Diseño de una arquitectura de clasificación biclase basada en las máquinas de vectores de soporte, MapReduce y Spark

Mario Alberto Giraldo Londoño

Tesis de maestría presentada para optar el título de Magíster en Ingeniería

Tutor

John Freddy Duitama, Doctor (PhD)
Julián David Arias, Doctor (PhD)

Universidad de Antioquia
Facultad de Ingeniería
Maestría en Ingeniería
Medellín, Antioquia, Colombia
2020

Cita	M. A. Giraldo Londoño [1]
Referencia Estilo IEEE (2020)	[1] M. A. Giraldo Londoño, “Diseño de una arquitectura de clasificación biclase basada en las máquinas de vectores de soporte, MapReduce y Spark”, Tesis de maestría, Maestría en Ingeniería, Universidad de Antioquia, Medellín, Antioquia, Colombia, 2020.

Maestría en Ingeniería, Cohorte XX.

Grupo de Investigación Intelligent Information Systems Lab..

UNIVERSIDAD DE ANTIOQUIA
Vicerrectoría de Docencia

Sistema de Bibliotecas

Repositorio Institucional: http://bibliotecadigital.udea.edu.co

Universidad de Antioquia - www.udea.edu.co

Rector: John Jairo Arboleda Céspedes.

Decano/Director: Jesús Francisco Vargas Bonilla.

Jefe departamento: Diego José Botia Valderrama.

AGRADECIMIENTOS

A Dios por trazar este camino y procurarme una experiencia de vida rica en amor, acompañamiento, aprendizaje… Su esencia ha recorrido cada una de las etapas de mi existencia.

A mi familia, por tolerar mis ausencias y creer, como yo, en la formación y el aprendizaje. Su apoyo, constancia y paciencia fue el aliciente para la culminación exitosa de esta meta.

A los asesores de este trabajo, los Doctores John Freddy Duitama Muñoz y Julián David Arias Londoño, por la compañía y valoración permanente, que se transformó en una contribución relevante para el desarrollo de esta investigación.

A la Universidad de Antioquia, en especial a la Facultad de Ingeniería, porque abrieron sus puertas para que irrumpiera en sus dinámicas internas e hiciera lecturas de sus experiencias y les diera un nuevo significado a la luz del objeto de estudio.

A todos, ¡Gracias!

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ABREVIACIONES Y NOMENCLATURAS

SVM:	*Support vector machine* o máquina de vectores de soporte.
SV:	*Support vector* o vectores de soporte.
KKT:	*Karush-Kuhn-Tucker conditions*, condiciones KKT o condiciones de Karush-Kuhn-Tucker.
SMO:	*Sequential minimal optimization* u optimización secuencial mínima.
HDFS:	*Hadoop distributed file system* o sistema de archivos distribuido de Hadoop
RDD:	*Resilient distributed dataset* o conjunto de datos distribuidos resilientes
ML:	*Machine learning* o aprendizaje automático

RESUMEN

La máquina de vectores de soporte o *support vector machine* (SVM, su sigla en inglés), es un método ampliamente utilizado en el campo del aprendizaje automático, debido a que ha logrado una alta capacidad de generalización al hacer predicciones correctas sobre nuevas muestras en diferentes aplicaciones, como lo muestra [1] y [2], entre otros. Su costo computacional puede ser del orden cuadrático respecto al número de muestras de entrenamiento. Este hecho hace que su uso para entrenar con grandes conjuntos de datos tenga un alto costo computacional; su implementación en modelos de programación en paralelo, como MapReduce y Spark, ha demostrado ser eficiente para mejorar dicho costo. Sin embargo, existen factores en el empleo de estas *framework* que impactan el desempeño de los algoritmos, tanto en cuanto al costo computacional como de precisión. El presente trabajo de tesis se propone una arquitectura para hacer un procesamiento distribuido de la SVM basadas en SMO (*Sequential Minimal Optimization*) y se analiza cómo el tamaño del subconjunto de datos y el número de tareas de mapeo impactan su desempeño en implementaciones bajo MapReduce y Spark. Además, se plantea un modelo de costos como una herramienta útil para la configuración del tamaño de los subconjuntos de datos de acuerdo con el *hardware* disponible y los datos a procesar.

Abstract

Support vector machine (SVM) is a classifier widely used in machine learning because of its high generalization capacity. Its computational cost can be of the quadratic order respect to number of training samples. This fact makes using SVM to train large data sets have a high computational cost. SVM implementations on distributed systems such as MapReduce and Spark have shown efficiency to improve computational cost; however, there are factors as to using these tools that have impact on the algorithms performance, both at computational cost and accuracy levels. This paper analyzes how data subset size and number of mapping tasks affect SVM performance on MapReduce and Spark. Also, a cost model as a useful tool for setting data subset size according to available hardware and data to be processed is proposed.

INTRODUCCIÓN

En la sociedad actual existe un marcado crecimiento en la generación de grandes cantidades de datos de todo tipo, tales como audio, vídeo, imágenes, mensajes de texto, mensajes de voz, entre otros. Esta información es generada por las personas y por las máquinas a través de las redes sociales, la biometría y la interacción con nuevos dispositivos como sensores, teléfonos inteligentes y sistemas de monitorización; por tanto, se hace necesario utilizar herramientas nuevas y variadas para el análisis de estos datos, que respondan a las exigencias de las empresas y de la sociedad misma. Una muestra de este crecimiento de datos es la que se explica en la infografía "*The 42 V's of Big Data and Data Science*" Tom Shafer, Elder Research, Inc.[1], en la cual mencionan que a diario se crean aproximadamente 2.5 quintillones de *bytes* de datos.

Paralelo a este crecimiento en la generación de datos y a la importancia que ha ido tomando el uso de técnicas de aprendizaje automático en el análisis de datos, en la declaración realizada por Pichai, oficial ejecutivo en jefe de Google durante el informe Q3 de octubre de 2015 [3], se evidencia que "el aprendizaje automático jugará un papel más importante en los servicios avanzados de Google, es el futuro, es un camino de transformación por el cual estamos repensando todo lo que estamos haciendo". Así, han surgido técnicas y tecnologías en esta última década, que permiten tomar los datos, almacenarlos, procesarlos y hacer diferentes tipos de análisis que no se podrían hacer con los *framework* de análisis tradicionales. Normalmente, estos análisis incorporan grandes cantidades de datos de todo tipo, pueden ser estructurados o no estructurados, provenir de diferentes fuentes y generalmente requieren ser analizados mediante algoritmos

[1] Ampliar consulta en: https://www.kdnuggets.com/2017/04/42-vs-big-data-data-science.html

de minería de datos, técnicas estadísticas y sistemas de aprendizaje, que permiten, entre otras cosas, determinar patrones en la información, clasificarlos en tiempo real y elaborar predicciones para apoyar la toma de decisiones en los negocios, en el gobierno y en la vida diaria.

La necesidad de analizar estos datos con el uso de técnicas estadísticas y de aprendizaje automático, ha traído nuevos retos en la construcción de frameworks de procesamiento de datos en paralelo, que permiten dar una solución más rápida en términos de tiempo a problemas computacionales complejos y utilizando grandes volúmenes de datos.

Algunos de los *framework* propuestos en la actualidad para el almacenamiento y procesamiento de datos distribuidos y en paralelo son Hadoop, MapReduce y Spark; las cuales incorporan un conjunto de librerías de aprendizaje automático que permiten abordar este tipo de problemas, permitiendo encontrar similitudes entre individuos, identificar patrones de comportamiento, establecer correlaciones y diferenciar grupos mediante técnicas de clasificación o agrupamiento.

Dentro del aprendizaje automático, la clasificación es uno de los problemas más relevantes que hay que resolver [4]. Consiste en diseñar un sistema que sea capaz de identificar a qué clase (de un conjunto de clases preestablecidas) pertenece un objeto dado. Uno de los modelos más usados en la literatura para resolver este tipo de problemas es el de las máquinas de soporte de vectores, las cuales fueron introducidas en 1995 por Cortes y Vapnik [5]. Inicialmente, se pensaron para solucionar problemas de clasificación biclase usando modelos lineales y en la actualidad se utilizan para solucionar problemas de clasificación multiclase y de regresión, tanto lineales como no lineales.

El entrenamiento de una máquina de vectores de soporte implica la solución de un

problema de optimización cuadrático, cuyo costo computacional usando algoritmos convencionales como el *interior-point* [6], puede ser del orden cuadrático respecto al número de datos de entrenamiento [7]. Este hecho hace que el uso de las máquinas de vectores de soporte para el análisis de grandes cantidades de datos tenga un alto costo computacional, ya que los requerimientos de memoria se incrementan considerablemente afectando el tiempo de entrenamiento [8]. Una solución que ayuda a reducir este costo computacional es el algoritmo de optimización mínima secuencial (SMO por sus siglas en inglés), propuesto por Platt [9] y mejorado por Keerthi [10]. SMO tiene dos componentes: un método analítico para resolver los dos multiplicadores de lagrange involucrados, y una heurística para elegir qué multiplicadores optimizar. Para ello propone una descomposición del problema de optimización cuadrático de la SVM en subproblemas. En cada paso, el algoritmo elige dos multiplicadores de lagrange para optimizar conjuntamente, encuentra los valores óptimos para estos multiplicadores, y actualiza el SVM para reflejar los nuevos valores óptimos. De esta forma, SMO obtiene una solución analítica del orden lineal que permite reducir el costo de entrenamiento de la máquina de vectores de soporte, a lineal en el uso de memoria y entre lineal y cuadrático en tiempo de computación [9]. Aunque el SMO mejora sustancialmente el tiempo de entrenamiento de la máquina de vectores de soporte, es un algoritmo secuencial y no fue pensado para trabajar en entornos de programación paralela; de hecho, su naturaleza secuencial hace de la paralelización del algoritmo un problema complejo de resolver. En la literatura se han propuesto diferentes versiones del algoritmo de optimización mínima secuencial para trabajar en entornos de programación en paralelo, con el propósito de analizar mayores volúmenes de datos con una optimización en el tiempo de entrenamiento y sin afectar la precisión.

En los últimos años se han publicado una serie de trabajos [11], [12], [21]–[23], [13]–[20] en los cuales se implementa la máquina de vectores de soporte

estándar, bajo el modelo de programación MapReduce y Spark. Aunque estos trabajos tienen mejoras significativas en el tiempo de entrenamiento, sin afectar sustancialmente el porcentaje de precisión en la clasificación (uno o dos puntos por debajo de los algoritmos secuenciales), en ningún caso se aborda el problema de definir el número óptimo de particiones del conjunto de datos de entrada para entrenar el modelo, o la configuración que debería ser usada en los *framework* de programación MapReduce y Spark, con el objetivo de obtener los mejores resultados posibles. Es necesario tener en cuenta que una aplicación en MapReduce y Spark tiene limitaciones en el *hardware* disponible, ya que aspectos como el costo de comunicación entre los nodos o el tamaño de la memoria RAM, pueden afectar significativamente el tiempo de ejecución de las tareas o procesos que se ejecuten en el sistema [24]–[32].

Por tal motivo, este estudio establece unos parámetros que permiten relacionar el *hardware* disponible, la cantidad de información a procesar y el ajuste de parámetros del modelo, por medio de una fórmula de costos que ayuda a definir una arquitectura de ejecución de la máquina de vectores de soporte en MapReduce y Spark, evaluando su impacto en el tiempo de entrenamiento y en la precisión del clasificador.

Para ello, el estudio se estructuró por capítulos. El primero está relacionado con la motivación, la necesidad y la delimitación del planteamiento del problema, así como con los objetivos generales y específicos. En el segundo se explica el problema de clasificación y su solución por medio del modelo máquina de vectores de soporte; así mismo se explica el uso de MapReduce y Spark en el análisis de grandes volúmenes de datos y finaliza con el estado del arte sobre las estrategias de implementación del modelo de máquina de vectores de soporte en MapReduce y Spark. El tercero contiene las fases de la investigación, la arquitectura propuesta, la fórmula del modelo de costos y la descripción de elementos

importantes relacionados con las implementaciones en MapReduce y Spark; igualmente, la descripción de las bases de datos utilizadas para las pruebas y el *hardware* y *software* utilizados. El cuarto capítulo contiene la explicación de los experimentos realizados y los resultados obtenidos; el último apartado está conformado por las conclusiones, y posteriormente se hacen recomendaciones para trabajos futuros. Se finaliza con la bibliografía referencial y los anexos como complemento a informaciones y *framework* trabajadas durante el proyecto investigativo.

CAPÍTULO I

PLANTEAMIENTO DEL PROBLEMA

Las técnicas de minería de datos y de aprendizaje automático (ML, sigla en inglés de *machine learning*) se han utilizado para diferentes propósitos como la detección de correos no deseados, el reconocimiento de voz, el comercio de acciones, la robótica y la publicidad, entre otros, permitiendo, entre otros aspectos, encontrar similitudes entre individuos, identificar patrones de comportamiento, establecer correlaciones y diferenciar grupos mediante técnicas de clasificación o agrupamiento. Uno de los problemas más importantes a resolver dentro del aprendizaje automático es el de clasificación [4], el cual consiste en diseñar un sistema que sea capaz de identificar a qué clase pertenece un objeto dado. Uno de los modelos de ML más ampliamente usados para abordar este tipo de problemas son las máquinas de vectores de soporte (en adelante SVM, su sigla en inglés por *support vector machines*) [5].

La SVM, dentro de su proceso de entrenamiento, requieren encontrar una función de separación o decisión que permita separar las clases a las que pertenecen los objetos dados. Encontrar esta frontera implica resolver un problema de optimización que para su solución requiere algoritmos de programación cuadrática (QP), cuyo desempeño es de orden cúbico en el número de muestras y de orden cuadrático en el número de variables. El tiempo de entrenamiento de este modelo crece exponencialmente en la medida en que aumenta el tamaño de la muestra a analizar [33].Aunque la solución que ofrece SMO requiere de una cantidad de memoria que es lineal en el tamaño del conjunto de entrenamiento, lo que permite a SMO manejar conjuntos de entrenamiento muy grandes, su tiempo de computación se mantiene entre lineal y cuadrático en el tamaño del conjunto de entrenamiento Adicional a esto, tanto en la SVM como en la mayoría de los

modelos de aprendizaje, se requiere que el entrenamiento se ejecute varias veces para poder estimar los errores de entrenamiento y predicción y a su vez, ajustar los hiperparámetros del algoritmo. Si se realiza una sola simulación no es posible tener confianza en los resultados obtenidos. Estos factores hacen que el proceso de entrenamiento sea cada vez más costoso en la medida que se tengan grandes volúmenes de datos.

Aunque algoritmos como el de optimización mínima secuencial (SMO, su sigla en inglés por *sequential minimum optimization*) [9], propuesto por Plat en 1998[2], permite reducir el costo computacional de la SVM, han sido diseñados inicialmente, para analizar conjuntos de datos pequeños. En este sentido, muchos algoritmos de aprendizaje fueron implementados para mejorar la capacidad de modelado de los datos, si bien en términos de costo computacional, no fueran los más eficientes.

La necesidad de aplicar estas técnicas en la analítica para grandes volúmenes de datos que surgen diariamente de las redes sociales, teléfonos móviles y sensores, entre otros, ha traído nuevos retos en la construcción de *framework* de almacenamiento y procesamiento de datos en paralelo y distribuidas, que permiten dar una solución más rápida, en términos de tiempo, a problemas computacionales complejos y con grandes volúmenes de datos.

Uno de los *framework* actuales para el manejo de grandes cantidades de datos es el sistema de archivos distribuidos Hadoop, sobre el cual se pueden utilizar nuevos *framework* de programación para el procesamiento de datos en paralelo como MapReduce y Spark. Muchos algoritmos de aprendizaje automático han sido migrados para que puedan trabajar bajo estos *framework* mejorando su

[2] Posteriormente mejorado por Keerthin [10].

escalabilidad.

MapReduce y Spark, son *framework* de programación que, gracias a la ejecución de tareas de mapeo, reducción, transformación y acción, de forma distribuida y en paralelo sobre sistemas como Hadoop, ayudan a mejorar los tiempos de ejecución de algunos algoritmos de aprendizaje automático, proporcionando técnicas y tecnologías modernas para construir entornos de ejecución en paralelo que permiten procesar grandes volúmenes de datos. Sin embargo, el uso de estos *framework* implica tener en consideración los costos relacionados con los recursos del entorno de Hadoop, como lo son el número de nodos disponibles, el costo de comunicación y la capacidad de procesamiento, memoria y almacenamiento, necesarios para el procesamiento oportuno de las aplicaciones ya que pueden afectar de manera significativa el tiempo de ejecución de las tareas o procesos que se ejecuten en el sistema [24]–[32].

En los últimos años (2010-2019) se han publicado una serie de trabajos [11], [12], [21]–[23], [13]–[20] sobre la implementación del modelo SVM estándar para clasificación biclase bajo los *framework* de programación MapReduce y Spark. Estos trabajos en las implementaciones tiene en comunes algunas fases y métodos, como la utilización de la librería LibSVM [34], la división del conjunto de datos en subconjuntos, la asignación de cada subconjunto a una tarea de mapeo, el entrenamiento del modelo en las fase de mapeo y la asignación de tareas de forma secuencial, en donde la salida de una o varias tareas de mapeo se asigna a la entrada de otra tarea de mapeo. Aunque estos trabajos evidencian mejoras considerables en el tiempo de entrenamiento, cabe resaltar que en ninguno de ellos establecen criterios generales relacionados con el costo del uso de dichos *framework*, que permitan definir en casos específicos cuál es la mejor estrategia de ejecución del entrenamiento de la SVM en MapReduce o Spark, de acuerdo con la naturaleza de los datos, el parámetro de regularización del modelo *C*, el cual actúa como una compensación entre el error de entrenamiento y la precisión

de la solución. Cuanto mayor sea *C*, menor será el error final de entrenamiento. Pero si aumenta demasiado *C*, corre el riesgo de perder las propiedades de generalización del clasificador, ya que intentará ajustar lo mejor posible todos los puntos de entrenamiento (incluidos los posibles errores de su conjunto de datos). Además, un valor de *C* grande, generalmente aumenta el tiempo necesario para el entrenamiento y la capacidad de cómputo con que se cuente.

Con fundamento en las ideas anteriores, surge la pregunta central de esta investigación: ***¿Cómo diseñar una arquitectura para la implementación en MapReduce y en Spark de un sistema de clasificación biclase basado en máquinas de vectores de soporte, que permita ajustar la estrategia a utilizar, con base en las especificaciones del conjunto de datos y los recursos disponibles, con miras a mejorar el tiempo de entrenamiento sin afectar la precisión?***

En concordancia con ello, este trabajo está orientado a establecer parámetros que permitan relacionar el *hardware* disponible, la cantidad de información a procesar y el parámetro del modelo, por medio de una fórmula de costos, que ayude a definir una arquitectura de ejecución de la SVM en MapReduce y Spark, evaluando su impacto en el tiempo de entrenamiento y en la precisión del clasificador. Entiéndase por arquitectura, la configuración adecuada de los parámetros de ejecución de la SVM en MapReduce y Spark, para un conjunto de datos determinado y unos recursos disponibles.

1.1. OBJETIVOS

1.1.1. General

Diseñar una arquitectura para la implementación en MapReduce y en Spark de un sistema de clasificación biclase basado en máquinas de vectores de soporte, que

permita ajustar la estrategia a utilizar, con base en las especificaciones del conjunto de datos y los recursos disponibles, buscando mejorar el tiempo de entrenamiento sin afectar la precisión.

1.1.2. Específicos

a. Analizar los criterios utilizados por las estrategias actuales de paralelización del modelo SVM bajo los *framework* de programación MapReduce y Spark usados en sistemas de clasificación biclase.

b. Definir un modelo de costos que permita establecer los parámetros que inciden en el desempeño del modelo SVM paralelizado bajo MapReduce, en términos de tiempo, precisión y costo computacional.

c. Diseñar una arquitectura para la implementación del modelo de costo propuesto.

d. Evaluar las arquitecturas diseñadas en dos bases de datos públicas, a través de metodologías de validación cruzadas estándar, así como de la medición del tiempo, la precisión y el costo computacional.

CAPÍTULO II

MARCO TEÓRICO Y ESTADO DEL ARTE

En este capítulo se hace una breve descripción de algunos conceptos del aprendizaje automático, esenciales para el desarrollo de esta tesis, enfocados principalmente en el modelo de máquinas de vectores de soporte. Posteriormente, se presentan los resultados encontrados en la revisión del estado del arte sobre las implementaciones del modelo SVM estándar en MapReduce y Spark.

2.1. MÁQUINAS DE VECTORES DE SOPORTE

El aprendizaje automático se define como un conjunto de técnicas y herramientas que utiliza diferentes métodos para detectar automáticamente patrones o características en un conjunto de datos determinado y luego utilizar estos patrones para predecir datos [35]. Como se observa en la Figura 1, durante el proceso de aprendizaje el sistema toma un conjunto de muestras de entrenamiento, luego utiliza un algoritmo de ML para que identifique patrones o características de este conjunto de muestras y finalmente, genera un modelo con los patrones detectados [36]. El desempeño de este modelo se mide por su precisión, es decir, por la proporción de los casos predichos que han sido clasificados correctamente. Con este modelo se puede predecir el comportamiento de datos futuros o tomar decisiones con base en nuevos datos de entrada.

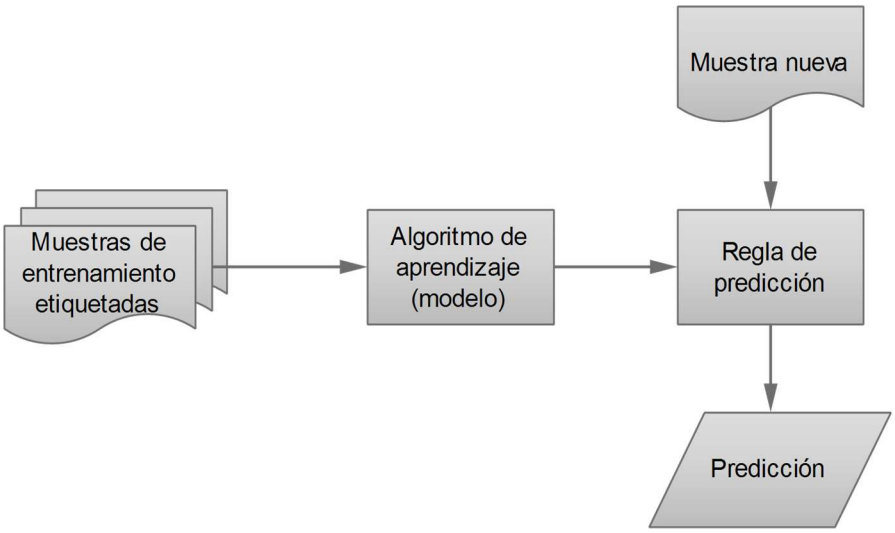


Figura 1. Proceso del aprendizaje automático supervisado. Tomada de [6]

Las técnicas de clasificación son quizá las más utilizadas en el aprendizaje automático y se recurre a ellas en diferentes áreas de la ciencia para resolver problemas del mundo real, como la detección y el reconocimiento de rostros, la clasificación de documentos e imágenes y el filtrado de correos, entre otros. El objetivo de la clasificación es generar una función de decisión $f(x)$ (denominada clasificador) a partir de un conjunto de entrenamiento dado, por medio del cual se podrá predecir la clase a la cual pertenece un objeto nuevo [35]. Cuando previamente se conoce a qué clase pertenecen los objetos del conjunto de entrenamiento, se hace referencia al aprendizaje supervisado; por su parte, cuando no se conocen las clases a las cuales pertenecen, se habla de aprendizaje no supervisado. En este último caso, el modelo de entrenamiento se forma a través de agrupaciones o *clusters* que se crean de acuerdo con las características de los datos de entrada.

Uno de los modelos de aprendizaje automático más ampliamente utilizado para clasificación son la SVM, que fueron propuestas por Cortes y Vapnik en 1995 y han demostrado buenos resultados en la solución de problemas como el reconocimiento de caracteres escritos a mano, la clasificación de imágenes y la

detección de rostros, entre otros [37]. El entrenamiento de este modelo implica encontrar un hiperplano que separe las muestras de dos o más clases con una superficie que maximice la distancia entre éstas. Sin embargo, como se aprecia en la Figura 2, la cual corresponde a un problema perfectamente separable linealmente, pueden existir infinitos planos que separen estas clases, en este caso las muestras de la clase “A” de las de la clase “B”.

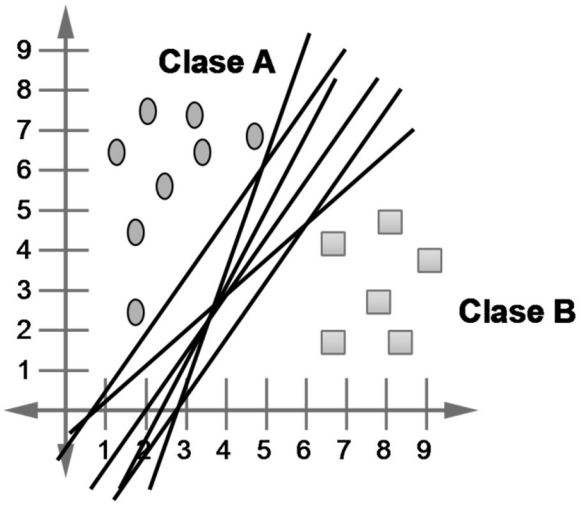


Figura 2. Infinitos planos que separan las clases A y B

Por consiguiente, se debe encontrar entre todos los hiperplanos el que haga que el margen de separación entre ellas sea máximo, es decir, el que maximice la distancia entre el hiperplano y los datos de ejemplo más cercanos a éste (Figura 3). A este hiperplano de separación con el margen máximo se le denomina hiperplano óptimo [38].

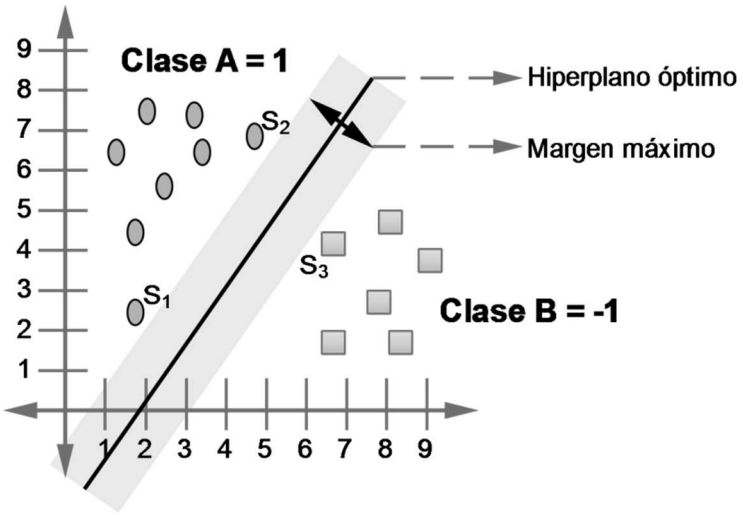


Figura 3. Hiperplano óptimo con el margen máximo

Este hiperplano de separación puede hallarse por medio de una clasificación lineal (Figura 4), para el ejemplo de la Figura 3, s1, s2 y s3 serían las muestras (los vectores de soporte) que definirían el hiperplano óptimo; en caso contrario, cuando las muestras no son separables linealmente, como se observa en la Figura 5 donde las muestras de la clase “A” y de la clase “B” están mezcladas, la clasificación debe hacerse proyectando las muestras de entrenamiento a un espacio transformado en el cual puedan ser linealmente separables y usando una variable de holgura, la cual permite que el modelo sea menos rígido y permitir ciertos errores, es decir, que algunos puntos de clase +1 sean clasificados como -1 y viceversa (Figura 6).

