecha y hora de la finalización. Accediendo a él podremos ver los aspectos medidos de la ejecución. Podemos ver un ejemplo del resultado en [F.1](#)

```

=====
Program arguments
-r ./Datasets/banana-5k/banana_norm.csv -f instanceSelection.lshis.LSHIS -c
classification.seq.knn.KNN -k 1 -cv 10 1
=====
Filter: LSHIS
Classifier: KNN
+++++
Reduction(%)    + 99.06916059101681
+++++
Accuracy(%)     + 68.60614397651686
+++++
Filter time(s)  + 0.117
+++++

```

Figura F.1: Ejemplo del fichero resultante de una ejecución.

Lanzamiento de ejecución o batería de ejecuciones desde la interfaz gráfica

Antes de empezar, merece la pena recordar que para llevar a cabo este paso es necesario que, tal y como se ha indicado en la sección F.3 destinada a la instalación, el fichero .jar que vamos a utilizar debe estar situado en la ruta `$SPARK_HOME/lib`.

El primer paso consistirá en lanzar la interfaz gráfica. Para ello, podemos hacer doble clic sobre el archivo .jar distribuido o ejecutar la siguiente sentencia en la línea de comandos:

```
$ java -cp $SPARK_HOME/lib/ISAlgorithms.jar  
    gui.SparkISGUI
```

De cualquiera de las maneras, tendremos ante nosotros una nueva ventana similar a la que se muestra en la imagen F.2. Pueden observarse cuatro áreas principales en el cuerpo de la aplicación, junto un menú superior y otro inferior.

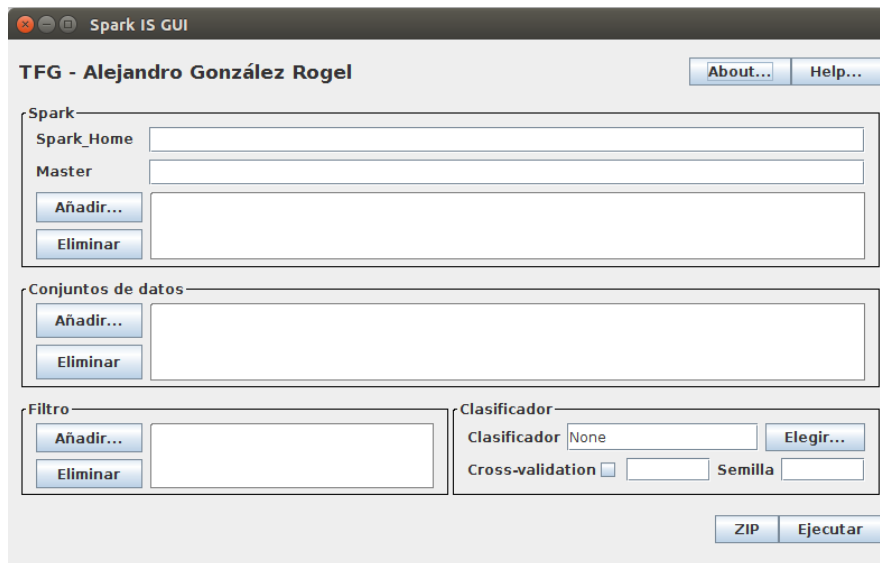


Figura F.2: Ventana principal de la interfaz gráfica.

- **Menú superior:** Contiene el título del proyecto y dos botones de carácter informativo. Si no comprendemos el funcionamiento de la interfaz, sería útil consultar el botón de ayuda de la propia interfaz, donde se realiza una explicación más detallada de cada uno de los campos existentes en la interfaz.
- **Cuerpo:** Se encuentra subdividido en cuatro grandes bloques, cada uno de ellos dedicado a un aspecto diferente del lanzamiento. Podemos apre-

ciar una estructura similar en la mayoría de ellos, que contienen un listado donde aparecerán todas las configuraciones que se hayan indicado hasta el momento, así como un par de botones para añadir o eliminar nuevas configuraciones.

- **Panel de configuración de Spark:** Nos permitirá seleccionar una o varias opciones de configuración para iniciar nuestra tarea en Spark. Algunas de estas opciones son comunes para todas las tareas que programemos, tales como “SPARK_HOME” y “Master”. Otras, variarán en cada experimento pueden añadirse o eliminarse pulsando sobre los botones “Añadir...” y “Eliminar” respectivamente.

