Escuela Politécnica Superior de Jaén
Máster Universitario en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior (Jaén)

Trabajo Fin de Máster

IMPLEMENTACIÓN DE MODELOS DE DATA STREAM SOBRE PLATAFORMAS BIG DATA

Alumno/a: **Puentes Marchal, Fernando**

Tutor/a: María Dolores Pérez Godoy
Pedro González García
Dpto.: Informática

Julio, 2019

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Dña. María Dolores Pérez Godoy y D. Pedro González García, tutores del Proyecto Fin de Máster titulado: Implementación de modelos de Data Stream sobre plataformas Big Data, que presenta Fernando Puentes Marchal, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Julio de 2019

El alumno:

PUENTES MARCHAL FERNANDO - 26505619M

Firmado digitalmente por PUENTES MARCHAL FERNANDO - 26505619M
Fecha: 2019.07.31 10:46:39 +02'00'

Fernando Puentes Marchal

Los tutores:

PEREZ GODOY MARIA DOLORES - 75093149N

Firmado digitalmente por PEREZ GODOY MARIA DOLORES - 75093149N
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-75093149N, givenName=MARIA DOLORES, sn=PEREZ GODOY, cn=PEREZ GODOY MARIA DOLORES - 75093149N
Fecha: 2019.07.31 11:07:52 +02'00'

María Dolores Pérez Godoy

GONZALEZ GARCIA PEDRO - 26210879X

Firmado digitalmente por GONZALEZ GARCIA PEDRO - 26210879X
Nombre de reconocimiento (DN): c=ES, serialNumber=26210879X, sn=GONZALEZ GARCIA, givenName=PEDRO, cn=GONZALEZ GARCIA PEDRO - 26210879X
Fecha: 2019.07.31 11:24:21 +02'00'

Pedro González García

AGRADECIMIENTOS

Me gustaría agradecer en primer lugar a mi familia, en especial a mis padres, Isabel y Fernando, por darme el empujón que necesitaba en los momentos importantes y motivarme a seguir mejorando cada día.

A mis tutores, Loli y Pedro, por guiarme, aconsejarme y la gran disponibilidad durante la realización del presente Trabajo Fin de Máster. También me gustaría agradecerles por ayudarme a conseguir el material y equipos necesarios para la realización del trabajo.

RESUMEN

Cada vez hay más dispositivos conectados a la red que continuamente están generando información en tiempo real. A partir de esta información se puede extraer conocimiento utilizando técnicas de ciencias de datos para obtener un modelo descriptivo o predictivo. Tradicionalmente estas técnicas analizaban conjuntos de datos estáticos almacenados en memoria, pero en la actualidad está adquiriendo cada vez más relevancia el uso de técnicas que analizan los datos en tiempo real, para ayudar en la toma de decisiones.

En la actualidad existen muchas plataformas de *software* libre que facilitan la creación de estas técnicas. La documentación sobre la idoneidad de elección de una de estas plataformas existentes, es escasa, lo que ha motivado la realización de un estudio de las sus características en este trabajo. Además, uno de los factores más importantes al elegir una plataforma de procesamiento es su rendimiento, por lo que también se incluyen dos comparativas con el fin de analizarlas.

En este trabajo también se incluye un estudio de las librerías de *machine learning* y de los algoritmos que éstas incluyen, disponibles en las plataformas analizadas. Tras realizar este estudio se puede apreciar que no hay muchos algoritmos de *machine learning* para analisis de datos en tiempo real dentro de estas librerías. Esta falta de librerías de algoritmos de *streaming* es lo que motiva otro de los objetivos de este trabajo que es la implementación de métodos utilizando la plataforma de procesamiento apache Spark.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN, MOTIVACIONES Y OBJETIVOS DEL PROYECTO

En esta sección se presentan la introducción, las motivaciones y los objetivos del proyecto. Además, se explica la metodología que se ha seguido para la realización del trabajo.

1.1. Introducción

Actualmente, nos encontramos en la era digital en la que hay una gran cantidad de dispositivos conectados a la red que generan una gran cantidad de datos de manera continua, como móviles, tabletas, ordenadores, televisiones, etc. Cada año este número aumenta y surgen nuevos dispositivos, lo que se traduce en más cantidad de información. Toda esta gran cantidad de información puede utilizar para extraer conocimiento y generar modelos que sean capaces de realizar tareas como recomendar contenido basado en los gustos de un usuario, detectar actividad anómala en una red, conocer los sentimientos y gustos de una persona, ayudar en el campo de la medicina o en el sector empresarial.

Debido a la gran cantidad de datos que se analizan, es necesario aplicar técnicas de minería de datos en un contexto de Big Data. Cuando se habla del término Big Data se hace referencia a un gran volumen de datos (pueden ser estructurados o no estructurados) que cumplen las siguientes propiedades (5Vs):

- **Volumen**: Como los datos son generados automáticamente por máquinas, la cantidad de datos que se maneja es enorme.
- **Velocidad**: Cada vez hay más máquinas generando datos en tiempo real, por lo que los datos se generan a una gran velocidad. En estos sistemas se requiere un tiempo de respuesta adecuado ya que los datos tienen un ciclo de vida muy corto, haciendo obsoleto lo que instantes antes era válido.
- **Variedad**: Esta gran cantidad de datos pueden ser estructurada, semi-estructurada o desestructurada y puede proceder de diferentes fuentes, como texto, imágenes, video, etc.

- **Veracidad**: Esta propiedad hace referencia a la credibilidad o confianza de los datos y afecta a la calidad de los resultados.
- **Valor**: Indica el conocimiento e información útil que se puede extraer de los datos.

La minería de datos se ha centrado tradicionalmente en el análisis de esta información almacenada en *datasets* estáticos, en los que toda la información está disponible y se analiza utilizando técnicas de análisis conocidas como técnicas en modo tradicional [1]. Sin embargo, cada vez hay más dispositivos que generan datos de manera continua y se necesitan otro tipo de técnicas que sean capaces de analizar esta información y generar respuestas en tiempo real para ayudar en la toma de decisiones. Estas son conocidas como técnicas en modo *streaming* [2] y procesan los datos tan pronto como estos llegan al sistema.

En la actualidad, existen muchas plataformas de *software* libre y propietarias que facilitan el procesamiento de los datos en modo *streaming* y son utilizados por muchas empresas y la comunidad investigadora. Además, muchas de ellas cuentan con librerías que contienen algoritmos y métodos para facilitar la creación de nuevos algoritmos de *machine learning*, tanto en modo tradicional como en *streaming*.

1.2. Motivación

Los datos llegan al sistema de manera continua en tiempo real a través de un *stream* de entrada, generado por una o más fuentes de datos [3]. Debido a la gran cantidad de datos que se generan, es necesario el uso de plataformas distribuidas para facilitar el procesamiento de *streams* de datos, por lo que en los últimos años han ido apareciendo plataformas para el procesamiento de datos. En un sistema de procesamiento de datos se pueden diferenciar las siguientes fases (Figura 1.1) [4]:

- **Recolección de datos**: En esta primera fase se utiliza una plataforma encargada de recolectar los datos generados de manera continua por una o más fuentes de datos, para unificarlos en un único *stream* de datos de entrada.

- **Procesamiento de datos**: Una vez que los datos se han unificado en un *stream* de datos de entrada, se lleva a cabo la fase de procesamiento, en la que se ejecuta un algoritmo de análisis de datos en *streaming* para extraer conocimiento en tiempo real.
- **Análisis y evaluación**: Esta última fase es opcional y se encarga de visualizar los resultados obtenido en la fase de procesamiento de datos usando algún tipo de herramienta de visualización en tiempo real.

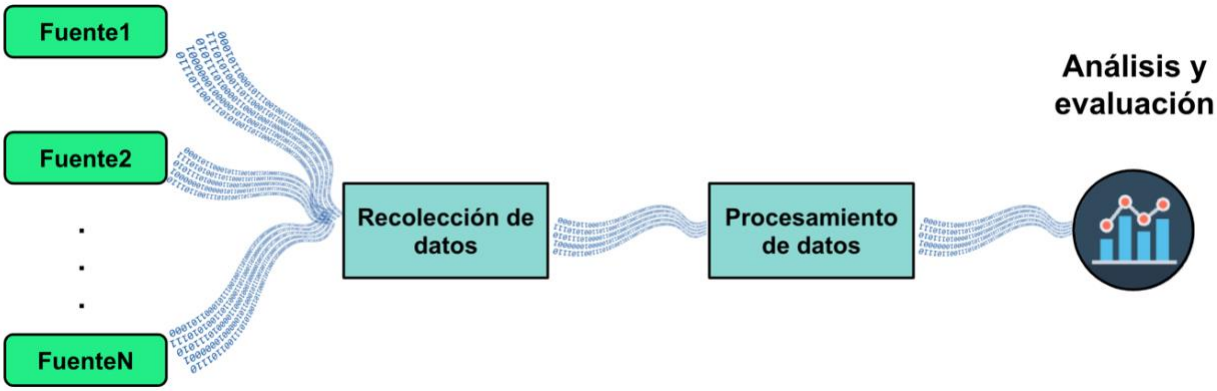

Figura 1.1. Fases de un sistema de procesamiento en streaming

Como se ha comentado en la subsección anterior, actualmente existe una gran variedad de plataformas, tanto de *software* libre como propietario, para realizar la tarea de cada una de las fases dentro de un sistema de procesamiento en *streaming*. Las plataformas propietarias ofrecen ventajas como mejor soporte y la posibilidad de implementar soluciones rápidamente. Sin embargo, hoy en día las plataformas de *software* libre son cada vez más utilizadas, porque permiten la creación de soluciones más eficientes sin restricciones de implementación, y facilita el acceso a toda la comunidad investigadora. Por estos motivos, y con el objetivo de ayudar a decidir que plataforma utilizar según las necesidades requeridas, en este trabajo se incluye un estudio sobre las plataformas de *software* libre más utilizadas para las fases de recolección y procesamiento.

Además, otro factor importante cuando se va a elegir una plataforma de procesamiento es el rendimiento de la misma, es decir, la cantidad de datos por segundo que es capaz de procesar y la latencia. En los últimos años se han realizado varias comparativas entre plataformas de procesamiento para procesar datos en modo *streaming*. Una de las más conocidas es el *Yahoo Streaming Benchmark*, que

se creó para comparar el rendimiento de apache Storm con otras plataformas. Por ello se ha replicado esta comparativa con las herramientas incluidas en este trabajo utilizando las versiones más recientes.

Otro aspecto muy importante a tener en cuenta al elegir una plataforma de procesamiento para *machine learning* es la existencia de librerías que faciliten la creación de este tipo de algoritmos. Por esta razón, se ha realizado un estudio de las librerías de *machine learning* disponibles para cada plataforma, junto con los algoritmos que contiene cada una de ellas. Además, como las comparativas existentes no han comparado el uso de estas plataformas con algoritmos de *machine learning* existentes en *streaming*, en este trabajo también se incluye una comparativa entre apache Spark y apache Flink utilizando el algoritmo *streaming k-Means*.

Con el paso del tiempo han ido apareciendo algoritmos en modo *streaming*. Estos algoritmos en la mayoría de los casos se han planteado para una ejecución secuencial sobre una única máquina que realiza todo el procesamiento. Debido a que existen enormes cantidades de datos que deben ser procesadas, es necesario utilizar técnicas de procesamiento distribuidas que permiten paralelizar los cálculos y reducir el tiempo de procesamiento. Dentro de la tarea de clasificación, se pueden utilizar modelos de árboles de decisión, modelos basados en reglas, vecinos más cercanos, máquinas de vectores de soporte, redes neuronales y *ensembles*. De todos ellos, los menos utilizados son los modelos de redes neuronales debido a su alto coste computacional. Existe, sin embargo, un grupo de algoritmos de diseño de redes neuronales, denominados ELM (*Extreme Learning Machine*) que se caracterizan por su velocidad para actualizar el modelo ante la llegada de nuevos datos. Sus implementaciones son secuenciales y no aprovechan las ventajas de la computación distribuida para procesar grandes volúmenes de datos. Por este motivo, se han realizado la implementación de algunos de estos algoritmos en la plataforma de procesamiento distribuido apache Spark *Streaming*. Finalmente, se ha realizado una comparativa de estos algoritmos con los originales para demostrar que al añadir más nodos la precisión de las soluciones se mantiene.

1.3. Objetivos del proyecto

El propósito de este trabajo es el análisis de diferentes plataformas de *software* libre disponibles para el análisis de datos en *streaming*, la instalación de las mismas en un clúster y realizar comparativas de rendimiento para ayudar a decidir cual elegir. Una vez realizada está comparativa, se han implementado una serie de modelos de redes neuronales en apache Spark *Streaming* y se han evaluado utilizando diferentes *datasets* reales y sintéticos (generados con MOA [5]).

Concretamente, los objetivos propuestos para este proyecto son los siguientes:

- Estudiar los conceptos de *Big data*, *Data stream* y ciencia de datos.
- Conocer las plataformas disponibles que permiten el manejo de grandes volúmenes de información, y el desarrollo de modelos.
- Estudiar las necesidades y elementos necesarios para la instalación de estas plataformas.
- Instalación de la plataforma seleccionada.
- Conocer las bibliotecas ya existentes de ciencias de datos aplicadas a *data stream*.
- Implementar nuevos modelos sobre la plataforma instalada.
- Analizar los resultados obtenidos con respecto a las técnicas tradicionales.

Además de los propuestos, se han cumplido otros objetivos, como la instalación y configuración de un clúster. Sobre este clúster, también se ha realizado la instalación y configuración de las plataformas de recolección y procesamiento en *streaming*. Este trabajo no solo incluye una revisión de comparativas existentes sobre estas plataformas de procesamiento, sino que se han realizado comparativas propias en un entorno de *machine learning*. Además, se han realizado 5 implementaciones de algoritmos de *machine learning* y se ha realizado un estudio muy completo con ellos utilizando diferentes *datasets*, tanto reales como sintéticos. Esto implica que también se han tenido que crear los *datasets* sintéticos, utilizando la herramienta MOA.

1.4. Metodología

Para la realización de este trabajo se ha seguido la siguiente metodología:

- Revisar documentación necesaria para realizar las tareas de este trabajo. Esto incluye los conceptos de *streaming*, las plataformas analizadas, comparativas sobre estas plataformas realizadas con anterioridad, librerías de *streaming* e implementación en apache Spark *Streaming* y Flink.
- Recopilar las principales características de las plataformas analizadas en este trabajo. Entre la información obtenida se incluyen los algoritmos disponibles en librerías de *machine learning*.
- Instalación y configuración de varios clústers con diferentes números de nodos. En esta instalación se incluyen las plataformas comparadas.
- Replicación de la comparativa *Yahoo Streaming Benchmark*, con la configuración de recursos *hardware* del clúster utilizado.
- Realización de una segunda comparativa con un algoritmo de *machine learning* en apache Spark *Streaming* y Flink. En este caso ha sido necesaria la implementación del algoritmo en Flink.
- Implementación de diferentes algoritmos de redes neuronales ELM (*Extreme Learning Machine*) en apache Spark *Streaming*.
- Evaluación de los algoritmos implementados con diferentes *datasets* reales y sintéticos.
- En paralelo a los puntos anteriores, escribir la documentación explicando los contenidos del trabajo.

2. DATA STREAM

Un ***stream*** es un flujo continuo de datos {X, Y} generado por una o más fuentes, donde X es un conjunto de atributos e Y es una clase (clasificación) o un valor numérico (regresión), que van llegando al sistema con el tiempo [3]. El etiquetado de instancias suele ser muy costoso, por lo que la etiqueta Y asociada a cada dato estará disponible un tiempo después de la llegada del dato al sistema. La velocidad con la que llegan los datos al sistema puede variar con el tiempo, es decir, en unos intervalos de tiempo se pueden recibir más datos que en otros. El valor de Y puede existir en todos los datos, en algunos o en ninguno, dando lugar a tres tipos de aprendizaje según este factor [6]:

- **Supervisado**: todos los datos tienen un valor para Y. Normalmente estos valores de Y llegan al sistema después de la llegada de sus atributos X (llega con retardo).
- **Semi-supervisado**: no todos los datos tienen un valor para Y. Llega al sistema el valor de Y para algunos datos (normalmente también con retardo). Algunas técnicas pueden asignar un valor Y a los datos que no lo tienen para luego utilizarlos en el aprendizaje.
- **No supervisado**: ningún dato tiene un valor para Y. El modelo inicial se entrena con un conjunto limitado de datos con valor Y. Después el modelo se actualiza con datos sin la posibilidad de conocer su valor Y.

Debido a las características presentes en el análisis de *streams* de datos, se pueden identificar las siguientes diferencias con respecto al análisis tradicional [2] [6]:

- Los datos aparecen en el *stream* de manera secuencial en tiempo real, mientras que en el modelo tradicional los datos proceden de ficheros almacenados en disco.
- El tamaño del *stream* es dinámico y potencialmente ilimitado debido a que siempre pueden aparecer nuevos datos. Por el contrario, en el modelo tradicional el contenido del *dataset* es finito y estático, ya que el *dataset* completo está almacenado en memoria.
- Debido a la existencia de múltiples fuentes y al envío por la red, no se puede controlar el orden de llegada de los datos. Por el contrario, en el

modelo tradicional todo el *dataset* está disponible, por lo que es posible leerlo en orden.

- Debido a que el *stream* de datos tiene un tamaño ilimitado, todos los datos del *stream* no pueden permanecer en memoria. Por este motivo, cada dato del *stream* se suele tener en cuenta una única vez para el aprendizaje y después es descartado o almacenado en disco (si se necesita un histórico). Por el contrario, en el modo tradicional un mismo dato se puede tener en cuenta varias veces para el aprendizaje.
- En un *stream* se pueden producir cambios en la distribución que genera los datos (el modelo ya no sirve y es necesario actualizarlo a la nueva distribución). Esto no ocurre en el modelo tradicional, donde todos los datos que están contenidos en el *dataset* están generados con la misma función de distribución.
- En *streaming* la etiqueta Y de cada dato estará disponible un tiempo después de la llegada del dato, mientras que, en el modo tradicional, donde los datos están almacenados, el valor Y está disponible desde la lectura del dato.

Estos motivos hacen que no sea posible la aplicación directa de los algoritmos utilizados en el modelo tradicional, ya que deben tener en cuenta estas características y trabajar con limitaciones de memoria y tiempo.