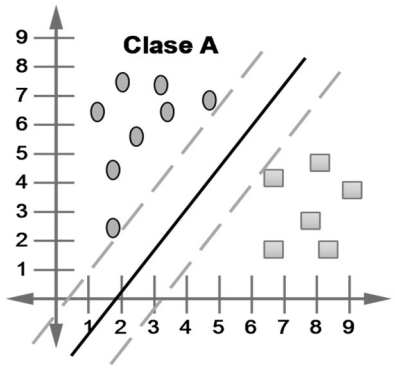


Figura 4. Conjunto linealmente separable

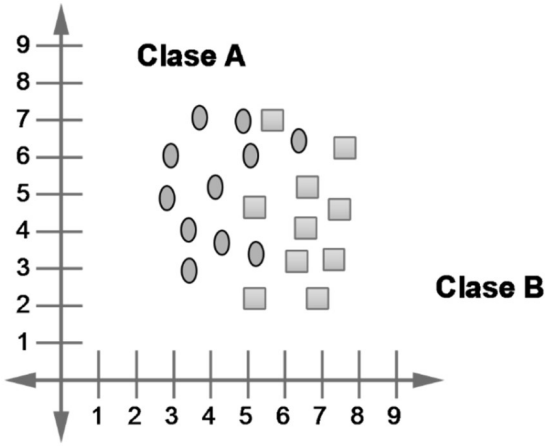


Figura 5. Conjunto no separable linealmente

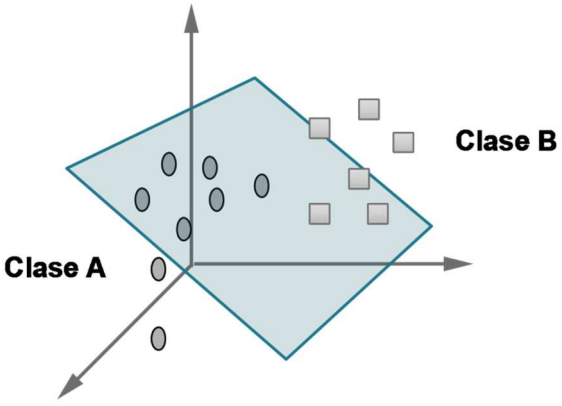


Figura 6. Transformación del espacio característico de R^2 a R^3

Ahora, consideremos un problema de clasificación lineal biclase, en el que los datos de entrenamiento están dados como $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ donde x_n es un vector de entrada, $x \in R^p$ y $y_n \in \{-1, +1\}$ es la etiqueta que le corresponde a cada una de las clases. El margen de separación entre ambas clases se puede definir con la ecuación:

$$f(x) \equiv \langle w.x \rangle + b \qquad (SEQEcuaciónARABIC1)$$

Donde w es el vector de coeficientes de peso, x el vector de entrada y b es un término de sesgo, también conocido como intercepto y especificar que $\langle w.x \rangle$ es el producto interno en el espacio R^p. Teniendo en cuenta que el hiperplano de separación entre estas dos clases es el plano definido como $f(x) = 0$ y los puntos más cercanos a ese hiperplano se encontrarían en los planos $f(x) \pm 1$, se tiene entonces que el margen de separación de este hiperplano se define como:

$m = \frac{2}{||w||}$ **,** siendo $\left|\left|w\right|\right|$ la norma l_2 del vector w.

Por consiguiente, como el objetivo es maximizar el margen m, desde el punto de vista matemático sería seria equivalente a minimizar la norma de w:

$$\min_{w,b} \frac{1}{2} \left|\left|w\right|\right|$$

Sujeta a una serie de restricciones que resultan de la condición de que las clases positivas $(\langle w.x_i \rangle + b) \geq 1 \; para \; y_i = +1$deben estar separadas de las clases negativas $(\langle w.x_i \rangle + b) \leq 1 \; para \; y_i = -1$, las cuales se pueden unir de la siguiente manera $y_i(\langle w.x_i \rangle + b) - 1 \geq 0 \; \forall i$. De esta forma tendríamos la siguiente ecuación:

$$\arg\min_{w,b} \frac{1}{2} \left|\left|w\right|\right|^2$$

Sujeto a: $y_i(\langle w^T \phi(x_i) \rangle + b) \geq 1 \;\; \forall i$ (1)

Esto deja un *problema de optimización primal [7]*, donde x_i es el i-esimo vector de

entrada y y_i es la salida correcta de la SVM para el i-esimo vector de entrada.

Para dar solución a este problema de optimización se recurre al método de multiplicadores de lagrange[3] y a las condiciones de optimización de primer orden, conocidas como "condiciones de Karush, Kuhn y Tucker" (KKT) [39] ver anexo 1; las cuales permiten generalizar problemas de mejora con restricciones de igualdad en problemas con restricciones simples.

De esta manera, aplicando el método lagrangiano al problema primal (2), se tendría una ecuación de la forma:

$$L(\alpha) = \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle x_i . x_j \rangle - \sum_{i=1}^{n} \alpha_i \quad (2)$$

Sujeta a:

$$\alpha_i \geq 0, \sum_{i=1}^{n} y_i \alpha_i = 0 \quad \forall_i = 1, \dots, n$$

Así, el problema primal (2) se convierte en un problema de programación cuadrática llamado problema dual, en el cual la función objetivo D depende únicamente de un conjunto de multiplicadores de lagrange (α_i y n representa el número de ejemplos de entrenamiento. Una vez se hallan los multiplicadores de lagrange de la ecuación (3), se obtiene el vector normal (w) mediante la ecuación:

$$w = \sum_{i=1}^{n} y_i \alpha_i x_i \qquad (3)$$

Y el término de sesgo (b) con:

$$b = \sum_{i=1}^{n} y_i \alpha_i \langle x_i . x_j \rangle - y_j \qquad (4)$$

[3] Según Bishop (2006) también son llamados multiplicadores indeterminados y se utilizan para encontrar los puntos estacionarios de una función de distintas variables sujetas a una o más restricciones.

De la aplicación de las condiciones KKT a este problema de optimización, se establece que las soluciones óptimas α , $(w \ , b \ )$ deben satisfacer

$$\alpha_i \ [y_i(\langle w \ . x_i \rangle + b) - 1] = 0, i = 1, \dots, i$$

Esto implica que solo los x_i para los cuales el margen funcional es uno $(y_i(\langle w. x_i \rangle + b \ ) = 1)$ y por consiguiente se encuentran más cerca del hiperplano, son aquellos que tienen un α_i distintos de cero. Todos los demás x_i tiene un α_i igual a cero. Por lo tanto, los puntos en los que $\alpha_i > 0$ son aquellos que están en el hiperplano óptimo y son los únicos que están involucrados en la ecuación del vector de peso w, ecuación $w = \sum_{i=1}^{n} y_i \alpha_i x_i$ (3). A estos puntos los llamaremos *vectores de soporte*.

En el caso de la clasificación no lineal (SVM no lineal) se usa el método basado en funciones Kernel que consiste en mapear el espacio de entrada a un espacio de representación de dimensión alta en el cual puedan ser linealmente separables a través de una función no lineal llamada función kernel [7]

$$x = \{x_1, x_2, \dots, x_n\} \rightarrow \phi(x) = \left\{\phi(\ [\![x]\!]\ _1, \phi(\ [\![x]\!]\ _2, \dots, \phi(\ [\![x]\!]\ _n\right\}$$

El uso de la función kernel permite proyectar los datos originales $\{x_1, x_2\}$ en un espacio de dimensión superior conocido como “espacio de características” , donde los datos son linealmente separables. Existen diferentes tipos de funciones kernel, algunas de ellas son la polinomial, la Gaussiana, la sigmoidal y la *radial basis function* (RBF) [7]. Se define una función kernel simétrica *K* tal que $K(x_i, x_j) =$, el producto interno se realiza sobre el espacio de características, de modo que el entrenamiento dependa solo de *K* y el mapeo ϕ no sea usado explícitamente. Por tanto, a través de la función kernel, la ecuación (3) quedaría expresada de la forma:

$$\min_{\alpha} D(\alpha) = \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j) - \sum_{i=1}^{n} \alpha_i \qquad (5)$$

Sujeta a:

$$\alpha_i \geq 0 \forall_i = 1, \dots, n$$

$$\sum_{i=1}^{n} y_j \alpha_i = 0$$

Con el objetivo de permitir ciertos errores en la clasificación y que el modelo sea menos rígido, se introduce una nueva variable denominada "variable de holgura", denotada por ξ y un parámetro de regularización de ξ denominado C, el cual es elegido por el usuario para penalizar dicho error; de tal manera que a medida que se aumenta el valor de C, mayor es la penalización a los errores y por consiguiente se permiten menos datos clasificados de manera equívoca. $\sum_i \xi$ sería una cota superior del número de errores de entrenamiento.

Al introducir ξ y C, en el problema de optimización primal, se tendría la siguiente ecuación:

$$min \frac{1}{2}||w|| + C\sum_{i=1}^{n} \xi_i, C > 0 \qquad (6)$$

Sujeto a:

$$y_i(\langle w.\phi x_i \rangle + b) \geq 1 - \xi_i \forall i = 1, \dots, n$$

$$\xi_i \geq 0 \forall i = 1, \dots, n$$

El problema dual quedaría expresado como:

$$\min_{\alpha} D(\alpha) = \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i y_i \alpha_j y_j K(x_i, x) - \sum_{i=1}^{n} \alpha_i \qquad (7)$$

Sujeto a:

$$\sum_{i=1}^{n} y_i \alpha_i = 0$$

$$0 \leq \alpha_i \leq C \forall_i = 1, \dots, n$$

Con base en la ecuación (8), el hiperplano óptimo de separación podría representarse por la siguiente función de decisión o función objetivo:

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i \, K(x_i, x) + b \qquad (8)$$

El problema de programación cuadrático inmerso en la ecuación $(f(x) = \sum_{i=1}^{n} \alpha_i y_i \, K(x_i, x) + b \qquad (8)$ involucra una matriz que tiene un número de elementos igual al cuadrado de los ejemplos de entrenamiento, lo que implica que en la medida en que aumenta el tamaño de la muestra, el tiempo de entrenamiento de este modelo crece exponencialmente [33]. Para reducir este costo computacional se han propuesto soluciones como el algoritmo de SMO [9], formulado por Plat en 1998 y mejorado por Keerthi [10] en 2001, el cual consiste en tomar el problema de programación cuadrática de la SVM y descomponerlo en una serie de pequeños problemas de programación cuadrática (QP), seleccionando un conjunto de trabajo que está compuesto por sólo dos muestras de entrenamiento.

Estos pequeños problemas se resuelven analíticamente, eliminando de esta forma el consumo de tiempo requerido por la solución del algoritmo de programación cuadrática con todas las muestras. Se reduce así el tiempo de entrenamiento de la SVM de cuadrático a lineal respecto al conjunto de muestras de entrenamiento [10], lo cual permite manejar conjuntos de datos más grandes que los manejados por la SVM estándar [7].

El SMO se formula como un algoritmo iterativo, el cual divide el problema de optimización de QP en una serie de pequeños subproblemas de optimización QP, en donde se seleccionan exactamente dos alfas de todos los *n* alfas disponibles para conformar *el conjunto de trabajo* a ser optimizado, encuentra los valores óptimos para estos alfas, y actualiza el SVM para reflejar los nuevos valores

óptimos. Dicho conjunto representa el problema de optimización más pequeño posible para el problema dado e involucra dos multiplicadores de lagrange que obedecen las restricciones de igualdad lineal que se muestran en la ecuación (8). Cada subproblema se resuelve analíticamente hasta la convergencia.

El algoritmo iterativo comprende los siguientes pasos:

1. Encontrar un multiplicador de lagrange α_1 que viole las condiciones de KKT, por medio de un ciclo externo (*outer loop*), el cual recorre todo el conjunto de entrenamiento y comprueba cada muestra en busca de violaciones de las condiciones KKT. Cuando encuentra la muestra se marca para optimizar.
2. Elegir un segundo multiplicador de lagrange α_2. Después de pasar por todo el conjunto de entrenamiento, el ciclo externo itera sobre todos los ejemplos cuyos multiplicadores de lagrange no son ni 0 ni C. Una vez más, cada ejemplo se compara con las condiciones de KKT y los ejemplos que violan las condiciones son elegibles para la optimización.
3. Optimizar el par (α_1, α_2).
4. Repetir pasos 1, 2 y 3 hasta que todos los multiplicadores de lagrange satisfagan las condiciones KKT, dentro de una tolerancia establecida (ξ), con lo cual termina el algoritmo.

Este enfoque analítico evita las técnicas de optimización de QP numéricas que consumen un tiempo considerable dependiendo del número de muestras de entrenamiento, dado que se evita el cálculo de matrices de alta dimensión, pues en cada iteración sólo requiere almacenar una matriz de 2 x 2. Este algoritmo escala entre lineal y cuadrático en el tamaño de conjunto de entrenamiento y dependiendo del problema del análisis de datos. El tiempo de cálculo de SMO está dominado por la evaluación de las muestras.

El SMO no fue pensado para trabajar en entornos de programación paralela, de hecho, su naturaleza secuencial, hace de la paralelización del algoritmo, un problema complejo de resolver. En la literatura se han propuesto diferentes versiones del SMO para trabajar en entornos de programación en paralelo como Hadoop, MapReduce y Spark, con el fin de analizar mayores volúmenes de datos con una optimización en el tiempo de entrenamiento y sin afectar la precisión. A continuación, se hará una breve descripción de estos entornos de programación.

2.2. APACHE HADOOP

Es una herramienta de código abierto de la Fundación Apache para la administración de *clusters*[4] de computación distribuida, utilizada para el manejo de grandes volúmenes de datos. Se basa en el sistema de archivos distribuido creado en Google y se encarga de hacer la gestión de datos de forma eficiente y confiable en un entorno de computación distribuida; es escalable, provee integridad de datos y es tolerante a fallos [40].

2.3. MapReduce

Es un *framework* de programación en paralelo para entornos como Hadoop, que gestionan la información de manera distribuida. Un proceso de MapReduce se ejecuta en dos fases: una de mapeo y otra de reducción, las cuales procesan los datos en paralelo.

Como se ilustra en la Figura 7, en la fase de mapeo la entrada de datos se divide en fragmentos (*splits*) que son asignados a una o varias tareas de la fase de

[4] *Cluster* (informática): conjuntos de computadoras (nodos) interconectadas entre sí normalmente por una red de alta velocidad, que actúan en conjunto como si fuesen una única computadora [58].

mapeo en conjuntos de tuplas *(key, value);* estas tareas se ejecutan de manera paralela e independiente. Cada tarea aplica una función definida por el programador sobre cada ítem del *split* o sobre el *split* completo y entrega los resultados ordenados de nuevo en un conjunto de tuplas *(key, value)*. En este nuevo conjunto la *key* identifica el conjunto entregado y el *value* es el resultado del procesamiento.

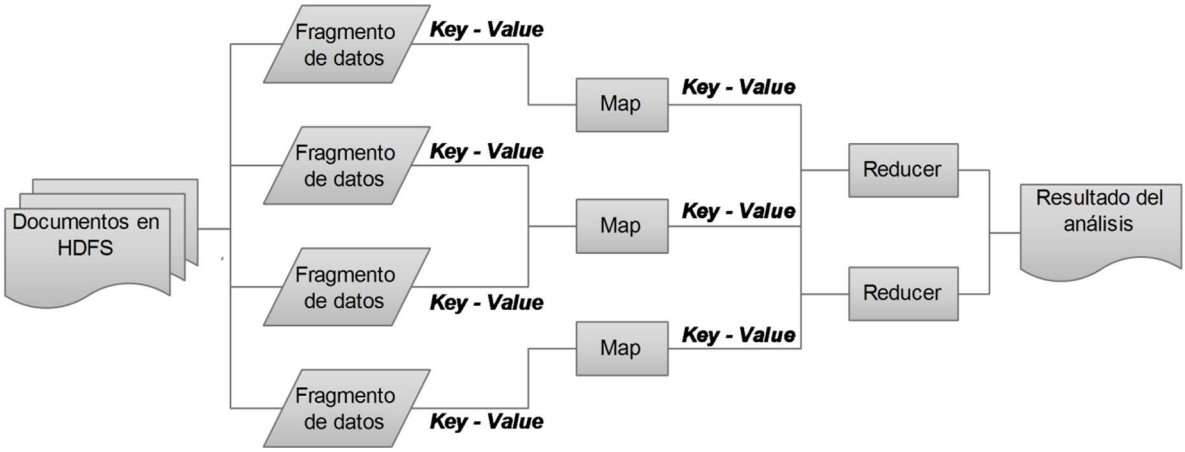


Figura 7. Proceso de datos con MapReduce

La fase de combinador o combiner también conocido como semi-reducer, se ejecuta después de cada tarea de mapeo. En esta fase se toma el conjunto de tuplas generados por la tarea de mapeo y hace una agrupación de los valores que tengan la misma *key*, para luego ser entregados a la fase de reducción.

La fase de reducción es igualmente ejecutada en varias tareas en paralelo llamadas *reducer*. En esta fase se toma el conjunto de tuplas entregadas en la fase de mapeo y se asocian las que tengan la misma *key*, agrupándolas en un solo resultado. En la reducción también se entrega el resultado en la forma *(key, value)*.

Por lo general, tanto la entrada como la salida del trabajo MapReduce se almacenan en un sistema de archivos distribuido. El marco de trabajo MapReduce se hace cargo de las tareas de ejecución, seguimiento y re-ejecución de las tareas fallidas [41].

Los factores que más influyen en el costo de procesamiento de un proceso MapReduce son la comunicación entre los nodos para transmitir la información a ser procesada y el factor de réplica de la información cuando varios procesos de reducción requieren la misma información. También es determinante conocer el tamaño de cada fragmento de datos (*split*) para que se acomode en la memoria principal del nodo que lo procesa y así evitar costos ocultos por escritura a disco.

Características que se deben tener en cuenta en el uso de esta herramienta

- Está diseñada para que cada tarea escriba los datos en disco, lo que puede representar un costo adicional en los procesos.
- Es eficiente para ejecutar procesos por lotes.
- Es deficiente para ejecutar procesos iterativos, debido que debe estar escribiendo en disco constantemente.
- Soporta programación en Java y Python.
- No incluye librerías para la utilización de algoritmos de ML.

2.4. SPARK

Es un marco de trabajo para computación distribuida, pensado para procesamiento de grandes volúmenes de datos, linealmente escalable y tolerante a fallas. Realiza los procesamientos en memoria con el uso del tipo de datos RDD (Resilient Distributed Dataset), que son una colección de objetos particionados e inmutables durante el procesamiento. Los trabajos en Spark se distribuyen en un maestro (*master*) y varios esclavos (*slave*). El maestro envía las tareas a los

esclavos y éstos devuelven los resultados al maestro (Figura 8). Los tipos de tareas que se ejecutan en Spark sobre los RDD, están divididas en dos categorías: transformaciones y acciones.

La definición de estas tareas se hace de forma secuencial en grupos de transformaciones y se ejecutan después de definir una acción (Figura 9). Esta modalidad de procesamiento facilita no tener que recargar los datos desde el inicio en cada transformación o acción que se ejecute, haciendo a Spark muy adecuado para los algoritmos iterativos, los cuales requieren múltiples pasadas sobre un conjunto de datos, así como para aplicaciones que precisan consultas rápidas sobre grandes conjuntos de datos [42] y para el entrenamiento de la SVM.

Características que se deben tener en cuenta en el uso de esta herramienta

- Durante cada tarea de transformación o acción mantiene los datos en memoria. Aunque esto le permite procesar con mayor rapidez los datos, también representa un costo en la capacidad de memoria disponible.
- Es eficiente para ejecutar procesos iterativos.
- Soporta varios lenguajes de programación (Java, Scala, Python y SQL).
- Provee una consola de programación interactiva.
- Procesa datos en diferentes tipos de formato (XML, csv, json, html).
- Incorpora bibliotecas de aprendizaje automático.
- Es tolerante a fallos.

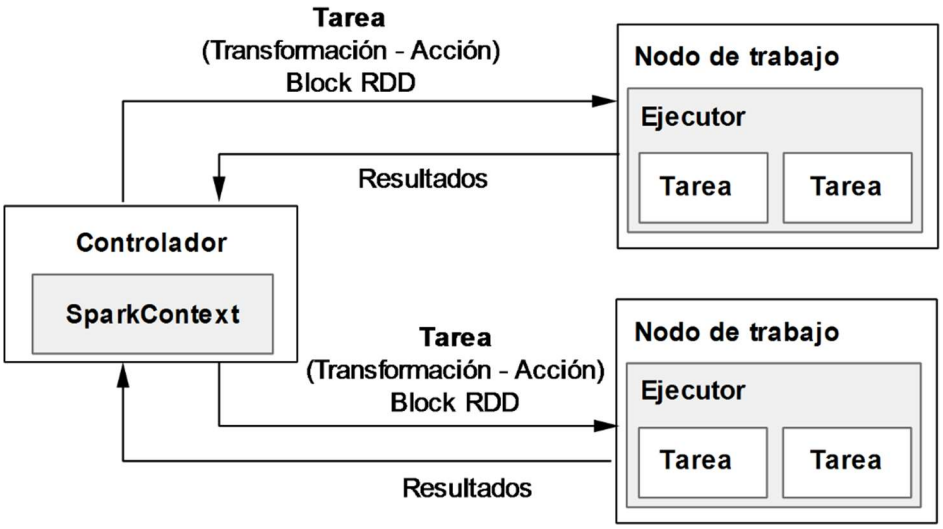


Figura 8. Flujo de trabajo con Spark

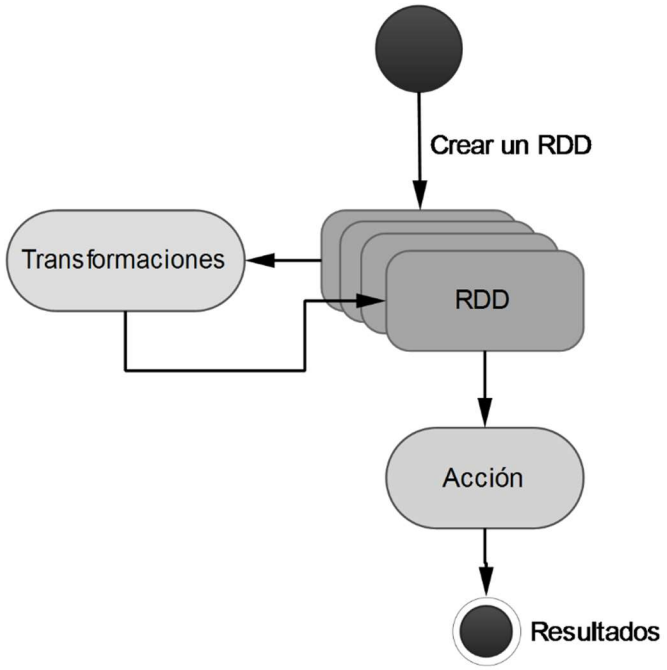


Figura 9. Análisis de datos con Spark

En los últimos años se han publicado un serie de trabajos [11], [12], [21]–[23], [13]–[20] en los cuales se implementa la SVM estándar bajo el *framework* MapReduce y Spark. En estos se utiliza el algoritmo SMO implementado por medio de la librería LibSVM [34]. En el apartado estado del arte, se reseñan de manera cronológica estos trabajos de origen nacional e internacional.

2.5. ESTADO DEL ARTE

La revisión de la literatura se hizo en las bases de datos Scopus, Science Direct, Pearson, IEEE Explore, Google academy y search.carrot2.org. En la búsqueda se utilizaron las siguientes categorías: SVM en paralelo, algoritmo de SVM, SVM en MapReduce y SVM en Spark. Se buscaron referencias entre los años 2009 y 2019.

Se encontraron 88 resultados de los cuales 56 eran propuestas de algoritmos para la solución del modelo SVM secuenciales, 32 eran propuestas para la solución del modelo SVM en paralelo, 13 de ellas utilizaban el *framework* MapReduce y sólo una utilizaba el marco de trabajo Spark, con la implementación que trae en la librería MLlib la SVM linear[5]. Es importante aclarar que no se tuvieron en cuenta los trabajos en los cuales se hizo alguna modificación al modelo estándar SVM.

2.5.1. Paralelización de la SVM: En estos se observa que son comunes algunas fases y métodos en la paralelización de la SVM, como dividir el conjunto de datos en pequeños subconjuntos (en forma aleatoria o con la técnica de balanceo *bootstraping*). Cada subconjunto de datos es asignado a una tarea de mapeo, de modo que el número de tareas dependerá del número de subconjuntos de datos generados; en [15] la asignación de tareas se hace en cascada, de forma que la salida de una o varias tareas de mapeo es asignada a la entrada de otra tarea de mapeo; en las demás se asignan las tareas de forma secuencial. La salida de cada tarea de mapeo es el vector de términos α ver ecuación (8) y el valor de sesgo (*b)* ver ecuación (5), los cuales definen un modelo SVM obtenido para dicha partición.

[5] https://spark.apache.org/docs/2.2.0/ml-classification-regression.html#linear-support-vector-machine

En la etapa de reducción se concatenan los vectores alfa generados por cada tarea de mapeo, dando origen a un vector alfa global. Se toma el valor promedio de *b* para todas las particiones con el fin de obtener un valor de *b* global. Para el entrenamiento se usa la librería SVM del paquete WEKA o la librería LibSVM.

Alham *et al.* [11], [13], adicional a las fases en común expuestas en 2.5.1, utilizaron un modelo de memoria *cache* para almacenar los vectores alfa generados en cada fase de entrenamiento, con el fin de concatenarlos luego en la fase de reducción. Esta implementación se usó para la clasificación de imágenes de animales y se comparó contra una implementación secuencial de la SVM. Reportaron una precisión del 93% (en comparación con el 94% obtenido mediante una implementación secuencial). La velocidad de entrenamiento del proceso paralelo fue doce veces mayor a la del proceso secuencial. Para el entrenamiento de la SVM emplearon la librería que trae el paquete WEKA.

Igualmente, Zhao *et al.* [12] implementaron la estrategia explicada en 2.5.1, a la cual llamaron MRPsvm, y la compararon contra una implementación paralelo desarrollada con el *software* MPI conocida como Pisvm y con la implementación secuencial de la SVM utilizando la librería LibSVM. Realizaron el entrenamiento con cinco conjuntos de datos (*Web, Adult, Mnist, Covtype y Kddcup99*). En todos los casos, MRPsvm obtuvieron tiempos superiores a Pisvm y reportaron una velocidad de entrenamiento cuatro veces mayor que la del algoritmo propuesto comparado con la LibSMV secuencial. Por ejemplo, con el conjunto de datos *web* reportaron tiempos de 306,4 segundos con LibSVM secuencial, 117,5 segundos con Pisvm y 65,8 segundos con MPRsvm. La precisión de los tres clasificadores es la misma; con el conjunto de datos *web* reportaron una precisión del 97,6%.