El botón “Añadir...” dará paso a un nuevo diálogo (ver imagen F.3) donde rellenar una serie de parámetros, ninguno de los cuales debería dejarse vacío. Una vez indicados todos los parámetros, aceptar esa configuración resultará en una nueva línea en la tabla de configuración de la pantalla principal.

El botón “Eliminar” suprime la fila de la tabla seleccionada, si es que hubiese alguna.

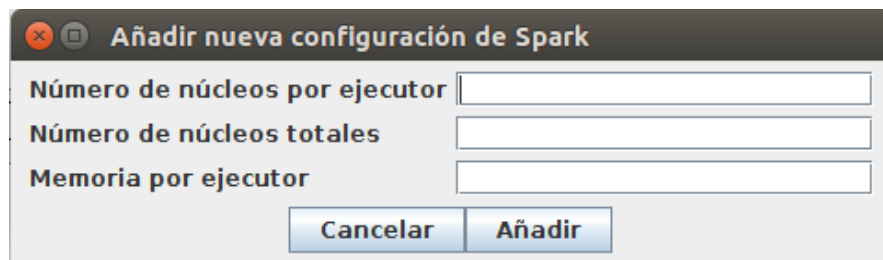


Figura F.3: Diálogo para la selección de opciones de Spark.

- **Panel de selección de conjunto de datos:** Cuenta con un listado de los conjuntos añadidos hasta el momento y una serie de botones que permiten añadir/eliminar los conjuntos de datos seleccionados para el experimento.

Al igual que en el caso anterior, pulsar sobre el botón “Añadir...” generará un diálogo (ver imagen F.4) que permitirá indicar un nuevo conjunto de datos y las opciones de lectura necesarias, si es que fuese necesario.



Figura F.4: Diálogo para la selección de un conjunto de datos.

- **Panel de selección de filtro:** Similar al panel anterior, muestra un listado de todas las configuraciones propuestas hasta la fecha y la posibilidad de añadir más o eliminar las ya existentes.

Mencionar una peculiaridad en este caso. Al presionar sobre el botón “Añadir...” se abrirá un panel con solo un campo para seleccionar un algoritmo selector de instancias. El resto de campos se generarán de manera automática cuando seleccionemos un algoritmo concreto (ver imágenes F.5 y F.6).

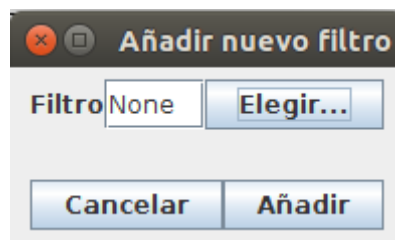


Figura F.5: Diálogo para la selección de filtro.

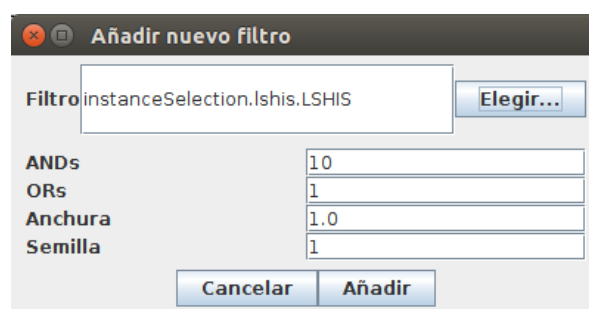


Figura F.6: Diálogo para la selección de filtro después de seleccionar un algoritmo.

- **Panel de selección de clasificador y validación cruzada:** Su funcionalidad es similar a la descrita en paneles anteriores, concretamente el panel de selección de filtro. Sin embargo, es este caso

solo es posible añadir un único clasificador (así como solo una única configuración de validación cruzada), por lo que las opciones que típicamente se mostraban en un diálogo separado, ahora se muestran en la propia pantalla (ver imagen F.7).

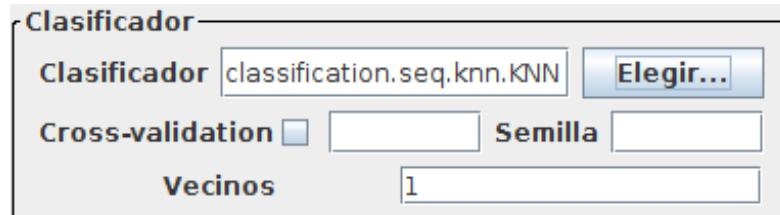
El panel de configuración, titulado "Clasificador", contiene los siguientes elementos: un campo de texto "Clasificador" con el valor "classification.seq.knn.KNN" y un botón "Elegir..."; un grupo de control "Cross-validation" con un botón desactivado; un campo de texto "Semilla" vacío; y un grupo de control "Vecinos" con un campo de texto que contiene el número "1".