En las siguientes subsecciones se van a describir diferentes conceptos básicos que son necesarios para comprender el análisis de datos en modo *streaming*. En la sección 2.1 se explica la definición de cambio de concepto y sus diferentes tipos. En la sección 2.2 se detalla el concepto de ventana y los diferentes tipos que se pueden encontrar. En la sección 2.3 y 2.4 se describen diferentes aspectos a tener en cuenta al desarrollar un algoritmo de *machine learning* en modo *streaming*. Finalmente, en la sección 2.5 se incluyen las diferentes métricas, *datasets* y técnicas de evaluación que se suelen utilizar para evaluar y comparar diferentes algoritmos en *streaming*.

2.1. Cambio de concepto

Los datos $\{X, Y_i\}$ que llegan por el *stream*, para i = 1, …, c, se generan con una determinada función de distribución de probabilidad $P^t(X, Y_i)$, que es la que indica la relación que existe entre los atributos de entrada (X) y el de salida (Y_i). A cada función de distribución se le conoce como **concepto**. El cambio de este concepto por otro diferente se denomina **cambio de concepto** (en inglés ***concept drift***). Dependiendo de si el concepto cambia o no durante el aprendizaje, se pueden tener dos escenarios: estacionario (no hay cambio de concepto) y no estacionario (pueden existir cambios de concepto). Suponed que durante un intervalo de tiempo t_0 los datos se generan con una determinada función de distribución $P^{t_0}(X, Y_i)$. A partir de t_0 se produce un cambio de concepto durante Δ unidades de tiempo. El concepto va cambiando, hasta estabilizarse en el nuevo concepto en el instante $t_1 = t_0+\Delta$. A partir de ese instante, los datos se generan con la función de distribución $P^{t_1}(X, Y_i)$, de modo que $P^{t_0}(X, Y_i) \neq P^{t_1}(X, Y_i)$ [6].

Para poder estimar $P(X, Y_i)$ en cualquier instante de tiempo t hay que considerar la probabilidad de clase previa $P(Y_i)$ y la probabilidad de clase condicional $P(X|Y_i)$, con la siguiente expresión [7]:

$$P(X, Y_i) = P(Y_i)P(X|Y_i)$$

Después, para tomar la decisión de a que clase pertenece a un dato X, el clasificador selecciona la máxima probabilidad posterior $P(Y_i|X)$ para una clase Y_i según la teoría de decisión Bayesiana [7]:

$$P(Y_i|X) = \frac{P(Y_i)P(X|Y_i)}{P(X)} \qquad \textit{(Fórmula 2.1)}$$

donde $P(X) = \sum_{i=1}^{c} P(Y_i)P(X|Y_i)$ para todas las clases Y_i (i=1,…,c).

En un entorno no estacionario se pueden encontrar diferentes tipos de cambio de concepto según el factor que se modifique en la fórmula 2.1 [7]:

- **Cambio de concepto real** (Figura 2.1 b): Se produce un cambio en $P(Y_i|X)$ que afecta a los límites de decisión y hace necesario la actualización del modelo para adaptarse al nuevo concepto.

- **Cambio de concepto virtual** (Figura 2.1 c): Se produce un cambio en $P(X|Y_i)$ o en $P(Y_i|X)$, pero este cambio no afecta a los límites de decisión del modelo, por lo que el rendimiento del clasificador no disminuye y no es necesario una adaptación.
- **Cambio de concepto en las clases**: Se produce un cambio en $P(Y_i)$, que puede resultar en la aparición de clases no balanceadas (Figura 2.1 d), nuevas clases emergentes (Figura 2.1 e) o fusión de clases existentes (Figura 2.1 f). Normalmente este tipo de cambio se asocia a cambio de concepto real si afecta a los límites de decisión del modelo o a cambio de concepto virtual si no afecta a dichos límites.

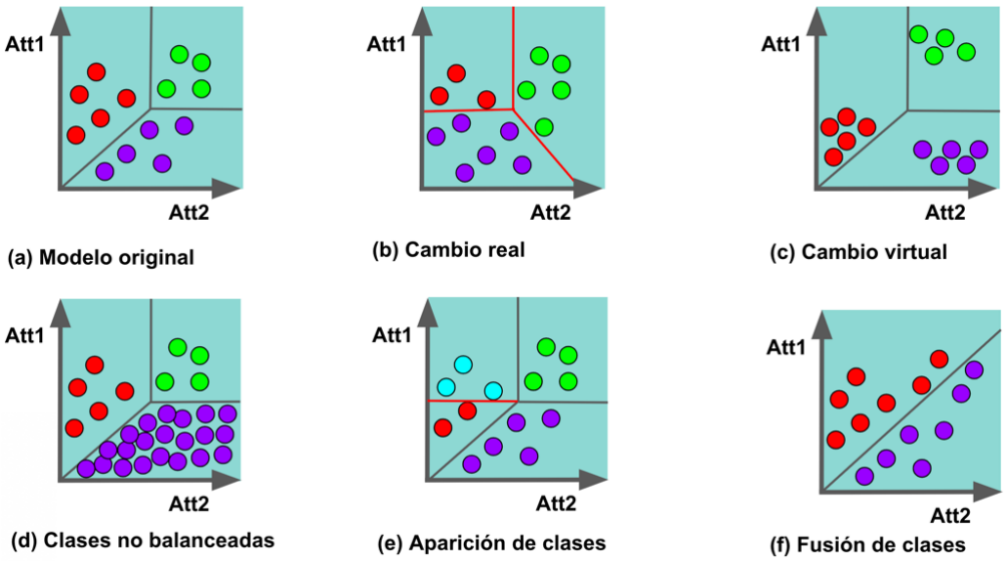

Figura 2.1. Tipos de cambio de concepto (I)

Un cambio de concepto de C1 a C2 se puede presentar de diferentes formas en un *stream* [8] [9]:

- **Abrupto** (Figura 2.2 a): El cambio de concepto entre C1 y C2 es inmediato o en un periodo de tiempo muy pequeño. Este tipo de cambio reduce rápidamente el rendimiento del clasificador y es más fácil de detectar.
- **Gradual** (Figura 2.2 b): Durante el tiempo que dura el cambio, los conceptos C1 y C2 están presentes. Los datos se generan con una combinación de C1 y C2, hasta que se estabiliza el concepto C2. Este cambio es más difícil de detectar debido a que el concepto cambia poco a poco y no todos los detectores son buenos para este tipo de cambios.

- **Incremental** (Figura 2.2 c): Durante el intervalo que dura el cambio de concepto solo existe un concepto, que es el anterior con pequeñas modificaciones, iniciando desde C1. Este concepto va modificándose incrementalmente hasta estabilizarse en el concepto C2.
- **Recurrente** (Figura 2.2 d): Este tipo de cambio se produce cuando vuelve a aparecer un concepto que ya apareció previamente. Como se ve en la imagen, primero hay un cambio de C1 a C2 y luego de C2 a C1 (vuelve a aparecer el concepto C1).
- **Anómalo** (Figura 2.2 e): Son cambios aleatorios que se producen durante un instante corto y luego se vuelve al concepto original. Este tipo de cambios se debe ignorar.
- **Ruido** (Figura 2.2 f): Representa fluctuaciones insignificantes del concepto que deben ser filtradas.
- **Combinado**: Se produce cuando aparecen más de un tipo de cambio de concepto en el mismo *stream*.

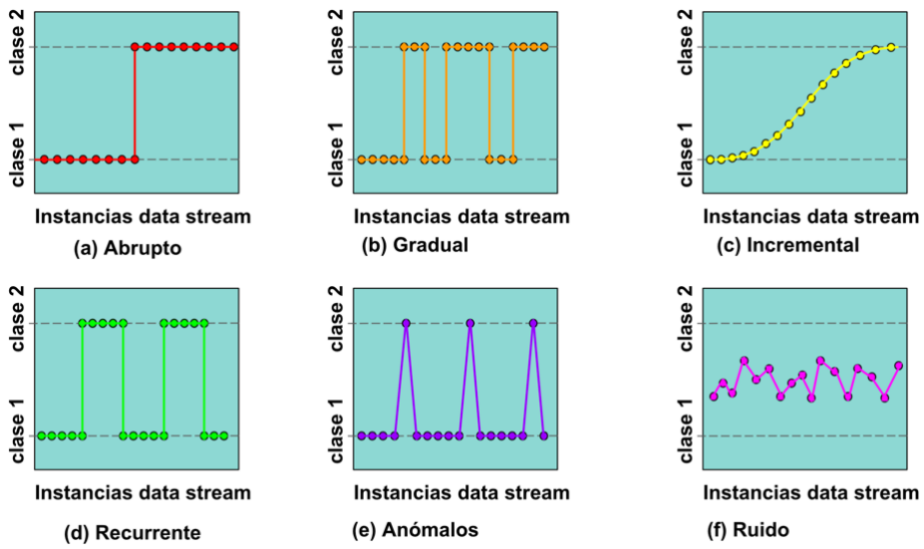

Figura 2.2. Tipos de cambio de concepto (II)

Según la severidad del cambio, que hace referencia a la gravedad del cambio que se ha producido en el modelo, se distinguen dos tipos de cambio de concepto:

- **Locales** (Figura 2.3 a): Los cambios ocurren en una subregión del espacio (si se observa todo el espacio, los cambios solo afectan a algunas regiones). Este tipo de cambios son más costosos de encontrar en términos de computo. Además, en muchas ocasiones es fácil confundir

este tipo de cambios con ruido, por lo que hay que tener especial cuidado a la hora de detectar este tipo de cambio.

- **Globales** (Figura 2.3 b): En este caso los cambios afectan a todo el espacio de instancias, por lo que es más fácil identificar este tipo de cambios y reaccionar a ellos.

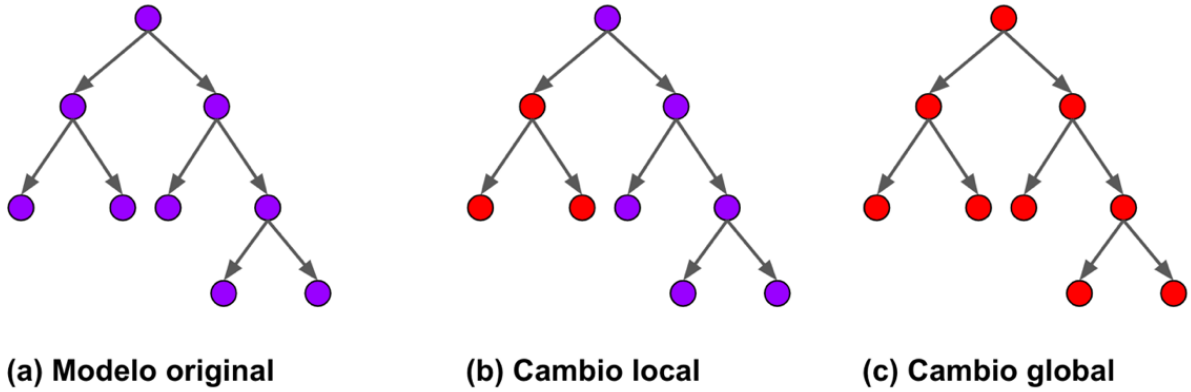

Figura 2.3. Tipos de cambio de concepto (III)

Según la previsibilidad de un cambio de concepto, que indica si la aparición de un nuevo concepto es totalmente aleatoria o se puede conocer cuando se va a producir, se distinguen dos tipos de cambios:

- **Predecible** (Figura 2.4 a): El cambio sigue un patrón que permite conocer cuando se va a producir el cambio.
- **No predecible** (Figura 2.4 b): El cambio es totalmente aleatorio.

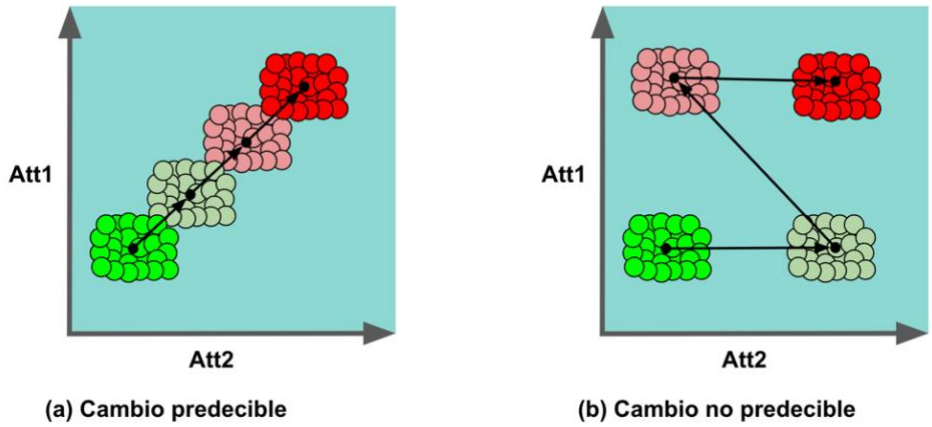

Figura 2.4. Tipos de cambio de concepto (IV)

2.2. Ventanas

Al realizar un procesamiento en *streaming* los datos se pueden procesar por bloques utilizando una estructura denominada **ventana**. Una ventana es una estructura que permite almacenar una serie de datos en ella para luego procesarlos en conjunto. Cada ventana tiene asociado una **longitud** y un **desplazamiento**, como muestra la Figura 2.5.

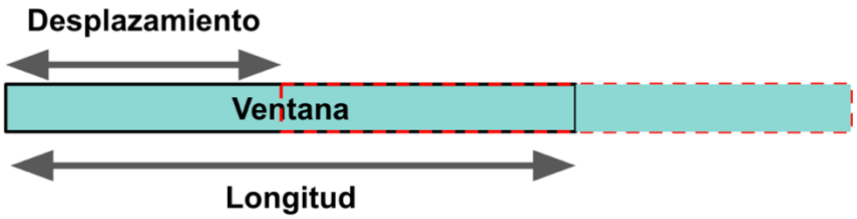

Figura 2.5. Parámetros de las ventanas

Según las unidades utilizadas para expresar la longitud y el desplazamiento de la ventana, existen dos tipos de ventanas [8] [7]:

- **Ventanas temporales**: La ventana está expresada en unidades de tiempo y los datos tienen asociados una **marca de tiempo** (***timestamp***) para asociarles una ventana. Cada ventana almacena todos los datos con un *timestamp* entre el inicio y el fin (inicio + duración) de la misma. Cuando se llega al final de una ventana, esta se procesa y se desplaza tantas unidades de tiempo como indique el desplazamiento (inicio + desplazamiento). En el ejemplo mostrado en la Figura 2.6 (a) aparece una ventana temporal de longitud 3 segundos y desplazamiento 2 segundos. Este ejemplo supone que llega un nuevo dato cada segundo y muestra el contenido de la ventana en cada instante.
- **Ventanas secuenciales**: La ventana está expresada en unidades de **instancias** (número de datos), por lo que en este caso no es necesario un campo *timestamp* adicional. Cada ventana almacena tantos datos como indique la longitud de la ventana (sin importar el tiempo que tarde en llenarse). Cuando una ventana está llena, esta se procesa y a continuación actúa como una **cola FIFO** (***First-In First-Out***), por lo que cada dato nuevo que llega al sistema sustituye al más antiguo de la ventana. Cada vez que se introducen en la ventana tantas instancias

nuevas como indique el desplazamiento, se realiza un procesamiento. En la Figura 2.6 (b) se puede apreciar un ejemplo de ventana secuencial en la que la longitud es 12 instancias y el desplazamiento es 1 instancia. En este caso se muestra la llegada de las diferentes instancias (no el tiempo) y se muestra el contenido de la ventana en cada momento.

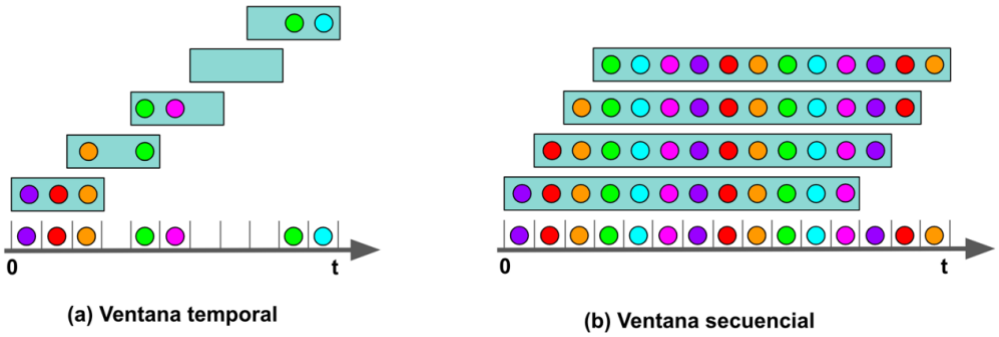

Figura 2.6. Ventanas temporales y secuenciales

Al utilizar una ventana, el valor de desplazamiento siempre es fijo, pero el valor del tamaño de la ventana puede variar o mantenerse fijo. Según este criterio, una ventana puede ser [8] [7]:

- **Ventana de tamaño fijo**: El tamaño de la ventana se mantiene inalterado durante toda la ejecución. Esta es la opción más simple al implementar una ventana. En la Figura 2.7 (a) se muestra un ejemplo en el que aparece un cambio de concepto, pero el tamaño de la ventana no cambia.
- **Ventana de tamaño variable**: El tamaño de la ventana va variando entre un mínimo y un máximo en función de algún criterio. El criterio habitual es utilizar un detector de cambio de concepto y utilizar su salida como referencia. Mientras no se detecte un cambio de concepto, la ventana va creciendo progresivamente hasta llegar al máximo. En el momento en el que se detecta un cambio de concepto, el tamaño de la ventana se establece al mínimo y solo se mantienen las instancias más recientes, que son las que representan el concepto actual. En la Figura 2.7 (b) se puede apreciar un ejemplo con el mismo cambio que el ejemplo anterior, pero en este caso el tamaño de la ventana va aumentando hasta que detecta un cambio. Tras detectar el cambio el tamaño de la ventana es el mínimo y a continuación vuelve a crecer.