You *et al.* [21], utilizaron el método de entrenamiento de SVM explicado en 2.5.1. para la predicción de la interacción de proteína-proteína, con LibSVM. En esta

propuesta analizaron el comportamiento del algoritmo con un solo nodo, y luego con 2, 3, 4 y 5 nodos; igualmente, el conjunto de entrenamiento fue dividido en 2, 3, 4, 5, 6 y 7 subgrupos. La precisión de este algoritmo con un nodo y 20 mil muestras fue del 94% con un tiempo de 17.100 segundos, en tanto que con 5 nodos y el mismo número de muestras fue del 93%, con un tiempo de 13.607 segundos.

Por su parte, Priyadarshini *et al.* [22] emplearon el método de entrenamiento de SVM explicado en 2.5.1, proceso que se hizo de forma iterativa. En esta propuesta para la SVM secuencial reportaron sólo el tiempo de entrenamiento de un conjunto de datos con 20.010 muestras, el cual fue de 22.786 segundos, no reportan datos de la precisión de este entrenamiento. La comparación la hacen con el resultado del entrenamiento de 5 conjuntos de datos (*Inosphere, Waveform, Forest Covtype, Adult y Heart*) variando el número de nodos de 1 a 4 nodos. En todos los casos el tiempo de entrenamiento disminuye en la medica que aumenta el número de nodos y la precisión aumenta. Por ejemplo, para el conjunto de datos *Forest Covtype* en el entrenamiento con un solo nodo la precisión fue de 80,43% y el tiempo de entrenamiento de 379,39 segundos, mientras que con cuatro nodos este se redujo a 90,34 segundos con una precisión del 88,10%.

De otro lado, el algoritmo presentado por Le Quoc *et al.* [23], en el cual complementaron MapReduce con Redis[6], proporciona una base de datos en memoria en formato *key-value*, utilizada para almacenar los parámetros del modelo de forma global (*SV, w y b*) y compartirla con los nodos en cada entrenamiento. Usaron, además, el método de entrenamiento de SVM explicado en 2.5.1, a diferencia de que los vectores de soporte y los parámetros del modelo se combinaron con los datos almacenados en la memoria, para volver a ser

[6] Redis Es un motor de base de datos en memoria de código abierto (con licencia BSD).

divididos y entregados a las siguientes tareas de mapeo. El proceso se hizo varias veces y terminó en el momento en que la precisión en la validación del modelo fue igual al paso anterior. No se implementó la fase de reducción. La precisión de esta implementación con 19 tareas de mapeo fue del 92% en la primera iteración y del 93% en la tercera.

Entretanto, Caruana [14] utilizó el método de descomposición de SVM explicado en 2.5.1, para la clasificación de correos marcados como no deseados. Este algoritmo fue extendido con un proceso de optimización basado en Ontología. El autor diseñó SPONTO, abreviatura de SPamONTOlogy, que actúa como una base de retroalimentación para los procesos de entrenamiento y clasificación, que permite controlar la disminución de la eficiencia en los ciclos de entrenamiento. Para el proceso de entrenamiento usó una base de datos llamada spamBase y la librería SVM del paquete WEKA. El tiempo requerido para entrenar el SMO de forma secuencial utilizando 128.000 instancias en un solo nodo de computadora fue de aproximadamente 563 segundos. La misma cantidad de instancias se procesó en aproximadamente 134 segundos mediante un *cluster* Hadoop MapReduce con cuatro nodos de computación. La precisión de este algoritmo fue del 92% comparado con un 94% de un SMO secuencial con 128 mil instancias. Con el uso de SPONTO la precisión fue en promedio del 96%.

Ahora bien, Sun *et al.* [15], aplicaron el método de descomposición de SVM explicado en 2.5.1, con una pequeña modificación en la salida de las tareas de mapeo. La salida de dos tareas de mapeo se envió a una tarea de reducción, en la que se combinaron los conjuntos de vectores de soporte los cuales fueron enviados nuevamente a otra tarea de mapeo, y así sucesivamente hasta que las tareas de mapeo se combinaron en una sola, describiendo un modelo de entrenamiento de la SVM en cascada [7]. Al final de las iteraciones se obtuvo un modelo global SVM. Para el entrenamiento utilizaron la librería LibSVM y las bases

de datos Adult[7] y Forest Cover Type[8]. En las pruebas del algoritmo se aumentó el número de nodos de 1 hasta 8 (1-2-4-8). La precisión con un nodo fue del 84%, en tanto que con ocho nodos fue del 82,99%. En la medida que aumentaron los subconjuntos de entrenamiento la precisión se vio afectada, dado que cuando se eleva el número de subgrupos, estos quedan con menos muestras de entrenamiento; sin embargo, el tiempo de entrenamiento mejora en la medida que se incrementa el número de nodos.

Zhao *et al.* [16] implementaron un algoritmo con dos trabajos MapReduce, uno para entrenar y otro para actualizar los vectores de soporte. En el primer trabajo de MapReduce utilizaron el método de descomposición de SVM explicado en 2.5.1, mediante la técnica de *bootstrap* para dividir el conjunto de entrenamiento en subconjuntos. En el segundo, tuvieron como entrada los vectores de soporte del primer trabajo de MapReduce y un nuevo subconjunto de entrenamiento. El proceso terminó cuando no existieron más subconjuntos de entrenamiento. La precisión de este algoritmo fue evaluada con el conjunto de datos *Mashrooms* [43] que contiene 4.000 datos y comparado contra una implementación de la SVM incremental (ISVM), reportando 97% de precisión de la implementación en paralelo contra 93% de la implementación incremental, mejorando un 4% comparado contra la ISVM y con 30.000 ejemplos del conjunto de datos Connect-4 [44], es 1% menor contra la incremental (76,46% paralelo contra 77,66%). El tiempo de entrenamiento de la implementación en paralelo con el primer conjunto de datos fue de 15,7 segundos contra 2,9 segundos en la implementación incremental, mientras que con el segundo conjunto de datos fue de 0,78 minutos en la implementación en paralelo contra 34,8 en la implementación incremental.

[7] La fuente de datos fue descargada del sitio *web* de NEC laboratory American Inc. http://ml.nec-labs.com/download/data/milde/.

[8] La fuente de datos fue descargada del sitio web http://ftp.ics.uci.edu/pub/machine-learningdatabases/covtype/.

Concluyeron que cuando el conjunto de entrenamiento tiene pocas muestras para entrenar como en este caso de 4.000 ejemplos, la implementación en paralelo toma mucho más tiempo que la secuencial, debido al costo de comunicación entre los nodos del *cluster*.

Kumar *et al.* [17], por su lado, usaron el método de descomposición de SVM explicado en 2.5.1, con la LibSVM. La velocidad de entrenamiento está asociada con el número de nodos utilizados. Con un archivo de 1 MB y dos nodos, el tiempo de entrenamiento fue de 50,6 segundos, con tres nodos fue de 42 segundos y con 4 nodos de 40,3 segundos. Este artículo no cuenta con datos de precisión de la SVM.

Alham *et al*. [18] continúan con la propuesta de trabajo [13] y en esta nueva implementación proponen crear copias del conjunto de datos de entrenamiento con la técnica de balanceo *bootstrapping* [20] y hacen una implementación del proceso de MapReduce en dos capas, en la que los modelos de entrenamiento generados en la primera capa se toman como entrada de la segunda capa. En la segunda capa se unen dos conjuntos de vectores de soporte se realiza de nuevo el entrenamiento y se genera un nuevo modelo SVM. La precisión de este algoritmo coincide con la reportada en trabajos anteriores del 96% contra un 94% de una SVM secuencial; la velocidad en el tiempo de entrenamiento sigue siendo menor que el de una SVM secuencial.

Catak *et al.* [19], sugieren un algoritmo que se ejecuta en un entorno en la nube. El conjunto de entrenamiento es dividido en subconjuntos; en la tarea de mapeo cada subconjunto es combinado con un conjunto de vectores de soporte globales que en la primera iteración es igual a cero; este nuevo subconjunto es enviado a la tarea de reducción donde se realiza el entrenamiento con la LibSVM y valida el modelo con el conjunto de test. Los nuevos vectores de soporte que resultan de

este entrenamiento se combinan con los vectores de soporte globales. El proceso termina cuando la precisión en la validación del modelo es igual a la anterior. Este algoritmo fue probado con cinco diferentes conjuntos de datos *German, Herat, Lonosphere y Satelite*, del repositorio de aprendizaje automático de la UCI[9]. La precisión con cada uno de los conjuntos de datos fue de 77%, 82%, 84% y 90% respectivamente, no hay reportes de comparación con otros modelos de entrenamiento de la SVM, ni reportan tiempo de entrenamiento.

Para concluir, Antoniades *et al.* [20] apuntan a un algoritmo para la clasificación de imágenes con SVM, en donde cada tarea de mapeo genera un conjunto de pares clave valor; la clave es el RGB de la imagen separada por coma y el valor es la clase a la que pertenece la imagen[10]. Este conjunto de datos es enviado a la tarea de reducción en la que se realiza el entrenamiento de un conjunto de clave/valor. De acuerdo con la complejidad del entrenamiento se pueden programar más tareas de reducción. Los resultados de los entrenamientos de cada tarea de reducción son almacenados en un archivo plano. Este algoritmo fue probado ejecutándolo en 1, 2 y 3 nodos con Hadoop, versus la ejecución de la SVM estándar. La precisión de este algoritmo fue del 90% tanto en el entrenamiento con 1, 2 y 3 nodos, como en la ejecución estándar.

[9] http://archive.ics.uci.edu/ml/datasets/

[10] thttp://archive.ics.uci.edu/ml/datasets/Skin+Segmentation

2.6. RESUMEN DE LAS ESTRATEGIAS IMPLEMENTADAS:

Tabla 1. Resumen de las estrategias implementadas del modelo SVM estándar en MapReduce.

Trabajo	Entrada de mapeo	Proceso de mapeo	Salida mapeo	Proceso de reducción	Precisión del algoritmo	Validado contra
Alham *et al.* [11]	*m* particiones, se particiona de forma aleatoria	LibSVM por nodo.	Matriz alfa. Valor de b	Une matrices alfa. Genera el vector de peso y el promedio de los valores de *b*.	Precisión= 93% paralelo – 94% secuencial Tiempo= 11 veces más rápido que serial.	SMO secuencial
Zhao *et al.* [12]	*m* particiones, sin técnica de particionamiento.	LibSVM por nodo y Técnica de almacenamiento de los vectores de soporte en memoria cache.	Matriz alfa. Valor de b	Une matrices alfa. Genera el vector de peso y el promedio de los valores de *b*.	Precisión= igual a la implementación LibSVM Tiempo= 4 veces más rápido que LibSVM.	LibSVM.
You *et al.* [21]	*m* particiones, sin técnica de particionamiento	Entrenar con LibSVM	Conjunto de vectores de soporte.	Une los vectores de soporte y armar un solo modelo.	94% con 20 mil muestras y 1 nodo. 93% 17.100 segundos. Con 5 nodos 93% y 13.607 segundos.	Incrementando el número de nodos

Trabajo	Entrada de mapeo	Proceso de mapeo	Salida mapeo	Proceso de reducción	Precisión del algoritmo	Validado contra
You *et al.* [21]	*m* particiones, sin técnica de particionamiento	Entrenar con LibSVM	Conjunto de vectores de soporte.	Une los vectores de soporte y armar un solo modelo.	94% con 20 mil muestras y 1 nodo. 93% 17.100 segundos. Con 5 nodos 93% y 13.607 segundos.	Incrementando el número de nodos
Priyadarshini *et al.* [22]	*m* particiones, sin técnica de particionamiento.	LibSVM	Conjunto de vectores de soporte por proceso de mapeo y vector de b.	Concatena los vectores de soporte y calcula el promedio de los valores de *b*.	Precisión 1 nodo 80.43%, 4 nodos 88.10%. No hay información de la SVM secuencial Tiempo: Secuencial con 20.010 muestras 22.786,32 segundos. Paralelo: conjunto de datos 1024MB 376,8 segundos.	LibSVM secuencial
Le Quoc *et al.* [23]	*m* particiones, sin técnica de particionamiento.	LibSVM	vectores de soporte, parámetros w y *b* del modelo.	No se implementa.	Precisión 92% en la iteración 1 y 93% en la iteración 3.	Con la precisión en cada iteración.

Trabajo	Entrada de mapeo	Proceso de mapeo	Salida mapeo	Proceso de reducción	Precisión del algoritmo	Validado contra
Alham *et al.* [13]	*m* particiones, sin técnica de particionamiento.	LibSVM por nodo.	Vector de peso y los valores de *b*.	Suma los vectores de peso y saca el promedio de los valores de *b*.	Precisión= 93% paralelo – 94% secuencial Tiempo= 11 veces mayor que SMO secuencial.	SMO secuencial
Caruana *et al.* [14]	*m* particiones, sin técnica de particionamiento	LibSVM por nodo.	Conjunto de vectores de soporte por proceso de mapeo.	Une los vectores de soporte. Genera el vector de peso y el promedio de los valores de *b*.	Precisión= 92% paralelo =94%secuencial Tiempo= 327750 instancias en 134 segundos	SMO secuencial.
Sun *et al.* [15]	*m* particiones, sin técnica de particionamiento.	LibSVM Modelo en cascada	Conjunto de vectores de soporte por proceso de Mapeo	Combina los vectores de soporte y los envía a otra tarea de mapeo.	Precisión= 1 nodo 84% - 8 nodos 83%	1-2-4- y 8 nodos en Hadoop
Zhao *et al.* [16]	*m* particiones, definidas bajo la técnica bootstrap.	LibSVM por nodo.	Conjunto de vectores de soporte por proceso de mapeo.	Une los vectores de soporte. Genera el vector de peso y el promedio de los valores de *b*.	Precisión= 93% paralelo – 97% secuencial con 4000 instancias 77% paralelo – 76% secuencial con 30.000 instancias.	SMO secuencial

Trabajo	Entrada de mapeo	Proceso de mapeo	Salida mapeo	Proceso de reducción	Precisión del algoritmo	Validado contra
Kumar *et al.* [17]	*m* particiones, sin técnica de particionamiento.	LibSVM	Vector de peso y valor de *b*	Suma los vectores de peso y calcula el promedio de los valores de *b*.		LibSVM
Alham *et al.* [18]	*m* particiones, técnica de balanceo bootstrap.	LibSVM por nodo. La última tarea de mapeo une los vectores de soporte y genera el modelo.	Conjunto de vectores de soporte por proceso de mapeo.	No utiliza fase de reduce, sólo tareas de mapeo.	Precisión= 96% paralelo - 94% secuencial Tiempo= 12 veces mayor que SVM secuencial.	SMO secuencial
Catak *et al.* [19]	*m* subconjuntos	Combinar el conjunto de muestras con los vectores de soporte globales.	Subconjunto de datos mezclado con los VS.	Realiza el entrenamiento con la LibSVM.	5 conjuntos de datos 77%, 82%,84% y 90%.	Con ninguno.
Antoniades *et al.* [20]	Sin técnica de particionamiento.	Armar grupos clave/valor (RGB/clase)	Grupos de datos de la forma (r,g,b)=clave clase=valor	Entrenamiento con la LibSVM	Sobre el 90%	Por el número de nodos del *cluster*.

En todos los trabajos se observan mejoras significativas en el tiempo de entrenamiento, sin que esto afecte sustancialmente el porcentaje de precisión en la clasificación (uno o dos puntos por debajo de los algoritmos secuenciales). Pero cabe resaltar que en ningún caso se aborda el problema de definir el número óptimo de particiones, o la configuración que debería ser usada en el *framework* MapReduce o Spark, con el objetivo de obtener mejoras en el tiempo de

entrenamiento ni se establecen parámetros que permitan relacionar el *hardware* disponible (Memoria y CPU), la cantidad de información a procesar y el ajuste de parámetro del modelo, que ayuden a definir una configuración adecuada de ejecución de la SVM en MapReduce y Spark, con el objetivo de obtener los mejores resultados posibles.

Es necesario tener en cuenta que una aplicación en MapReduce y Spark tiene limitaciones de ejecución debido al *hardware* disponible. Por tal motivo aspectos como el costo de comunicación entre los nodos, el tamaño de la memoria RAM y la capacidad de procesamiento, pueden afectar de manera significativa el tiempo de ejecución de las tareas o procesos que se ejecuten, como se menciona en los trabajos que se describen a continuación.

González-Vélez *et al.* [45], analizaron el rendimiento de una aplicación en MapReduce de ordenamiento y escritura aleatoria en Hadoop, en un *cluster* con un número de nodos de trabajo que varia entre 1 y 16 nodos en cada ejecución. Usaron el mismo tipo de conjunto de datos dividido en tres tamaños diferentes 1GB, 2GB y 4GB con un algoritmo. En cada prueba se estableció el mismo número de tareas de mapeo (8) para los tres conjuntos de datos. Los autores concluyeron que, aunque el tiempo de ejecución aumenta en la medida que se incrementa el número de nodos, dejando fijo el número de tareas de mapeo, este crecimiento no es del todo lineal, y la eficiencia en la ejecución de las tareas también varía, los cual se ve reflejado en el número de tareas fallidas reportadas y reasignadas durante el experimento. Los autores explican que esta variabilidad en el tiempo y la eficiencia se debe al movimiento de datos entre los nodos y a que el nodo maestro prefería los nodos que tenían las mejores características en procesamiento y memoria o que respondían más rápido a las llamadas del nodo maestro para asignar las tareas.

Por su parte, Wang *et al.* [46] usaron la herramienta MRPerf para simular un ambiente MapReduce y explorar el efecto de la localidad de datos en el rendimiento general de un proceso; en el análisis incluyeron aspectos como la topología de red y las fallas de *software* y *hardware*. Concluyeron que cuando los datos no están ubicados de forma local en el nodo, las tareas de mapeo consumen mayor tiempo de ejecución, debido a que este tiene que hacer una recuperación de datos remotos. La ubicación de los datos afecta significativamente el tiempo de ejecución, que llega a ser de hasta un 284% cuando los datos están ubicados en una red externa, en comparación con una ubicación local de los mismos, lo que clasifica como un factor importante la ubicación de los datos.

Igualmente, Rajaraman *et al.* [47], introdujeron un modelo de medición del costo para los algoritmos MapReduce, en el cual establecieron que los factores predominantes en el costo de un proceso de este tipo, son el volumen de datos a ser transferidos entre los nodos en la fase de mapeo y de reducción y el factor de replicación que sufre la información procesada. Entiéndase por factor de replicación la razón entre volumen de datos que entran a la fase de mapeo y el volumen de datos que entran a la fase de reducción. En el modelo propuesto por Rajaraman y su equipo, se ejemplifica en un algoritmo que hace un *join* entre tres tablas de una base de datos relacional. El *join* se puede realizar con uno o dos procesos (una o dos fases) MapReduce. El costo de comunicación del *join* de una fase está determinado por el tamaño de las tablas de entrada y el factor de replicación de datos que requiere el algoritmo (cantidad de cubetas intermedias generadas, cuyo tamaño depende de la memoria disponible por nodo). El costo de comunicación del *join* de dos fases se obtiene a partir del tamaño de las tablas de entrada y del tamaño del *join* resultante (el número de las tuplas que coinciden en las dos tablas). Luego de calculados los costos de cada *join* (de una y de dos fases) se comparan para poder determinar cuál de las dos estrategias es preciso

implementar de acuerdo con la configuración de la máquina que se tenga y el tamaño de los datos de entrada. Se tiene una restricción y es la memoria disponible por nodo para las tareas de reducción, y el factor de replicación de la información en el join de una fase; ambos factores inciden en la selección de la estrategia de un paso o de dos pasos.

Adicionalmente, Priyadarshini *et al.*, K. Morton *et al.,* y Murphy *et al.* [24], [25], [26] explican el uso de programas como ParaTimer y Parallax, para obtener información detallada de métricas sobre el uso de memoria, CPU y red, relacionadas con la ejecución de trabajos MapReduce y poder tener un perfil del comportamiento de la aplicación. Estos programas se instalan en un sistema operativo Linux y son monitorizadas al momento de lanzar un trabajo MapReduce para obtener valores de consumo de memoria, CPU y red.

En los trabajos de Z. Zhang *et al.* y F. Tian *et al.* [27], [28] se analiza el flujo de ejecución de una aplicación MapReduce (mapeo, combinador y reducción) utilizando modelos como CRESP [29], ARIA [30] y AROMA [32], que permiten estimar el tiempo de ejecución de un trabajo a partir de pruebas realizadas con un conjunto reducido de nodos y de datos de entrada. Por su lado, Vapnik [29] propuso un modelo denominado CRESP que permite dimensionar los recursos óptimos que necesita un *cluster* para ejecutar trabajos sobre Hadoop.

Verma *et al. [31]*, tomaron como base un trabajo previo donde presentaron el modelo ARIA (*Automatic Resource Inference and Allocation*) [30] con el cual calcularon los límites inferior y superior del tiempo de ejecución de un trabajo MapReduce y lo extendieron. En este trabajo los autores se enfocan en la simulación de las decisiones del *job master* y en las políticas de asignación de recursos, añadiendo factores de escalabilidad para obtener el tiempo de ejecución para datos de mayor tamaño mediante una regresión lineal simple, y establecer un

tiempo estimado de la finalización de la aplicación de acuerdo con el *hardware* disponible.

Keerthy *et al.* [32] plantearon el modelo AROMA, sistema que permite obtener los recursos óptimos para que un trabajo sobre Hadoop logre los objetivos establecidos. Blackard *et al.* [46] presentaron el simulador MRPerf, el cual realiza una simulación del comportamiento de aplicaciones MapReduce, para poder determinar el impacto de la utilización de distintas topologías de red en la ejecución de aplicaciones MapReduce.

En estos trabajos se evidencia la importancia de la medición de parámetros del sistema como la cantidad de datos de entrada, escritos y movidos entre el *map* y el *reduce* (*shuffle*), además del tiempo de procesamiento (CPU) y el consumo de memoria, entre otros, para establecer un tiempo estimado de finalización en la ejecución de trabajos de MapReduce dependiendo del *hardware* disponible.

Por tal motivo, para la definición de la formula de costos de nuestra implementación del algoritmo SVM en MapReduce y Spark tomamos como base el modelo ARIA propuesto por Verma [31] *et al*., con el objetivo de crear un modelo de costos específico para este tipo de trabajos que tienen un alto costo computacional. El modelo ARIA se fundamenta en el teorema de *Makespan* propuesto por Verma *et al*. [31] con base en ideas propuestas por Graham [48]. Y tiene como propósito obtener la secuencia óptima para la puesta en marcha de una serie de tareas por parte de un conjunto de trabajadores, que permita minimizar su tiempo de finalización. Verman utiliza este trabajo para establecer una metodología que le permita predecir el tiempo de finalización de un trabajo con un conjunto dado de n tareas que son procesadas por k servidores (o k contenedores en entornos MapReduce y Spark).

Para la explicación del modelo ARIA se utiliza el concepto el concepto de número de *slots*[11] empleado por YARN (*Yet-Another-Resource-Negotiator*)[12] para planificar la ejecución de los trabajos de MapReduce y Spark de acuerdo con los recursos de un *cluster* y el número de contenedores que se puedan crear. Un contenedor representa la unidad básica para la asignación de recursos como la memoria y el número de núcleos virtuales de CPU. Para que *YARN* pueda administrar los recursos del *cluster* se deben configurar algunos valores relacionados con la memoria, la CPU y los contenedores en los archivos mapred-site.xml y yarn-site.xml, para que sean asimilados por cada trabajo MapReduce y Spark.

Los parámetros que se deben configurar son los siguientes:

Núcleos virtuales de CPU

- *mapreduce.map.cpu.vcores*: número de núcleos virtuales de CPU requeridos por cada tarea de mapeo.
- *mapreduce.reduce.cpu.vcores*: número de núcleos virtuales de CPU requeridos por cada tarea de reducción.
- *yarn.app.mapreduce.am.resource.cpu-vcores:* número de núcleos virtuales de CPU requeridos por el *master* de un *job*.
- *yarn.scheduler.minimum-allocation-vcores:* número más pequeño de núcleos virtuales de CPU que pueden asignarse para un contenedor.
- *yarn.scheduler.maximum-allocation-vcores:* mayor número de núcleos virtuales de CPU que se pueden asignar a un contenedor.

[11] Se usa en Hadoop para denotar el número de tareas que se pueden ejecutar simultáneamente en un nodo.

[12] Es una tecnología de administración central de recursos que reconcilia la forma en que las aplicaciones utilizan los recursos del sistema de Hadoop con los agentes de administración de nodo que monitorizan las operaciones de procesamiento de nodos individuales del *cluster* [40].

- *yarn.nodemanager.resource.cpu-vcores:* número de núcleos virtuales de CPU que se pueden asignar para los contenedores.

Memoria

- *yarn.app.mapreduce.am.resource.mb:* memoria en MB requerida para el *master* de un *job*.
- *mapreduce.map.memory.mb:* cantidad de memoria física en MB, asignada a cada tarea *map* de un *job*.
- *mapreduce.reduce.memory.mb:* cantidad de memoria física en MB, asignada a cada tarea de reducción.
- *yarn.scheduler.minimum-allocation-mb:* cantidad más pequeña de memoria física,*¿* en MB, que puede solicitarse para un contenedor.
- *yarn.scheduler.maximum-allocation-mb:* máxima cantidad de memoria física en MB, que se puede asignar a un contenedor.

Contenedores

- *yarn.nodemanager.resource.memory-mb*: cantidad de memoria física, en MiB, que se puede asignar a los contenedores.
- *yarn.scheduler.minimum-allocation-mb*: cantidad más pequeña de memoria física, en MiB, que puede solicitarse para un contenedor.
- *yarn.scheduler.increment-allocation-mb*: incremento de memoria de un contenedor.
- *yarn.scheduler.maximum-allocation-mb*: máxima cantidad de memoria física, en MiB, que se puede solicitar para un contenedor.
- *yarn.nodemanager.resource.cpu-vcores*: número de núcleos de CPU virtuales que se pueden asignar para los contenedores.

- *yarn.scheduler.minimum-allocation-vcores*: número más pequeño de núcleos de CPU virtuales que pueden solicitarse para un contenedor.
- *yarn.scheduler.increment-allocation-vcores*: incremento de núcleos de CPU virtual de un contenedor.
- *yarn.scheduler.maximum-allocation-vcores*: mayor número de núcleos de CPU virtuales que se pueden asignar a un contenedor.

En el modelo ARIA Verma utiliza la metodología aplicada por Graham, para acotar los límites inferiores y superiores del tiempo de ejecución de una aplicación MapReduce, a partir de los parámetros extraídos en la calibración y caracterización de la aplicación. Las tareas de *map* y *reducer* son consideradas como una serie de *n* tareas de duración t_iy los contenedores creados por Yarn son considerados los *k* servidores en donde se ejecutarán las *n* tareas.