Figura F.7: Panel para la configuración del clasificador y la validación cruzada.

- **Menú inferior:** Cuenta con dos botones para ejecutar diferentes acciones. La modalidad “ZIP” será descrita en la subsección F.5, pero en esta sección nos interesa presionar el botón “Ejecutar”.

Antes de lanzar una ejecución surge la pregunta de cuantas ejecuciones diferentes se han programado. Se generarán tantas ejecuciones como posibles combinaciones entre componentes se puedan crear con las configuraciones propuestas. Así pues, si, por ejemplo, se han indicado dos configuraciones de Spark, dos conjuntos de datos y dos filtros se realizarán $2 \times 2 \times 2 = 8$ ejecuciones, puesto que el clasificador será siempre el mismo.

Una vez hemos cumplimentado todos los campos requeridos (los únicos opcionales son los que se refieren a la validación cruzada) y presionamos sobre el botón “Ejecutar” veremos como en la esquina inferior izquierda se muestra un nuevo texto indicando la ejecución que se está realizando. Un ejemplo de un experimento completamente configurado y en funcionamiento puede verse en la imagen F.8

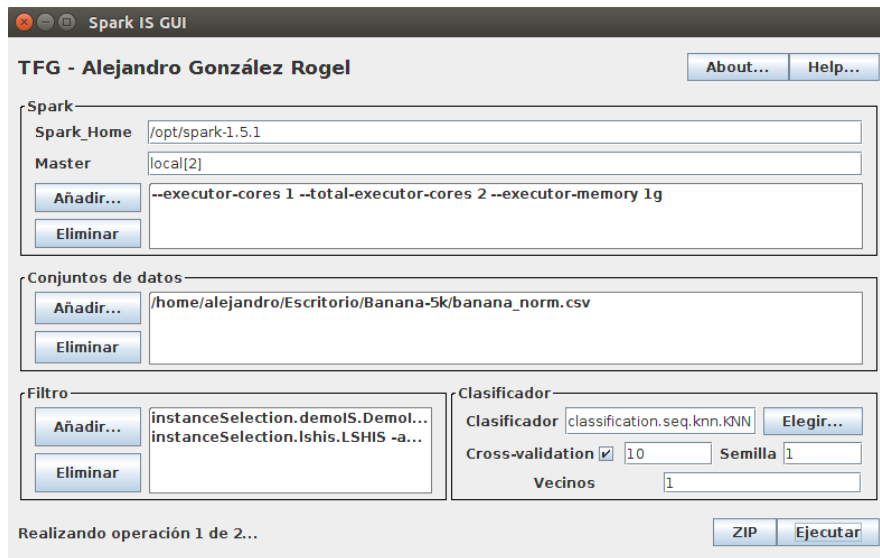


Figura F.8: Ventana principal de la interfaz gráfica rellena.

No se pueden lanzar varias baterías de experimentos a la vez, así que si intentásemos crear una nueva ejecución y ejecutarla mientras la anterior continúa funcionando recibiremos un mensaje de error.

Una vez terminadas todas las ejecuciones, la interfaz nos informará de ello mediante un nuevo diálogo. Podemos ir a consultar el resultado en el directorio “results” que se ha generado en la carpeta desde la que hemos lanzado la ejecución. Si hemos iniciado el programa haciendo doble clic sobre el fichero .jar este directorio probablemente se encuentre en la carpeta \$HOME del usuario.

Definición de una batería de ejecución y lanzamiento por separado

El procedimiento inicial es similar al marcado en el método de lanzamiento anterior, es decir, una vez abierta la interfaz gráfica hemos de indicar todas las configuraciones de ejecución que deseemos. La diferencia esta vez la encontramos al final de la operación, donde en lugar de presionar sobre el botón “Ejecutar”, lo haremos sobre el botón “ZIP”

Esta operación, que tal vez puede tardar unos instantes si hemos seleccionado conjuntos de datos muy grandes, generará una carpeta de nombre “zip” en el directorio desde donde hemos invocado la interfaz gráfica. Dentro de la carpeta encontraremos un nuevo archivo .zip que contendrá un script de ejecución .sh junto con todos los conjuntos de datos que necesitemos para realizar las ejecuciones que hemos definido previamente.