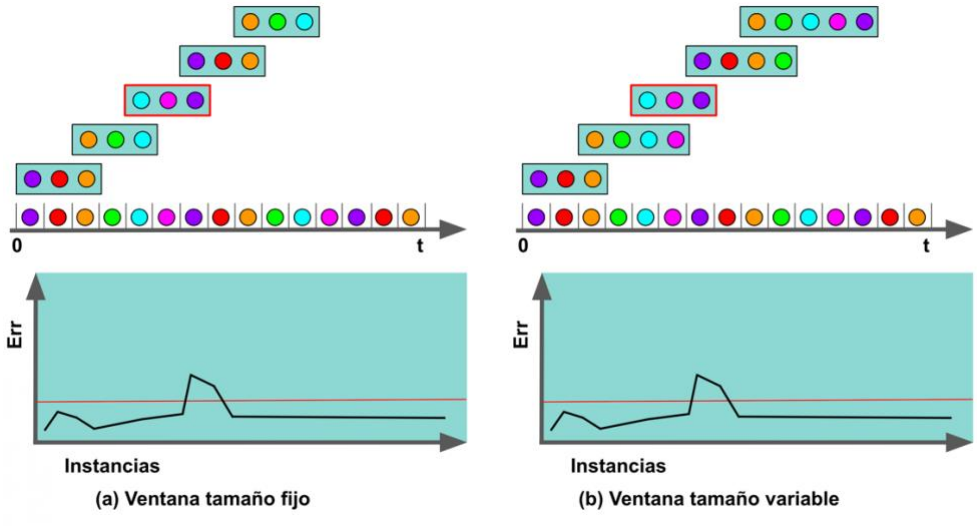

Figura 2.7. Ventanas de tamaño fijo y variable

Al utilizar ventanas para manejar el cambio de concepto se pueden utilizar diferentes estrategias, algunas de ellas son mejores para detectar un determinado tipo de cambio. Las opciones utilizadas en la literatura son [7]:

- **Una ventana**: En este caso se pueden encontrar dos estrategias:
  - **Ventana deslizante** (Figura 2.8 a): Es equivalente a utilizar una ventana deslizante de tamaño fijo.
  - **Ventana *landmark*** (Figura 2.8 b): Es equivalente a utilizar una ventana deslizante de tamaño variable, en la que el desplazamiento solo se produce cuando se detecta un cambio de concepto.
- **Dos ventanas**: La idea es utilizar una ventana de referencia para almacenar datos del concepto actual y una con los datos del instante actual, para comprobar si hay cambio de concepto comparando la medida de rendimiento para ambas ventanas. En este caso existen los siguientes enfoques:
  - ***Ventanas separadas*** (Figura 2.8 c): La ventana de referencia se mantiene con unos datos fijos mientras no se detecte un cambio de concepto y la ventana actual se va desplazando. Cuando se detecta un cambio de concepto, los datos de la ventana actual sustituyen a los de la ventana de referencia y el proceso continúa. Este enfoque es útil para detectar cambios graduales.

- **Ventanas adyacentes**: Las ventanas de referencia y la actual guardan una relación (normalmente los datos de la ventana actual sustituyen o se añaden a los de la ventana de referencia). En este caso se pueden encontrar los siguientes tipos según si el tamaño de las ventanas es fijo o variable:
  - **Fijo/Fijo** (Figura 2.8 d): El tamaño de ambas ventanas es fijo. Este tipo de ventana es útil para detectar cambios de concepto abruptos.
  - **Variable/Fijo** (Figura 2.8 e): El tamaño de la ventana de referencia es variable, mientras que el de la ventana actual es fijo. Esta estrategia es más útil para detectar cambios de concepto graduales.
  - **Variable/Variable** (Figura 2.8 f): El tamaño de ambas ventanas es variable. Este enfoque es útil para detectar tanto cambios de concepto graduales como abruptos.
- **Ventanas solapadas** (Figura 2.8 g): En esta estrategia las ventanas de referencia y la actual tienen datos en común, por lo que puede ser de interés en situaciones en las que hay pocos datos o cuando los datos no están balanceados.
- **Ventanas discontinuas** (Figura 2.8 h): La ventana de referencia tiene un tamaño variable y solo se utiliza un subconjunto de datos de esta (seleccionados siguiendo algún criterio) para comparar con la ventana de referencia. Este enfoque es útil para detectar cambios locales.

- **Tres ventanas** [10]: En este enfoque se utiliza un conjunto de 3 ventanas de tamaño variable (pequeña, mediana y grande) para detectar diferentes tipos de cambio de concepto. Cada ventana se centra en una velocidad de cambio:
  - **Pequeña**: Se utiliza para detectar cambios muy rápidos.
  - **Mediana**: Se utiliza para detectar cambios lentos.
  - **Grande**: Se utiliza para detectar cambios muy lentos.

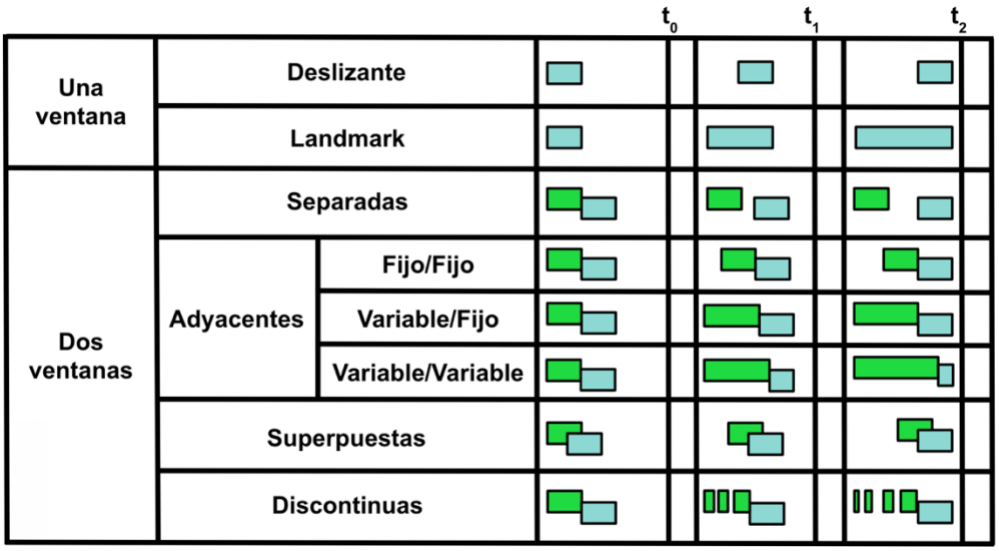

Figura 2.8. Estrategias con ventanas

2.3. Proceso de aprendizaje

Otro de los factores que se deben elegir cuando se desarrolla un algoritmo para realizar tareas de *machine learning* en *streaming* es el número de clasificadores que va a utilizar el modelo. Según este factor hay dos opciones [7]:

- **Un clasificador**: El modelo solo está formado por un único clasificador que debe ser incremental para poder actualizarse con nuevos datos. Puede utilizar un **mecanismo de olvido** para olvidar la información que ha quedado obsoleta y dar más importancia a los datos recientes. Puede ser una buena opción que permite que el sistema no sea muy complejo, pero no es recomendable para problemas de cambios de concepto recurrentes. Cuando aparece un concepto recurrente, esta opción lo considera como un nuevo concepto y se adapta a él sin utilizar los beneficios del conocimiento previo.
- **Un conjunto de clasificadores (*ensemble*)**: En este caso el modelo está formado por varios clasificadores, permitiendo una mejor generalización que los modelos con un único clasificador. Cada clasificador del *ensemble* puede ser entrenado con los mismos datos o con diferentes, dependiendo del clasificador base utilizado. Cuando se clasifica un dato utilizando un *ensemble*, cada clasificador toma una decisión. Después, estas decisiones se combinan para generar la decisión final, normalmente

utilizando una estrategia de voto ponderado o mayoritario. El éxito de un método de *ensemble* para manejar el cambio de concepto depende de dos criterios: **diversidad** (diferencias entre los modelos del *ensemble*) y **adaptabilidad** (capacidad del *ensemble* para adaptarse a nuevos conceptos). Ambos pueden conseguirse a través de tres factores:

- **Gestión de los datos de entrenamiento**: Indica como se utilizan los datos para entrenar cada modelo. Existen tres tipos:
  - **Basado en bloques**: Los datos de entrenamiento se presentan por bloques de datos en cada instante. Generalmente el tamaño de los bloques es igual.
  - **Basado en peso**: En este enfoque cada dato tiene un peso asociado y los clasificadores se entrenan con ellos. Pueden ser de gran interés para problemas de clase no balanceada o cambios locales.
  - **Datos filtrados**: Esta técnica se basa en seleccionar determinados datos del conjunto de entrenamiento siguiendo algún criterio de selección específico. Incluso se pueden realizar filtrados de atributos. Esto permite que cada clasificador se pueda entrenar con diferentes zonas del espacio de datos.
- **Gestión de la estructura**: Indica si se modifica o no la estructura del *ensemble*. Hay dos opciones:
  - **Estructura fija**: El número de clasificadores en el *ensemble* no cambia. En este caso se pueden utilizar diferentes técnicas para adaptarse al concepto, como entrenar un nuevo clasificador con cada nuevo bloque de datos y reemplazar al peor (según un criterio de rendimiento) del *ensemble*.
  - **Estructura variable**: El número de clasificadores en el *ensemble* se adapta automáticamente. De esta forma se pueden añadir nuevos modelos sin tener que eliminar uno existente, permitiendo utilizarlos en el futuro. Este tipo es útil para cambios de concepto recurrentes.

- **Gestión de la decisión final**: Indica como se combina la decisión de los diferentes clasificadores para generar la decisión final. En este caso existen tres opciones:
  - **Ponderación dinámica**: Cada clasificador tiene un peso asociado que se actualiza dinámicamente según un criterio (normalmente su precisión/error). La combinación de las decisiones se hace por votación ponderada, multiplicando la decisión de cada clasificador por su peso asociado.
  - **Selección dinámica**: De todos los clasificadores del *ensemble* se elige uno para tomar la decisión final. Esta selección se realiza siguiendo un criterio que normalmente es elegir el clasificador con mayor precisión.
  - **Combinación de las anteriores**: En este caso se seleccionan varios clasificadores según un criterio y luego se combinan sus decisiones mediante votación ponderada.

2.4. Proceso de adaptación

Los datos llegan continuamente al sistema y en ellos se pueden producir cambios de concepto, como se ha explicado previamente. Para que el modelo pueda adaptarse a los nuevos conceptos es necesario desarrollar una estrategia de adaptación. Cuando se diseña un mecanismo de adaptación se deben tener dos factores en cuenta [7]:

- **Cuándo adaptar el modelo**: Este factor indica en qué momento se debe adaptar el modelo con los datos más recientes para adaptarse al concepto. En este caso hay dos opciones:
  - **Enfoque "a ciegas"**: El modelo se adapta periódicamente cada cierto tiempo o número de instancias. No utiliza un detector de cambio de concepto, ya que para adaptarse a nuevos conceptos se basa en la idea de actualización continua. Un inconveniente de este enfoque es que la velocidad de adaptación es constante, ya que la actualización suele utilizar un método incremental que

actualiza el modelo poco a poco. Esta idea es buena ante cambios graduales, ya que va adaptándose poco a poco, pero no para cambios abruptos, porque el modelo se adapta de la misma forma que en un cambio gradual y la adaptación a un cambio abrupto requiere un tiempo. Además, otro inconveniente es que el coste computacional es mayor que un enfoque informado debido a que se realizan actualizaciones del modelo constantemente.

- **Enfoque informado**: El modelo se adapta solo cuando se detecta un cambio de concepto. En este caso si se utiliza un detector de cambio de concepto que es el que indica cuando actualizar el modelo. Normalmente estos detectores monitorizan el rendimiento del modelo para detectar un cambio de concepto. En este caso se debe tener especial cuidado con las falsas alarmas, ya que si se detectan muchas falsas alarmas puede reducir considerablemente el rendimiento del sistema. En caso de detectar un cambio se puede decidir si se actualiza el modelo actual incrementalmente o se entrena uno nuevo desde cero.

- **Cómo olvidar la información obsoleta**: Este factor indica la velocidad con la que se olvida la información obsoleta para adaptarse mejor al concepto actual. Si se produce un cambio abrupto la velocidad de adaptación debe ser mayor que para un cambio gradual. En este factor se pueden identificar dos opciones:
  - **Olvido abrupto (o memoria parcial)**: En este enfoque se utiliza una ventana (temporal o secuencial) que contiene los datos más recientes recibidos por el *stream*. El modelo solo conoce los datos que están dentro de la ventana, por lo que cuando entran nuevos datos los aprende y cuando salen datos de la ventana los olvida. Otra forma puede ser muestrear determinadas instancias para mantener un resumen del concepto del *stream*.
  - **Olvido gradual (o memoria completa)**: En este caso no se olvida completamente ningún dato. Cada dato tiene asociado un peso (**factor de penalización**) según su antigüedad, que va disminuyendo con el paso del tiempo. Los datos más recientes tienen un peso mayor que los más antiguos, por lo que se les da

más importancia a los datos más recientes, aunque no se olviden los datos obsoletos. Según la velocidad con la que se olvida la información obsoleta el factor de penalización puede ser lineal o exponencial.

2.5. Evaluación

Un aspecto muy importante cuando hay varios algoritmos disponibles para hacer la misma tarea es realizar una comparativa para ver el comportamiento de los mismos y cuál es el mejor [11]. En este caso, la comparación debe tener en cuenta que el concepto puede cambiar con el tiempo. Además, se deben utilizar diferentes *datasets* para evaluar alguna métrica y verificar el rendimiento del algoritmo ante problemas de diferente tipo. Para poder obtener las métricas en *streaming* se debe utilizar una técnica de evaluación que se ajuste a las características de procesamiento continuo que se presenta. En las siguientes subsecciones se describen estos aspectos.

2.5.1. Datasets

La elección de *datasets* es un factor importante a la hora de evaluar un algoritmo. Normalmente los *datasets* están orientados a un determinado problema, como clasificación o predicción. Además, influyen otros factores como número de atributos, atributos numéricos o nominales, número de clases, etc. Por estos motivos hay que elegir *datasets* que estén orientados al problema que intenta resolver el algoritmo a evaluar. En *streaming* se pueden utilizar dos tipos de *datasets* [12]:

- **Reales**: Son *datasets* obtenidos en situaciones reales y que sirven para ver como se comporta el algoritmo ante un problema real.
- **Sintéticos**: Son *datasets* obtenidos artificialmente mediante algún software, como por ejemplo **MOA** [5], que son totalmente configurables y permiten saber en que momentos exactos del *dataset* se producen los cambios de concepto y el tipo de cambio. Estos *datasets* son muy útiles para evaluar el comportamiento del algoritmo ante la presencia de cambios de concepto.

Es muy importante combinar el uso de *datasets* reales y sintéticos para evaluar bien el algoritmo. En [12] se ha realizado un estudio en el que se muestran diferentes *datasets* utilizados en la tarea de clasificación. En la Tabla 2.1 se incluyen los más utilizados.

Datasets reales	Datasets sintéticos
KDD Cup 99	SEA
Image Segmentation	STAGGER
Adult	SINE
Covertype	CIRCLES
Poker	Hyperplane
Electricity	GAUSS
Wine	LED display
SEER Breast Cancer	Waveform

Tabla 2.1. Datasets reales y sintéticos más utilizados

2.5.2. Métricas

Para evaluar el rendimiento de un algoritmo en *streaming* es necesario monitorizar alguna métrica que indique lo bueno que es el algoritmo en cada momento. En *streaming* se suelen utilizar las mismas métricas que en el modo tradicional [6]:

- **Precisión**: Proporción de instancias correctamente clasificadas.
- **Error**: Proporción de instancias incorrectamente clasificadas.
- ***Recall*** / ***Sensitivity***: Es la precisión de una clase específica.

Normalmente se utiliza el error o la precisión del modelo. Además de medir estas métricas, es interesante tener en cuenta otras como:

- **Coste computacional**: Monitorizar el uso de memoria y CPU a lo largo del tiempo.
- **Tiempo de actualización**: Monitorizar el tiempo que tarda el modelo en entrenar con los nuevos datos.
- **Tiempo de decisión**: monitorizar el tiempo que tarda el modelo en tomar la decisión al clasificar cada instancia.

2.5.3. Técnicas de evaluación

En el entorno tradicional, donde está disponible todo el *dataset*, la técnica de evaluación más utilizada para evaluar las métricas es la **validación cruzada**. Sin embargo, en un contexto en *streaming* donde no están disponibles todos los datos, se tienen recursos computacionales limitados y aparecen cambios de concepto, esta técnica no es directamente aplicable. Por este motivo es necesario considerar otras técnicas que se adapten a este entorno. Se suelen utilizar dos enfoques según si el escenario es estacionario o no [6] [11]:

- ***Holdout***: Esta técnica se utiliza en entornos estacionarios, donde no hay cambios de concepto. Esta técnica consiste en mantener un conjunto de datos que es independiente del modelo (no se ha utilizado para entrenar el modelo) y, a intervalos regulares, se evalúa con este conjunto independiente.
- ***Prequential* (*Test-then-Train*)**: Esta técnica se utiliza en entornos no estacionarios con cambios de concepto, aunque también se puede utilizar en entornos estacionarios. Cada nuevo dato que llega al sistema se utiliza primero para evaluar el modelo (***Test***) y luego para entrenar el modelo (***Train***). De esta forma, el modelo se está evaluando con datos que no ha visto antes y no es necesario mantener un conjunto de datos independiente. Además, en este método no es necesario evaluar todas las instancias, por lo que se favorece su utilización en entornos semi-supervisados. Este enfoque es un **método pesimista**, porque le da la misma importancia a todos los datos en lugar de darle más importancia a los datos recientes, obteniendo un valor para la métrica superior al real. Por este motivo se suele utilizar esta técnica con un mecanismo de olvido:
  - **Ventana**: Se utiliza una ventana de tamaño fijo para estimar la métrica para las instancias dentro de la ventana. En este caso se utiliza una fórmula incremental para añadir los nuevos datos y eliminar los que salen de la ventana. La fórmula utilizada es la siguiente:

$$S(i) = \frac{1}{w}\sum_{k=i-w+1}^{i} e_k \qquad \textit{(Fórmula 2.2)}$$

donde:

S(i) es la métrica en el instante i

W es el tamaño de la ventana

e_k es el error de la muestra k

- **Factor de olvido**: Utiliza un factor de penalización (0-1) para estimar la métrica. En este caso no se olvida totalmente la aportación de cada instancia, sino que se le asigna un peso cada vez menor. La fórmula utilizada es la siguiente:

$$S(i) = x_i + \alpha S(i-1) \qquad \textit{(Fórmula 2.3)}$$

donde:

S(i) es la métrica en el instante i

$\mathbf{x_i}$ es el valor de la métrica para el dato actual

α es el factor de penalización

De las dos técnicas de evaluación, la más utilizada en el contexto de *streaming* es el método *prequential* con un mecanismo de olvido para dar más importancia a los datos más recientes. La métrica más evaluada es el error y para verificar el comportamiento de un algoritmo ante cambios de concepto es necesario utilizar *datasets* sintéticos.