Según Verma, primero se debe crear un perfil de la aplicación en un escenario concreto, obteniendo información relevante al trabajo MapReduce como el tiempo de ejecución de las fases de *map* y *reduce*, la cantidad de datos de entrada, escritos, y movidos entre el *map* y el *reduce* (shuffle) y el tiempo de procesamiento (CPU), entre otros. Luego, con base en los datos tomados se determina el tiempo medio $(t|\ \ |avg)$ y el tiempo máximo $(t|\ \ |max)$ de las *n* tareas de *map* y *reduce*, finalmente con estos valores se calculan los límites inferior (T_{low}) y superior (T_{up}) de cada una de las fases, como se explica a continuación.

Para la fase de mapeo el límite inferior (Tm_{low}) está determinado por el número total de tareas de mapeo (n_m), dividido por el número de contenedores asignados (k_c) y multiplicado por el tiempo promedio de ejecución de las *n* tareas de mapeo (tm_{avg}). En el mejor de los casos todas las *n* tareas estarán distribuidas equitativamente en los *k* contenedores, por lo que el total del trabajo se ejecutará

lo más rápido posible. Por consiguiente, el límite inferior estará representado por la siguiente fórmula:

$$\text{Límite Inferior:} \qquad Tm_{low} = \frac{n_m}{k_c} tm_{avg} \qquad (9)$$

Este límite inferior permitirá conocer cuál es el tiempo mínimo de ejecución de un trabajo MapReduce, suponiendo que todas las *n* tareas estarán distribuidas equitativamente en los *k* contenedores.

Para calcular el límite superior (Tm_{up}) se considera el peor de los casos, que es la tarea con mayor tiempo de ejecución (tm_{max}); por consiguiente, al número total de tareas se le resta esta tarea y se suma al final de la ecuación. Se obtiene la siguiente fórmula para el límite superior:

$$\text{Límite superior:} \qquad Tm_{up} = \left(\frac{n_m}{k_c} - 1\right).tm_{avg} + tm_{max} \quad (10)$$

La misma lógica de la fase de mapeo se aplica a la fase de reducción, lo que arroja las siguientes fórmulas para los límites inferior (Tm_{low}) y superior (Tm_{up}):

$$\text{Límite inferior:} \qquad Tr_{low} = \frac{n_r}{k_c} tr_{avg} \qquad (11)$$

$$\text{Límite superior:} \qquad Tr_{up} = \left(\frac{n_r}{k_c} - 1\right).tr_{avg} + tr_{max} \qquad (12)$$

Finalmente, para obtener los límites inferior (Tt_{low}) y superior (Tt_{up}) del tiempo total de finalización del trabajo MapReduce se unen las fórmulas de los límites inferiores y superiores de cada fase:

$$\text{Límite superior total:} \qquad Tt_{up} = Tm_{up} + Tr_{up} \qquad (13)$$

Límite inferior total: $$Tt_{low} = Tm_{low} + Tr_{low} \quad (14)$$

Estas fórmulas representan predicciones optimistas y pesimistas del tiempo de finalización del trabajo MapReduce. Verma calcula estos tiempos para aplicaciones que tienen un costo computacional bajo, como el conteo de palabra.

CAPÍTULO III

SOLUCIÓN PROPUESTA Y VALIDACIÓN

En este capítulo se detalla la solución propuesta y los experimentos realizados para evaluar el tiempo y la precisión de la arquitectura planteada e implementada, sistematizando los resultados obtenidos.

3.1. SOLUCIÓN PROPUESTA

Esta tesis se enmarca en el paradigma cuantitativo con un enfoque descriptivo. Todo el modelo de implementación está basado en los *framework* de MapReduce y Spark. Se implementó una arquitectura con el algoritmo LibSVM y se realizaron varias pruebas usando bases de datos de problemas reales, comparando los resultados con las distintas técnicas propuestas en la literatura.

Para determinar qué aspectos del uso de MapReduce y Spark afectan el entrenamiento de la SVM, se procedió a realizar una implementación propia utilizando la librería LibSVM, la cual permitió posteriormente incluir los criterios de ejecución bajo MapReduce y Spark. Igualmente, se realizó la búsqueda de los archivos fuentes de las implementaciones en MapReduce de la SVM de los artículos mencionados en el estado del arte; se encontró el código fuente de la estrategia explicada en Sun *et al.* [15]. Se implementó de acuerdo con las instrucciones que indican estos autores en su wiki cascadeSVM[13]. En el momento de la ejecución generó errores en la estructura de uno de los archivos requerido como archivo de entrada. A la fecha de la implementación de las estrategias propuestas no se hallaron más códigos fuentes disponibles en la Internet de algoritmos de la SVM estándar en MapReduce que se pudieran replicar.

[13] Ampliar información en: https://code.google.com/p/cascadesvm/wiki/

En la implementación con el *framework* MapReduce, se contó con tres tareas: configuración, mapeo y reducción. En la tarea de configuración se definieron los tipos de datos de entrada y salida de las tareas de mapeo y reducción respectivamente, se definieron los nombres de las clases de mapeo y reducción, las rutas de los archivos de entrada y de salida y los parámetros de configuración de la SVM.

En la implementación con el marco de trabajo Spark, se utilizó una estructura de datos llamada *DataFrame* y funciones tipo *UDF* y ventana. Igualmente, se implementaron dos tareas, una de mapeo y otra de reducción. Los modelos resultantes del entrenamiento se guardaron como objetos tipo *DataFrame* comprimidos en formato *parquet*.

La implementación propia de la arquitectura de clasificación biclase con el modelo SVM consistió en el diseño y la evaluación de cuatro estrategias para la implementación del modelo en MapReduce y una estrategia para la implementación en Spark:

- Estrategia uno en MapReduce: entrenamiento sin combinador
- Estrategia dos en MapReduce: entrenamiento con combinador
- Estrategia tres en MapReduce: entrenamiento en el reducer
- Estrategia cuatro en MapReduce: entrenamiento iterativo
- Estrategia en Spark: entrenamiento iterativo

Estas estrategias se explican en detalle en la sección 3.2 (Arquitectura propuesta), una vez se introduzcan los conceptos básicos de la estrategia de implementación.

Para el entrenamiento con cada una de las estrategias se decidió dividir el

conjunto de datos de entrenamiento de forma aleatoria; no se optó por la técnica de *bootstrapping* usada en los artículos de J. Zhao *et al*. [16] y N. Alham *et al*. [18], debido a que el remuestreo con reemplazo implica que cada subgrupo tenga el mismo número de elementos que la muestra original [39], lo que aumentaría la cantidad de muestras a entrenar en factor de los subgrupos que se deseen crear, afectando el tiempo de entrenamiento. Este aumento en la cantidad de muestras de entrenamiento para un algoritmo como la SVM, donde el costo computacional depende del número de muestras de entrenamiento, afectaría significativamente el tiempo de entrenamiento. La cantidad de subgrupos en los que se dividió el conjunto de entrenamiento dependió de la cantidad de tareas de mapeo que se querían ejecutar y su tamaño fue determinado en *bytes*. Para evitar que algún subgrupo quedara con muestras de una sola clase y se presentaran errores en el entrenamiento, se realizó un preprocesamiento de los datos en el cual se eliminan las muestras repetidas y luego se organizan por clases de forma balanceada, logrando una mejor distribución de las muestras de entrenamiento dentro del conjunto de datos. Este preprocesamiento fue implementado para las cuatro estrategias en MapReduce y la estrategia en Spark.

Como metodología de validación para las diferentes estrategias, se usó la metodología de validación cruzada (*cross-validation*) mencionada en los artículos [15], [16] y [49] con 5 *folds*. Finalmente, se calculo la media aritmética de los resultados de eficiencia de cada iteración para obtener un resultado único de eficiencia.

En la identificación de los conjuntos de datos utilizados en los trabajos expuestos anteriormente, se identificó la base de datos de cobertura de bosques [50], para hacer el entrenamiento y poder tener un punto de comparación de los resultados obtenidos. Otros conjuntos de datos como los utilizados para la clasificación de imágenes no se encontraron disponibles y otros cuentan con menos de 1.000

datos para analizar, lo cual no los hace candidatos para utilizarlos en un algoritmo de MapReduce debido a que con tan pocos datos puede resultar más costoso en tiempo que una implementación secuencial.

3.2. ARQUITECTURA PROPUESTA

A continuación, se explicará el diseño de la arquitectura de entrenamiento con el modelo SVM para clasificación biclase en MapReduce y Spark con base en las estrategias analizadas en los antecedentes y propuestas en este estudio.

3.2.1. Diseño e implementación en MapReduce y Spark

El diseño de la paralelización de la SVM consiste de dos procesos. El primero está relacionado con el preprocesamiento de los datos y el segundo con el entrenamiento de la SVM. En ambos se implementaron dos tareas, una de mapeo y otra de reducción, las cuales son ejecutadas de forma paralela en los nodos que conforman el *cluster*.

En el proceso de entrenamiento el número de tareas de mapeo es equivalente al número de subconjuntos creados. A su vez, este número es determinado por el modelo de costos propuesto en este trabajo y que se detalla en la sección 3.5.

A continuación, se explica en detalle la implementación del preprocesamiento de los datos, de las tareas de mapeo y reducción.

- **Etapa de preprocesamiento**

El preprocesamiento de los datos consiste en leer el archivo con los datos de entrenamiento en cada tarea de mapeo y separar cada muestra en un conjunto *clave-valor*, donde la clave es la muestra y el valor es la etiqueta de la muestra. En

la fase de reducción se concatenan las muestras por etiqueta y se van guardando en un archivo de texto de forma balanceada ver Figura 10.

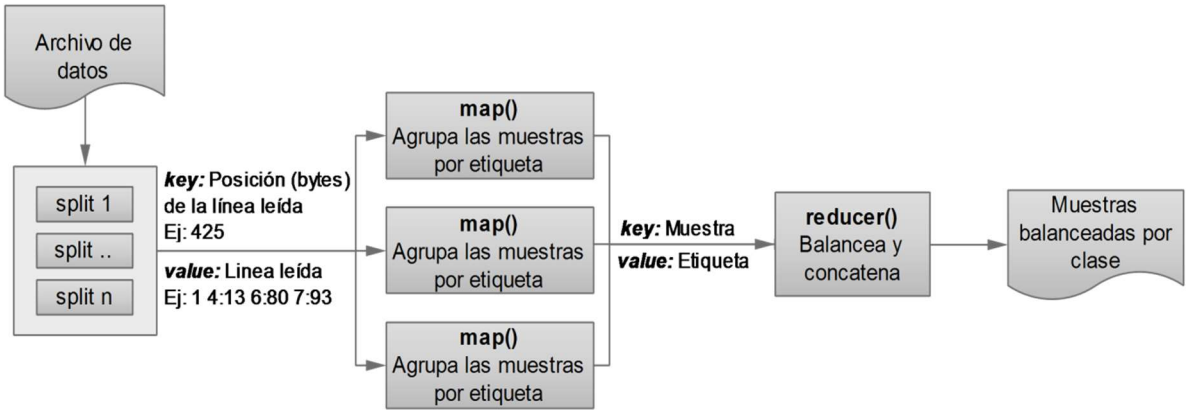


Figura 10. Preprocesamiento de los datos en MapReduce

```
1  Proceso MapperPreprocess
2
3      SubProceso map(Text key, Text value)
4
5          //el "value" contiene cada una de las muestras de entrenamiento
6          Leer value
7
8          yi <- value.etiquta
9          xi <- value
10
11         //Se guarda cada uno de los modelos entrenados en formato (key, value)
12         Escribir (xi, yi)
13
14     FinSubProceso
15
16 FinProceso
```

Figura 11. Tarea de mapeo en el preprocesamiento de los datos

En la Figura 11 se muestra en seudocódigo de la tarea de mapeo del procesamiento, en la cual se lee cada una de las muestras de entrenamiento y se almacena la muestra y la etiqueta en dos variables *xi, yi* respectivamente, líneas 8, 9 y 12.

```
Proceso ReducerPreprocess
    nroMuestrasA <- 0
    nroMuestrasB <- 0
    ClaseMayor <- ""
    ClaseMenor <- ""
    Dimension MuestrasA[0]
    Dimension MuestrasB[0]
    cont, i, j, k, p <-0
    SubProceso reducer(Text key, Lista (value))
        //El "value" contiene el modelo de entrenamiento de la tarea de map
        leer value
        Si value.etiqueta== "1" Entonces
            //Se concatenan todos los SV de cada modelo
            MuestrasA[cont]<-value.muestra
            nroMuestrasA <- nroMuestrasA + 1
        Sino
            //Se concatenan todos los SV de cada modelo
            MuestrasB[cont]<-value.muestra
            nroMuestrasB <- nroMuestrasB + 1
        Fin Si
        cont <- cont + 1
    Fin SubProceso
    Si nroMuestrasA >= nroMuestrasB  Entonces
        claseMayor <- "1"
        claseMenor <- "-1"
        totalxClaseA <- nroMuestrasA/nroMuestrasB
        totalxClaseB <- 1
        muestrasRestantes <- nroMuestrasA - (totalxClaseA*nroMuestrasB)
    Sino
        claseMenor <- "-1"
        claseMenor <- "1"

        totalxClaseA <- nroMuestrasB/nroMuestrasA
        totalxClaseB <- 1
        muestrasRestantes <- nroMuestrasB - (totalxClaseB*nroMuestrasA)
    Fin Si
    Mientras i<= (nroMuestrasA + nroMuestrasB) Hacer
        Mientras j <= totalxClaseA Hacer
            //Se crea un archivo solo con los vectores de soporte
            Escribir (Text(muestrasEnttoBalanceadas))
            j <- j+1
        Fin Mientras
        j <- 0
        Mientras k<=totalxClaseB Hacer
            //Se crea un archivo solo con los vectores de soporte
            Escribir (Text(muestrasEnttoBalanceadas))
            k <- k+1
        Fin Mientras
        k <- 0
        i <- i+1
        Mientras p<=muestrasRestantes Hacer
            //Se crea un archivo solo con los vectores de soporte
            Escribir (Text(muestrasEnttoBalanceadas))
            p <- p+1
        Fin Mientras
    Fin Mientras
FinProceso
```

Figura 12. Tarea de reducción en el preprocesamiento de los datos

En la **¡Error! No se encuentra el origen de la referencia.** Figura 12 se aprecia el seudocódigo de la tarea de reducción, que consiste en leer cada una de las muestra, separarlas por clase (etiqueta) y almacenarlas en dos estructuras de listas separadas y contarlas (líneas de la 11 a la 21 en la **¡Error! No se encuentra el origen de la referencia.**); luego en las líneas de la 23 a la 35 se busca cuál de las clases tiene mayor número de muestras y en qué proporción respecto a la clase con menor muestra. Con estos datos en las líneas 26 y 33 se calcula la proporción de la clase mayor respecto a la clase menor y finalmente en las líneas 38 a la 55 se crea un archivo de texto con las muestras balanceadas de acuerdo con la proporción calculada.

➢ **Tarea de mapeo**

Todas las estrategias tienen implementadas la misma funcionalidad en la tarea de mapeo, la cual se presenta de forma gráfica en la Figura 13 y en pseudocódigo en la Figura 14. Cada tarea de mapeo lee un subconjunto de los datos de entrenamiento de tamaño *n*, llamado *split* (línea 8 en la Figura 14), en formato <*key,value*>; en donde *key* representa la posición en *bytes* de la muestra leída y *value* representa la muestra leída (línea 11 en la Figura 14). En la medida en que se leen los datos, el algoritmo crea dos vectores, uno que contiene las etiquetas de las muestras y otro que agrupa los valores de las características de cada muestra (líneas 14 y 15 en la Figura 14). Con estos vectores y los parámetros del modelo SVM se hace el entrenamiento utilizando el método *svm_train()* de la LibSVM (línea 19 en la Figura 14). En cada entrenamiento se genera un submodelo de entrenamiento con un conjunto de vectores de soporte, un conjunto de coeficientes alfa y un valor de sesgo o intercepto *b*. Estos datos son enviados a la tarea de reducción o de combinador en formato <*key,value*> (línea 24 en la Figura 14), donde *key* es el texto “modelo_SVM” y *value* es el submodelo serializado en formato Avro [51].

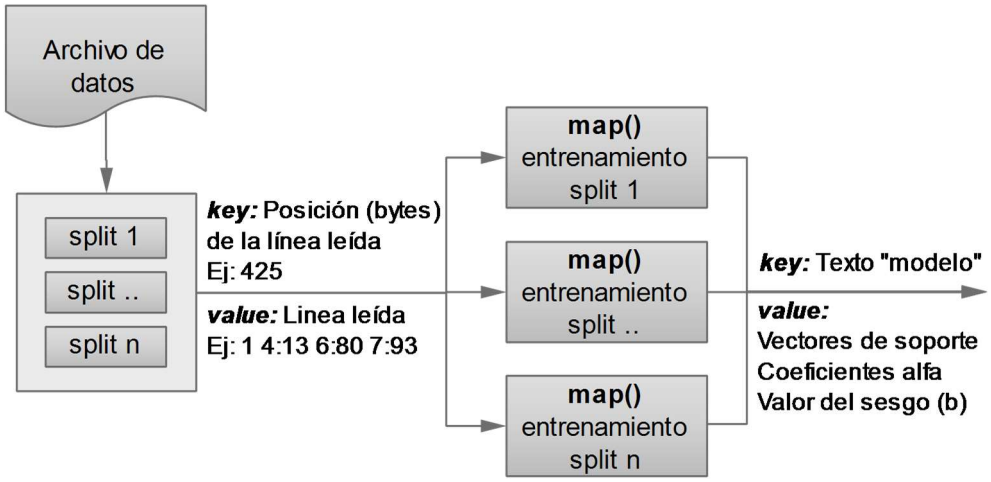


Figura 13. Tarea de mapeo SVM en MapReduce

```
Proceso MapperSVM

    //vector para guardar los valores de las características de la muestra
    Dimension vectorX[0]
    //vector para guardar las etiquetas de las muestras
    Dimension vectorY[0]

    SubProceso map(Text key, Text value)

        //el "value" contiene el subconjunto de entrenamiento
        Leer value

        Para i<-0 Hasta Longitud(value) Hacer
            vectorY[i] <- etiqueta
            vectorX[i] <- muestra
        Fin Para

        //se entrena cada split con la libsvm
        modelo <- svm_train(vectorX, vectorY)
        //Se serializa el objeto modelo a un objeto AVRO
        modeloMap <- AvroValue(modelo)

        //Se guarda cada uno de los modelos entrenados en formato (key, value)
        Escribir (Text("modelo_SVM"), modeloMap)

    FinSubProceso

FinProceso
```

Figura 14. Pseudocódigo. Tarea de mapeo en MapReduce

➢ **Tarea de reducción**

En la Figura 15 se presenta de forma gráfica la información que llega de las tareas de mapeo y entra a la tarea de reducción y en la Figura 16 se detalla el pseudocódigo de esta tarea. En esta fase se leen cada uno de los modelos entrenados previamente como se aprecia en la línea 11 de la Figura 16, donde la variable *value* contiene cada uno de los submodelos entrenados; en las líneas 13 y 14 de la Figura 16 se concatenan los vectores de soporte y los coeficientes alfa; en tanto que en la línea 17 de la Figura 16 se suma la cantidad de vectores de soporte de cada submodelo y en las líneas 19 y 26 se calcula el valor del sesgo o intercepto *b* del modelo con un promedio sopesado debido a que este valor depende del número de vectores de soporte de cada submodelo. Al final del proceso en la línea 32 de la Figura 16, se obtiene un modelo entrenado final.

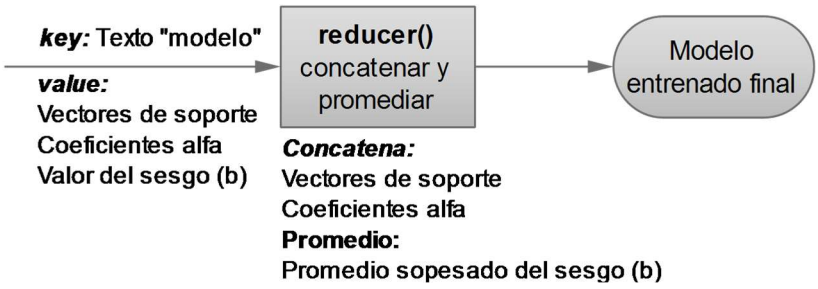


Figura 15. Tarea de reducción MapReduce

```
Proceso ReducerSVM
    totalSV <- 0
    intercepto <- 0
    Dimension SV[0]
    Dimension TA[0]
    //Se crea el objeto que tendrá el modelo final de entrenamiento
    modeloFinal <- new Modelo()
    cont<-0
    SubProceso reducer(Text key, Lista (value))
        //El "value" contiene el modelo de entrenamiento de la tarea de map
        leer value
        //Se concatenan todos los SV de cada modelo
        SV[cont]<-value.vectoresSoporte
        //Se concatenan todos los valores alpha de cada modelo
        TA[cont]<-value.terminosAlpha
        //se suman la cantidad de SV de cada modelo
        totalSV <- totalSV + value.Nro_SV
        //Se calcula el intercepto de cada modelo y se acumula
        intercepto <- intercepto + (value.intercepto * value.Nro_SV)

        cont <- cont + 1

    Fin SubProceso
    //Se divide el acumulado del intercepto por el total de SV para obtener
    //el promedio sopesado.
    intercepto <- intercepto/totalSV
    //Se almacena el totoal de SV en el modelo final
    modeloFinal <- SV[]
    //Se almacena el totoal de valores alpha en el modelo final
    modeloFinal <- TA[]
    //Se guarda el promedio del intercepto en el modelo final
    modeloFinal <- intercepto
    //Se guarda el modelo final
    Escribir (Text(modeloFinal))

FinProceso
```

Figura 16. Pseudocódigo. Tarea de reducción en MapReduce

A continuación, se explican las cuatro estrategias de entrenamiento propuestas en MapReduce.

3.2.1.1. Entrenamiento sin combinador (E1MR)

En esta estrategia las tareas de mapeo realizan el entrenamiento de los subconjuntos de datos (Figura 13) y entregan a la salida (Figura 17), el conjunto de vectores de soporte, el conjunto de coeficientes alfa y el valor de sesgo o intercepto *b* de cada uno de los submodelos entrenados. En la tarea de reducción

se concatenan los vectores de soporte y los coeficientes alfa y se calcula el valor del sesgo *b* con un promedio sopesado, obteniendo un modelo entrenado final.

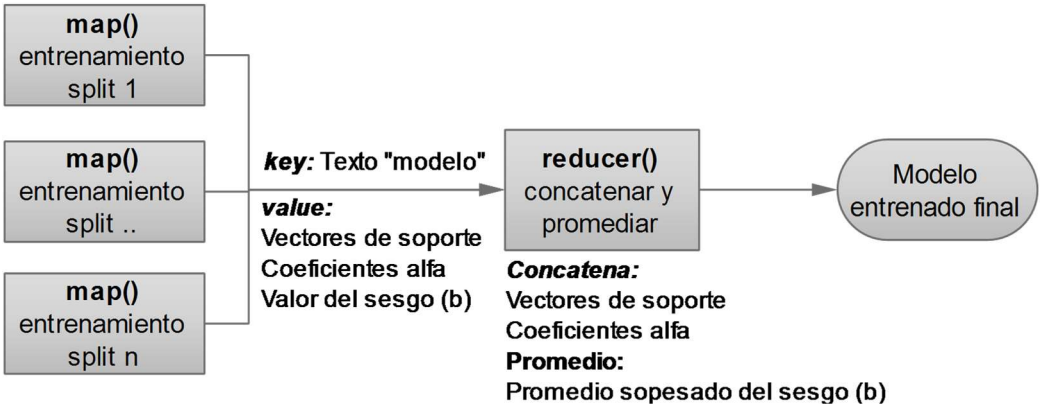


Figura 17. Estrategia uno MapReduce: entrenamiento sin combinador

3.2.1.2. Entrenamientos con combinador (E2MR)

En esta estrategia, como se puede ver en la Figura 18, las tareas de mapeo entregan al combinador un conjunto de vectores de soporte, un conjunto de coeficientes alfa y un valor de sesgo *b* correspondientes a cada uno de los submodelos. En el combinador se realiza una concatenación de los vectores de soporte y de los coeficientes alfa, y se calcula el valor de *b* con un promedio sopesado, obteniendo un modelo intermedio que es enviado a la tarea de reducción en la cual se unen estos modelos intermedios en un solo modelo final. El combinador se implementa con el fin de reducir la cantidad de información que se envía al *reducer*.

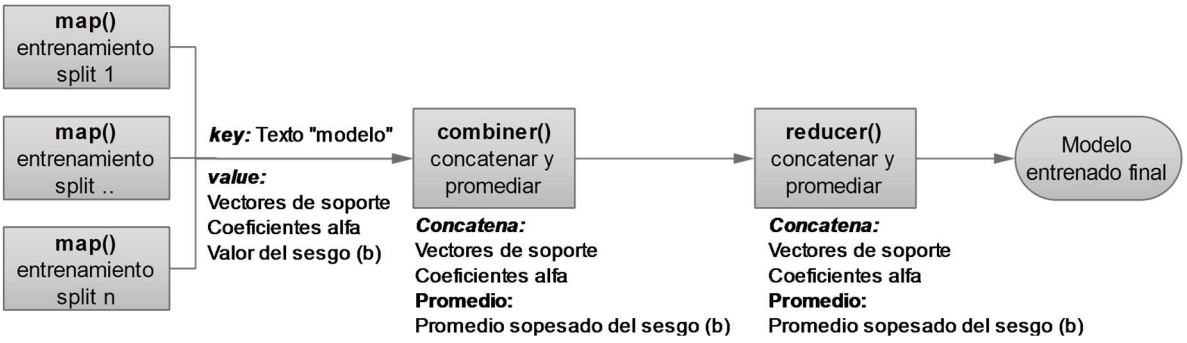


Figura 18. Estrategia dos MapReduce. Entrenamiento con combinador

3.2.1.3. Entrenamiento en el reducer (E3MR)

En esta estrategia, como se observa en la Figura 19, la tarea de mapeo entrega los datos de cada uno de los submodelos a la tarea de reducción, en donde se concatenan solamente los vectores de soporte y se realiza un nuevo entrenamiento para obtener un modelo final.