Ese fichero lo moveremos a donde sea necesario, típicamente un servicio clúster donde se encuentre Spark instalado junto con nuestra librería en el directorio `$SPARK_HOME/lib`.

Una vez allí, el archivo puede descomprimirse usando el comando:

```
$ unzip <fichero-zip>
```

Es posible que para descomprimir el archivo necesitemos un paquete “unzip” instalado. Si no existe, podremos obtenerlo con el comando

```
$ sudo apt-get install unzip
```

Finalmente, hemos de ejecutar el archivo `.sh` que acabamos de descomprimir. Para ello, desde consola, podemos movernos a la ruta donde se encuentre el fichero y escribir en terminal:

```
$ chmod +x Bateria_de_Ejecucion.sh
$ ./Bateria_de_Ejecucion.sh
```

Deberían comenzar las ejecuciones en Spark, una tras otra, hasta que finalicen todas.

Lanzamiento en el servicio Google Cloud Dataproc

Google Cloud Dataproc es uno de los muchos servicios que ofrece la plataforma Google Cloud (<https://cloud.google.com/>). Para acceder a todos ellos es necesario contar con una cuenta Google, cuyo proceso de registro no será descrito en este manual. Igualmente, existen diferentes maneras de utilizar este servicio. En este apartado se describirá únicamente la manera de uso gráfica, por ser la más intuitiva para realizar labores sencillas.

Lo primero que debemos saber es que los servicios ofrecidos en Google Cloud (incluido el propio Google Cloud Dataproc) suelen estar asociados a un proyecto concreto, aunque, si acabamos de registrarnos, ya contaremos con un proyecto creado. De querer generar uno nuevo podemos hacerlo en el menú superior, clic sobre el nombre del proyecto actual y clic sobre “Crear proyecto” en el menú desplegable que generará la acción anterior (ver imagen F.9).

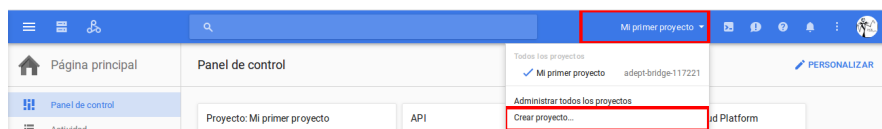


Figura F.9: Creación de un nuevo proyecto en Google Cloud.

Es importante indicar que, en el caso de estar realizando esta tarea con la versión de prueba de Google Cloud, no deberemos crear un nuevo proyecto,

pues la condición gratuita se aplica solo al proyecto generado por defecto por Google.

Ahora mismo, probablemente nos encontremos ante la página de inicio de nuestro proyecto (ver imagen F.10), aunque no entraremos a explicar el funcionamiento de la misma.

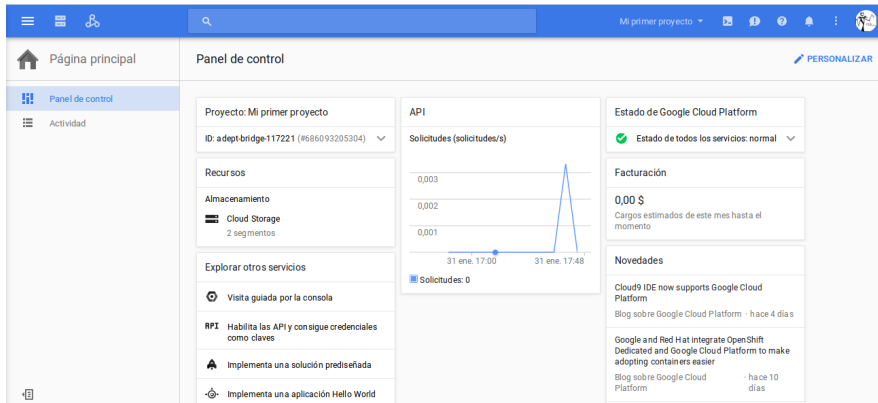


Figura F.10: Página de inicio de un proyecto en Google Cloud.

El segundo paso será almacenar nuestro archivo .jar, junto con los conjuntos de datos, en algún lugar que Google y sus máquinas puedan alcanzar. Para ello, lo más sencillo será usar un segundo servicio de Google: Google Cloud Storage. Podemos acceder a él desde el menú superior, haciendo clic sobre el icono situado más a la izquierda y seleccionando el servicio “Storage” de entre las opciones que muestre el menú emergente (ver imagen F.11).