3. PLATAFORMAS PARA ANÁLISIS DE DATOS EN STREAMING

Las plataformas de análisis de datos en *streaming* son herramientas de carácter distribuido que permiten implementar un procesamiento paralelo en tiempo real, aprovechando los recursos computacionales de un clúster. En la actualidad cada vez hay más fuentes, como sensores o móviles, que emiten datos de manera continua. Estos datos deben analizarse en tiempo real, por lo que es necesario el uso de este tipo de herramientas. Debido a la existencia de muchas plataformas para el análisis de datos en *streaming* no es fácil elegir la adecuada. Por este motivo, en esta sección se realiza un estudio de las características de las principales plataformas para análisis de datos en *streaming*. En la Tabla 3.1 se incluyen plataformas de software propietario recogidas por los informes de Gartner [13] [14] y las plataformas más importantes de software libre.

Plataformas	
Software propietario	**Software libre**
Alooma	Apache Apex
Amazon Kinesis	Apache Beam
APPAMA Streaming Analytics	Apache Flink
Arcadia Data Enterprise	Apache Flume
Attunity	Apache Gearpump
Azure Stream Analytics	Apache Ignite
Big Data Streaming	Apache Kafka
Confluent Platform	Apache Nifi
dA Platform	Apache S4
Data Beaming	Apache Samza
Data Stream Manager	Apache Spark
e-Server	Apache Storm
Esper Enterprise Edition	Elastic Logstash
EVAM	
Google Cloud Dataflow	
Hortonworks DataFlow (HDF)	
IBM Streams	
Inetco Systems Analytics	
Interana	
Nexla	
Oracle Stream Analytics	
RapidMiner Platform	
RB Light	
SAP Event Stream Processor	
Spring Cloud Data Flow (SCDF)	
StreamAnalytics	
Streamlio	
Striim	
Talend Data Streams	
Unified Analytics Platform	

Tabla 3.1. Plataformas de software propietario

En este trabajo se han analizado plataformas de software libre, porque son las que tienen más importancia para la comunidad investigadora. Todas ellas tienen su código disponible en el repositorio *maven* [15], lo que facilita la importación de dependencias.

Este apartado se divide en tres subsecciones. En la subsección 3.1 se van a analizar diferentes plataformas para la recolección de datos y en la subsección 3.2 algunas plataformas para el procesamiento de datos. Finalmente, la subsección 3.3 muestra las librerías de *machine learning* analizadas en este trabajo.

3.1. Plataformas de recolección de datos

En entornos de *streaming* normalmente van a existir diferentes fuentes que van a estar generando simultáneamente *streams* de datos hacia el sistema de procesamiento. Al tener varios *streams* de entrada, es necesario realizar una tarea de recolección o ingesta de datos antes del procesamiento. La recolección de datos consiste en obtener los datos de los diferentes *streams* de entrada para unificarlos y generar un único *stream* de entrada al sistema de procesamiento. Estos datos se pueden generar de uno en uno o por lotes, y son introducidos en el sistema de procesamiento en el mismo orden en el que llegan a la etapa de recolección.

En este tipo de plataformas, se pueden encontrar diferentes propiedades que diferencian unas de otras. A continuación, se detallan las propiedades que se incluyen en este análisis:

- **Capacidad de pre-procesamiento**: Indica si la plataforma permite modificar los datos que recolecta mediante alguna transformación.
- **Abstracción de datos**: Es la estructura de datos que utiliza la plataforma para separar la especificación de los datos de su implementación.
- **Garantía de entrega**: Como los envíos de datos se hacen a través de la red, pueden producirse problemas durante el envío que hacen que aparezcan datos perdidos o duplicados. Esta propiedad mide la aparición de este problema. Hay tres tipos diferentes:

    - **Como máximo una vez**: Cada dato se puede enviar 0 o 1 vez. En este caso se puede dar el fenómeno de datos perdidos.
    - **Exactamente una vez**: Cada dato se envía exactamente 1 vez. Este es el caso ideal en el que no pueden aparecer datos perdidos ni duplicados.
    - **Al menos una vez**: Cada dato se puede enviar 1, 2, 3, etc veces. En este caso pueden aparecer datos duplicados.
- **Utilidad de monitorización**: Permite conocer si la plataforma tiene una interfaz (normalmente web) en la que se puede visualizar el estado del flujo en tiempo real.
- **Tipo de topología**: En un sistema de recolección normalmente van a existir diferentes fuentes y diferentes destinos. Esta característica hace referencia a la posibilidad de que aparezcan o se eliminen fuentes/destinos durante la ejecución. Hay dos tipos de topología:
    - **Estática**: Una vez que se ha ejecutado el sistema de recolección, no se pueden incluir nuevas fuentes/destinos. Para hacer esto es necesario reiniciar el sistema.
    - **Dinámica**: Permite que aparezcan nuevas fuentes/destinos durante la ejecución sin necesidad de reiniciar el sistema.
- **Replicación de datos**: Consiste en tener una copia de los datos en diferentes nodos del clúster. En caso de que un nodo caiga, los datos se obtienen de otro nodo y el sistema sigue funcionando.
- **Integración con plataformas de procesamiento**: Propiedad que indica qué plataformas de procesamiento tienen soporte para la plataforma de recolección, es decir, si la plataforma de procesamiento tiene implementado un receptor para leer el *stream* generado por la plataforma de recolección.

El objetivo de esta sección es analizar y comparar diferentes plataformas para realizar la tarea de recolección de datos (primera etapa), destacando sus principales características. En particular, esta sección analiza apache Kafka, Flume y Nifi.

3.1.1. Apache Kafka

Apache Kafka es una plataforma distribuida para *streaming* creada por LinkedIn en 2011 que permite recolectar datos de diferentes fuentes. Como muestra la Figura 3.1, en una arquitectura de Kafka se pueden diferenciar tres elementos principales:

- ***Broker***: Es el elemento central, que se encarga de almacenar datos y estar escuchando las peticiones que llegan del resto de elementos. El *broker* está compuesto por una serie de colas de mensajes, cada una con un identificador único denominado ***topic***.
- **Productor**: Se encarga de enviar datos a uno o más *topics*.
- **Consumidor**: Se encarga de recibir datos en tiempo real de uno o más *topics*.

Figura 3.1. Esquema general de apache Kafka

En esta plataforma, los datos se denominan ***records*** (registros), que son pares clave-valor en formato texto, donde la clave normalmente se forma con una marca de tiempo y el valor es una cadena de texto en un formato específico (normalmente JSON).

Kafka trabaja siguiendo una **metodología publicación/subscripción** [16], en la que los productores publican datos en uno o más *topics* y los consumidores se subscriben a uno o más *topics* para recibir en tiempo real los datos que se publican en estos *topics*. Esto hace que tenga una estructura más dinámica, ya que se pueden incluir nuevos productores/consumidores o eliminar alguno ya existente sin necesidad de reiniciar el elemento central.

Cada *topic* se puede dividir en una o más **particiones** (como muestra la Figura 3.2), que son una secuencia ordenada e inmutable de *records*. Según la clave del *record*, un dato es enviado a una partición u otra y se mantiene almacenado ella durante un periodo de tiempo denominado **periodo de retención**. Tras este periodo de tiempo, los datos son eliminados de la partición. Cada partición tiene un identificador numérico secuencial denominado ***offset***, que identifica de manera única cada *record* dentro de la partición. El valor de *offset* es mantenido por los consumidores, permitiendo que se pueda leer datos de las mismas particiones a diferentes velocidades.

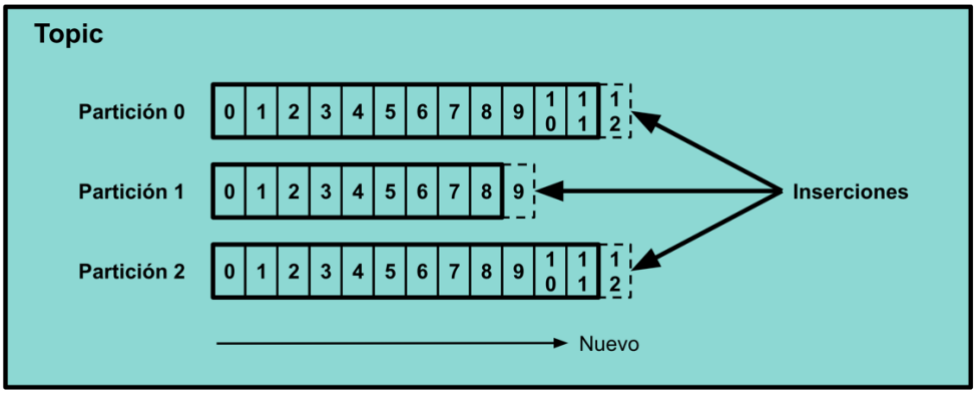

Figura 3.2. Estructura de un topic

Kafka permite la distribución de los datos, por lo que las particiones se distribuyen entre diferentes nodos del clúster. Cada partición tiene un nodo que actúa como ***leader*** y cero o más nodos que actúan como ***followers***. El *leader* recibe todas las peticiones de lectura/escritura y los *followers* copian al *leader*, realizando las mismas acciones. Si el *leader* se cae, automáticamente un *follower* pasa a ser el nuevo *leader*. Además, Kafka permite la replicación de los datos, por lo que las particiones pueden tener copias en varios nodos y si un nodo cae se utiliza otro. Esto proporciona tolerancia a fallos.

Esta plataforma es muy potente para la recolección de datos, pero también permite realizar tareas de procesamiento gracias a la ***API Kafka Streams***. Esta API permite leer datos de un *topic*, realizar alguna tarea de procesamiento y almacenar el resultado en otro *topic*. Aunque permite realizar tareas de procesamiento, no es tan potente como las plataformas de procesamiento vistas en la subsección 3.2.

Kafka es una plataforma de recolección muy utilizada tanto por la comunidad investigadora como por empresas. Por este motivo, tiene una gran cantidad de documentación e información sobre problemas resueltos, lo que acelera el desarrollo de aplicaciones y la corrección de errores. Además, al ser tan utilizada, tiene compatibilidad con todas las plataformas de procesamiento vistas en este trabajo.

3.1.2. Apache Flume

Apache Flume es un servicio distribuido para mover, agregar y recolectar eficientemente flujos de datos de diferentes fuentes y almacenarlos en un sistema de datos centralizado. Como muestra la Figura 3.3, en Flume se utilizan **agentes** compuestos por diferentes elementos que pueden ser de tres tipos [17]:

- **Fuente**: Recolecta datos de una fuente de datos externa (por ejemplo, un servidor web) y los almacena con un formato específico en uno o más canales.
- **Canal**: *Buffer* pasivo que almacena los datos recibidos de la fuente hasta que son consumidos por un elemento denominado sumidero, que se comenta más adelante. Este *buffer* intermedio es necesario porque en general las velocidades de lectura y escritura son diferentes. En Flume existen diferentes tipos de canales según el almacenamiento que se utiliza:
  - **En memoria**: Almacena los datos en memoria. Es más rápido, pero si el nodo cae los datos se pierden y no se pueden recuperar.
  - **En ficheros**: Almacena los datos en disco. Es más lento, pero si el nodo cae los datos se pueden recuperar, proporcionando tolerancia a fallos.
  - **Kafka**: Almacena los datos en Kafka. Kafka proporciona replicación, por lo que si un nodo cae los datos se pueden recuperar de otro nodo, proporcionando tolerancia a fallos.
  - **JDBC**: Almacena los datos en un almacenamiento persistente que está respaldado por una base de datos.

- **Memoria desbordable**: Es un híbrido de los dos primeros. Utiliza la memoria como almacén primario y el disco como secundario. Cuando la memoria está llena los nuevos datos se almacenan en disco. Actualmente este tipo de canal es de uso experimental y no se recomienda su uso.

- **Sumidero (*Sink*)**: Es el responsable de eliminar un dato del canal y enviarlo al sistema destino para almacenarlo. Un dato no es eliminado del canal hasta que no es almacenado en el sistema destino. Solamente puede haber un canal asociado a cada sumidero.

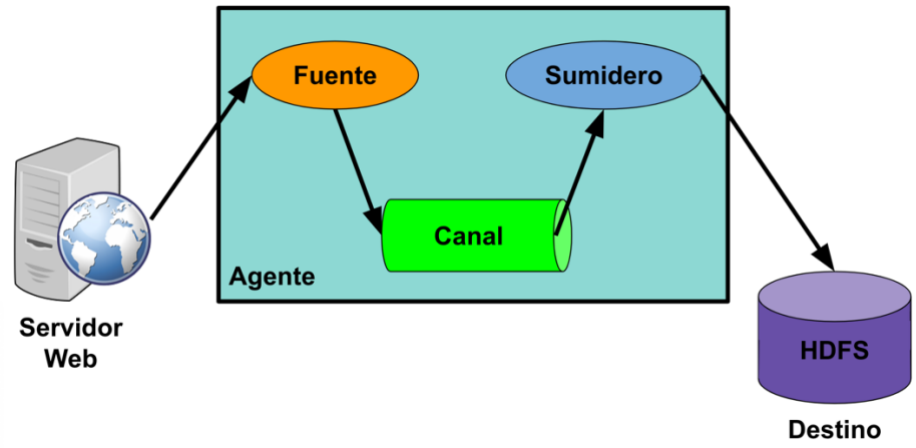

Figura 3.3. Esquema general de apache Flume

Un agente puede contener varios elementos de cada tipo. Además, los agentes se pueden encadenar (Figura 3.4) formando un **flujo multi-salto**, donde los datos van pasando de un agente a otro hasta llegar al sistema destino.

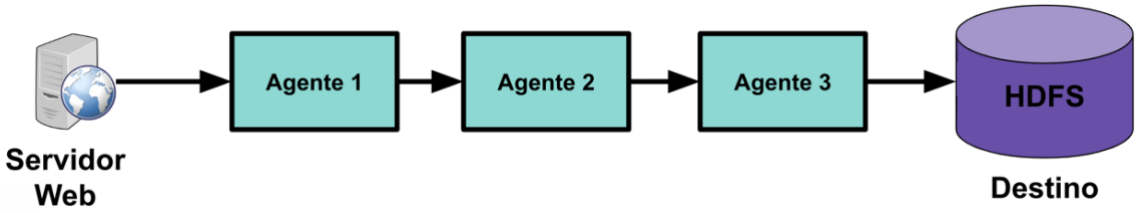

Figura 3.4. Cadena de agentes

En Flume los agentes se configuran en un archivo de texto almacenado localmente. Este archivo contiene los parámetros necesarios para cada elemento de los diferentes agentes, que debe incluir, por cada elemento: un nombre, el tipo y un conjunto de propiedades específicas según el tipo de elemento.

Otra característica es que puede integrarse fácilmente con Apache Hadoop [18]. Por este motivo, su principal uso es mover grandes cantidades de datos de uno o más sistemas origen a un sistema HDFS, que es el sistema de ficheros distribuido de Hadoop que permite compartir ficheros muy grandes entre varias máquinas.

Esta plataforma no presenta una estructura dinámica (como en el caso de Kafka), ya que en Flume es necesario especificar todas las fuentes origen y destino que van a interactuar con el sistema y estas no se modifican durante la ejecución. Si se quiere añadir/eliminar alguno de ellos, hay que reiniciar el sistema. Esto hace que Kafka sea una plataforma con un propósito más general que Flume.

3.1.3. Apache Nifi

Apache Nifi es una plataforma de recolección que permite automatizar el movimiento de datos de un sistema de ficheros origen a otro sistema de ficheros destino. En la arquitectura de Nifi se utilizan unos elementos denominados ***processors***, que interaccionan con las fuentes origen, realizan operaciones sobre los datos o envían datos al sistema destino [19]. Los *processors* están conectados entre sí mediante unas colas denominadas **conexiones**, que permite a varios procesos interaccionar a diferentes velocidades. Además, esta plataforma proporciona una interfaz de usuario web muy potente e intuitiva que facilita la creación y configuración de dichos elementos. Además, está interfaz web actúa como un monitor para visualizar el estado del flujo en tiempo real y detectar posibles errores que se hayan producido. En la Figura 3.5 se puede apreciar un ejemplo de aplicación con Nifi en la interfaz web.

En esta plataforma los datos que se mueven a través del sistema se denominan ***FlowFiles***. Cada *FlowFile* contiene un *map* de pares clave-valor, donde la clave es un atributo y el valor es el contenido de dicho atributo.

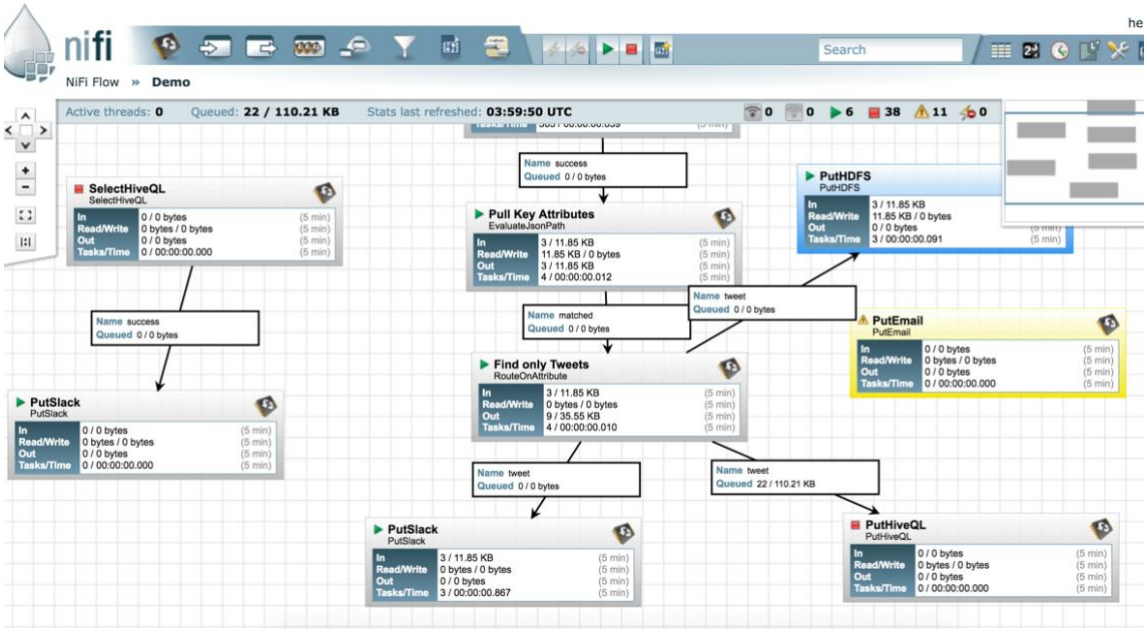

Figura 3.5. Esquema general de apache Nifi

Fuente: https://community.hortonworks.com/articles/45706/using-the-new-hiveql-processors-in-apache-nifi-070.html

Cuando Nifi se utiliza en un clúster (como muestra la Figura 3.6), todos los nodos realizan las mismas tareas con diferentes bloques de datos (los datos se reparten entre los diferentes nodos). Nifi no permite la replicación de los datos, por lo que, si un nodo cae, el flujo es redireccionado a otro nodo. Los datos que estaban en ese nodo se mantienen en él hasta que se active de nuevo.