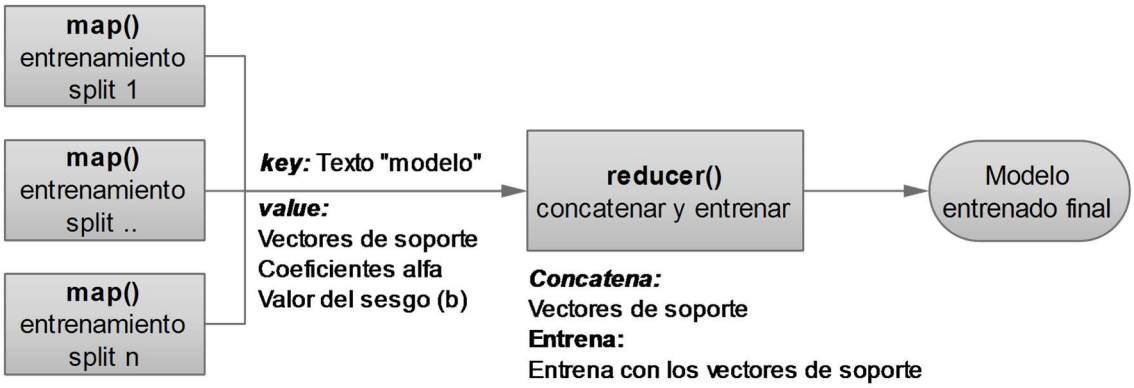


Figura 19. Estrategia tres MapReduce. Entrenamiento en el *reducer*

Al realizar pruebas con esta estrategia se generaban errores, que indicaban que la máquina virtual de Java (JVM)[14] se había quedado sin memoria debido a la cantidad de vectores de soporte que se requería almacenar en una estructura de

[14] La JVM es una máquina virtual que permite que una computadora ejecute programas Java - https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html

datos Java para efectuar el nuevo entrenamiento. Por esta razón, se decidió cambiar la estrategia adicionando una tarea de combinador (Figura 20) por cada tarea de mapeo, en la cual no sólo se concatenan los vectores de soporte, sino que adicionalmente se realizara un reentrenamiento en esta misma tarea. En la tarea de reducción se continúan concatenando los vectores de soporte y los coeficientes alfa y se efectúa el cálculo del promedio sopesado de *b*, construyendo el modelo de entrenamiento final.

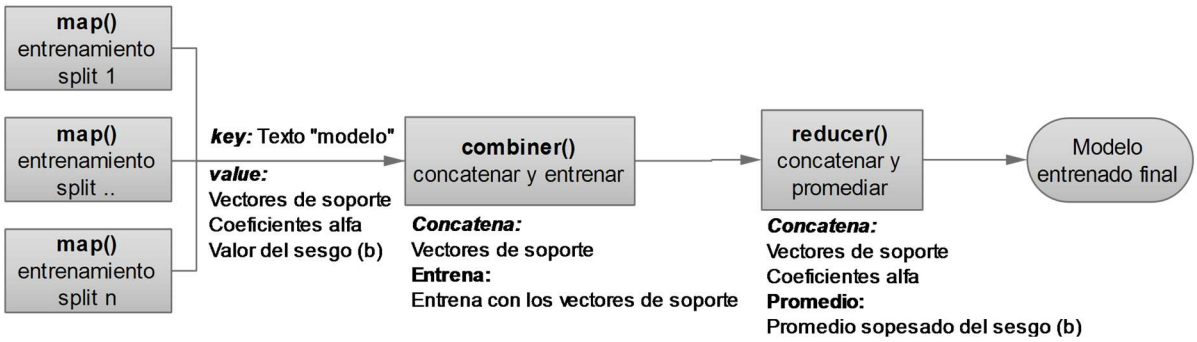


Figura 20. Estrategia tres MapReduce. Entrenamiento en el combinador

3.2.1.4. Entrenamiento iterativo (E4MRI)

En esta estrategia se implementaron dos trabajos de MapReducer tomando como base la estrategia E1MR (Figura 17). En el primer trabajo, la tarea de mapeo realiza el entrenamiento con cada *split* y entrega los datos de cada uno de los submodelos a la tarea de reducción, en donde solamente se concatenan los vectores de soporte y son entregados al segundo trabajo MapReduce. En el segundo trabajo, la tarea de mapeo hace un entrenamiento con los nuevos datos (vectores de soporte) y entrega nuevamente el resultado de los submodelos a la tarea de reducción, en la cual se concatenan los vectores de soporte y los coeficientes alfa. El valor de *b* se calcula con un promedio sopesado y se construye el nuevo modelo de entrenamiento.

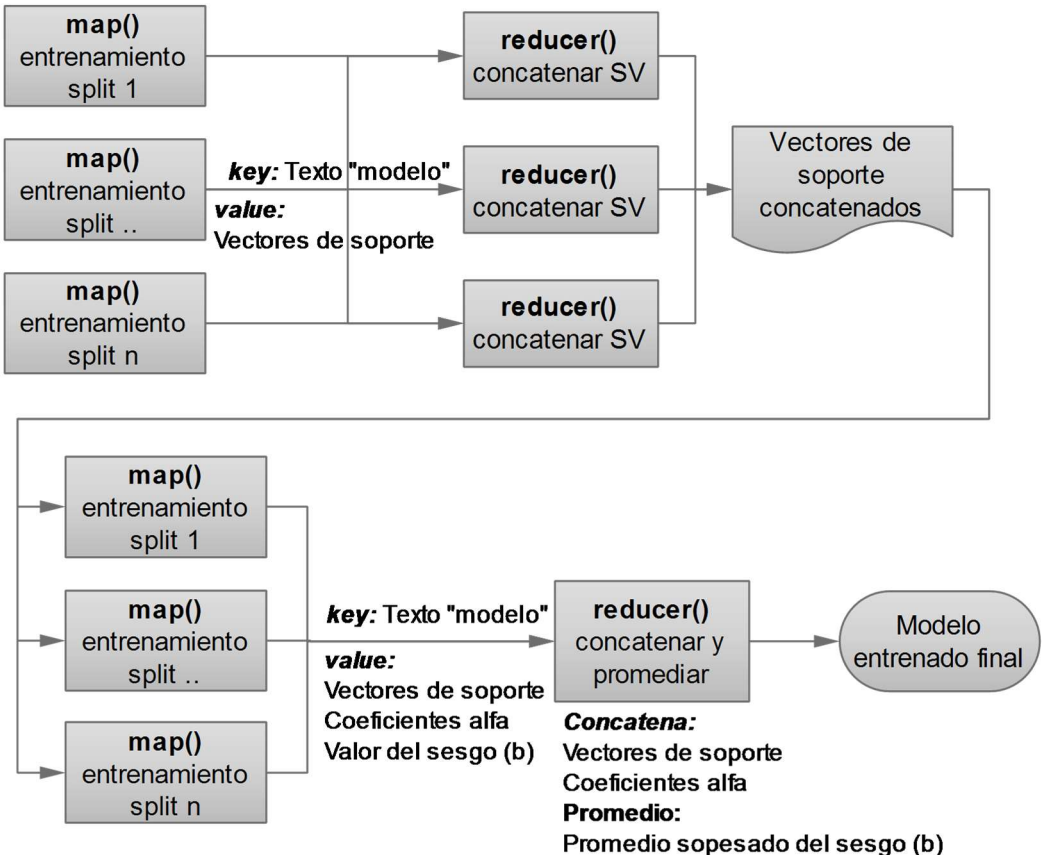


Figura 21. Estrategia cuatro MapReduce. Entrenamiento en dos fases

Para garantizar que las muestras quedarán balanceadas y asegurar que los vectores de soporte obtenidos en el primer trabajo de MapReduce estuvieran balanceados antes de entregarlos al segundo trabajo de MapReduce, se incluyó la etapa del preprocesamiento de los datos al inicio de cada trabajo (Figura 22). Esto permitió lanzar trabajos MapReduce de forma iterativa con dos condiciones de parada:

- Que el número de SV no disminuya en la iteración actual, es decir, que el entrenamiento nuevo arroje la misma cantidad de vectores de soporte que el proceso anterior.
- Alcanzar un número máximo de iteraciones; en este caso la quinta iteración.

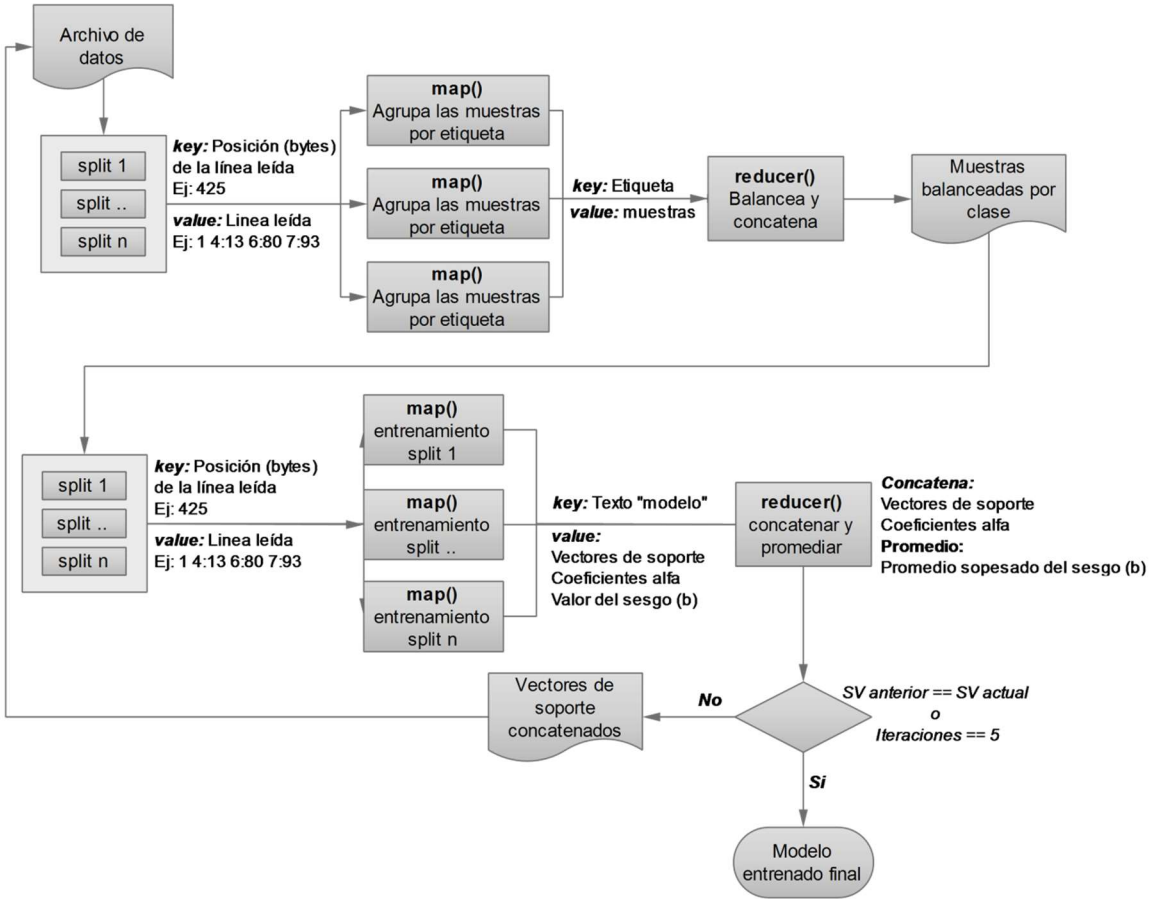


Figura 22. Estrategia cuatro MapReduce. Entrenamiento iterativo

Para la implementación en Spark se tomó como base la funcionalidad de la estrategia E4MRI, haciendo uso de los *RDD* y *dataframe*. La tarea de mapeo se implementó como una transformación, mientras que la tarea de reducción como una acción.

3.2.1.5. Entrenamiento iterativo en Spark (E5SPI)

Para evitar los problemas vistos en la estrategia E4MRI, también se adiciona una fase de preprocesamiento de los datos para distribuir las muestras de forma balanceada y evitar que se intente hacer un entrenamiento con muestras de una sola clase (Figura 23).

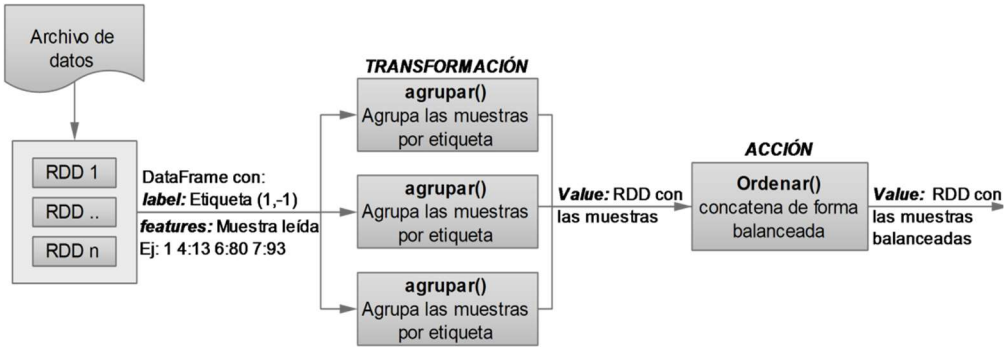


Figura 23. Preprocesamiento de datos en Spark

Inicialmente, el conjunto de datos es leído y cargado en un *dataframe* en formato <*key,value*>, en donde *key* representa la etiqueta de la muestra leída y *value* representa la muestra leída. Este *dataframe* es dividido en el número de subconjuntos que se desean entrenar. En cada tarea de mapeo se lee un subconjunto de datos y se crea un vector que contiene las etiquetas y una matriz que contiene los datos de las características de cada muestra; con el vector, la matriz y los parámetros del modelo SVM se realiza el entrenamiento utilizando la librería LibSVM. Se genera un submodelo entrenado con un conjunto de vectores de soporte, un conjunto de coeficientes alfa y un valor de *b*. Esta información es enviada a la tarea de reducción en formato <*key,value*> en un *dataframe*. En la tarea de reducción, que en *Spark* sería la acción, se realiza una concatenación de los vectores de soporte y los coeficientes *alfa* y se calcula el valor de *b* con un promedio sopesado, obteniendo un modelo entrenado final.

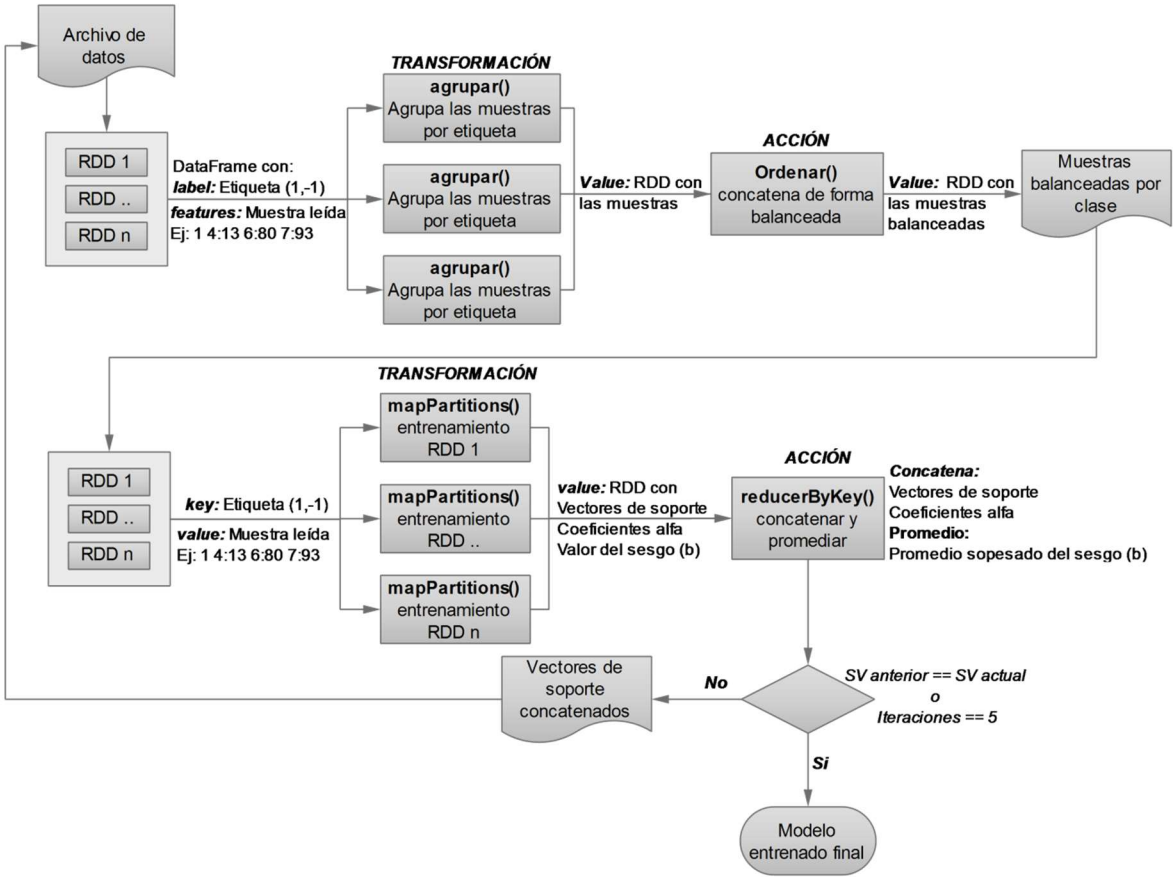


Figura 24. Estrategia en Spark. Entrenamiento iterativo

Luego del preprocesamiento de los datos, se hace el entrenamiento. En la tarea de reducción se hacen las mismas validaciones que en E4MRI para definir la parada del proceso de entrenamiento (Figura 24). Una vez finalizado el proceso de entrenamiento, se concatenan los vectores de soporte y los coeficientes alfa y se calcula el valor de *b* con un promedio sopesado, generando un modelo entrenado final. Si no se cumplen ninguna de las dos condiciones, se concatenan todos los vectores de soporte y se continúa el proceso de entrenamiento con los vectores de soporte de los nuevos modelos.

3.3. BASES DE DATOS UTILIZADAS

Para los experimentos se utilizaron las siguientes bases de datos:

Covertype [50] es un archivo biclase de 581.012 muestras de entrenamiento; cada muestra contiene 54 características continuas y un campo adicional con la etiqueta. El objetivo del problema de clasificación es determinar el tipo de cobertura forestal al que corresponde cada muestra. Son datos de áreas silvestres ubicadas en el Bosque Nacional del Colorado en Estados Unidos.

Bosson Higgs [52], es un archivo con 11 millones de muestras o instancias simuladas, llamados eventos, cada uno de los cuales contiene 28 características, una columna de ID y otra columna de etiquetas. El objetivo del problema de clasificación con este conjunto de datos consiste en poder identificar un suceso llamado “evento de señal”, dentro de un conjunto de eventos llamados “eventos de fondo”. Todas las variables contienen valores continuos, menos el Id que es un entero consecutivo y la etiqueta que toma dos valores cero (0) para los eventos de señal y uno para los eventos de fondo (1).

3.4. RECURSOS DE HARDWARE Y SOFTWARE

Las pruebas de la arquitectura diseñada se hicieron en un *cluster* de Hadoop conformado por un nodo principal llamado *NameNode* y cuatro nodos secundarios denominados *DataNode*. El nodo principal contó con 3 tres *cores* de 2.3 GHz, 23 GB de memoria RAM y 3.7 Teras en disco. Cada uno de los nodos secundarios tenían 2 cores de 2.3 GHz, 10 GB de memoria RAM y 3.8 Teras en disco (Figura 25).

Todos los nodos tenían instalado el sistema operativo Centos 6.5, Hadoop 2.6, Spark 1.6, integrados en la distribución de Cloudera versión 5.4 [53].

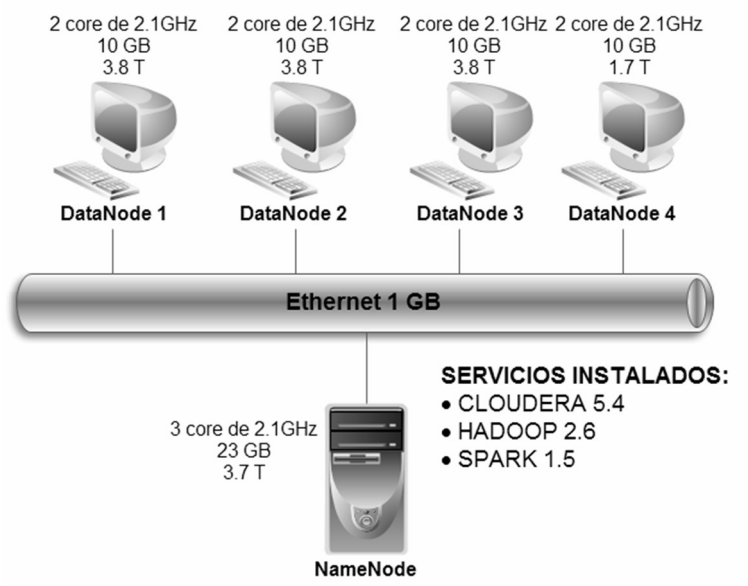


Figura 25. Recursos de *hardware* y *software*.

3.5. MODELO DE COSTOS

El objetivo del modelo de costos es proporcionar una herramienta para la gestión de los recursos de cómputo en la ejecución de trabajos MapReduce y Spark, con el fin de optimizar el rendimiento en entornos heterogéneos donde los recursos de *hardware* son limitados, en este caso específico, optimizar el tiempo de entrenamiento en la ejecución del algoritmo SVM en MapReduce y Spark, con los recursos descritos en la sección 3.4.

Con base en el modelo ARIA[15] propuesto por Verma [31] y el concepto de contenedores[16] manejado por el planificador de recursos de Hadoop llamado Yarn (*Yet Another Resource Negotiator*) y explicado en la sección del estado del arte, se creó un perfil de la aplicación. Se hicieron entrenamientos con la SVM en

[15] *Automated Resource Inference and Allocation*.
[16] Un contendor representa la unidad básica de asignación de recursos (memoria y CPU) para la ejecución de un proceso.

MapReduce y Spark con el mismo *hardware*, variando la cantidad de muestras de entrenamiento y el tamaño del *split*, con el propósito de obtener información relevante como tiempo de finalización de las tareas de mapeo Tm y de reducción Tr, tiempo total de entrenamiento Tt, tamaño del conjunto de datos de entrada cd, tamaño de los subconjuntos de datos $split$ y tiempo de procesamiento $ccpu$, entre otros, de los entornos de ejecución MapReduce y Spark. Con estos datos se calculó el promedio de ejecución de una tarea de mapeo y reducción, así como los tiempos máximos (límite superior) que tarda la ejecución de una aplicación, con k contenedores y *z* tareas de mapeo o reducción.

Para el análisis se debe tener en cuenta que el número de contenedores depende directamente de la capacidad de procesamiento que se tenga y de las configuraciones de memoria y cpu realizadas en el administrador de recursos YARN y el número de *mappers* está determinado por el tamaño del *split* y el tamaño del conjunto de datos. Es importante saber cuántos contenedores se usen ya que esto ayuda a estimar el tiempo de entrenamiento. Para estimar cuántos se deben usar se utiliza la fórmula de costos planteada en el trabajo.

Según el modelo ARIA y la ecuación (11), para encontrar el límite superior Tm_{up} de la fase de mapeo, se debe calcula el tiempo promedio tm_{avg} de finalización de las tareas de mapeo y el tiempo máximo tm_{max} de finalización de una tarea de mapeo que está determinado por el número total de tareas de mapeo (n_m), dividido por el número de contenedores asignados (k_c). En el ejemplo que expone Verma [31], estos tiempos representan predicciones optimistas y pesimistas del tiempo de finalización del trabajo de MapReduce y son calculados para aplicaciones que tienen un costo computacional bajo, como en el caso del contador de palabra [54]. En este caso, esta fase tiene un alto costo computacional debido a que en ella se realiza el entrenamiento con el modelo SVM; por consiguiente, el tiempo de total de ejecución dependerá del tiempo de

finalización del entrenamiento con un conjunto de datos determinado, un valor de *C* y una función kernel específica. Por ende, la suma de estos dos valores en la formula será reemplazada por una función $f(cpu, split)$, que representará el tiempo promedio de finalización de las tareas de mapeo en función del consumo de CPU (milisegundos) y el tamaño del *split* (MB). De acuerdo con esto, la fórmula para calcular el límite superior de la fase de mapeo sería:

$$Tm_{up} = \left(\frac{n_m}{k_c} - 1\right) . f_m(cpu, split) \qquad (15)$$

Para el cálculo de la función $f(cpu, split)$ se realiza un análisis inferencial con los datos de $Tm,\ Tr,\ Tt,\ cd,\ split$ *y* $ccpu$, para hallar la función que exprese el tiempo promedio tm_{avg} de ejecución de una tarea de mapeo en términos del consumo de CPU como variable dependiente y el *split* como variable independiente. Esta función es particular para cada caso de entrenamiento, por la particularidad existente en los datos de entrenamiento, el valor de *C* y los valores de los parámetros de la función kernel que se utilice, que en este caso sería el parámetro gamma (γ) de la función kernel RBF.

Como ya no se está sumando un tiempo máximo de finalización, el cual representaba una tarea de mapeo adicional a las establecidas en n_m, se elimina el menos uno de la primera parte de la ecuación $\frac{n_m}{k_c} - 1$.

$$Tm_{up} = \left(\frac{n_m}{k_c}\right) . f_m(cpu, split) \qquad (16)$$

El número de tareas de mapeo n_m estará determinado por la división entre el tamaño total del conjunto de datos y el tamaño del *split*:

$$n_m = \frac{cd}{split} \qquad (17)$$

Al reemplazar n_m en la ecuación $Tm_{up} = \left(\frac{n_m}{k_c}\right).f_m(cpu, split)$(16), el tiempo máximo de finalización de la fase de mapeo estará determinado por:

$$Tm_{up} = \left(\frac{cd}{k_c * split}\right).f(cpu, split) \quad (18)$$

En la implementación E4MRI y E5SPI se programó una sola tarea de reducción, y como se explicó en la sección 3.2, se creó un modelo final a partir de los submodelos entrenados en la tarea de mapeo. Por consiguiente, el tiempo de finalización de esta fase estuvo determinado por el número datos de entrada, que en este caso sería la sumatoria del número de vectores de soporte de cada submodelo entrenado. Según esto, el límite superior de la fase de reducción está dado por:

$$Tr = f_r(sv, cpu) \quad (19)$$

Al reemplazar 19 y 20 en 14, el tiempo total de entrenamiento queda expresado como:

$$T_t = \left(\frac{cd}{k_c * split}\right) f_m(cpu, split) + f_r(sv, cpu) \quad (20)$$

Como resultado se obtiene la ecuación $T_t = \left(\frac{cd}{k_c * split}\right) f_m(cpu, split) + f_r(sv, cpu)$ (20), que explica el tiempo total de entrenamiento en función del tamaño del conjunto de datos de entrenamiento, los contenedores disponibles, el consumo de CPU y el tamaño del *split*.

Para construir el perfil de la aplicación se tomaron observaciones para cada una de las variables $Tm, Tr, Tt, cd, split$ *y* $ccpu,$ de las pruebas realizadas con con la implementación de la estrategía E1MR y los conjuntos de datos del Bosson Higgs y la cobertura de bosques. Para ambos conjuntos de datos se usó la función del kernel RBF. Para el Bosson Higgs se usó *un valor C* de 10 y de *gamma* de 1. Estos valores fueron definidos después de un proceso de ajuste de parámetros del modelo (ver Figura 29. Comportamiento de la eficiencia, variando C y *gamma*.

Mejor valor C=10, gamma=1.), el cual consistió en varios entrenamientos variando el valor de *C* entre (0.1, 1 , 10) y el valor de *gamma* entre (0.001, 0.1, 1, 10 ,100), siendo estos valores (*C=10 y g=1*) los que presentaban mayor eficiencia. Para el conjunto de datos cobertura de bosques se usó un valor de 32 tanto para *C* como para *gamma*. Este valor se tomó como referencia del artículo de C. Liu et, al. [37] con el propósito de tener un punto de comparación. El tamaño del *split* tomó valores de 5 a 15 MB con incrementos de 1MB. El entrenamiento se hizo mediante la técnica de validación cruzada con 5 *folds*.