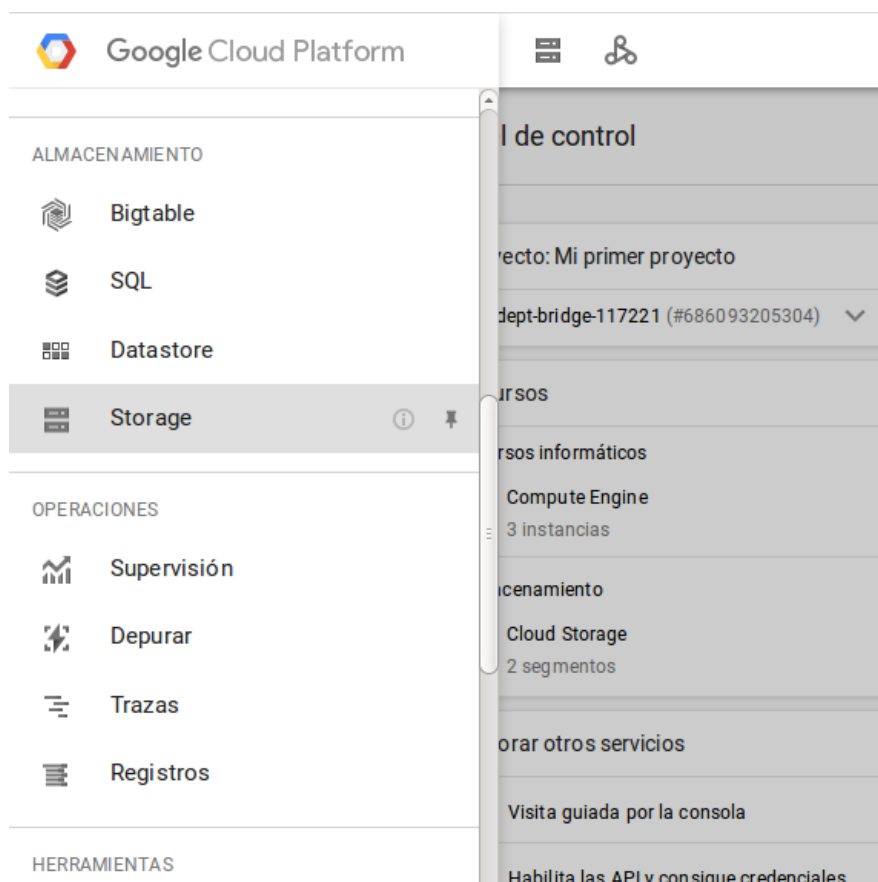


Figura F.11: Menú de selección de servicios de Google Cloud.

Si el proyecto es nuevo, es posible que tengamos que reservar un nuevo segmento de memoria para poder guardar nuestros datos. Simplemente hacemos clic sobre la opción “Crear segmento” que veremos en el centro de nuestra pantalla y rellenamos los campos de “Nombre”, “Clase de almacenamiento” y “Ubicación” acorde a nuestras preferencias. Una vez terminado este paso, podemos acceder a nuestro nuevo espacio de memoria y administrarlo según nuestro gusto mediante las opciones del menú superior o arrastrando hacia la ventana del navegador todos aquellos elementos que deseamos que sean subidos al servicio de almacenamiento.

Ahora, necesitamos dar ciertos permisos a nuestro proyecto para poder asignarle un clúster. Para ello, vamos a visitar el servicio Google Compute Engine de la misma manera que antes visitamos “Storage”, presionando el icono de la izquierda en el menú superior y buscando el servicio entre la lista de posibles. Lo primero que se nos preguntará en cuando carguemos el nuevo servicio es si deseamos asignar una serie de permisos a nuestro proyecto, a lo

cual aceptaremos.

Con todo preparado, vamos a acceder finalmente al servicio que realmente nos interesa: Google Cloud Dataproc. Repetimos la misma operación anterior: vamos al menú desplegable y buscamos servicio “Dataproc”. Encontraremos ante nosotros una pantalla similar a la que muestra la imagen F.12.