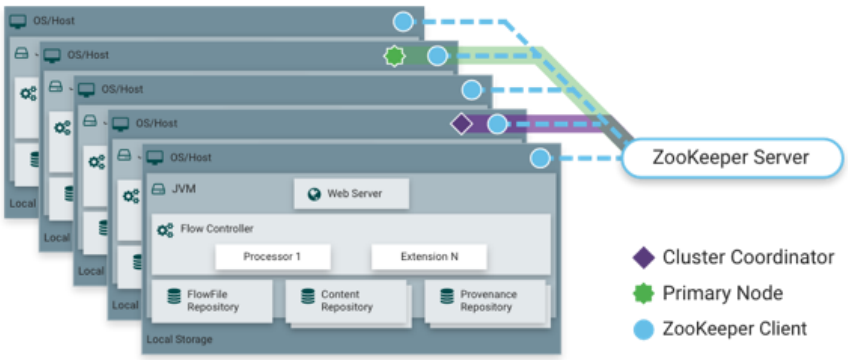

Figura 3.6. Apache Nifi en modo clúster

Fuente: https://nifi.apache.org/docs/nifi-docs/html/administration-guide.html

Al igual que pasa con Flume, en Nifi la estructura del sistema es estática, es decir, las fuentes origen y destino son fijas y no se pueden modificar durante la ejecución. Si se desea modificar hay que reiniciar el sistema. Esto hace que Kafka tenga un propósito más general.

3.1.4. Resumen de las características de las plataformas de recolección

En esta sección se han analizado tres plataformas distribuidas para la recolección de datos en *streaming*. Para cada una de ellas, se han explicado sus principales características y su modelo de funcionamiento. La Tabla 3.2 recoge un resumen de las características de estas plataformas. Todas ellas se pueden utilizar en Windows/Linux/Macos y permiten realizar tareas de pre-procesamiento sobre los datos, pero solo Kafka permite realizar tareas de procesamiento (con un peor rendimiento que las plataformas de procesamiento vistas en la subsección 3.2) gracias a su *API Kafka Streams*. Ofrecen garantías de entrega al menos una vez, por lo que los datos pueden tener duplicados y solo soportan Java como lenguaje de programación. Todas proporcionan tolerancia a fallos, pero en el caso de Flume solo si se utilizan canales basados en ficheros. Solo Nifi tiene una utilidad de monitorización de forma nativa, aunque Kafka tiene software de terceros para monitorizar el flujo, como *Burrow* de LinkedIn. Solo Kafka soporta replicación de datos y tiene una estructura dinámica, por lo que Kafka tiene un propósito más general que Flume y Nifi. Como Kafka es muy utilizada por la comunidad investigadora y las empresas, es la que tiene más documentación y más compatibilidad con las plataformas de procesamiento.

Características	**Plataformas**		
	Kafka	**Flume**	**Nifi**
Versión analizada	2.1.0 (11/2018)	1.9.0 (01/2019)	1.8.0 (10/2018)
Pre-procesamiento	Si	Si	Si
Procesamiento	Si*	No	No
Garantía de entrega	Al menos una vez	Al menos una vez	Al menos una vez
Tolerancia a fallos	Si	Si*	Si
Lenguajes de programación	Java	Java	Java
Utilidad de monitorización	No*	No	Si
Estructura dinámica	Si	No	No
Sistemas operativos	Windows Linux MacOS	Windows Linux MacOS	Windows Linux MacOS
Documentación	Mucha	Poca	Poca
Replicación de datos	Si	No	No
Abstracción de datos	*Record*	*Event*	*FlowFile*
Tipo de topología	Dinámica	Estática	Estática
Integración con plataformas de procesamiento	Todas	Spark Streaming	Flink

Tabla 3.2. Características de las plataformas de recolección

Para la etapa de recolección de datos Kafka es la más utilizada debido a que ofrece muchas ventajas sobre el resto de plataformas de recolección. La primera a destacar es su estructura dinámica, que permite añadir/eliminar fuentes origen y

destino sin necesidad de reiniciar el sistema. Otro punto a favor de Kafka es la replicación de datos, que permite la continuidad del sistema en caso de que algún nodo del clúster caiga. Además, al ser tan utilizada por la comunidad investigadora y las empresas tiene una gran cantidad de documentación y tiene compatibilidad con todas las plataformas de procesamiento que se van a describir en la subsección 3.2, por lo que se facilita y acelera el desarrollo de aplicaciones en *streaming*. Por estos motivos, para la realización de las pruebas en este trabajo se ha decidido utilizar apache Kafka como plataforma de recolección de datos.

3.2. Plataformas de procesamiento de datos

Una vez que se recolectan los datos y se introducen al sistema como un *stream* único, es necesario utilizar una plataforma de procesamiento (al igual que en la etapa anterior, de carácter distribuido) para procesar este *stream* de datos de entrada. Este procesamiento se lleva a cabo mediante la utilización de un algoritmo de *machine learning* en *streaming* con el objetivo de descubrir conocimiento.

En las plataformas de procesamiento también se pueden encontrar diferentes propiedades que las caracterizan. A continuación, se describen algunas de estas propiedades con el objetivo de explicar su significado:

- **Modo de procesamiento**: Indica el modo de procesamiento que se va a realizar, que puede ser de dos tipos:
  - **Tradicional**: Se procesa un conjunto de datos finito y estático que está almacenado que está almacenado en un sistema de ficheros. Este conjunto se analiza en conjunto.
  - ***Streaming***: En este caso no hay un *dataset* finito, sino que los datos van llegando al sistema con el tiempo, siendo analizados en cuanto alcanzan el sistema. Debido al tamaño del *stream* (infinito), los datos solo se tienen en cuenta una vez durante el procesamiento y luego son descartados. Dentro de este modo los datos se pueden analizar **dato a dato** o por pequeños bloques, denominados ***micro-batches***.

- **Garantía de procesamiento**: Hace referencia al número de veces que un mismo dato puede ser procesado por la plataforma. Aunque en teoría cada dato solo es procesado una vez y luego se descarta en el contexto de *streaming*, en la práctica esto podría no cumplirse. Esto se debe a que los datos tienen que enviarse entre máquinas y pueden ocurrir problemas, produciendo datos duplicados o datos perdidos. Hay tres tipos de garantías de procesamiento:
    - **Como mucho una vez**: Los datos se pueden procesar una vez o ninguna. Pueden aparecer datos perdidos.
    - **Exactamente una vez**: Los datos se procesan solo una vez, ni más ni menos. En este caso se garantiza que no aparecen datos duplicados ni perdidos.
    - **Al menos una vez**: Los datos se pueden procesar 1, 2, 3, etc veces. En este tipo de garantía pueden aparecer datos duplicados.
- **Abstracción de datos**: Es la estructura de datos que utiliza la plataforma para separar la especificación de los datos de su implementación.
- **Utilidad de monitorización**: Permite conocer si la plataforma tiene una interfaz (normalmente web) en la que se puede visualizar el estado del clúster en tiempo real.
- **Modificaciones durante la ejecución**: Indica si se pueden añadir/eliminar nodos al clúster durante la ejecución del sistema, sin necesidad de reiniciar el clúster para ello.
- **Tipo de procesamiento**: Al realizar el procesamiento, cada nodo del clúster que realiza el procesamiento puede almacenar información previa o no. En esta propiedad hay dos posibilidades:
    - **Con estado**: Cada nodo realiza un procesamiento basándose en el conocimiento adquirido previamente y después actualiza el estado con los nuevos resultados.
    - **Sin estado**: En este caso los nodos procesan cada dato sin tener en cuenta los resultados previos (no se almacenan los estados previos).

- **Unidad de procesamiento**: Cuando una plataforma de procesamiento obtiene los datos del *stream* de entrada, esta puede obtenerlos dato a dato o por bloques (utilizando una ventana).
- **Uso de ventanas**: Indica si la plataforma tiene soporte para el uso de ventanas, de modo que se pueda realizar un procesamiento por bloques. Hay dos tipos de ventanas que se pueden usar en una plataforma de procesamiento:
    - **Basadas en tiempo**: El usuario define un tiempo como tamaño de ventana y otro como desplazamiento. Esta ventana almacena todos los datos que lleguen en cada intervalo de tiempo y los procesa en conjunto.
    - **Basadas en secuencia**: El usuario define un número de instancias como tamaño de ventana y otro como desplazamiento. Esta ventana almacena datos hasta que se llena, después los procesa y por último se desplaza.
- **Librerías de *machine learning***: Para poder realizar algoritmos de *machine learning* las plataformas de procesamiento ofrecen funciones a alto nivel para facilitar el desarrollo. Además de estas funciones, pueden tener librerías que contengan algoritmos y métodos implementados para ayudar en el desarrollo de algoritmos para esa plataforma.

En esta sección se analizan y comparan diferentes plataformas para realizar la tarea de procesamiento (segunda etapa), destacando sus principales características. En concreto se analizan apache Spark (modo *streaming* y modo *structured streaming*), Storm, Flink, Samza, Apex y Beam.

3.2.1. Apache Spark Streaming

Apache Spark es un motor de código abierto para el procesamiento distribuido, con diferentes modos de funcionamiento. Uno de estos modos es ***Spark Streaming*** [20], una extensión del núcleo de Spark que permite el procesamiento escalable, de alto rendimiento y con tolerancia a fallos de flujos continuos de datos. Los datos llegan al sistema por uno o más flujos de entrada que son procesados utilizando alguna

transformación implementada en *Spark Streaming*. Las transformaciones implementadas actualmente son las que se incluyen en la Tabla 3.3.

Transformaciones
map
flatMap
filter
repartition
union
count
reduce
countByValue
countByKey
join
cogroup
transform
updateStateByKey

Tabla 3.3. Transformaciones sobre un RDD

Cuando se programa en Spark, se utiliza una estructura de datos de solo lectura denominada ***Resilient Distributed Dataset*** (RDD), que es un conjunto de datos distribuido entre los diferentes nodos del clúster y que pueden ser procesados en paralelo. Los RDDs soportan dos tipos de operaciones: transformaciones y acciones. Cuando se realiza una transformación sobre un RDD, se obtiene como resultado un nuevo RDD con los cambios realizados y el original se mantiene inalterado. Cuando se realiza una acción sobre un RDD, esto devuelve el resultado de realizar algún cálculo sobre dicho RDD.

Spark Streaming funciona como muestra la Figura 3.7. Primero se debe especificar un intervalo de tiempo denominado **intervalo de *batch*** y *Spark Streaming* se encarga de dividir el flujo por intervalos (***micro-batches***) de esa duración utilizando una ventana temporal. Por ejemplo, si el intervalo de *batch* es de 1 segundo, cuando la aplicación se inicia, *Spark Streaming* almacena todos los datos recibidos cada segundo en un RDD diferente (datos del segundo 0-1 al RDD0, datos del segundo 1-2 al RDD1, datos del segundo 2-3 al RDD2, …). De esta manera, cada segundo tendremos un RDD con los datos que han llegado en ese intervalo. Esta secuencia de RDDs se abstrae en una estructura superior denomina ***Discretized Stream*** (DStream). Cada vez que *Spark Streaming* genera un nuevo RDD, este es procesado por el motor de Spark, tal y como se hace en el modo tradicional. Cada vez que el motor de Spark procesa un RDD se genera un nuevo RDD a la salida (Figura 3.8), por lo que a la salida se tendrá otro DStream.

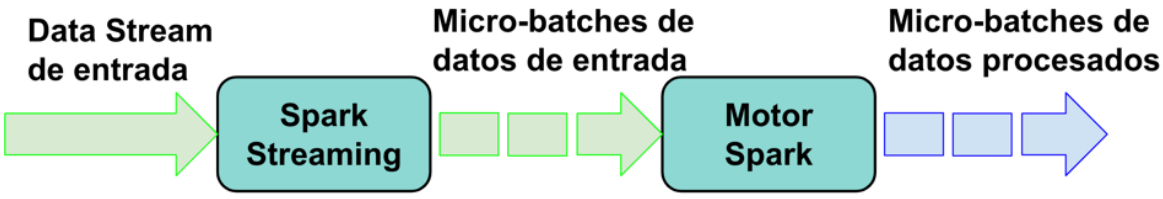

Figura 3.7. Esquema general de apache Spark Streaming

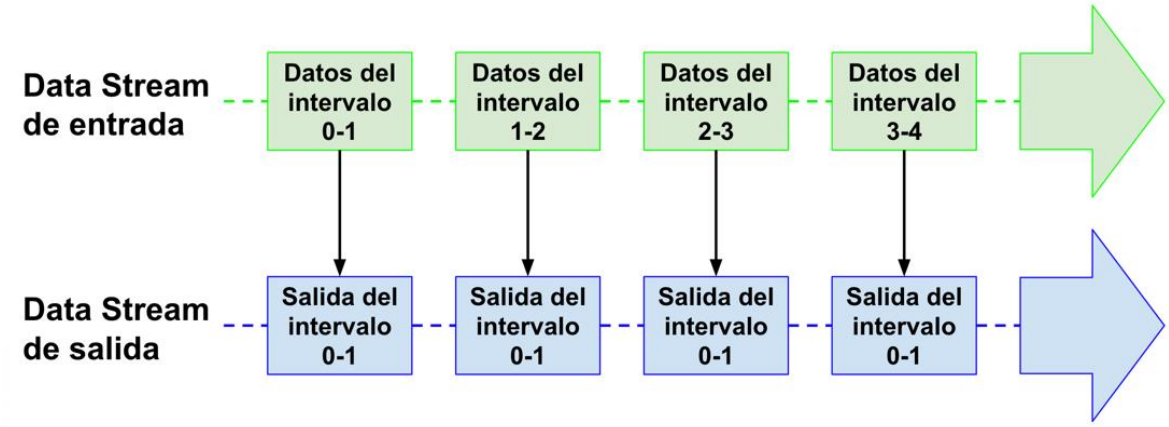

Figura 3.8. Transformación de un DStream

El flujo de datos al sistema se puede obtener de diversas fuentes utilizando uno de los receptores implementados en *Spark Streaming*. Estos receptores simplifican la conexión con el sistema destino y la obtención de los datos utilizando funciones de alto nivel, sin que el desarrollador tenga que implementar nada. En caso de que se necesite recibir los datos de un sistema que no está incluido en *Spark Streaming*, se ofrece la posibilidad de implementar un receptor personalizado. Actualmente los receptores disponibles son los siguientes:

- Sistemas de ficheros.
- Conexiones TCP.
- Kafka.
- Flume.
- Kinesis.

Debido a su modelo de *micro-batches*, cuando los datos llegan al sistema tienen que esperar a que termine la duración del *batch* para poder ser procesados. Esto hace que la latencia (tiempo que transcurre desde que un dato llega al sistema hasta que es procesado) de *Spark Streaming* sea mayor que en el resto de plataformas de procesamiento.

Con respecto a las librerías de *machine learning*, *Spark Streaming* tiene compatibilidad con MLLIB, streamDM y AmidstToolbox, que se describen en la subsección 3.3. Esta es la plataforma de procesamiento de datos en *streaming* que tiene más algoritmos de *streaming* implementados actualmente.

Spark Streaming también permite el uso de ventanas temporales. Para poder utilizar una ventana temporal se deben especificar el **tamaño** y el **desplazamiento de la ventana**, ambos deben ser múltiplos del intervalo de *batch*. Por ejemplo, en la Figura 3.9 el intervalo de *batch* es 1, el tamaño de la ventana es 3 y el desplazamiento es 2. Esto permite procesar varios RDDs como si fueran uno, es decir, para este caso primero los datos del RDD0, RDD1 y RDD2 se unirían en un mismo RDD y luego este RDD sería procesado, generando un RDD a la salida. Después la ventana se desplaza 2 unidades, manteniendo el último RDD de la ventana anterior. A continuación, se haría lo mismo con RDD2, RDD3 y RDD4. Las operaciones de ventana soportadas por *Spark Streaming* están incluidas en la Tabla 3.4.

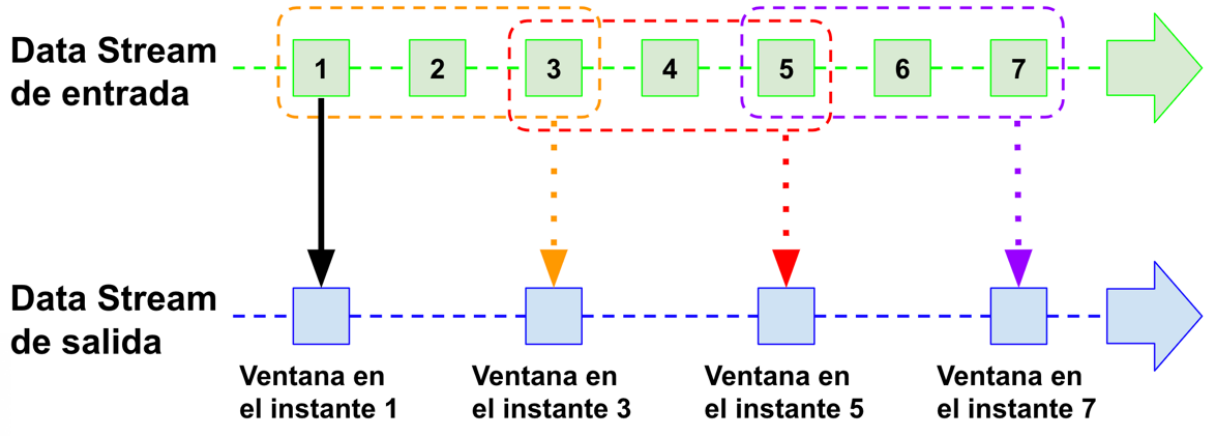

Figura 3.9. Ventanas temporales en Spark Streaming

Operaciones de ventana
window
countByWindow
reduceByWindow
countByKeyAndWindow
reduceByKeyAndWindow

Tabla 3.4. Operaciones sobre ventanas en Spark Streaming

3.2.2. Apache Spark Structured Streaming

Otro modo de funcionamiento que tiene Spark para el análisis de datos en *streaming* es ***Spark Structured Streaming*** [21], que está implementado sobre el motor de Spark SQL y permite el procesamiento de flujos de datos de forma similar a como se hace en el modo tradicional, sobre conjuntos de datos estáticos. En *Structured Streaming* se utiliza una tabla ilimitada de entrada que contiene los datos que llegan por el *stream*. Cada vez que llega un nuevo dato por el *stream*, este se añade como una nueva fila al final de la tabla, como se puede apreciar en la Figura 3.10. En este modo de funcionamiento también se procesan los datos por *micro-batches* y, después de ser procesados, se descartan de la tabla de entrada. En este caso no es necesario especificar el intervalo de *batch*. Si no se especifica, *Structured Streaming* procesa los nuevos datos tan pronto como sea posible (cuando termina el procesamiento previo).