Para calcular la función del tiempo de entrenamiento promedio de la fase de mapeo $f_m(cpu, split)$ y de la fase de reducción, se tomaron los datos de cada uno de los entrenamientos realizados con los dos conjuntos de datos, como se aprecia en las Tabla 2 y Tabla 3.

Tabla 2. Datos del entrenamiento con el conjunto de datos del Bosson Higgs.

Split **(megabytes)**	**Tareas de mapeo**	**Contenedores utilizados**	**Tiempo de mapeo total (milisegundos)**	**Tiempo de CPU total (milisegundos)**	**Tiempo de reducción (milisegundos)**
15	5	4	4.478.907	15.418.480	502.403
14	6	4	4.185.656	15.072.080	443.571
13	6	4	3.457.805	12.795.780	411.545
12	6	6	2.202.193	11.087.670	395.907
11	7	4	3.124.980	9.626.290	394.790
10	8	4	2.766.357	9.203.920	403.140
9	8	2	4.510.195	8.515.330	301.885
8	9	2	4.503.304	7.530.070	318.886
7	11	2	3.196.890	5.177.300	331.654
6	12	2	2.671.012	4.938.920	345.535
5	15	6	849.169	4.955.140	245.230

Tabla 3. Datos de entrenamiento con el conjunto de datos cobertura de bosques

Split (megabytes)	Tareas de mapeo	Contenedores utilizados	Tiempo de mapeo total (milisegundos)	Tiempo de CPU total (milisegundos)	Tiempo de reducción (milisegundos)
12	27	8	5.578.532	39.999.560	983.362
11	29	8	4.938.358	35.694.060	862.161
10	32	8	4.164.268	31.032.620	698.549
9	36	18	2.059.329	25.182.940	561.810
8	40	23	1.127.246	20.849.610	172.441
7	46	23	882.043	16.529.480	208.658
6	54	23	808.674	14.360.400	242.399
5	64	20	505.559	11.745.560	118.321

Mediante un análisis de regresión simple se obtuvo un modelo x-cuadrado de la forma $y = a + b * x^2 + \varepsilon$ que describe la relación entre el consumo de CPU por tarea de mapeo y el tamaño del *Split*, es decir $f_m(cpu, split)$. Este modelo permitió encontrar la función de aproximación al tiempo promedio de finalización de las tareas de mapeo.

Para las pruebas realizadas con el conjunto de datos del Bosson de Higgs, se obtuvo la siguiente función:

$$y = -103603 + 13516.7 * split^2 \qquad (21)$$

Que corresponde a la Tabla 4 y a la Figura 26. Consumo de CPU por *split* con los datos del Bosson , con un R^2 de 98,1%:

Tabla 4. Consumo de CPU por tarea de mapeo y por tamaño del *split* del Bosson Higgs.

Split **(megabytes)**	**Consumo de CPU por tarea de mapeo (milisegundos)**
15	3.083.696
14	2.512.013
13	2.132.630
12	1.847.945
11	1.375.184
10	1.150.490
9	1.064.416
8	836.674
7	470.664
6	411.577
5	330.343

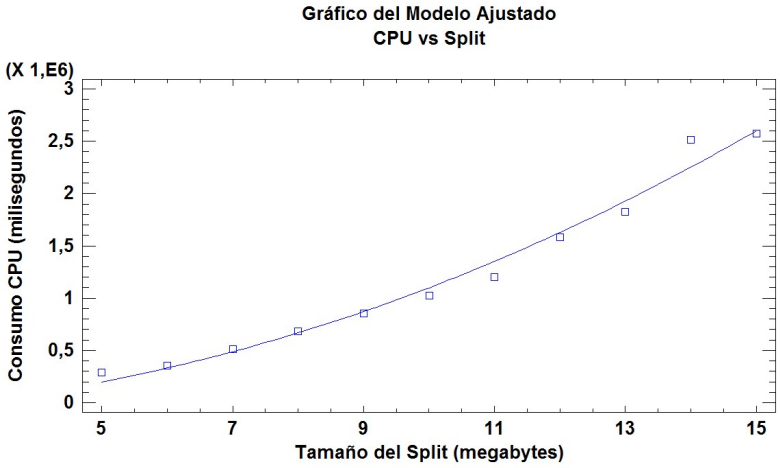


Figura 26. Consumo de CPU por *split* con los datos del Bosson Higgs.

La función de aproximación al tiempo promedio de finalización de las tareas de mapeo obtenida con los datos de las pruebas realizadas con el conjunto de datos de cobertura de bosques es:

$$y = -157513 + 11244{,}7 * split^2 \qquad (22)$$

Que corresponde a la Tabla 5 y a la Figura 27, con un R^2 de 99,3%.

Tabla 5. Consumo de CPU por tarea de mapeo y por tamaño del *split* en cobertura de bosques

Split **(megabytes)**	**Consumo de CPU por tarea de mapeo**
12	1.481.465
11	1.230.830
10	969.769
9	699.526
8	521.240
7	359.337
6	265.933
5	183.524

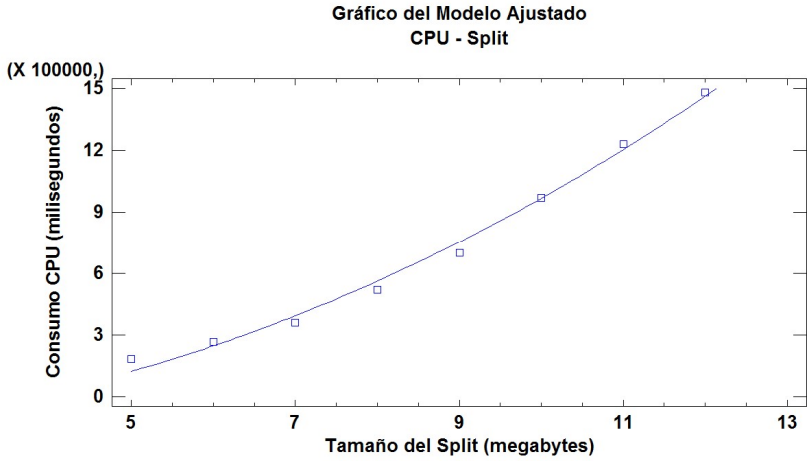


Figura 27. Consumo de CPU por tarea de mapeo y por tamaño del *split* con los datos de cobertura de bosques

El estadístico R^2 en ambos casos es superior al 98%, mostrando así que el modelo explica gran cantidad de la varianza total. Estos modelos permiten cuantificar la relación entre el consumo de CPU y el tamaño del *split*.

Los valores de *a y b* de las funciones 22 y 23 nos permiten evidenciar cómo el comportamiento en el tiempo de entrenamiento de la SVM depende del conjunto de datos y el parámetro del modelo utilizado como se muestra en las Figura 27 y Figura 28.

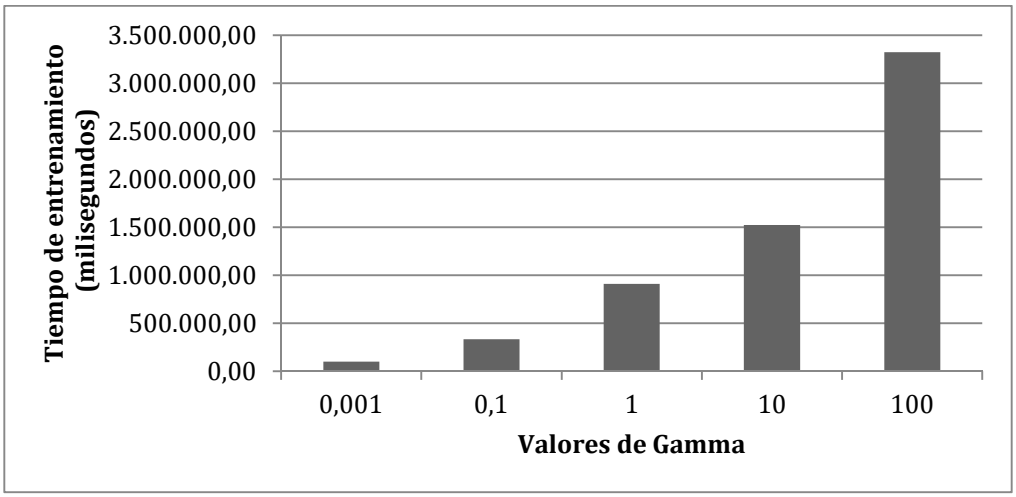


Figura 28. Consumo de CPU *vs*. parámetro del kernel RBF (*gamma*)

Conocidas las funciones 22 y y 23 es posible obtener una estimación del tiempo total de entrenamiento para un conjunto de datos determinado, con un tamaño de *split* fijo y un número de contenedores disponibles. A continuación, se reemplazan las funciones 22 y 23 en la ecuación $T_t = \left(\frac{cd}{k_c * split}\right) f_m(cpu, split) + f_r(sv, cpu)$ (20):

Para el conjunto de datos del Bosson de Higgs se tendría la siguiente fórmula:

$$T_t = \frac{cd}{(split * k| \quad |c)*(-103603+13516.7*split^2)+T_r} \qquad (23)$$

Entre tanto, para el conjunto de datos de cobertura de bosques se tendría la siguiente fórmula:

$$T_t = \frac{cd}{(split * k| \quad |c)*(-157513+1124\ ,7*spl\ \ ^2)+T_r} \qquad (24)$$

Para realizar una estimación del tiempo total de entrenamiento se tomaron los datos de entrenamiento del fold2 que no utilizaron para calcular la fórmula anterior,

para luego compararlos con el valor real, a fin de validar si el modelo tiene una buena estimación del consumo de CPU. Una vez hecha la estimación, se halló el error promedio de las observaciones, cuyo resultado es el siguiente.

Tabla 6. Estimación del tiempo de entrenamiento fold2 en el Bosson Higgs

Split (mb)	Número de mapeos	Tiempo real	Tiempo calculado	Porcentaje de error
15	5	4322018	4908885	9,6%
14	6	3715401	3818505	2,8%
13	6	3845779	3816259	0,8%
12	6	2287195	2149935	6,0%
11	7	3260867	3063835	6,0%
10	8	2659588	2808151	5,6%
9	8	4602788	4956249	7,7%
8	9	4200909	4188062	0,3%
7	11	3192633	3352292	5,0%
6	12	2903367	2680987	7,7%
5	15	845110	796669	5,7%

Como se observa en la Tabla 6 al aplicar la ecuación $T_t = \frac{cd}{(split * k\left| \right. \left| c\right)*(-103603+13516.7*split^2)+T_r}$ (23) para los datos del fold2 del Bosson Higgs, se obtienen errores promedio, y máximo y mínimo del 5,2%, 9,6% y 0,3%.

Tabla 7. Estimación del tiempo de entrenamiento fold2 en la cobertura de bosques

Split (mb)	Número de mapeos	Tiempo real	Tiempo calculado	Porcentaje de error
12	28	5330797	5444118,26	2,1%
11	30	4506019	4426182,53	1,8%
10	33	4149283	3845416,333	7,3%
9	37	1961601	2059606,86	5,0%
8	41	1201892	1141624,348	5,0%
7	47	905312	983710,6	8,7%
6	55	743949	781428,4609	5,0%
5	66	476565	452508,9565	5,0%

En la Tabla 7 se aprecia cómo al aplicar la ecuación $T_t = \frac{cd}{(split * k| \quad |c)*(-157513+11244{,}7*split^2)+T_r}$ (24) para los datos del fold2 de cobertura de bosques, se obtiene un error promedio del 5%, un error máximo del 8,7% y un error mínimo del 2,1%.

El error promedio puede variar debido a factores que quedaron por fuera de este estudio como el número de tareas fallidas o el tiempo de transferencia de información entre los nodos que puede variar según el tráfico que la ocupe.

Este análisis permite concluir que teniendo un perfil de un trabajo de entrenamiento de la SVM y aplicando la ecuación 21 es posible calcular un tiempo estimado de ejecución del modelo SVM relacionando el parámetro del modelo y la función kernel, que afectan directamente consumo de CPU y el tiempo de entrenamiento, con el conjunto de datos de entrada, el tamaño del *split* y los recursos disponibles (contenedores disponibles). Adicionalmente, con esta fórmula se puede conocer el tamaño máximo del *split* que se podría utilizar para entrenar un conjunto de datos *x*, en un tiempo determinado *t* y con un número de contenedores disponibles *z*.

A continuación, para ejemplificar este caso se usó el programa GAMS (*General Algebraic Modelling System*) para llevar la ecuación a un problema de optimización en el cual se desea conocer cuál es el máximo tamaño del *split* que se debe utilizar para entrenar un conjunto de datos de 305 MB en una hora, con ocho contenedores disponibles. Para ello, se tomó el conjunto de datos de cobertura de bosques, que pesa 305 megabytes, y se definió como tiempo máximo de entrenamiento una hora y ocho contenedores disponibles. La restricción del problema fue el tamaño del *split,* que debe tener un valor mínimo de 5MB y un valor máximo 20 MB, estos datos son tomados de las pruebas que se realizaron con la estrategia E1MR.

Se definió la función objetivo como:

$$Maxf(t, split) = \frac{305.4}{(split * 8)} * (-157513 + 11244{,}7 * split^2) + 561810$$

Sujeta a:

$split \geq 5$; (tamaño mínimo del *split* en megabytes)

$split \leq 20$; (tamaño máximo del *split* en megabytes)

$Tt \geq 1$; (tiempo mínimo de entrenamiento 1 milisegundo)

$Tt \leq 3600000$; (tiempo máximo de entrenamiento en milisegundos)

Estos valores fueron tomados de los entrenamientos realizados en las pruebas con el conjunto de datos de la cobertura de bosques.

El código en el programa GAMS fue el siguiente:

```
variables Tt;

positive variables
split;
split.L=5;

equations
objetivo, restri1, restri2, restri3, restri4;

objetivo.. Tt =E= ((305.4 / split) / 8) * ((not 157513) + (11244.7 * POWER(split,2))) + 561810;

restri1..  split=G=5;
restri2..  split=L=20;
restri4..  Tt =G= 1;
restri3..  Tt =L= 3600000;

model MINTIEMPO /all/;

solve MINTIEMPO using NLP maximizing Tt;
```

La solución que se obtiene luego de ejecutar el programa es un valor de 7,07 para la variable *split*:

```
                          LOWER      LEVEL      UPPER     MARGINAL

---- VAR Tt                -INF  3.6000E+6       +INF        .
---- VAR split              .        7.078       +INF        .
```

Lo anterior indica que para entrenar un conjunto de datos de 305 MB en una hora y con ocho contenedores disponibles se debe utilizar un tamaño de *split* de 7 MB. De esta forma se conoce el tamaño del *split* óptimo para entrenar la SVM en un tiempo requerido y con ciertos recursos disponibles. Al comparar este resultado con los datos obtenidos previamente en los entrenamientos realizados, entrenar ese tamaño de muestra con ocho contenedores disponibles, tomó un tiempo de 2.885.500,15 milisegundos (48 minutos), dato que permite concluir que al aplicar la fórmula de costos y el proceso de optimización se puede estimar el tiempo de entrenamiento con un margen de error de aproximadamente un 20%, este margen de error puede ser debido a que las funciones halladas para cada conjunto de datos no explican el 100% del comportamiento de los datos. Igualmente como se hizo este ejercicio para el conjunto de datos de cobertura de bosques, se puede realizar con la ecuación $T_t = \frac{cd}{(split * k \mid \quad \mid c)*(-103603+13516.7*split^2)+T_r}$ (23) para el conjunto de datos del *Bosson de Higgs*.

CAPÍTULO IV

RESULTADOS

Este apartado contiene los resultados de las pruebas realizadas durante la implementación de la arquitectura de la SVM en MapReduce y Spark y que sirvieron como insumo para la definición de la formula de costos propuesta en el capitulo 3. El proceso realizado fue el siguiente.

a) Se hicieron pruebas de funcionalidad, en las que se evaluó el desempeño de cada una de las estrategias implementadas, con el objetivo de seleccionar aquella que tuviera mejor desempeño respecto al tiempo de entrenamiento y utilizar dicha estrategia para las pruebas posteriores.
b) Luego, se evaluó el comportamiento de la memoria y la cpu de la estrategia seleccionada, variando el tamaño del *split*.
c) Después, se evalúa la escalabilidad de la estrategia seleccionada variando el tamaño de la muestra de entrenamiento.
d) Finalmente, se hicieron pruebas sobre la estrategia seleccionada que permitieron evaluar la precisión en función del tamaño del Split, el numero de muestras de entrenamiento y el número de iteraciones del entrenamiento.

Estas pruebas nos permitieron identificar los factores a tener en cuenta en la definición de la formula de costos, como el número de tareas de mapeo, el número de contenedores disponibles y el tamaño del *split*. Igualmente, nos permitieron crear un perfil del entrenamiento de la SVM en MapReduce y Spark.

En el final del capítulo se presentan los resultados de la evaluación y se comparan respecto a la implementación de otros autores.

4.1. PRUEBA DE FUNCIONALIDAD

Esta prueba se llevó a cabo con el propósito de probar y validar que las estrategias propuestas cumplieran con las funcionalidades diseñadas y evaluar cuál de ellas presentaba mejor desempeño respecto al tiempo de entrenamiento.

Las pruebas se hicieron con un archivo que contenía datos del *Bosson de Higgs* y que está disponible en *UCI Machine Learning Repository* [44]. De este archivo se tomó un subconjunto de aproximadamente 250.000 muestras con 28 características cada una. De este subconjunto se tomó el 80%, aproximadamente 200.000 muestras, para entrenar y el 20% restante, 50.000 para validación. La función kernel utilizada para realizar el entrenamiento de la SVM fue la función RBF.

Para la selección del valor de *gamma*, el parámetro de penalidad *C* y el tamaño del *split*; se efectuó un proceso de ajuste de parámetros del modelo, el cual consistió en realizar varios entrenamientos con cada estrategia variando el valor de *C* entre (0.1, 1, 10), el valor de *gamma* entre (0.001, 0.1, 1, 10, 100) y el tamaño del *split* entre 5 y 13 MB, aumentando de 1 en 1. Los valores de *C*=10, *gamma*=1 y tamaño del *split* en 13 MB (corresponde aproximadamente a 38 mil muestras) fueron los que presentaron mayor precisión. En la Figura 29. Comportamiento de la eficiencia, variando C y *gamma*. podemos ver el comportamiento de la eficiencia, variando *C* y *gamma* de la estrategia E1MR, donde la mayor eficiencia se alcanza con valores de *C* de 1 y de gamma de 10. Cabe anotar que cuando se tomaba un valor de tamaño del *split* mayor a 13MB, algunas de las estrategias propuestas generaron un error de *time out*, debido a que el tiempo que toma entrenar esa cantidad de muestras es superior al tiempo

de respuesta del nodo esclavo al nodo maestro[17]; en ese caso el nodo maestro lo toma como una falla de comunicación con el nodo y cancela la tarea.

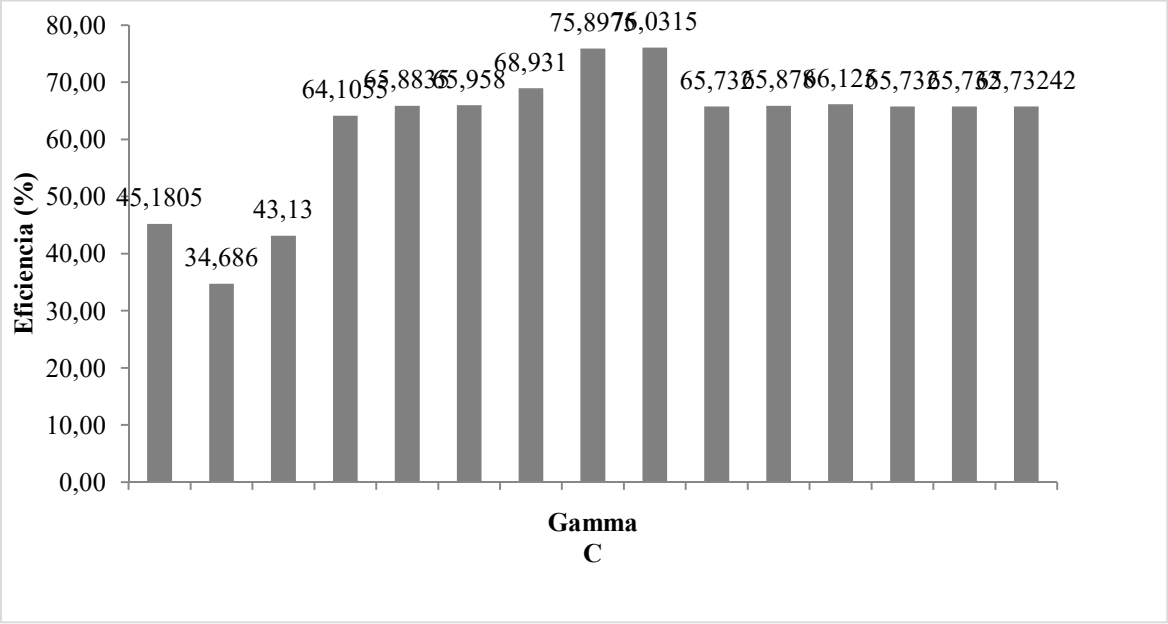


Figura 29. Comportamiento de la eficiencia, variando C y *gamma*. Mejor valor C=10, gamma=1.

Para esta prueba el número de nodos se deja fijo en 4 y el número de procesadores por nodos se deja fijo en 2 y la memoria se deja fija en 20GB por nodo.

Una vez establecidos los valores de *C* en 1, gamma en 10 y el tamaño del *split* en 13MB, procedemos a realizar el entrenamiento con cada estrategia y a tomar los tiempos de entrenamiento de cada una de ellas. Adicionalmente, en esta prueba se tomó el número de vectores de soporte como validador de la cantidad de vectores de soporte que quedaban después del entrenamiento con cada estrategia.

[17] Este tiempo de respuesta es configurado en la variable de entorno de Hadoop mapreduce.task.timeout y tiene un valor por defecto de 600.000 ms.

Estrategia uno (E1MR)*:* entrenamiento sin combinador. Terminó el entrenamiento en 15 minutos y 48 segundos y el modelo seleccionó 119.517 vectores de soporte.

Tabla 8. Prueba de funcionalidad de la estrategia uno.

Split (MB)	*# Mapeos*	*# Reductores*	*# Vectores de soporte*	*Tiempo de entrenamiento*
13 MB	6	1	119.517	15m:48s

Estrategia dos (E2MR)*:* entrenamiento con combinador; terminó el entrenamiento en 16 minutos y 48 segundos. Se seleccionaron 119.517 vectores de soporte (Tabla 9).

Tabla 9. Prueba de funcionalidad de la estrategia dos.

Split (MB)	*# Mapeos*	*# Reductores*	*# Vectores de soporte*	*Tiempo de entrenamiento*
13 MB	6	1	119.517	16m:48s

Estrategia tres (E3MR): entrenamiento en el combinador; terminó el entrenamiento en 16 minutos y 11 segundos. Se seleccionaron 119.517 vectores de soporte (Tabla 10).

Tabla 10. Prueba de funcionalidad de la estrategia tres.

Split (MB)	*# Mapeos*	*# Reductores*	*# Vectores de soporte*	*Tiempo de entrenamiento*
13 MB	6	1	119.517	16m:11s

Estrategia cuatro (E4MRI): Entrenamiento iterativo, se crearon inicialmente seis tareas de mapeo, en la segunda iteración bajó a cuatro tareas, en la tercera iteración el número de tareas descendió a tres, en la cuarta y quinta iteración se redujo a dos, y en todas las iteraciones se hizo una sola reducción. El número de vectores de soporte fue disminuyendo en cada iteración, lo mismo que el tiempo

de entrenamiento; en este caso el algoritmo se detuvo en la iteración número cinco que fue el valor máximo de iteraciones que se configuró. El entrenamiento terminó en 1 hora, 10 minutos y 35 segundos; se seleccionaron al final 72.748 vectores de soporte (Tabla 11).

Tabla 11. Prueba de funcionalidad de la estrategia cuatro.

# Jobs	*Split (MB)*	*# Mapeos*	*# Reductores*	*# Vectores de soporte*	*Tiempo de entrenamiento*
1-Preprocesamiento	13 MB	6	1	119.517	44s
1-Entrenamiento	13 MB	6	1		15m:57s
2-Preprocesamiento	13 MB	4	1	82.124	34s
2-Entrenamiento	13 MB	4	1		14m:42s
3-Preprocesamiento	13 MB	3	1	75.655	34s
3-Entrenamiento	13 MB	3	1		21m:20s
4-Preprocesamiento	13 MB	2	1	73.764	26s
4-Entrenamiento	13 MB	2	1		9m:30s
5-Preprocesamiento	13 MB	2	1	72.748	26s
5-Entrenamiento	13 MB	2	1		9m:06s

Una vez finalizadas las pruebas con las cuatro estrategias de MapReduce se puede observar que todas tuvieron el mismo comportamiento en el mismo número de vectores de soporte (119.517). Con la estrategia E1MR se obtuvo el mejor tiempo de entrenamiento que fue de 15 minutos y 48 segundos y el mayor tiempo se obtuvo con la estrategia E2MR, que fue de 16 minutos 48 segundos. La diferencia de nueve segundos entre E1MR y E4MRI obedeció a que E4MRI cuenta con una fase adicional que es la del preprocesamiento.

Adicionalmente como se observa en la Tabla 12, la cual contiene los tiempos totales de entrenamiento de cada estrategia MapReduce con los cinco *fold*, podemos observar que las estrategias en las que no se implemento la fase del *combiner* (E1MR y E4MRI), tuvieron los mejores tiempos de entrenamiento, con lo

cual podemos concluir que, para este caso en particular, la fase de *combiner* no ayuda a reducir el tiempo de entrenamiento.

Tabla 12. Resumen de tiempos por estrategia.

Estrategias	Total tiempo de entrenamiento
E1MR	1:21:21
E2MR	1:24:00
E3MR	1:22:57
E4MRI	1:22:13

Estrategia E5SPI: con esta estrategia diseñada en Spark, el entrenamiento se hizo con una sola iteración, el cual terminó en 11 minutos 47 segundos, incluyendo el tiempo de preprocesamiento, este tiempo es menor que el obtenido con la estrategia E1MR (Figura 30). Esto permite comprobar cómo Spark ayuda a reducir el tiempo de entrenamiento ya que evita la escritura en disco durante su ejecución, procesando todo en memoria, mientras que MapReduce tiene que leer y escribir en disco en cada tarea de mapeo y reducción.

Tabla 13. Entrenamiento estrategia en *Spark*.