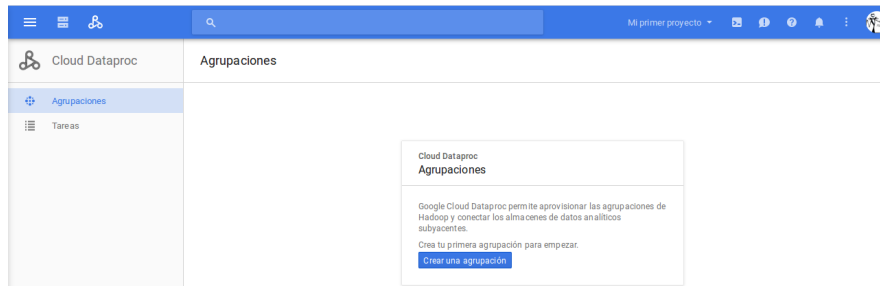


Figura F.12: Pantalla principal de Google Cloud Dataproc totalmente vacía.

Ahora, vamos a crear nuestro primer clúster. Presionamos sobre el botón “Crear nueva agrupación” que veremos en el centro de la pantalla y rellenamos el formulario que permitirá crear un clúster. Es importante saber que:

- Es conveniente situar la ubicación del clúster en la misma que el segmento de memoria que está destinado a usar y que ya hemos definido anteriormente.
- Si estamos ejecutando desde la versión gratuita de Google Cloud, el tamaño del clúster estará limitado a 8 CPUs, incluyendo el nodo maestro.
- En el apartado desplegable “Opciones de acceso, inicialización, versión, red, segmento y trabajadores prioritarios” podemos seleccionar la “versión de imagen” del clúster, lo que definirá la versión de los productos que vamos a usar. Este proyecto se ha probado con la versión 0.2 que incluye Spark 1.5.2
- También en el apartado “Opciones de acceso, inicialización, versión, red, segmento y trabajadores prioritarios” hay una opción llamada “Segmento de aplicación de fases de Cloud Storage”. Aquí debemos indicar el segmento de memoria que hemos creado anteriormente. Si no especificamos nada se creará uno por defecto.

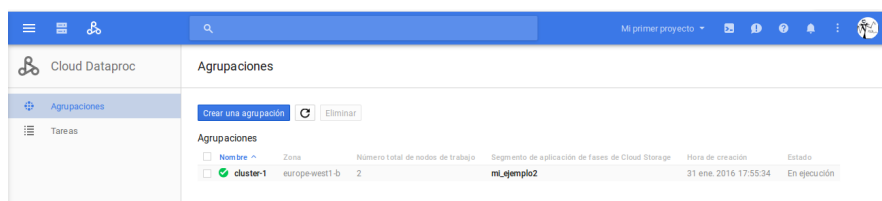


Figura F.13: Pantalla principal de Google Cloud Dataproc tras la creación de un clúster.

Bien, una vez terminemos de configurar todos los datos de nuestra red de máquinas tendremos nuestro primer clúster creado, por lo que únicamente falta definir la tarea que queremos ejecutar. Para ello, en el menú lateral izquierdo podremos ver una sección llamada “Tareas” (ver imagen F.13). Nos podemos dirigir a ella y crearemos una nueva tarea del mismo modo que hemos creado anteriormente un segmento de memoria o una agrupación, presionando el botón de creación que veremos en el centro de la pantalla y rellenando un formulario similar al que muestra la imagen F.14. De nuevo, es importante tener en cuenta que:

- Debemos definir el “Tipo de tarea” como “Spark”
- La ruta al archivo .jar (o al fichero que contenga el conjunto de datos cuando introduzcamos las opciones) tendrá una estructura similar a “gs://nombreSegmento/directorio/fichero” si hemos utilizado el servicio Google Cloud Dataproc.

Podemos ver un ejemplo de una configuración en la imagen F.14.

The screenshot shows the Google Cloud Dataproc console interface. On the left, there is a sidebar with a menu containing 'Agrupaciones' (Clusters) and 'Tareas' (Jobs). The main area is titled 'Enviar una tarea' (Submit a job). It contains several configuration fields: 'Agrupación' (Cluster) set to 'cluster-1', 'Tipo de tarea' (Job type) set to 'Spark', 'Archivos .jar (Opcional)' (Optional JAR files) with one file 'gs://dataproc-8aaa433e-f183-4011-a9ea-949a78c0e9c1-eu/ISAlgorithms.jar' and a text input field for more files, 'Clase principal o .jar' (Main class or JAR) set to 'launcher.ExperimentLauncher', and 'Argumentos (Opcional)' (Optional arguments) with a list of arguments: 'ISClassExec', '-f', 'gs://dataproc-8aaa433e-f183-4011-a9ea-949a78c0e9c1-eu/banana_norm.csv', '-f', and 'instanceSelection.demos.DemoIS'.