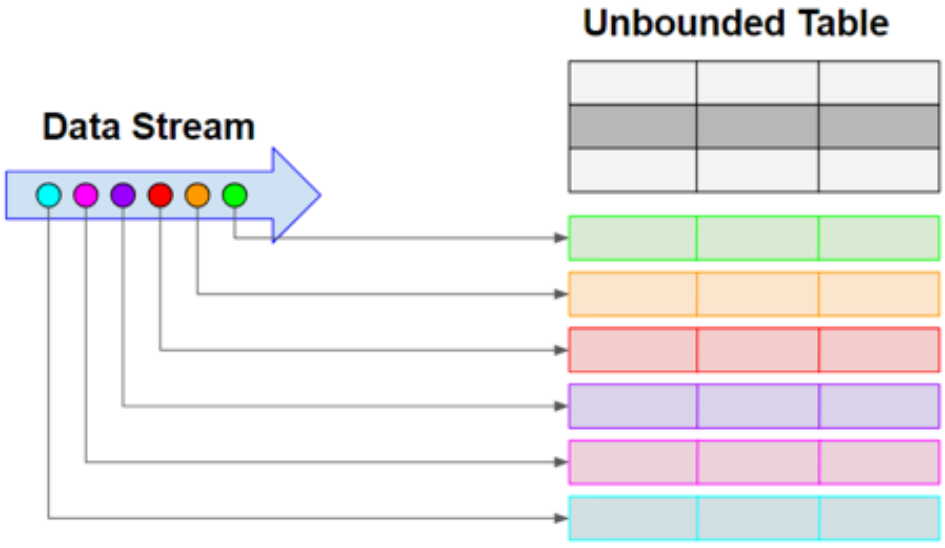

Figura 3.10. Estructura de datos de Spark Structured Streaming

La estructura de datos utilizada en *Structured Streaming* para abstraer esta tabla es ***Dataset/DataFrame***. Un *Dataset* es una colección de datos distribuida, que proporciona los beneficios de los RDDs junto con los de Spark SQL y que puede ser manipulada mediante la aplicación de operaciones (Tabla 3.5). Un *DataFrame* es un *Dataset* organizado en columnas, parecido a una tabla en el modelo relacional de bases de datos.

Operaciones
select
filter
groupBy
groupByKey
join
consultas SQL
window
watermark
count
dropDuplicates

Tabla 3.5. Operaciones sobre un Dataset/DataFrame

Sobre esta tabla de entrada se realizan consultas que se resuelven de forma incremental, generando un resultado tras cada procesamiento que actualiza la tabla de salida. El **modo de salida** define qué datos de la tabla de salida son almacenados en el almacenamiento externo y, dependiendo del tipo de consulta, se podrá utilizar uno u otro. En *Structured Streaming* hay 3 modos de salida diferentes:

- **Modo *append***: Solo se almacenan las nuevas filas de la tabla de salida que han aparecido desde la última actualización. Únicamente las consultas que añaden nuevas filas a la tabla de salida (y que no modifican las existentes) soportan este modo.
- **Modo *complete***: Todos los resultados de la tabla de salida se almacenan en cada actualización. Las consultas de agregación soportan este modo.
- **Modo *update***: Solo se almacenan aquellas filas de la tabla de salida que han modificado su valor desde la última actualización.

Al igual que *Spark Streaming*, en este modo de funcionamiento también hay receptores implementados para obtener los flujos de datos. Actualmente los siguientes receptores están disponibles:

- Sistemas de ficheros.
- Kafka.
- Conexiones TCP.
- *Rate*.

Structured Streaming también permite el uso de ventanas temporales, especificando una duración y un desplazamiento de ventana. En este caso, los datos tendrán asociado un campo de *timestamp* que indican el momento en el que fueron generados y la tabla de salida tendrá un campo adicional que indica la ventana asociada a cada resultado. En la Figura 3.11 se puede ver un ejemplo que consiste en un *WordCount*. En este ejemplo se utiliza una ventana de 10 minutos y un desplazamiento de 5 minutos. La tabla de salida se actualiza cada 5 minutos, y solo se actualizan los resultados de las ventanas correspondientes (por ejemplo, los datos que llegan en los minutos 12:05-12:10 actualiza las ventanas 12:00-12:10 y 12:05-12:15).

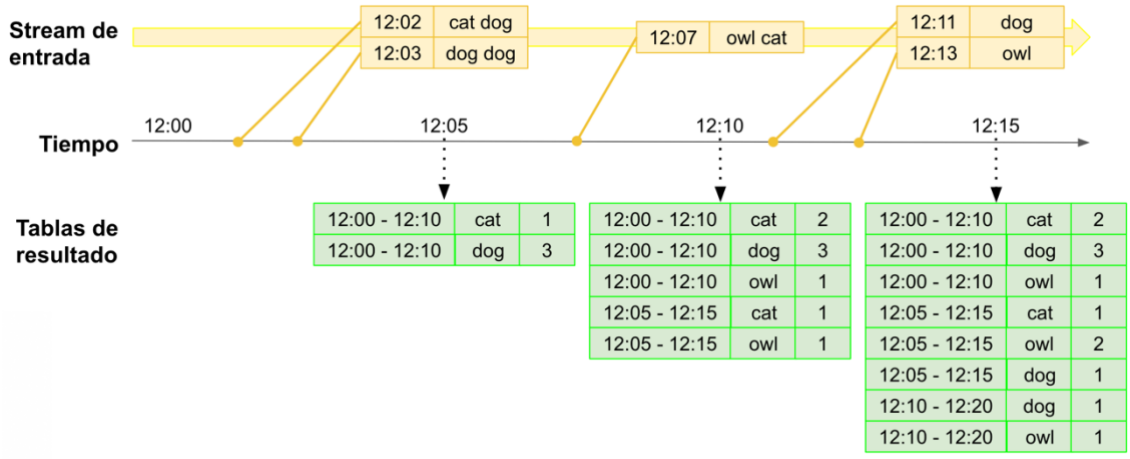

Figura 3.11. Ventanas temporales en Spark Structured Streaming

Para permitir que los datos tengan un cierto retardo, se puede fijar un límite de retardo denominado ***Watermark***. Un dato puede tener retardo por dos motivos: llega tarde al sistema; o *Structured Streaming* está procesando cuando llega el dato y este ha permanecido en memoria durante un tiempo. Por ejemplo, el dato de color rojo en la Figura 3.12 es un dato que llega en el minuto 12:13 pero según su *timestamp* se ha generado a las 12:04 (el retardo del dato es de 12:13 – 12:04 = 00:09). Si no se utiliza *watermark*, el dato es descartado porque ha llegado con retardo. Por el contrario, si se fija un *watermark* de 10 minutos o más, este dato es aceptado porque su retardo es menor que el *watermark* y se procesa en la ventana correspondiente (12:00-12:10).

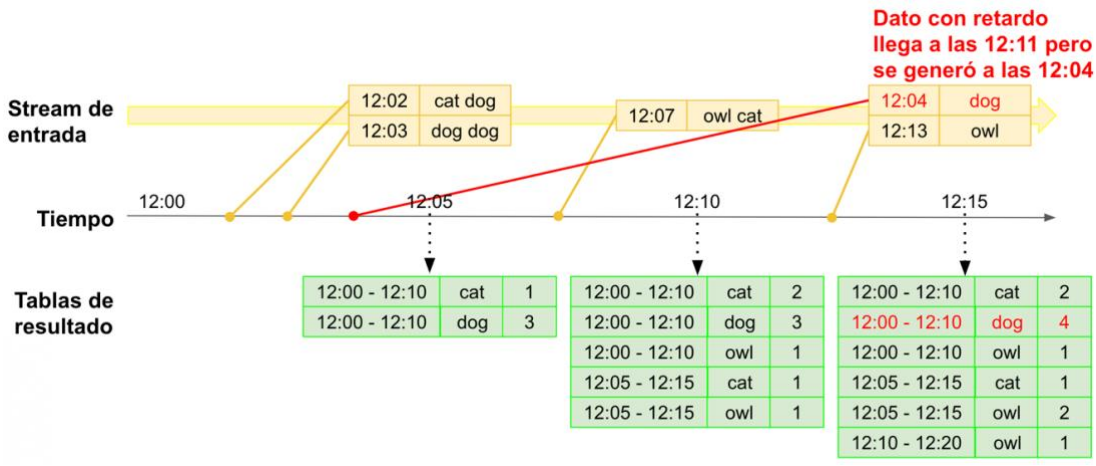

Figura 3.12. Concepto de watermark

Las consultas en *Structured Streaming* procesan los datos en *micro-batches* de hasta 100 milisegundos, aunque a partir de la versión 2.3 de Spark se ha introducido un nuevo modo denominado **procesamiento continuo**, que permite obtener latencias por debajo de 1 milisegundo con garantías de procesamiento de al menos una vez (puede procesar algunos datos varias veces). Sin embargo, este es un modo experimental que aun está en desarrollo.

3.2.3. Apache Storm

Apache Storm es un sistema de computación distribuida en tiempo real que procesa los datos de entrada dato a dato. Un *stream* en Storm es una secuencia ilimitada de datos, en la que cada dato es una **tupla** que contiene un conjunto ordenado e inmutable de pares clave-valor. Estos datos pasan por una serie de nodos que realizan tareas de procesamiento sobre ellos (en la Figura 3.13 se puede ver un ejemplo). Como se puede observar en este ejemplo, en Storm hay dos tipos de nodos [22]:

- ***Spout***: Regulan la entrada de datos al sistema. Este nodo lee datos de una fuente externa y los almacena en una cola, para introducirlos al sistema a la velocidad deseada. Un nodo puede generar más de un *stream*.
- ***Bolt***: Recibe un *stream* de entrada, realiza operaciones/transformaciones sobre los datos de dicho *stream* y genera un *stream* de salida. Opcionalmente, puede generar tuplas adicionales. Lo recomendable es

que cada nodo *Bolt* realice una única tarea (si se necesita realizar varios cálculos, lo mejor es realizar cada cálculo en un *Bolt* diferente).

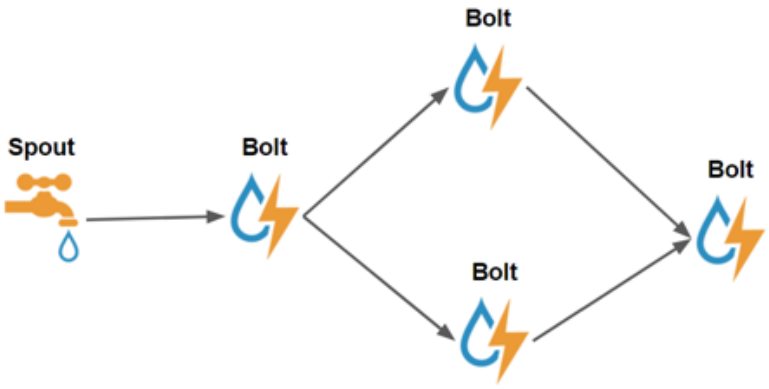

Figura 3.13. Esquema general de apache Storm

En esta plataforma los nodos se ejecutan como tareas, permitiendo que la función de cada nodo se pueda ejecutar en paralelo (varias máquinas ejecutando la tarea de un nodo). Esto es útil en situaciones en las que existen varios nodos con tareas diferentes y alguno de ellos requiere más tiempo para realizar los cálculos (debido al coste computacional).

Un inconveniente que tiene Storm con respecto al resto de plataformas de procesamiento es que hay que implementar todas las funcionalidades de los nodos a bajo nivel. Esto se debe a que no cuenta con funciones de alto nivel, como *map*, *filter*, *reduce*, etc. Con respecto a la conexión con sistemas externos, en la documentación de Storm se ofrecen una serie de guías para facilitar su integración con estos sistemas, como por ejemplo Kafka, HBase, HDFS, Hive, Cassandra, Redis, etc.

Con respecto a las librerías de *machine learning*, Storm solo cuenta con una denominada **SAMOA**, pero actualmente esta librería está en desarrollo.

Storm permite el uso de ventanas basadas en secuencia y ventanas basadas en tiempo. En ambos casos se debe especificar el tamaño o duración de la ventana y el desplazamiento. Las ventanas basadas en tiempo funcionan igual que en Spark *Structured Streaming* (permite el uso de *watermarks*), pero las ventanas basadas en secuencia funcionan de manera diferente. En lugar de especificar los valores en unidad de tiempo hay que especificarlos en número de instancias. Por ejemplo, una ventana de tamaño 10 y un desplazamiento de 5 significa que hasta que no se acumulen 10 instancias en la ventana no se realiza procesamiento. Tras realizar cada

procesamiento, la ventana se desplaza 5 instancias (mantiene las 5 más recientes) y vuelve a acumular datos hasta tener 10.

Storm tiene dos modos de funcionamiento: modo normal (los datos se procesan dato a dato) y otro modo que usa un nivel de abstracción superior denominado ***Trident*** (los datos se procesan por *micro-batches*). El segundo modo ofrece garantías de procesamiento exactamente una vez, aunque la latencia incrementa debido a que el sistema tiene que esperar a recolectar un *micro-batch*.

3.2.4. Apache Flink

Apache Flink es una plataforma de procesamiento para aplicaciones de flujos de datos que permite que los datos sean procesados dato a dato o por *micro-batches*. Además, también permite realizar procesamiento en modo tradicional, con *datasets* estáticos.

En esta plataforma las aplicaciones se crean mediante un grafo acíclico dirigido (la Figura 3.14 muestra un ejemplo), en el que los vértices son los nodos y las aristas que unen estos vértices son *streams*. Al igual que en Storm, en Flink los nodos se ejecutan como tareas, permitiendo que se puedan paralelizar entre diferentes máquinas para distribuir la carga de trabajo. Cada nodo puede tomar uno o más *streams* como entrada y generar uno o más *streams* de salida. En Flink hay tres tipos de nodos:

- **Fuente**: Regula la entrada de datos al sistema, de forma similar a los nodos *Spout* de Storm. Actualmente Flink tiene soporte para los siguientes sistemas: Kafka, Kinesis, RabbitMQ, Nifi, ActiveMQ y Netty.
- **Operador**: Realizan transformaciones sobre los datos de entrada y generan datos de salida, de forma similar a los nodos *Bolt* de Storm. Las transformaciones soportadas actualmente se muestran en la Tabla 3.6.
- **Sumidero (*Sink*)**: Envía los resultados a un sistema destino. Actualmente Flink tiene soporte para los siguientes sistemas: Kafka, Cassandra, Kinesis, Elasticsearch, HDFS, RabbitMQ, Nifi, Twitter, ActiveMQ, Flume, Redis y Akka.

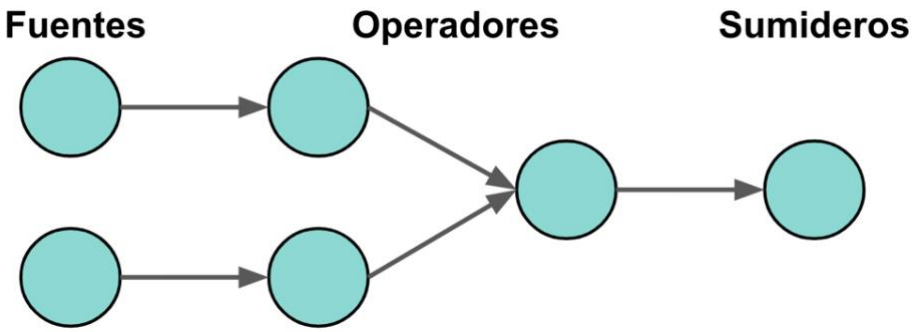

Figura 3.14. Esquema general de apache Flink

Transformaciones		
map	*fold*	*project*
flatMap	*sum*	*window*
filter	*min/max*	*windowAll*
keyBy	*minBy/maxBy*	*windowApply*
reduce	*intervalJoin*	*windowReduce*
union	*coMap*	*windowFold*
connect	*coFlatMap*	*windowJoin*
split	*select*	*windowCoGroup*

Tabla 3.6. Transformaciones sobre un stream de entrada en Flink

En Flink los datos que viajan por el *stream* se denominan **tuplas**, que son un conjunto de pares clave-valor. La estructura de datos que se utiliza para abstraer el *stream* se denomina ***DataStream***. En Flink un *stream* puede tener varios estados y, dependiendo de las transformaciones realizadas sobre el *stream*, se puede pasar a un estado u otro (como muestra la Figura 3.15). Los diferentes estados que puede tener el *stream* son [23]: ***DataStream***, ***KeyedStream***, ***WindowedStream***, ***AllWindowedStream***, ***ConnectedStream***, ***SplitStream*** y ***IterativeStream***.