Numero de muestras	*Split*	Tiempo de entrenamiento (hh:mm:ss)
250.000	**13 Mb**	**00:11:47**

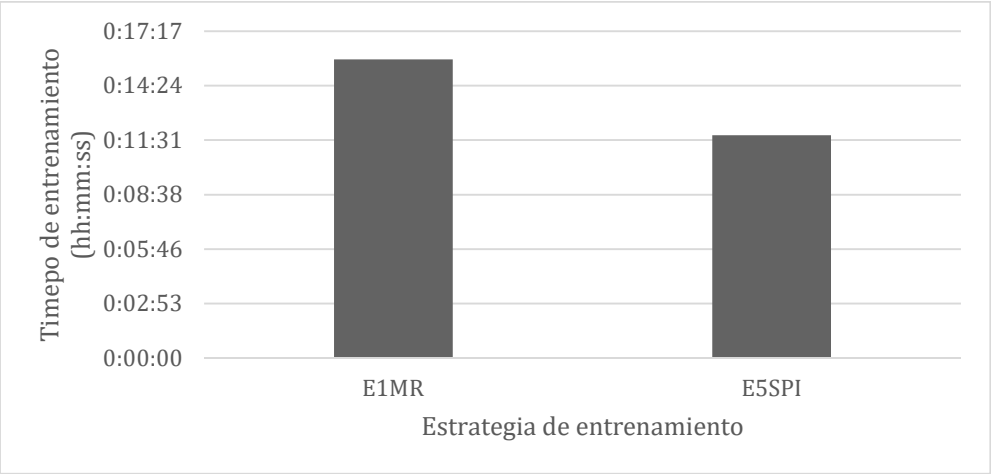


Figura 30. Tiempo de entrenamiento estrategia *MapReduce vs. Spark*.

De esta prueba de funcionalidad podemos concluir que la estrategia con menor tiempo de entrenamiento fue la E5SPI con una sola iteración, seguida de la estrategia en MapReduce E1MR. Esto nos muestra como el marco de trabajo de spark ayuda a reducir el tiempo de entrenamiento. Es preciso tener en cuenta que el tamaño máximo del *split*, puede estar limitado por el tiempo de respuesta entre el nodo esclavo y el nodo maestro, el cual igualmente afecta el tiempo total de entrenamiento, ya que, al incrementar este tiempo de respuesta, se estaría elevando igualmente el tiempo de entrenamiento.

Para las siguientes pruebas se utiliza la estrategia E1MR y la estrategia E5SPI con una sola iteración.

4.2. CONSUMO DE MEMORIA Y CPU VARIANDO EL TAMAÑO DEL SPLIT

El objetivo de esta prueba es evaluar como afecta el tamaño del *split* el comportamiento de las estrategias seleccionadas respecto al consumo de memoria y *cpu*. Para lo cual se utilizaron los mismos valores de *gamma* y C de la prueba de funcionalidad. El conjunto de datos fue de 200.000 muestras y el tamaño del *split* inicio en 5MB y finalizó en 15MB, con un incremento de 1MB.

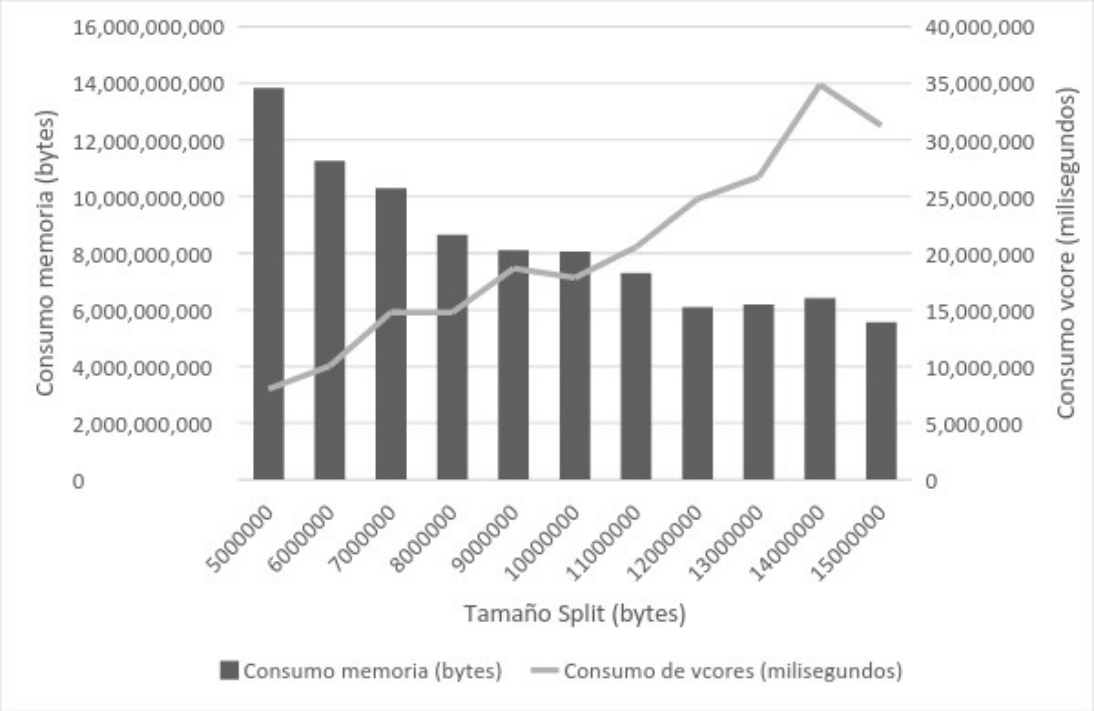


Figura 31. Consumo de memoria, consumo de vcore *vs*. tamaño del *split*

Como podemos observar en la (Figura 31) si se aumenta el tamaño del *split*, el consumo de memoria disminuye debido a que se generan menos tareas de mapeo, las cuales podrán ser atendidas simultáneamente por el número de nodos disponibles, pero al aumentar el tamaño del *split* se tendrán más muestras para entrenar y por consiguiente, el tiempo de entrenamiento será mayor.

4.3. ESCALABILIDAD DE LAS ESTRATEGIAS SELECCIONADAS

El objetivo de esta prueba fue evaluar la escalabilidad de las estrategias de entrenamiento en Spark y MapReduce*,* aumenta el tamaño de las muestras y analizar como afecta esto el comportamiento de la memoria y la CPU. Para lo cual se utilizó el archivo de *Bosson de Higgs* y se dividió en cuatro grupos: 250.000, 500.000, 1 millón y 1 millón y medio de muestras. Adicionalmente, se utilizó una

función kernel RBF con un *gamma* de 1 y el parámetro C del modelo se dejó en 10; de igual forma, el tamaño del *split* se dejó fijo en 13Mb. El valor del parámetro *gamma* del kernel RBF se dejó fijo para no tener dos factores de cambio dentro del análisis, ya que este parámetro impacta directamente el tiempo de entrenamiento de la SVM.

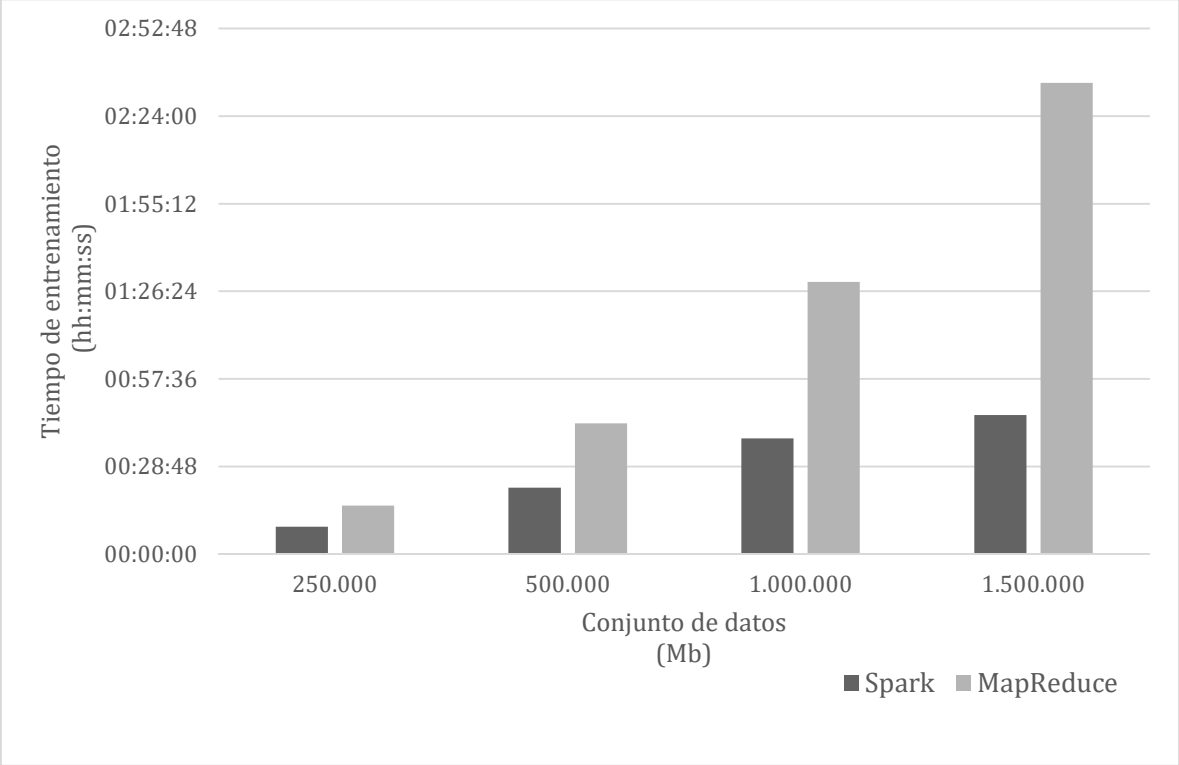


Figura 32. Tiempo de entrenamiento en Spark *vs*. tiempo de entrenamiento en E1MR MapReducer.

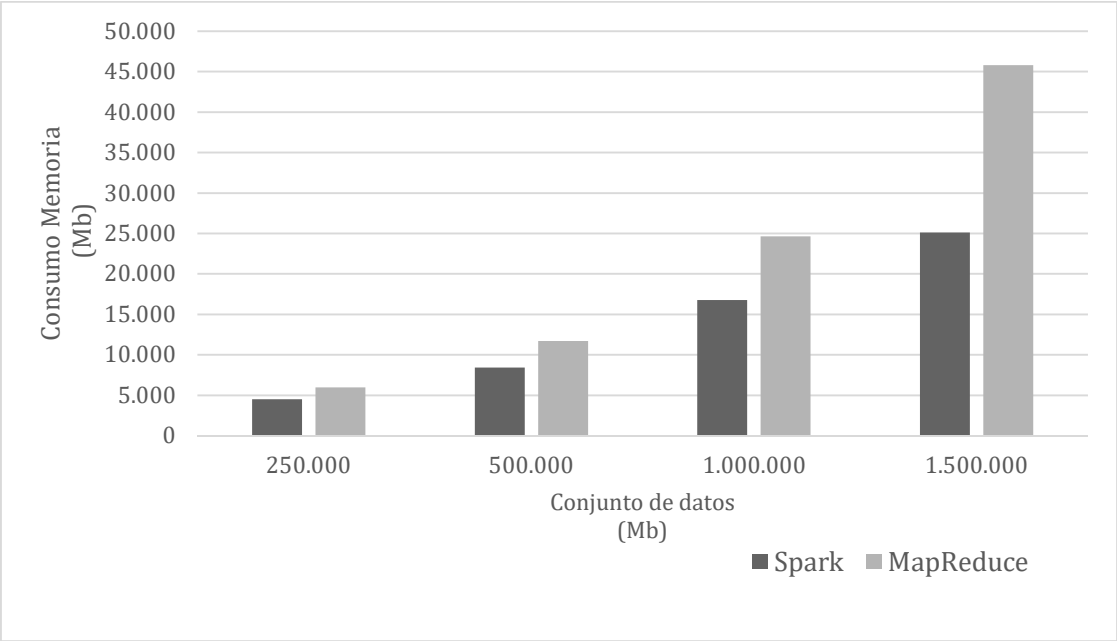


Figura 33. Consumo Memoria variando el conjunto de entrenamiento

Una vez realizadas la pruebas con E1MR y E5SPI, como vemos en la Figura 32, a medida que se aumenta el número de muestras, el tiempo de entrenamiento en Spark incrementa de forma lineal, mientras que en MapReduce lo hace de forma exponencial, y ocurre lo mismo con el consumo de memoria (Figura 33). Uno de los factores para que se presente este comportamiento en MapReduce es que al aumentar el conjunto de entrenamiento y dejar fijo el tamaño del *split*, las tareas de mapeo aumentan, y ya que el número de nodos es fijo, estos no pueden ejecutar todas las tareas de mapeo de forma simultánea, fenómeno que hace que se empiecen a encolar las tareas restantes y se ejecuten en la medida que van liberando recursos. Esto conlleva a un mayor incremento del tiempo de entrenamiento y en el consumo de memoria.

Para controlar este incremento en el tiempo de entrenamiento se debe tener en cuenta que la cantidad de tareas de mapeo no sea mayor al número de contenedores que pueden ser creados en el *cluster*. Si el número de tareas de

mapeo es mayor al número de contenedores y se desea mantener el tiempo constante se deberán incrementar el número de nodos o los recursos de memoria y la CPU de los nodos existentes. El consumo de memoria muestra un comportamiento normal ya que al aumentar el numero de muestras y dejar el numero de nodos fijos, se requiere de mayor capacidad en memoria para almacenar las muestras de entrenamiento.

4.4. PRUEBAS DE PRECISIÓN

Esta prueba tiene como objetivo evaluar el comportamiento de la precisión de las estrategias seleccionadas en función del tamaño del *split*, el número de vectores de soporte y el número de iteraciones.

Para esta prueba se partió de los valores de C y *gamma* obtenidos en las pruebas de funcionalidad sección 4.1 y se utiliza la estrategia E4MR en lugar de E1MR ya que esta realiza el entrenamiento de forma iterativa.

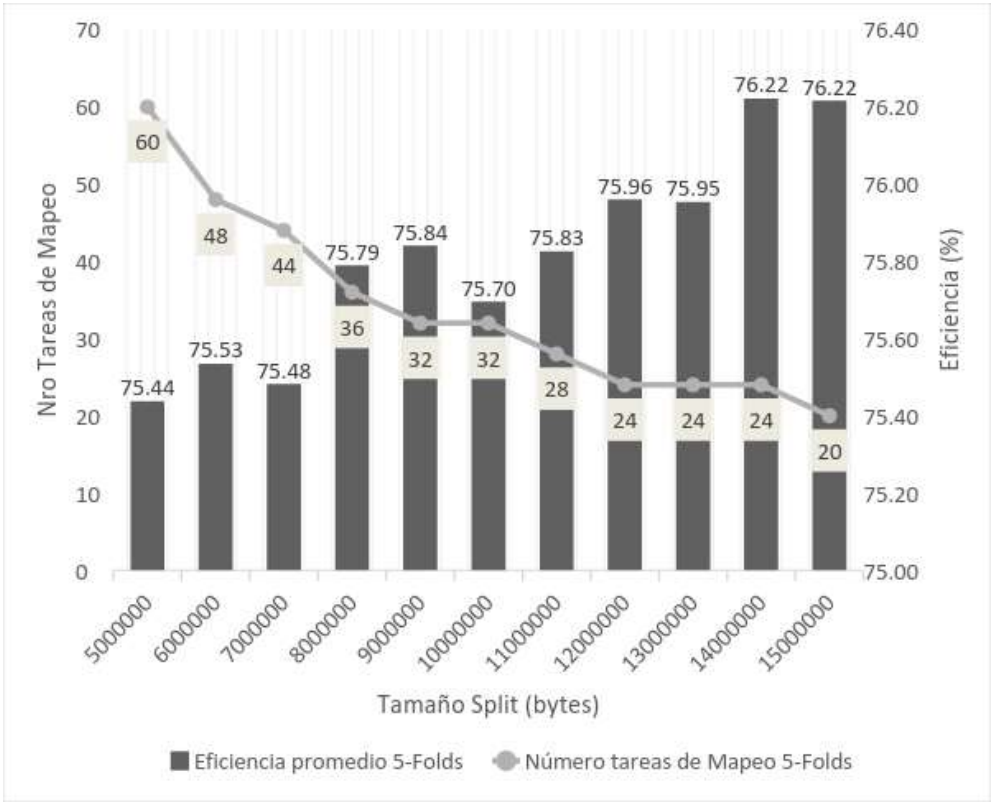


Figura 34. Eficiencia; número de tareas de mapeo *vs*. tamaño del *split*

Como vemos en la (Figura 34), al dejar el tamaño de la muestra de entrenamiento fijo en 500.000 muestras y aumentar el tamaño del *split*, la eficiencia incrementa en la medida que el *split* va haciéndose más grande. Con un *split* de 5MB la eficiencia fue de 75,4% mientras que con un *split* de 15MB, fue de 76,2%.

En la figura 36 se evidencia que en cada iteración el número de tareas de mapeo y el número de vectores de soporte va disminuyendo, pero la eficiencia va teniendo pequeñas variaciones de aumento y decremento. Con esta estrategia se puede obtener una eficiencia mayor o igual en cada iteración y con menos vectores de soporte al final Figura 36 y Figura 37). Se utilizó un tamaño de muestra de 1 millón y 1 millón y medio respectivamente; se dejó un *split* fijo de 10 MB y no se colocó límite en el número de iteraciones. Al final del entrenamiento en la iteración 25 y

26 respectivamente, la eficiencia tuvo una variación de aproximadamente 3 puntos por debajo o por encima de la eficiencia promedio.

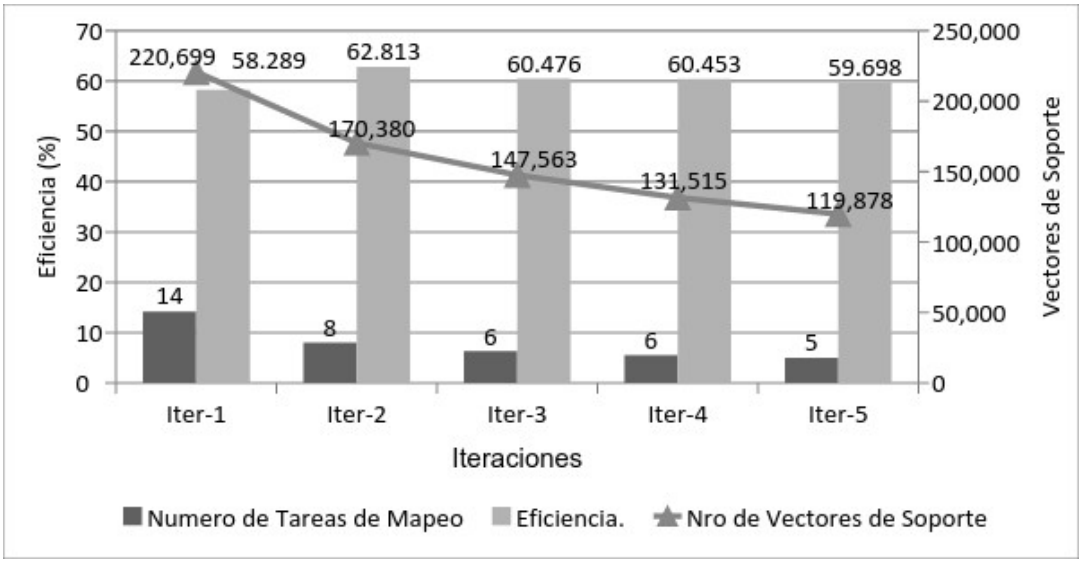


Figura 35. Número de tareas de mapeo, eficiencia, número de vectores de soporte *vs.* número de iteraciones

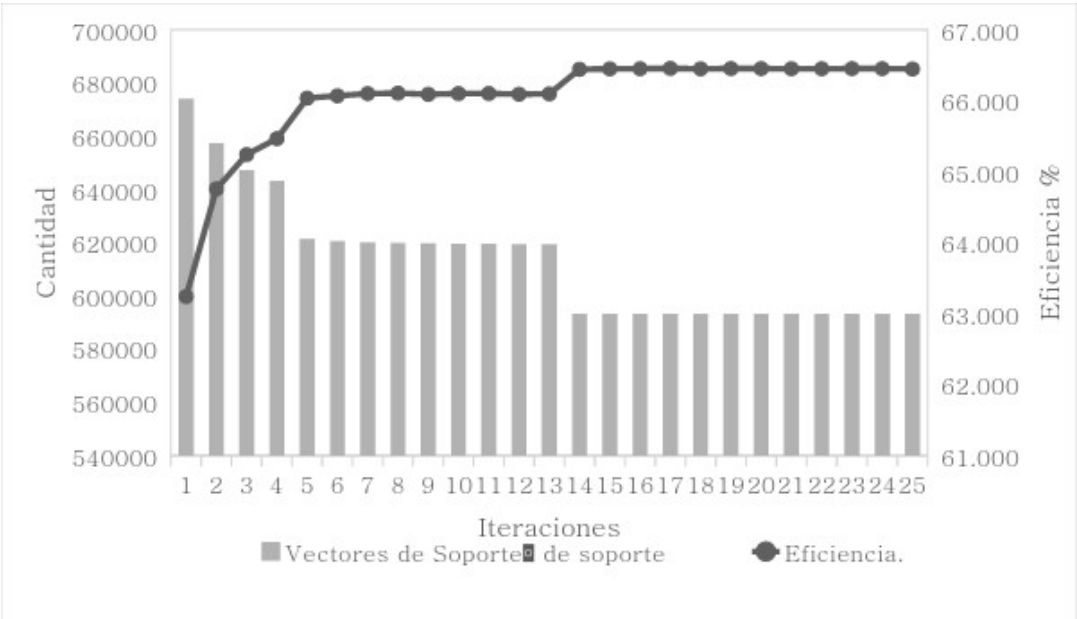


Figura 36. Vectores de soporte y eficiencia *vs*. número de iteraciones con 1 millón de muestras.

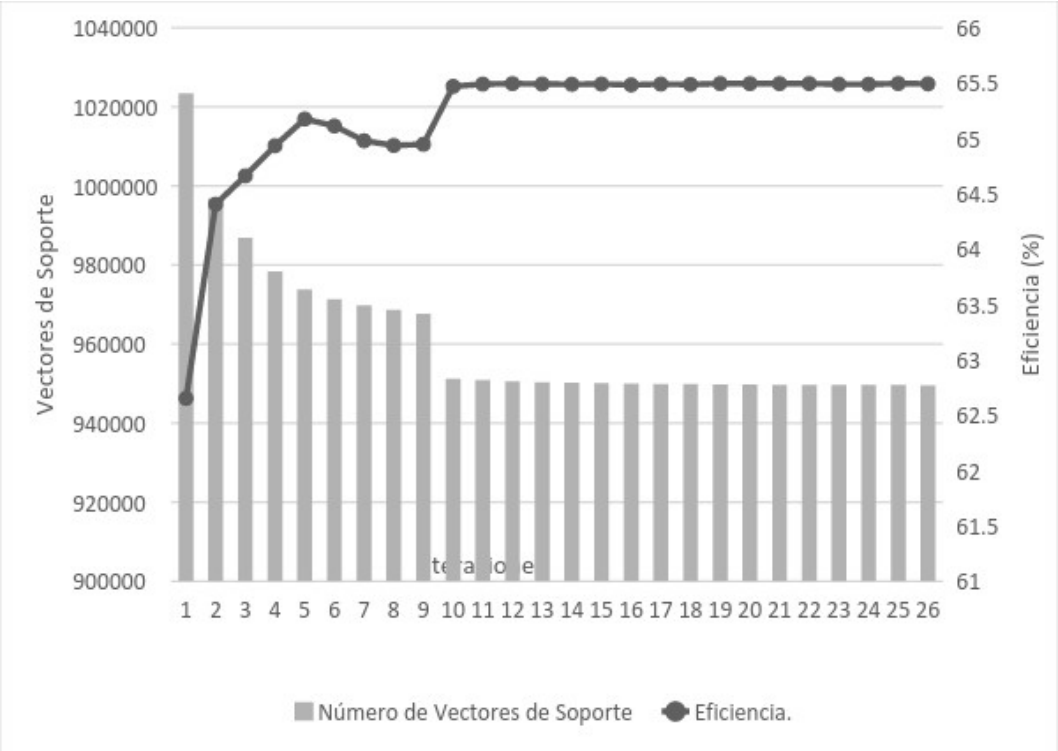


Figura 37. Vectores de soporte y eficiencia *vs.* número de iteraciones con 1 millón y medio de muestras.

4.5. COMPARACIÓN CON OTRAS SOLUCIONES PROPUESTAS

Para evaluar las estrategias E4MR y E5SPI propuestas, se tomó como referencia el artículo [49], en el que se hicieron pruebas de eficiencia y tiempo de entrenamiento con los algoritmos MBSSVM, LibSVM, CascadeSVM y MSMSVM, utilizando la base de datos de cobertura de bosques *covtype-binary*[18], así como la función kernel RBF. Los valores de *C* y *gamma* fueron de 32 cada uno.

[18] Ampliar información en https://archive.ics.uci.edu/ml/datasets/covertype

Reportaron los siguientes datos:

Tabla 14. Datos reportados por [49].

Algoritmo	Tiempo(s)	Precisión
MBSSVM	40	0,6913
CascadeSVM	1998	0,8923
LibSVM	83631	0,9615
MSM-SVM	33	0,9604

Estos datos se compararon con los obtenidos con la estrategia E4MRI y E5SPI (

Tabla 15), haciendo una sola iteración.

Tabla 15. Eficiencia y tiempo de entrenamiento con E4MRI y S5SPI.

Algoritmo	Tiempo(s)	Precisión
Estrategia E4MRI	42	0,96204
Estrategia E5SPI	35	0,96204

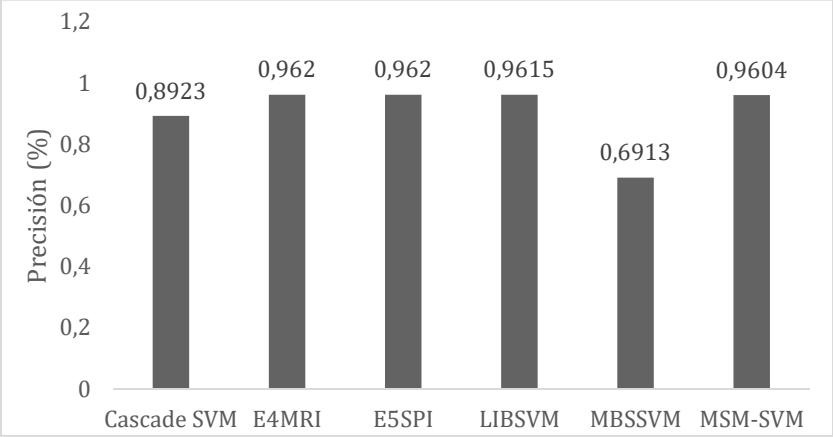


Figura 38. Comparativo de la precisión con otros algoritmos.

Con ambas estrategias se logró una eficiencia del 96,2%, 0,2 mayor que la reportada en [49] (Figura 38). El tiempo de entrenamiento con E4MRI fue de 42 segundos, en tanto que con E5SPI fue de 35 segundos, tiempos mayores a los reportados con la estrategia MSM-SVM.

CONCLUSIONES

Este estudio abordó uno de los problemas más importantes a resolver dentro del aprendizaje automático, como es la clasificación. Para abordar este problema empleamos la máquina de vectores de soporte, por ser uno de los métodos mas ampliamente utilizados en el aprendizaje automático, debido a su alta capacidad de generalización, al hacer predicciones correctas sobre nuevas muestras en diferentes aplicaciones, como lo muestra [1] y [2], entre otros. El entrenamiento de la máquina de vectores de soporte implica un alto costo computacional que llega hacer del orden cuadrático respecto al número de muestras de entrenamiento. La utilización de nuevos *framework* para el procesamiento de datos en paralelo como MapReduce y Spark han permitido aplicar este modelo de clasificación en la analítica para grandes volúmenes de datos, demostrando mejoras significativas en el tiempo de entrenamiento; sin embargo, existen factores en el empleo de estos *framework* que impactan el desempeño de los algoritmos.