Configuración	Valor
Agrupación	cluster-1
Tipo de tarea	Spark
Archivos .jar (Opcional)	gs://dataproc-8aaa433e-f183-4011-a9ea-949a78c0e9c1-eu/ISAlgorithms.jar
Clase principal o .jar	launcher.ExperimentLauncher
Argumentos (Opcional)	ISClassExec, -f, gs://dataproc-8aaa433e-f183-4011-a9ea-949a78c0e9c1-eu/banana_norm.csv, -f, instanceSelection.demos.DemoIS

Figura F.14: Ejemplo de una tarea configurada.

Lanzada la aplicación seremos redirigidos a una nueva pantalla donde podremos observar una línea de comandos tal y como si hubiésemos lanzado la aplicación en nuestra propia máquina (ver imagen F.15 para ejemplo). Mientras la ejecución tiene lugar podemos realizar cualquier otro tipo de operación, incluido definir nuevas tareas.

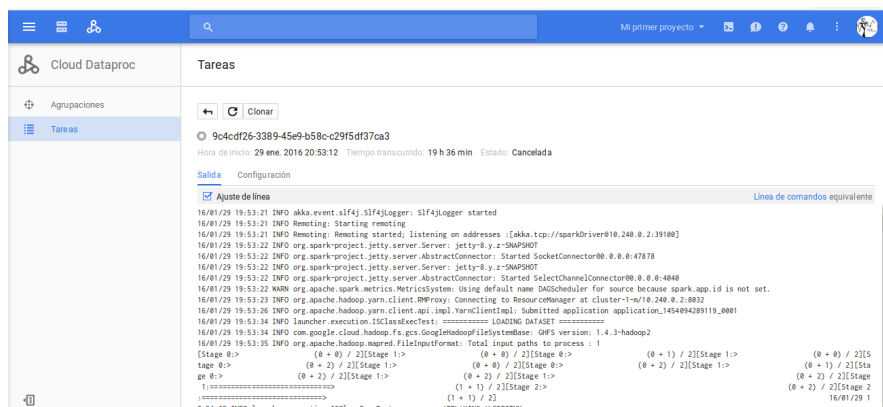


Figura F.15: Tarea en ejecución en Dataproc.

Una vez terminada la tarea que hayamos programado, queda ver el fichero resultante de la aplicación, para lo que deberemos conectarnos al nodo maestro. Accedemos de nuevo a la vista principal Google Dataproc y seleccionaremos nuestro clúster. Esto nos lleva a una nueva pantalla con información sobre nuestra agrupación, aunque no hablaremos mucho de ella (ver imagen F.16). Aquí haremos clic sobre “Instancias de VM” y sobre el botón “SSH” que aparecerá junto al nombre del nodo maestro.

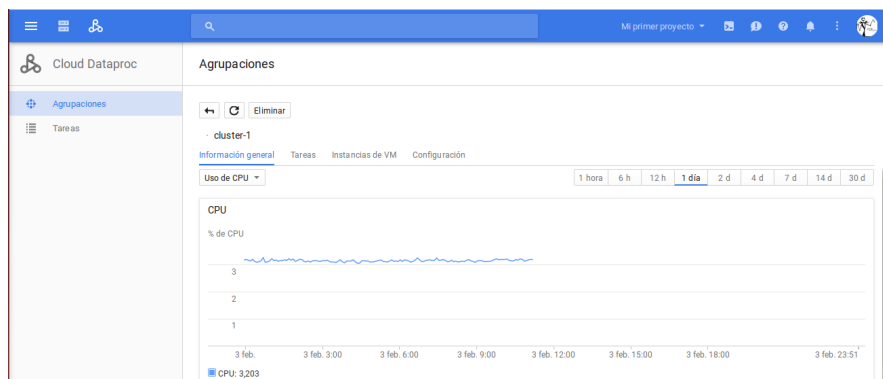


Figura F.16: Pantalla principal de nuestro clúster en Dataproc.