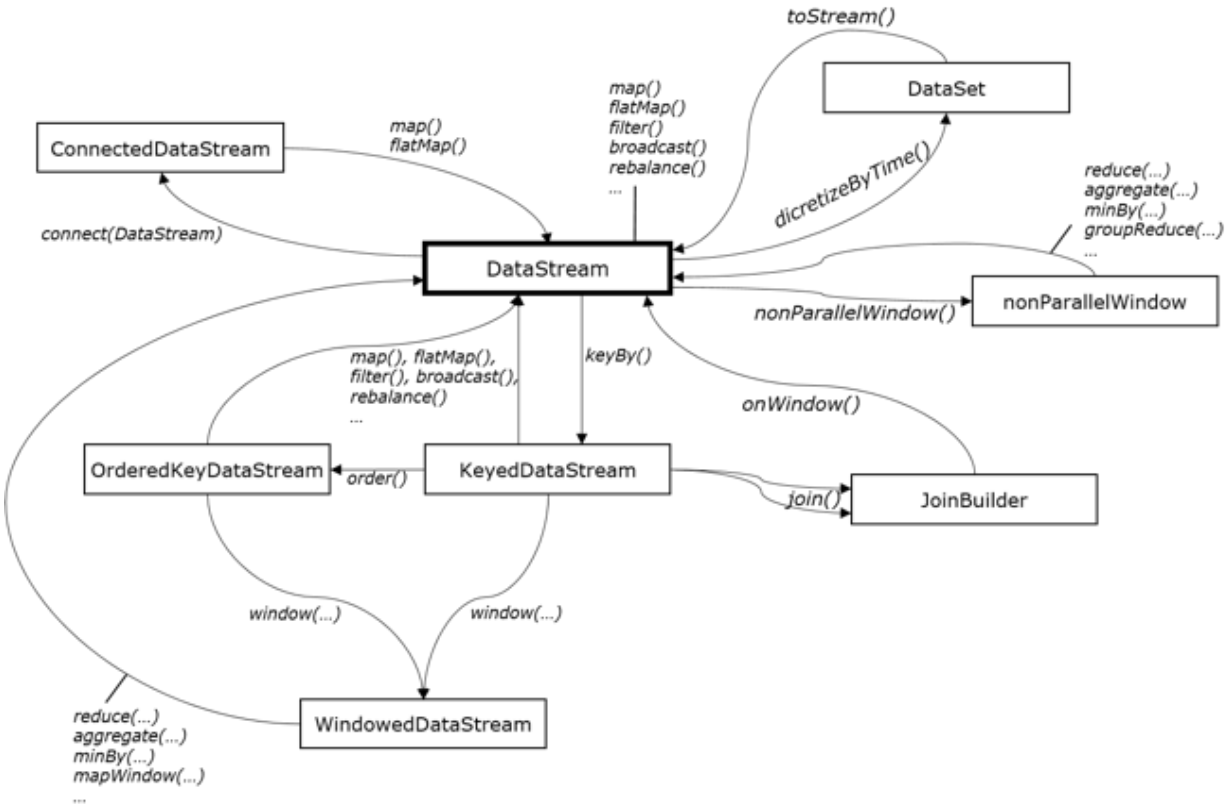

Figura 3.15. Estados de un stream en Flink

Fuente:
https://cwiki.apache.org/confluence/display/FLINK/Streams+and+Operations+on+Streams

Con respecto a las librerías de *machine learning*, Flink cuenta con su propia librería, denominada **FlinkML**, **SAMOA** y **AmidstToolbox**. Aunque hay diferentes algoritmos implementados en FlinkML, todos son para *machine learning* en modo tradicional.

Uno de los puntos fuertes de Flink es el uso de ventanas, ya que permite utilizar ventanas basadas en tiempo y ventanas basadas en secuencia. Además, tiene algunos tipos de ventanas implementadas y ofrece facilidades para crear ventanas personalizadas. Dentro de una misma ventana también se puede distinguir entre eventos de diferentes tipos. Junto con las ventanas basadas en tiempo también se pueden utilizar *watermarks* para permitir que los datos lleguen con un cierto retardo.

3.2.5. Apache Samza

Apache Samza es una plataforma de procesamiento distribuido en tiempo real que utiliza Kafka y **YARN** [24], que es una versión mejorada de apache **Hadoop**. Ofrece una API fácil de usar que permite realizar un procesamiento en modo tradicional o en modo *streaming*.

Un *stream* es una secuencia ilimitada de datos que puede tener múltiples productores escribiendo datos en él y múltiples consumidores que leen datos de él. Cada *stream* se puede dividir en múltiples particiones, por lo que el número de consumidores puede ser ajustado al procesamiento para escalar como estos datos son procesados. Cada partición es una secuencia ordenada e inmutable de datos, por lo que los nuevos datos se añaden al final de la partición.

Para facilitar la obtención de datos de fuentes externas, Samza tiene algunos conectores implementados. Actualmente los sistemas incluidos son [25]:

- Kafka
- Microsoft Azure Eventhubs
- Kineses
- HDFS
- Elasticsearch

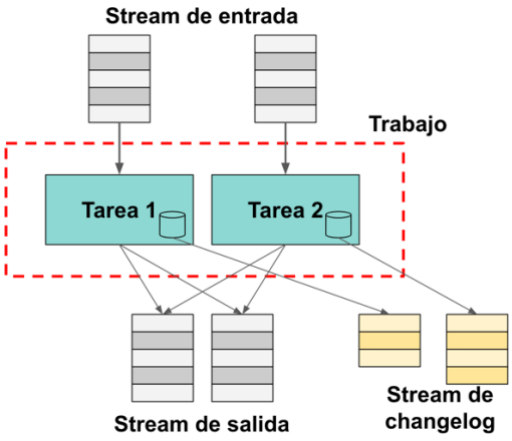

Figura 3.16. Esquema general de apache Samza

Una aplicación en Samza es un conjunto de transformaciones (**trabajos**) que se aplican sobre un *stream* de datos de entrada y que genera un *stream* de datos de salida. Samza escala la aplicación dividiéndola en múltiples tareas, que realizan

operaciones en el *stream* de datos. Cada tarea consume los datos de una partición del *stream* de entrada. Cada aplicación cuenta con un coordinador que gestiona la asignación de tareas entre las diferentes máquinas. En la Figura 3.16 se puede observar un ejemplo con un trabajo que contiene dos tareas.

Con respecto a las librerías de *machine learning*, solo tiene compatibilidad con **SAMOA** pero, como se ha comentado con anterioridad, aun está en desarrollo.

Esta plataforma proporciona tolerancia a fallos mediante la utilización de un ***checkpoint* incremental**, que almacena el estado de cada tarea con el objetivo de poder recuperar el sistema en caso de que se produzca algún fallo o un nodo caiga.

Samza permite realizar un procesamiento con estado o sin estado. Para poder realizar un procesamiento con estado, cada tarea tiene un almacén local en el que se almacena el estado de las particiones procesadas por la tarea. Además, se ofrece la posibilidad de almacenar el estado de forma remota.

3.2.6. Apache Apex

Apache Apex es una plataforma de procesamiento nativa de YARN que permite procesar los datos en modo tradicional o en *streaming*. Además, proporciona una API simple, que permite a los desarrolladores escribir código genérico y reutilizable. El código se incluye en la aplicación y la plataforma automáticamente gestiona diversos asuntos operativos, como la gestión de estados, tolerancia a fallos, escalabilidad, seguridad, métricas, etc.

El núcleo de Apex se complementa con **Malhar**, una librería que contiene funciones lógicas y conectores. Esta librería proporciona acceso a diferentes sistemas de ficheros (como HDFS, S3, NFS, FTP), sistemas de mensajes (como Kafka, ActiveMQ, RabbitMQ, JMS) y bases de datos (como MySql, Cassandra, MongoDB, Redis HBase, CouchDB, generic JDBC). Además, la librería contiene algunos códigos de prueba que muestran las capacidades y características de los operadores.

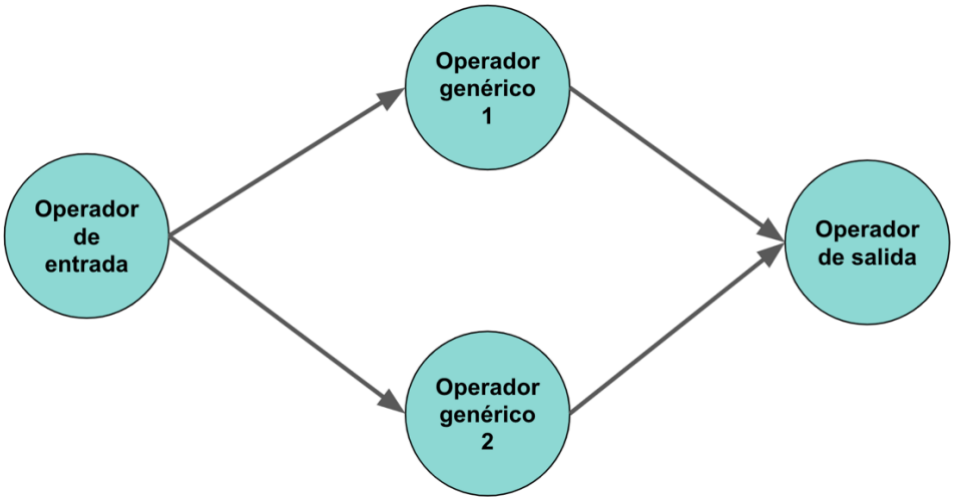

Figura 3.17. Esquema general de apache Apex

Una aplicación en Apex se representa como un grafo acíclico dirigido en el que los vértices son los nodos y las aristas que unen los nodos son *streams* de datos (en la Figura 3.17 se muestra un ejemplo). Los datos que viajan por el *stream* se denominan **tuplas**, que tienen un formato específico. Por otro lado, los nodos se denominan **Operadores**, que toman uno o más *streams* de entrada, realizan un procesamiento y generan cero, uno o más *streams* de salida. En Apex hay tres tipos de nodos que tienen una funcionalidad similar a los nodos de las plataformas anteriores [26]: **operador de entrada**, **operador genérico** y **operador de salida**. Cada nodo puede tener una serie de puertos de entrada (puerto utilizado para recibir un *stream*) y puertos de salida (puerto utilizado para generar un *stream*). El *stream* consiste en tuplas que viajan de un puerto de salida a un puerto de entrada. La plataforma garantiza que el envío de datos entre nodos es ordenado.

Apex permite que los datos se procesen dato a dato, aunque también permite procesar los datos por *micro-batches* utilizando ventanas temporales. Además, proporciona tolerancia a fallos mediante el uso de ***checkpoints***, que almacena el estado del sistema para la recuperación. Cuando un nodo cae su recuperación es instantánea gracias al uso de un **mecanismo de *Heartbeat***, que periódicamente comprueba el estado de los nodos e intenta levantarlo en caso de detectar un nodo caído.

3.2.7. Apache Beam

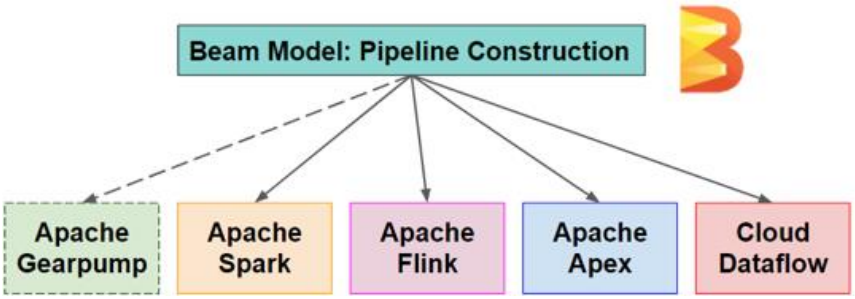

Figura 3.18. Runners disponibles en apache Beam

Apache Beam es una plataforma de procesamiento en tiempo real que permite a los desarrolladores definir un procesamiento independiente del sistema de procesamiento de datos (denominado ***runner***) que se utilice. Beam permite a los usuarios cambiar el *runner* utilizando el mismo código para el procesamiento implementado. Los siguientes *runners* están disponibles [27]: Apache Apex, Apache Flink, Apache Spark, Apache Gearpump y Google Cloud Dataflow. La Figura 3.18 muestra un esquema general de Beam. Al poder elegir entre diferentes *runners*, algunas características del sistema como la latencia y las garantías de procesamiento varían según el *runner* utilizado.

En esta plataforma las aplicaciones se implementan mediante una abstracción denominada ***Pipeline***, que contiene todas las tareas de procesamiento de datos, desde el principio hasta el final. Los datos en Beam se abstraen en una estructura denominada ***PCollection***, que puede ser un conjunto de datos finito (procede de una fuente estática, como un archivo) o ilimitado (procede de una fuente de datos continua). Las diferentes transformaciones que pueden incluirse en un *pipeline* se denominan ***PTransform***, que reciben uno o más *PCollections*, realiza un procesamiento y genera cero o más *PCollections* de salida.

3.2.8. Resumen de las características de las plataformas de procesamiento

En esta sección se han analizado siete plataformas para el procesamiento de datos en tiempo real presentando sus principales características y diferencias, que están resumidas en la Tabla 3.7. Todas las plataformas permiten gestionar el sistema de procesamiento a través de una utilidad web de monitorización, proporcionan

tolerancia a fallos y son multiplataforma. Con respecto a las garantías de procesamiento, Spark, Flink, Apex y Storm (con *Trident*) ofrecen garantía de procesamiento exactamente una vez, que es lo ideal para asegurar que el procesamiento no tiene pérdidas ni duplicados. Storm (sin *Trident*) y Samza ofrecen garantía de procesamiento al menos una vez. Estas plataformas tienen soporte para programar en diferentes lenguajes, excepto Samza y Apex, que solo cuentan con soporte para Java. De entre todas las plataformas analizadas, algunas pertenecen al conjunto de plataformas que utilizan un modelo con *micro-batches* (Spark, Storm con *Trident* y Flink) y otras que utilizan un modelo dato a dato (Storm sin *Trident*, Flink, Samza, Apex y Beam). Cada plataforma utiliza su propia estructura para abstraer el *stream* de datos y los datos. Todas soportan el uso de ventanas temporales, pero solo Storm y Flink soportan el uso de ventanas basadas en secuencia. Con respecto al procesamiento con estado o sin estado, todas ofrecen la posibilidad de elegir el tipo de procesamiento deseado, excepto Spark *Structured Streaming* (solo procesamiento con estado) y Storm (solo procesamiento sin estado). Finalmente, otro factor que puede ser muy importante al elegir una plataforma para *machine learning* es la existencia de librerías que tengan algoritmos implementados y/o que faciliten el desarrollo. En este aspecto, Spark *Streaming* y Flink son las plataformas de procesamiento de datos de este trabajo con más algoritmos implementados, pero Spark *Streaming* tiene más que Flink tanto para el modo tradicional como el *streaming*.

Para la etapa de procesamiento de datos, Spark es la más utilizada por la comunidad investigadora, debido a que permite realizar un procesamiento con estado o sin estado con garantías de procesamiento exactamente una vez. Además, tiene una gran cantidad de documentación, cuenta con una librería de *machine learning* propia (MLLIB) y tiene otros componentes (como SparkSQL) que se pueden integrar en las aplicaciones, lo que facilita el desarrollo de aplicaciones tanto para un entorno tradicional como para *streaming*.

Características	Plataformas						
	Spark Streaming	Spark Structured Streaming	Storm	Flink	Samza	Apex	Beam
Versión analizada	2.4.0 (11/2018)	2.4.0 (11/2018)	1.2.2 (06/2018)	1.7.1 (12/2018)	1.0.0 (11/2018)	3.7.0 (04/2018)	2.9.0 (12/2018)
Garantías de procesamiento	Exactamente una vez	Exactamente una vez	Al menos una vez Exactamente una vez	Exactamente una vez	Al menos una vez	Exactamente una vez	Depende
Tolerancia a fallos	Si	Si	Si	Si	Si	Si	Si
Lenguajes de programación	Java Scala Python R	Java Scala Python R	Java Scala Python Ruby	Java Scala Python	Java	Java	Java Scala Python
Utilidad de monitorización	Si	Si	Si	Si	Si	Si	Si
Sistemas operativos	Windows Linux MacOS	Windows Linux MacOS	Windows Linux	Windows Linux MacOS	Windows Linux	Windows Linux MacOS	Windows Linux MacOS
Documentación	Muchísima	Mucha	Mucha	Mucha	Poca	Poca	Mucha
Modificaciones durante la ejecución	No	No	Si	No	No	Si	No
Unidad de procesamiento	Micro-batch	Micro-batch	Dato a dato Micro-batch	Dato a dato Micro-batch	Dato a dato	Dato a dato	Dato a dato
Abstracción de datos	DStream	Dataset DataFrame	Tuple	DataStream	Message	Tuple	PCollection
Ventanas	Temporales	Temporales	Temporales Secuenciales	Temporales Secuenciales	Temporales	Temporales	Temporales
Tipo de procesamiento	Sin estado Con estado	Con estado	Sin estado	Sin estado Con estado	Sin estado Con estado	Sin estado Con estado	Sin estado Con estado
Librerías de machine learning	MLLIB streamDM Amidst-Toolbox	No	SAMOA	FlinkML SAMOA Amidst-Toolbox	SAMOA	No	No

Tabla 3.7. Características de las plataformas de procesamiento

3.3. Librerías de machine learning

Uno de los factores más importantes cuando se va a elegir una plataforma para procesamiento es la existencia de librerías de *machine learning*, que incluyen algoritmos implementados y métodos que facilitan el uso de la plataforma y el desarrollo de aplicaciones. En este trabajo se han analizado las siguientes librerías:

- **MLLIB** (MLL) [28] es la librería de *machine learning* de apache Spark que está incluida en el proyecto Spark, por lo que es testeado y actualizado con cada nueva versión. Además, contiene muchos tipos de algoritmos y funciones, como cálculo de estadísticos y normalización, útiles agilizar el desarrollo de algoritmos de *machine learning*.
- **StreamDM** (SDM) [29] es una librería de código abierto desarrollada por de Huawei Noah's Ark para realizar minería de datos en *streaming* utilizando apache Spark *Streaming*. Esta librería contiene varios algoritmos de *streaming* implementados y algunos generadores de datos para crear los *streams* de entrada y probar los algoritmos.
- **FlinkML** (FML) [23] es la librería de *machine learning* de apache Flink, que proporciona algoritmos de *machine learning* escalables, una API intuitiva y herramientas que ayudan a minimizar el uso de estos algoritmos.
- **SAMOA** (SAM) [30] es un *framework* distribuido para *machine learning* en *streaming* que contiene una abstracción de programación para algoritmos de *machine learning* en *streaming*. La idea de este *framework* es implementar algoritmos de *machine learning* en *streaming* sin tener que saber que plataforma de procesamiento va a ejecutarlos. Actualmente está en estado de incubación.
- **Amidst Toolbox** (AMT) [31] es una librería probabilística y escalable de *machine learning* desarrollada en Java, con un enfoque especial en análisis de datos en *streaming*. Esta librería permite especificar modelos que pueden ser aprendidos utilizando un algoritmo Bayesiano, tanto en un contexto tradicional como en *streaming*. Es una librería que puede utilizarse en Spark y Flink, pero el bloque de Spark, denominado Spark's link, está en desarrollo.

En la Tabla 3.8 se incluyen los algoritmos disponibles para cada una de las librerías de *machine learning* analizadas en este trabajo, tanto para procesamiento tradicional como en *streaming*. En total hay 20 algoritmos en MLLIB, 9 en streamDM, 6 en FlinkML, 6 en SAMOA y 14 en Amidst Toolbox. Como se puede observar, actualmente no hay implementados muchos algoritmos de *machine learning* en *streaming*.