La mayoría de las soluciones propuestas en los últimos años (2010-2019) sobre la implementación de la máquina de vectores de soporte estándar para clasificación biclase bajo estos *framework*, proponen dividir el conjunto de datos en pequeños subconjuntos, realizar el entrenamiento de la máquina de vectores de soporte en las tareas de mapeo, generando un modelo por cada subconjunto de datos y en la tarea de reducción obtener el modelo final. En todos los trabajos se observan mejoras significativas en el tiempo de entrenamiento, sin que esto afecte sustancialmente el porcentaje de precisión en la clasificación. Igualmente se pudo observar que en ningún caso se aborda una estrategia para definir el tamaño óptimo del subconjunto de datos, ni se establecen parámetros que permitan relacionar el *hardware* disponible, la cantidad de información a procesar y el ajuste del parámetro del modelo.

Por tal motivo y teniendo en cuenta que, aspectos como la cantidad de datos de entrada, escritos y movidos entre las tareas de mapeo y reducción, y el consumo de procesamiento y de memoria, afectan de manera significativa el tiempo de ejecución de los algoritmos que se implementan en MapReduce y Spark, se diseñó una arquitectura para la implementación en estos *framework* de un sistema de clasificación biclase basado en máquinas de vectores de soporte, y se evaluó el desempeño de ésta en el procesamiento con grandes volúmenes de datos.

Para el diseño de la arquitectura, se analizaron varias estrategias de paralelización del modelo de maquinas de vectores de soporte en MapReduce y Spark, de las cuales, las estrategias de MapReduce en las que no se implementó el combiner (E1MR y E4MRI), obtuvieron un mejor tiempo de entrenamiento como se observa en la Tabla 12. Lo que nos permite concluir, qué para este caso particular, el uso de esta fase incrementa el tiempo de entrenamiento ya que se realiza un proceso adicional de escritura en disco antes de enviar los datos al reducer. La estrategia implementada en spark (E5SPI), obtiene un mejor tiempo de entrenamiento respecto a E1MR, evidenciando cómo el uso de Spark ayuda a reducir aun más el tiempo de entrenamiento, ya que evita la escritura en disco durante el entrenamiento. La eficiencia lograda con las estrategias E1MR y E5SPI en los diferentes conjuntos de entrenamiento, muestra que la implementación en MapReduce y Spark permiten mejorar el desempeño del modelo sin afectar drásticamente la precisión.

A partir del análisis de estas estrategias, se comprobó que la implementación en paralelo de la maquina de vectores de soporte en MapReduce y Spark trabaja de manera eficiente en grandes conjuntos de datos, en comparación con la maquina de vectores de soporte secuencial y que, con el entrenamiento iterativo se puede reducir el número de vectores de soporte, obteniendo una eficiencia igual o mayor.

Adicionalmente que el uso de estos *framework* permite diseñar soluciones escalables como se evidencia en la Figura 32 donde se aumentó el tamaño de la muestra de entrenamiento de 250 mil a 1 millón 500 mil muestras, sin que se presentaran fallas en el entrenamiento. Así mismo, se observó que al aumentar el tamaño del *split* se obtiene una mayor eficiencia del modelo, debido a que cada subconjunto de datos tiene más muestras para entrenar; pero este tamaño está condicionado por la memoria disponible en el nodo responsable de ejecutar las tareas de entrenamiento. Por ello para optimizar el tiempo de entrenamiento lo ideal es calcular la cantidad de subgrupos y que estos sean inferiores o equivalente al número de nodos disponibles.

Por consiguiente, para definir el valor óptimo del split, se definió un modelo de costos específico para este tipo de trabajos que tienen un alto costo computacional, tomando como base el modelo de rendimiento analítico para trabajos en MapReduce llamado ARIA. Por medio del modelo de costos se logró establecer el tamaño máximo del split que se podría utilizar para entrenar un conjunto de datos, en un tiempo determinado y con un número de contenedores disponibles. Aplicando el modelo costos con los datos recopilados del entrenamiento con la estrategia E1MR, se logró estimar el tamaño óptimo del split para entrenar la maquina de vectores de soporte con un conjunto de datos de 305 MB, en una hora y con ocho contenedores disponibles, con un margen de error en el tiempo de entrenamiento de aproximado un 20%, con relación al tiempo obtenido en las pruebas realizadas con el mismo tamaño de split (7MB) y el mismo conjunto de datos. De esta forma podemos estimar tiempos de entrenamiento de la maquina de vectores de soporte dependiendo de los recursos disponibles y el conjunto de datos. Este modelo puede ser reproducible para cualquier conjunto de datos y cualquier implementación de la maquina de vectores de soporte en MapReducer y Spark.

Los experimentos realizados en este estudio y el uso del modelo de costos nos permitió comprobar que no solo es necesario conocer el funcionamiento de los diferentes *frameworks* de programación para el análisis de grandes volúmenes de datos como MapReducer y Spark y aprender a programar en ellos, sino también identificar qué aspectos de los nuevos *frameworks* impactan el desempeño de las aplicaciones, como en este caso, que el número de tareas de mapeo y el tamaño del *split* afectan el tiempo de entrenamiento y la eficiencia del clasificador. Otro aspecto importante a destacar en el uso de estos *frameworks* es la topología y la velocidad de la red del *cluster*. En este trabajo para todas las pruebas se usó una topología en árbol con una velocidad de 1GB, al igual que en otros trabajos [55], [56], usados como puntos de referencia para la presente tesis.

RECOMENDACIONES

En futuros estudios se aconseja integrar las estrategias con un algoritmo para la selección de instancias, que permita eliminar instancias repetidas, erróneas o poco útiles, creando un conjunto de datos más pequeño con el objetivo de hacer más eficiente el procesamiento de datos con las maquinas de vectores de soporte, conservando o mejorando el porcentaje de predicción.

Así mismo, se debería estudiar la inclusión de una condición adicional de parada del entrenamiento iterativo que permita finalizar el entrenamiento encontrando un equilibrio entre el menor número de vectores de soporte y la mayor eficiencia.

Finalmente, se alienta a que, tanto en la Universidad de Antioquia como en la ciudad, se continúe trabajando con estos *framework* de analítica de datos de tecnología de punta en el ámbito internacional, las cuales permitirán posicionarnos más ampliamente en este campo.

REFERENCIAS

[1] X. Shen, H. Wu, and Q. Zhu, “Training Support Vector Machine through redundant data reduction,” in *Proceedings of the 4th International Conference on Internet Multimedia Computing and Service*, 2012, pp. 25–28.

[2] I. Steinwart, D. Hush, and C. Scovel, “Training SVMs without offset,” *J. Mach. Learn. Res.*, vol. 12, pp. 141–202, 2011.

[3] “Google reports strong profit, says it’s ‘rethinking everything’ around machine learning | ITworld,” *IT WORLD*, 2015. [Online]. Available: http://www.itworld.com/article/2996619/google-reports-strong-profit-says-its-rethinking-everything-around-machine-learning.html. [Accessed: 01-Feb-2016].

[4] L. Bottou and L. Chin-Jen, “Support Vector Machine Solvers,” p. 27, 1995.

[5] V. Vapnik and C. Cortes, “Support-Vector Networks,” vol. 297, pp. 273–297, 1995.

[6] S. Nowozin and S. J. Wright, *Optimization for Machine Learning*. 2012.

[7] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik, “Parallel Support Vector Machines : The Cascade SVM,” *Adv. Neural Inf. Process. Syst.*, pp. 521–528, 2005.

[8] J. Martínez-Trinidad *et al.*, *Support Vector Machines for Pattern Classification*, vol. 6256. 2010.

[9] J. C. Platt, “Sequential Minimal Optimization : A Fast Algorithm for Training Support Vector Machines,” pp. 1–21, 1998.

[10] S. . Keerthi, S. k. Shevade, C. Bhattacharyya, and K. R. K. Murthy, “Improvements to platt’s SMO algorithm for svm classifier design,” *Neural Comput.*, vol. 13, pp. 637–649, 2001.

[11] N. K. Alham, M. Li, S. Hammoud, Y. Liu, and M. Ponraj, “A Distributed SVM for Image Annotation,” in *2010 7th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2010*, 2010, no. Fskd, pp. 2983–

2987.

[12] H. Zhao and F. Magoulès, “Parallel Support Vector Machines on multi-core and multiprocessor systems,” in *Artificial Intelligence and Applications / 718: Modelling, Identification, and Control*, 2011, no. Aia, p. 7.

[13] N. K. Alham, M. Li, Y. Liu, and S. Hammoud, “A MapReduce-based distributed SVM algorithm for automatic image annotation,” *Comput. Math. with Appl.*, vol. 62, no. 7, pp. 2801–2811, Oct. 2011.

[14] G. Caruana, M. Li, and M. Qi, “A MapReduce based parallel SVM for large scale spam filtering,” *2011 Eighth Int. Conf. Fuzzy Syst. Knowl. Discov.*, pp. 2659–2662, Jul. 2011.

[15] Z. Sun and G. Fox, “Study on Parallel SVM Based on MapReduce.” p. 7, 2012.

[16] J. Zhao, Z. Liang, and Y. Yang, “Parallelized Incremental Support Vector Machines Based on MapReduce and Bagging Technique,” in *Proceedings of 2012 IEEE International Conference on Information Science and Technology, ICIST 2012*, 2012, no. 2, pp. 297–301.

[17] A. Kumar and R. P. G, “Verification and validation of MapReduce program model for parallel K-Means algorithm on Hadoop Cluster,” *Intenational J. Comput. Appl.*, vol. 72, no. 8, pp. 48–55, 2013.

[18] N. K. Alham, M. Li, Y. Liu, and M. Qi, “A MapReduce-based distributed SVM ensemble for scalable image classification and annotation,” *Comput. Math. with Appl.*, vol. 66, no. 10, pp. 1920–1934, Dec. 2013.

[19] F. O. Catak and M. E. Balaban, “CloudSVM: Training an SVM classifier in cloud computing systems,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, vol. 7719 LNCS, pp. 57–68.

[20] A. Antoniades and C. C. Took, “A Google approach for computational intelligence in big data,” in *Proceedings of the International Joint Conference on Neural Networks*, 2014, pp. 1050–1054.

[21] Z. You, J. Yu, L. Zhu, S. Li, and Z. Wen, “A MapReduce based parallel SVM for large-scale predicting protein – protein interactions,” *Neurocomputing*, vol. 145, pp. 37–43, 2014.

[22] A. Priyadarshini, “A Map Reduce based Support Vector Machine for Big Data Classification,” *Int. J. Database Theory Appl.*, vol. 8, no. 5, pp. 77–98, 2015.

[23] D. Le Quoc, V. D’Alessandro, B. Park, L. Romano, and C. Fetzer, “Scalable Network Traffic Classification Using Distributed Support Vector Machines,” in *IEEE 8th International Conference on Cloud Computing*, 2015, pp. 1008–1012.

[24] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, “Estimating the progress of MapReduce pipelines,” *ICDE ’10 26th IEEE Int. Conf. Data Eng.*, pp. 681–684, 2010.

[25] K. Morton, M. Balazinska, and D. Grossman, “ParaTimer: A Progress Indicator for MapReduce DAGs.,” *SIGMOD ’10 ACM Int. Conf. Manag. Data*, pp. 507–518, 2010.

[26] H. Herodotou *et al.*, “Star sh: A Self-tuning System for Big Data Analytics,” *CIDR*, pp. 261–272, 2011.

[27] Z. Zhang, L. Cherkasova, and B. T. Loo, “Parameterizable benchmarking framework for designing a MapReduce performance model,” *Concurr. Comput. Pract. Exp. 26, 12*, pp. 2005–20026, 2014.

[28] F. Tian and K. Chen, “Towards optimal resource provisioning for running MapReduce programs in public clouds,” *CLOUD ’11 IEEE Int. Conf. Cloud Comput.*, pp. 155–162, 2011.

[29] K. Chen, J. Powers, S. Guo, and F. Tian, “CRESP: Towards Optimal Resource Provisioning for MapReduce Computing in Public Clouds.,” *IEEE Trans. Parallel Distrib. Syst. 25, 6*, pp. 1403–1412, 2014.

[30] A. Verma, L. Cherkasova, and R. H. Campbell, “Resource provisioning framework for MapReduce jobs with performance goals,” *Middlew. ’11 12th ACM/IFIP/USENIX Int. Middlew. Conf. (Lisbon, Port. 2011),* pp. 165–186,

2011.

[31] A. Verma, L. Cherkasova, and R. H. Campbell, “ARIA: Automatic Resource Inference and Allocation for MapReduce Environments,” *ICAC ’11 8th ACM Int. Conf. Auton. Comput. (Karlsruhe, Ger. 2011).*, 2011.

[32] P. Lama and X. Zhou, “AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud,” in *Proceedings of the 9th international conference on Autonomic computing - ICAC ’12*, 2012, p. 63.

[33] E. Y. Chang, K. Zhu, H. Wang, and H. Bai, “PSVM: Parallelizing support vector machines on distributed computers,” *Adv. Neural Inf. Process. Syst.*, vol. 20, no. 2, pp. 1–8, 2007.

[34] C.-C. Chang and C.-J. Lin, “LIBSVM: a library for Support Vector Machines,” *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1--27:27, May 2011.

[35] K. P. Murphy, “Machine Learning,” MIT Press, Ed. 2012, p. 26.

[36] J. Arias, “Modelamiento y Simulacion de Sistemas, Algunos conceptos de modelos de sistemas.” 2014.

[37] C. Liu, B. Wu, Y. Yang, and Z. Guo, “Multiple submodels parallel support vector machine on spark,” in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 945–950.

[38] A. M. Andrew, “AN INTRODUCTION TO SUPPORT VECTOR MACHINES AND OTHER KERNEL-BASED LEARNING METHODS by Nello Christianini and John Shawe-Taylor, Cambridge University Press, Cambridge, 2000, xiii+189 pp., ISBN 0-521-78019-5 (Hbk, £27.50). -,” *Robotica*, vol. 18, no. 6, pp. 687–689, 2000.

[39] C. M. Bishop, *Pattern Recognition and Machine Learning*. 2006.

[40] The Apache Software Foundation, “Welcome to Apache™ Hadoop®!,” 2014. [Online]. Available: http://hadoop.apache.org/.

[41] T. White, *Hadoop: The definitive guide*, vol. 54. 2012.

[42] W. Sandy Ryza, U. Laserson, S. Owen, and J. Wills, *Advanced Analytics*

with Spark Advanced Analytics with Spark Spark PATTERNS FOR LEARNING FROM DATA AT SCALE Advanced Analytics with Spark. .

[43] Jeff Schlimmer, “UCI Machine Learning Repository: Mushroom Data Set,” 1987. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Mushroom. [Accessed: 09-Feb-2017].

[44] F. Cai and V. Cherkassky, “Generalized SMO algorithm for SVM-based multitask learning.,” *IEEE Trans. neural networks Learn. Syst.*, vol. 23, no. 6, pp. 997–1003, Jun. 2012.

[45] H. González-Vélez and M. Kontagora, “Performance evaluation of MapReduce using full virtualisation on a departmental cloud,” *Int. J. Appl. Math. Comput. Sci.*, vol. 21, no. 2, pp. 275–284, 2011.

[46] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, “A simulation approach to evaluating design decisions in MapReduce setups,” *Proc. - IEEE Comput. Soc. Annu. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst. MASCOTS*, pp. 5–15, 2009.

[47] A. Rajaraman and J. D. Ullman, “Mining of Massive Datasets,” *Lect. Notes Stanford CS345A Web Min.*, vol. 67, p. 328, 2011.

[48] R. L. Graham, “Bounds for Certain Multiprocessing Anomalies,” *Bell Syst. Tech. J. 45, 9*, pp. 1563–1581, 1966.

[49] F. Ö. Çatak and M. E. Balaban, “A MapReduce based distributed SVM algorithm for binary classification.” Cornell University Library y Data Science Association, Turquía, pp. 1–19, 2013.

[50] J. A. Blackard and University Colorado State, “Covertype Data Set,” *UCI Machine Learning Repository*, 1998. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Covertype. [Accessed: 05-Feb-2016].

[51] T. White, *Hadoop The Definitive Guide*, vol. 4th Editio. 2008.

[52] P. Baldi, P. Sadowski, and D. Whiteson, “Searching for Exotic Particles in High-energy Physics with Deep Learning,” *Nature Communications 5*, 2014. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/HIGGS. [Accessed:

06-Feb-2016].

[53] Cloudera, “Cloudera.” [Online]. Available: http://www.cloudera.com/. [Accessed: 06-Feb-2016].

[54] F. P. Apache, “Apache Hadoop.” [Online]. Available: https://wiki.apache.org/hadoop/PoweredBy.

[55] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, “A simulation approach to evaluating design decisions in MapReduce setups,” *MASCOTS ’09 IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, pp. 1–11, 2009.

[56] Z. Zhang, L. Cherkasova, and B. T. Loo, “Performance Modeling of MapReduce Jobs in Heterogeneous Cloud Environments,” *CLOUD ’13 6th IEEE Int. Conf. Cloud Comput.*, pp. 839–846, 2013.

[57] A. J. Izenman, *Modern Multivariate Statistical Techniques*. 2001.

[58] Y. Sánchez, “Clustering y Grid Computing,” pp. 1–16, 2007.

ANEXOS

Anexo 1.

Teorema KKT (Karush–Kuhn–Tucker): Da las condiciones necesarias para obtener una solución óptima para un problema de optimización general. Una condición necesaria y suficiente para que un punto w sea óptimo es la existencia de α , β tal que:

$$\frac{\partial L(w\ ,\alpha\ ,\beta\ )}{\partial w} = 0$$
$$\frac{\partial L(w\ ,\alpha\ ,\beta\ )}{\partial \beta} = 0$$
$$\alpha_i g_i(w\ ) = 0, i = 1, \dots, k$$
$$g_i(w\ ) \leq 0, i = 1, \dots, k$$
$$\alpha_i \geq 0, i = 1, \dots, k$$

La ecuación $\alpha_i\ g_i(w\ ) = 0, i = 1, \dots, k$ es conocida como la condición KKT [57].

Anexo 2.

Análisis documental del estado del arte

En primer lugar, fue necesario detectar las palabras clave o descriptores que permitían la recuperación de publicaciones relacionadas con el tema de estudio, MapReduce y Support Vector Machine. Ello se logró por medio de una revisión rápida de Google Académico, en donde se analizaron específicamente las palabras clave. De esta manera se concluyó que los términos más implementados en la literatura científica son: MapReduce, Hadoop, Spark, máquinas de vectores de soporte, Máquinas de vectores de apoyo, Support Vector Machine y SVM.

Con los términos identificados, se establecieron las siguientes frases compuestas en español e inglés:

- MapReduce basado en máquinas de vectores de soporte

- Hadoop basado en máquinas de vectores de soporte
- Spark basado en máquinas de vectores de soporte
- MapReduce basado en Máquinas de vectores de apoyo
- Support Vector Machine based on MapReduce
- Support Vector Machine based on Hadoop
- Support Vector Machine based on Spark
- SVM in Hadoop
- SVM in MapReduce
- SVM in Spark

Con base en esta información se establecieron las siguientes ecuaciones de búsqueda:

(“Support Vector Machine” OR SVM) AND Hadoop)

(“Support Vector Machine” OR SVM) AND MapReduce)

(“Support Vector Machine” OR SVM) AND Spark)

(“Máquinas de vectores de soporte” OR “Máquinas de vectores de apoyo” OR SVM) AND Hadoop)

(“Máquinas de vectores de soporte” OR “Máquinas de vectores de apoyo” OR SVM) AND MapReduce)

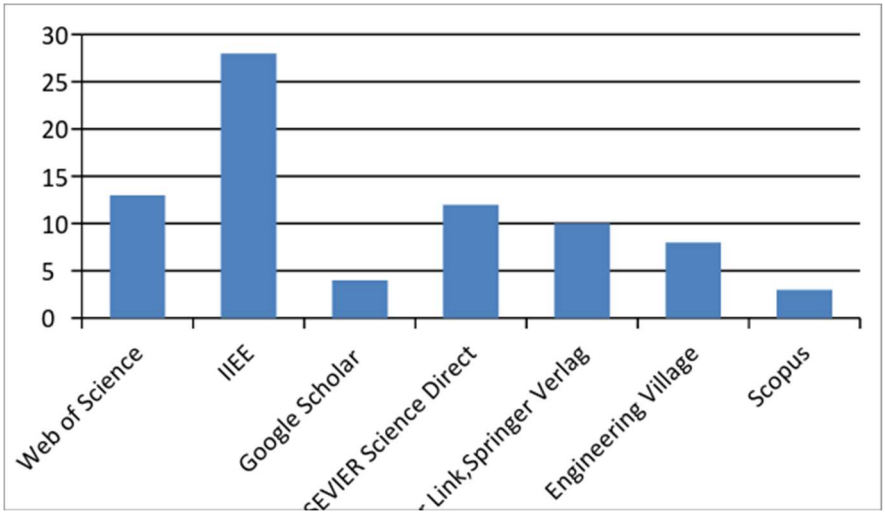
(“Máquinas de vectores de soporte” OR “Máquinas de vectores de apoyo” OR SVM) AND Spark)

Con las estrategias de búsqueda establecidas, se procedió a identificar las fuentes de información confiable que permitirían la recuperación de información científica; por ello, se revisaron las diferentes bases de datos que ofrece el Sistema de Bibliotecas de la Universidad de Antioquia, en el área de “Ingeniería y tecnología”.

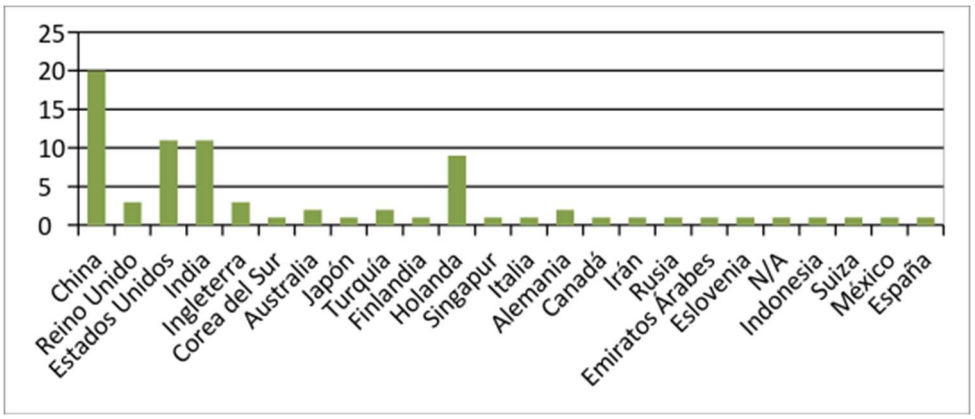
Las bases de datos que vinculaban el tema de investigación dentro de sus contenidos fueron las siguientes:

- ✓ *Access engineering*
- ✓ *ACM Digital Library (Association of computing machinery)*
- ✓ *Alfaomega*
- ✓ *Current Contents*
- ✓ *DOAJ (Directory of Open Acces Journals) Libre acceso*
- ✓ *EBSCO*
- ✓ *Emerald*
- ✓ *Engineering Village*
- ✓ *IEEE Explore*
- ✓ *IOP - Institute of Physics*
- ✓ *Nature*
- ✓ *Redalyc*
- ✓ *Scielo*
- ✓ *Science Direct*
- ✓ *Scopus*
- ✓ *Web of Science*
- ✓ *Wiley InterScience*
- ✓ Wilson
- ✓ Zentralblatt Math

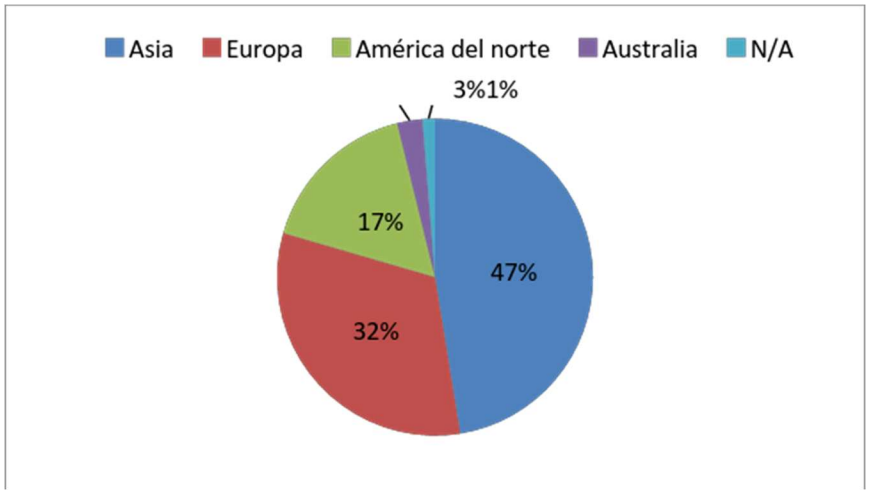
Resultados



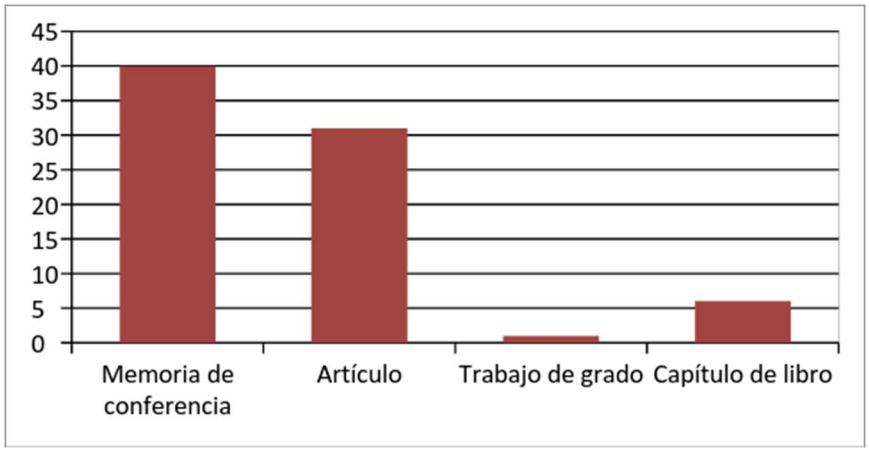
Gráfica. Recuperación de información según las bases de datos-



Gráfica. Recuperación de información según el país de publicación



Gráfica. Recuperación de información según continente



Gráfica. Recuperación de información según tipo de publicación

Criterios utilizados

A los resultados que arrojaban estas bases de datos se les aplicaban los siguientes delimitantes de búsqueda:

Tipo de publicación

Eran válidos todos los formatos de publicación: capítulo de libro, artículo científico, memorias de conferencias, entre otros.

Año de publicación

Publicaciones entre 2010-2019.

Idioma

Inglés y español.

Lugar de publicación

Sin restricciones.