Esta última acción abrirá una nueva ventana del navegador que emulará una consola de comandos. No es objetivo de esta sección hablar de todas las posibles opciones que podemos realizar, únicamente veremos el fichero resultante ejecutando el siguiente comando:

```
$ cat
/tmp/nombre_tarea/results/nombre_fichero_resultados
```

Parámetro	Descripción
Tipo de ejecución	
ISClassExec	Define una tarea en la que intervenga un algoritmo de selección de instancias y un clasificador
ISClassExecTest	Define una tarea en la que intervenga un algoritmo de selección de instancias y un clasificador y, además mide el tiempo de ejecución de la labor de filtrado.
Parámetros del lector	
-f	Indica que el atributo de clase es el primero de los atributos de cada instancia.
-hl + Int	Indica que existe una cabecera en el fichero y cuantas líneas forman dicha cabecera.
Parámetros para el selector de instancias	
instanceSelection.lshis.LSHIS	
-and + Int	Número de funciones AND.
-or + Int	Número de funciones OR.
-w + Double	Anchura de los <i>buckets</i> .
-s + Long	Semilla utilizada para generar números aleatorios.
instanceSelection.demoIS.DemoIS	
-rep + Int	Número de votaciones a realizar.
-alpha + Double	Valor alpha.
-np + Int	Número de particiones en las que se dividirá el conjunto de datos original.
-dsperc + Double	Porcentaje del conjunto de datos utilizado para calcular el error durante el cómputo del fitness.
-s + Long	Semilla utilizada para generar números aleatorios.
Parámetros para el clasificador	
classification.seq.knn.KNN	
-k + Int	Número de vecinos más cercanos

Cuadro F.1: Cheatsheet

Cheatsheet

A continuación se va realizar un listado con todas las posibles opciones que actualmente pueden formar parte de la sentencia de invocación del programa.

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 Garrrell, 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] Álar 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.
- [4] Álar Arnaiz-González, José F. Díez Pastor, César García Osorio, and Juan J. Rodríguez. 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.
- [5] 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.
- [6] P. Baldi, P. Sdowski, and D. Whiteson. Searching for Exotic Particles in High-energy Physics with Deep Learning, 2014-7-2.
- [7] Donnie Berkholz. The emergence of spark, 2015. <http://redmonk.com/dberkholz/2015/03/13/the-emergence-of-spark/>.

- [8] Louis Columbus. 84 % of enterprises see big data analytics changing their industries' competitive landscapes in the next year, 2014. www.forbes.com/sites/louiscolumbus/2014/10/19/84-of-enterprises-see-big-data-analytics-changing-their-industries-competitive-landscapes-in-the-next-year/#d25708c32502.
- [9] 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>.
- [10] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Petter Reutemann, and Ian H. The weka data mining software: An update. *SIGKDD Explorations*, 11, 2009.
- [11] Fraunhofer IAIS. Fraunhofer Intelligent Data Analysis Group Benchmark Repository. <http://www.iais.fraunhofer.de/index.php?id=32&L=1>.
- [12] Google Inc. Google Cloud Dataproc. <https://cloud.google.com/dataproc/>.
- [13] Jim Jagielski. Apache Software Foundation-Organization summary, 2015. <https://www.openhub.net/orgs/apache>.
- [14] 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.
- [15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. "O'Reilly Media, Inc.", 2015. chapter 7.
- [16] M. Lichman. UCI Machine Learning Repository. *University of California, Irvine, School of Information and Computer Sciences*, 2013. <http://archive.ics.uci.edu/ml>.
- [17] Oracle. What are the system requirements for java? <http://java.com/en/download/help/sysreq.xml>.
- [18] Apache Spark. Apache Spark docs - MLUtils. <https://spark.apache.org/docs/1.5.1/api/scala/index.html#org.apache.spark.mllib.util.MLUtils>.
- [19] Apache Spark. Hardware Provisioning. <https://spark.apache.org/docs/1.5.1/hardware-provisioning.html>.

- [20] Apache Spark. Spark Standalone Mode. <http://spark.apache.org/docs/latest/spark-standalone.html>.
- [21] “WebUpd8” team. Oracle Java Installer. <https://launchpad.net/~webupd8team/+archive/ubuntu/java>.
- [22] Wallace Ugulino, Débora Cardador, Katia Vega, Eduardo Velloso, Ruy Milidiú, and Hugo Fuks. Wearable Computing: Accelerometers’ Data Classification of Body Postures and Movements. <http://groupware.les.inf.puc-rio.br/public/papers/2012.Ugulino.WearableComputing.HAR.Classifier.RIBBON.pdf>, visited 2015-10-28.
- [23] The University of Waikato Weka. Can I use CSV files?, 2009. <https://weka.wikispaces.com/Can+I+use+CSV+files%3F>, visited 2015-10-28.
- [24] École Polytechnique Fédérale de Lausanne (EPFL). Scala 2.12 roadmap, 2015. <http://www.scala-lang.org/news/2.12-roadmap>.