Algoritmos	MLL	SDM	FML	SAM	AMT
SVM	x	x	x		
Logistic Regression	x	x			
Linear Regression	x	x	x		
Streaming Logistic Regression	x				
Streaming Linear Regression	x				
DecisionTree	x				
RandomForest	x				
Gradient Boosted Trees	x				
NaiveBayes	x	x			x
Isotonic Regression	x				
ALS	x		x		
K-means	x				
Gaussian Mixture	x				x
PIC	x				
LDA	x				
Bisecting K-means	x				
Streaming K-means	x				
FPGrowth	x				
Association Rules	x				
PrefixSpan	x				
Perceptron		x			
CluStream		x			
Hoeffding Decision Trees		x		x	
OnlineBagging		x		x	
StreamKM++		x			
SGD			x		
k-NN			x		
SOS			x		
AMRules				x	
OnlineBoosting				x	
Adaptive Bagging				x	
Distributed Stream Clustering				x	
TAN					x
AODE/HODE					x
Dynamic NB					x
Gaussian Discriminant Analysis					x
LCM					x
Bayesian Linear Regression					x
FactorAnalysis					x
DynamicLCM					x
HMM					x
KF					x
SwitchingKF					x

Tabla 3.8. Algoritmos de machine learning disponibles en librerías

4. COMPARATIVAS DE PROCESAMIENTO REALIZADAS

En este apartado se han realizado dos comparativas propias que comparan las diferentes plataformas de procesamiento de datos analizadas en la subsección 3.2. Con este motivo, se realizan dos comparativas para medir el rendimiento de las plataformas en dos problemas concretos. En la primera se utiliza una comparativa que se ha utilizado en otros estudios [32] [33] [34] [35] conocida como ***Yahoo Streaming Benchmark***, que evalúa la **latencia** y el **rendimiento** de las plataformas. La latencia es el tiempo que una instancia permanece en el sistema desde que llegó hasta que es procesada. Por otro lado, el rendimiento mide el número de instancias que se procesan por unidad de tiempo (normalmente segundos). La segunda comparativa mide el rendimiento de Spark *Streaming* y Flink en un problema de *machine learning* en *streaming*, en el que se ejecuta un algoritmo de *clustering* denominado ***streaming k-means*** y se mide el tiempo que tarda el algoritmo en procesar diferentes *datasets*.

Para poder realizar cada comparativa, ha sido necesario elaborar y poner en marcha dos configuraciones de clúster (una para cada comparativa), en los que se han instalado y configurado todas las plataformas de procesamiento utilizadas en cada comparativa.

4.1. Comparativa I: Yahoo Streaming Benchmark

En esta subsección se va ha detallar la primera comparativa realizada en la que se utiliza *Yahoo Streaming Benchmark* para comparar las plataformas analizadas en la subsección 3.2. En primer lugar, se describe como funciona esta metodología junto con otros estudios que se han realizado previamente en los que se ha utilizado *Yahoo Streaming Benchmark*. En las siguientes subsecciones se describe el clúster utilizado, detallando la arquitectura de nodos utilizados, como se ha realizado su configuración e instalación y la puesta en marcha del sistema para ejecutar las experimentaciones. Finalmente, en la última subsección se explican las pruebas realizadas y se analizan los resultados obtenidos.

4.1.1. Introducción

Yahoo Streaming Benchmark [32] fue desarrollado en 2016 por *Yahoo Inc* para comparar el rendimiento de apache Storm frente a otras plataformas de procesamiento. Esta comparativa simula una simple aplicación de mensajería que tiene 100 campañas, 10 anuncios por campaña y utiliza una ventana temporal de 1 segundo. Los anuncios llegan al sistema y el objetivo es contar, para cada campaña, el número de eventos relevantes dentro de cada ventana temporal. Mientras se realiza este proceso, se mide la latencia de los datos de entrada para diferentes velocidades de llegada. Cada anuncio (dato de entrada) tiene los siguientes atributos:

- ***user_id***: Identificador de usuario.
- ***page_id***: Página desde la que se accede al anuncio.
- ***ad_id***: Identificador del anuncio.
- ***ad_type***: Tipo de anuncio. Los valores que puede tomar este atributo son: *banner*, *modal*, *sponsored-search*, *mail* o *mobile*.
- ***event_type***: Tipo de evento. Los valores que puede tomar este atributo son: *view*, *click* o *purchase*.
- ***event_time***: *Timestamp* en el que se generó el evento.
- ***ip_address***: Dirección IP desde la que se accede al anuncio.

El flujo de operaciones que se realizan en *Yahoo Streaming Benchmark* es el que se muestra en la Figura 4.1, en la que se pueden diferenciar las siguientes operaciones:

1. **Leer eventos de Kafka**: Se obtienen los eventos del *stream* de entrada.
2. **Extraer la información del JSON**: Los datos llegan al sistema en formato JSON, por lo que en este paso los datos se convierten a objetos que facilitan su procesamiento, extrayendo la información de los atributos.
3. **Filtrar eventos irrelevantes**: Durante la ejecución llegan anuncios al sistema con diferente valor en el atributo *event_type*. Esta operación filtra los eventos que llegan por el *stream* de entrada y solo deja pasar aquellos que tienen el valor *view* en este atributo.

4. **Transformar/Proyectar atributos**: Cada dato contiene siete atributos. En este paso se proyectan aquellos atributos que son relevantes para el procesamiento. Estos atributos son *ad_id* y *event_time*.
5. **Unión**: En Redis hay una tabla que contiene el *campaign_id* asociado a cada *ad_id*. En esta operación se obtienen los datos de esta tabla y se le añade a cada dato el *campaign_id* que le corresponde.
6. **Agregación en ventana temporal**: En esta comparativa se utiliza una ventana temporal. En este paso se cuenta el número de *campaigns* dentro de cada ventana.
7. **Almacenar resultados**: Finalmente, los resultados obtenidos en el paso anterior se almacenan en Redis.

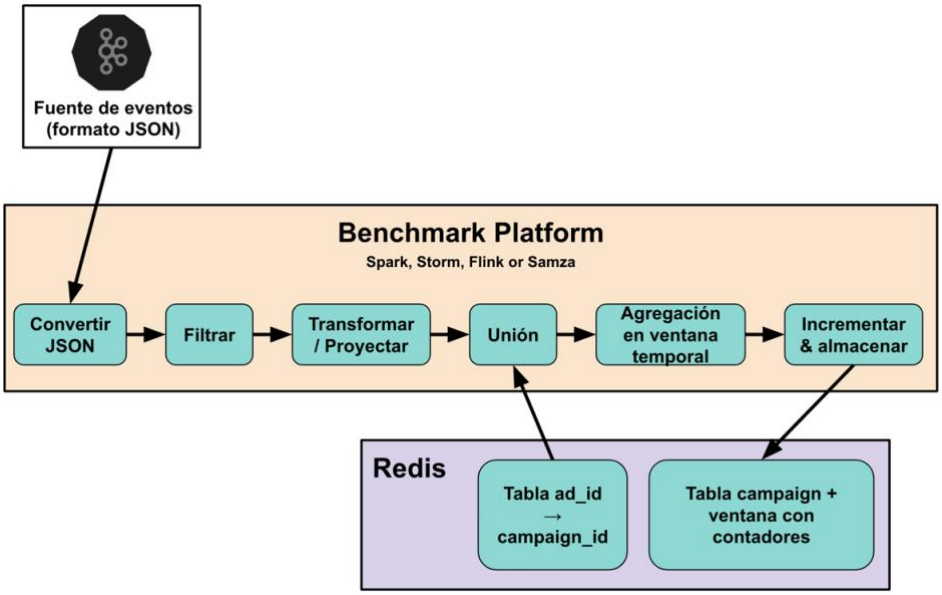

Figura 4.1. Esquema general de Yahoo Streaming Benchmark

En esta prueba hay varios productores generando datos hacia Kafka a una velocidad (eventos por segundo) determinada. Estos datos pasan por el sistema anterior y se realizan las operaciones que lo forman. Como se ha comentado antes, se utiliza una ventana de 1 segundo para contar el número de eventos que pertenece a cada campaña. Por este motivo, cada segundo se realiza este conteo y los resultados se almacenan en **Redis**, que se utiliza como base de datos. Esta prueba se ejecuta durante un tiempo determinado y, tras ese tiempo, los resultados de Redis se almacenan en ficheros y se vacía el contenido de Redis, para poder realizar otra prueba. Para obtener la latencia de cada ventana se utiliza la siguiente fórmula:

Latencia = (Última_actualización - Inicio) – Duración (*Fórmula 4.1)*

donde **Ultima_actualización** es la hora en la que se realiza la última actualización de la ventana, **Inicio** es la hora de inicio de la ventana y **Duración** es el tamaño de la ventana.

Como se ha comentado al inicio de esta subsección, el primer estudio realizado con *Yahoo Streaming Benchmark* fue [32]. En este estudio se comparan las siguientes plataformas: Spark *Streaming*, Storm y Flink (las versiones se pueden ver en la Tabla 4.1). Para cada plataforma, se realizan 10 ejecuciones generando los datos a diferentes velocidades (entre 50.000 y 170.000 eventos por segundo). Cada ejecución dura 30 minutos y durante una misma ejecución la cantidad de eventos por segundo se mantiene fija. Al finalizar la prueba, se aumenta la velocidad de generación de datos y se repite otra ejecución. Este proceso se repite hasta llegar a 10 ejecuciones. Cuando se realizan las 10 ejecuciones de una plataforma, se repiten las mismas 10 ejecuciones con otra plataforma. Para realizar las pruebas han utilizado una configuración homogénea de los nodos del clúster. Cada nodo tiene 2 procesadores Intel E5530 2.4GHz, 16 núcleos y 24GB de memoria RAM. En total se utilizan 30 nodos: 11 nodos de procesamiento (10 esclavos y 1 maestro); 5 nodos que funcionan como almacenes Kafka; 1 nodo Redis; 3 nodos Zookeeper; y 10 nodos productores de datos.

Plataforma	Comparativa [32]	Comparativa [34]	Comparativa propia
Spark Streaming	1.5.1	2.3.0	2.4.0
Spark Structured Streaming	No utilizada	2.3.0	2.4.0
Storm sin Trident	0.11.0	1.2.1	1.2.2
Storm con Trident	No utilizada	1.2.1	No utilizada
Flink	0.10.1	1.5.0	1.7.1
Kafka Streams	No utilizada	1.1.0	No utilizada
Samza	No utilizada	No utilizada	1.0.0

Tabla 4.1. Plataformas analizadas en las comparativas

En 2018, se realizó un proyecto fin de máster [34] en el que también se utiliza esta metodología. En este caso, utiliza las mismas plataformas que en el estudio anterior (con las versiones más recientes) y añade Spark *Structured Streaming*, Storm con *Trident* y *Kafka Streams* (las versiones se pueden ver en la Tabla 4.1). Para cada plataforma, se realizan 15 ejecuciones generando los datos a diferentes velocidades (entre 10.000 y 150.000 eventos por segundo). Cada ejecución dura 10 minutos y durante una misma ejecución la cantidad de eventos por segundo se mantiene fija. Al

finalizar la prueba, se aumenta la velocidad de generación de datos y se repite otra ejecución. Este proceso se repite hasta llegar a 15 ejecuciones. Cuando se realizan las 15 ejecuciones de una plataforma, se repiten las mismas 15 ejecuciones con otra plataforma. Para realizar las pruebas han utilizado nodos con diferentes configuraciones según su función. En total utilizan 29 nodos: 10 nodos de procesamiento (9 esclavos y 1 maestro, cada uno con 16 núcleos y 32GB de memoria RAM); 5 nodos que funcionan como almacenes Kafka (cada uno con 16 núcleos y 32GB de memoria RAM); 1 nodo Redis (con 4 núcleos y 8GB de memoria RAM); 3 nodos Zookeeper (cada uno con 4 núcleos y 8GB de memoria RAM); y 10 nodos productores de datos (cada uno con 2 núcleos y 4GB de memoria RAM).

En este trabajo se ha utilizado la misma metodología, utilizando versiones más recientes de las plataformas (se pueden observar en la Tabla 4.1) y añadiendo una nueva, que es apache Samza. En la siguiente subsección se detalla la arquitectura del clúster, describiendo los nodos utilizados y como interactúan entre ellos durante la prueba. En esta comparativa no se ha utilizado apache Apex porque esta plataforma utiliza software de DataTorrent y la web que da soporte a esta librería ha cerrado, por lo que lleva un tiempo sin actualizarse.

4.1.2. Arquitectura del clúster

La arquitectura del clúster utilizado para realizar esta comparativa se puede observar en la Figura 4.2. Como se ha indicado en la subsección anterior, en la arquitectura del clúster se pueden distinguir cinco grupos de nodos según la función que desempeñan:

- **Productores de datos**: Generan los datos a una determinada velocidad (eventos por segundo). Estos datos se almacenan en el almacén de Kafka. Cada productor de datos se ejecuta en paralelo, por lo que al generar los datos se debe tener esto en cuenta ya que hay que indicar la velocidad a cada uno. Por ejemplo, si hay 2 productores y queremos ejecutar una prueba con 20.000 eventos por segundo, se debe especificar a cada productor que genere 10.000 eventos por segundo.

- **Almacén de Kafka**: Es donde se almacenan los mensajes que envían los productores y de donde obtienen el *stream* los nodos de procesamiento. En este caso, aunque se tengan varios nodos de Kafka solo se utiliza un almacén. En estos nodos es donde se instala Kafka.
- **Procesamiento**: Es el conjunto de nodos que se encargan de ejecutar las plataformas de procesamiento utilizando una metodología maestro-esclavo. En estos nodos es donde se instala Spark, Storm, Flink y Samza.
- **Zookeeper**: Se encargan de gestionar el estado de los demás nodos (excepto Redis), para coordinar los servicios que ofrecen. En este bloque se pueden utilizar varios nodos para replicar los datos y proporcionar tolerancia a fallos.
- **Redis**: Es el nodo que ejecuta Redis, que es una base de datos en la que se van a almacenar los resultados. En este caso se suele utilizar solo un nodo.

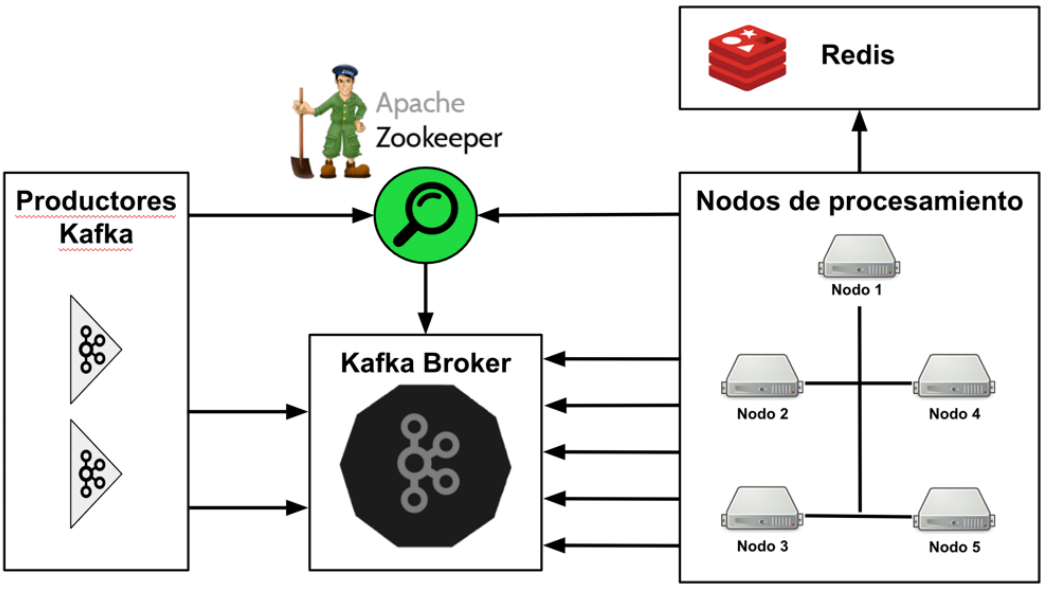

Figura 4.2. Arquitectura del clúster en la comparativa I

Para realizar esta comparativa se ha creado una arquitectura en la que una serie de productores de datos generan eventos que son enviados y almacenados en el almacén de Kafka. Una vez que los mensajes están almacenados en Kafka, los nodos de procesamiento empiezan a leer y procesar los datos en paralelo. Finalmente, los resultados de este procesamiento se almacenan en Redis. Por otro lado, Zookeeper gestiona y coordina los diferentes servicios del clúster.

En el clúster utilizado para realizar la experimentación todos los nodos tienen una configuración homogénea. Cada nodo tiene un procesador Intel Core i5 750 2.67GHz, 4 núcleos y 4GB de memoria RAM. Además, se han utilizado conexiones de 1GB/s entre los nodos. En total se han utilizado 10 nodos: 5 nodos de procesamiento (4 esclavos y 1 maestro); 1 nodo que funciona como almacén Kafka; 1 nodo Redis; 1 nodo Zookeeper; y 2 nodos productores de datos.

Una vez que se conoce cual es la arquitectura del clúster necesario para realizar la experimentación, en la siguiente subsección se va a explicar el proceso de instalación y configuración de los diferentes nodos del clúster.

4.1.3. Instalación y configuración del clúster

Todo lo necesario para realizar la instalación se proporciona junto a este proyecto. Es necesario descargar este proyecto y almacenarlo en una carpeta (se aconseja en el directorio de usuario).

Antes de crear la red privada con las máquinas que van a formar el clúster, es necesario realizar unas instalaciones previas a través de internet. Para ello, se proporciona un script denominado ***initialSetup.sh*** que contiene los comandos necesarios para estas instalaciones. Lo que se instala en el sistema es:

- **Actualizar el sistema**: Por si el sistema no está actualizado.
- **Java 8**: Es necesario utilizar Java para poder ejecutar algunos programas.
- **Maven**: Se utiliza para compilar el proyecto.
- **ssh-server**: Se utilizar para poder ejecutar comandos entre máquinas del clúster.
- **lein**: Es necesario para que los productores de Kafka puedan enviar los datos a la velocidad deseada.
- **R**: Se utiliza para generar las gráficas con los resultados finales.

Una vez descargado el proyecto y realizada la instalación inicial, ya se pueden conectar las máquinas a una misma red para que se puedan comunicar entre ellas. Para este propósito se utiliza un *switch* y 10 cables de red *Gigabit ethernet*. Para que

las máquinas tengan conexión entre sí es necesario configurar una dirección IP para cada máquina dentro de la misma red, como muestra la Figura 4.3. Para realizar el cambio de IP se han realizado los siguientes pasos por cada máquina:

- Ir a "**Editar las conexiones…**".
- Hacer clic en "**Añadir**".
- Elegir la opción "**Cableada**" y hacer clic en "**Crear…**".
- Poner un nombre a la red. Por ejemplo: "**Clúster**".
- Seleccionar la opción "**Ajustes de IPv4**".
- Asignar en "**Método**" la opción manual y asignar una dirección **10.0.0.x/24**, como muestra la Figura 4.4. A cada máquina se le asigna un valor de x diferente, del 1 al 10.
- Hacer clic en "**Guardar**" para crear la red cableada.

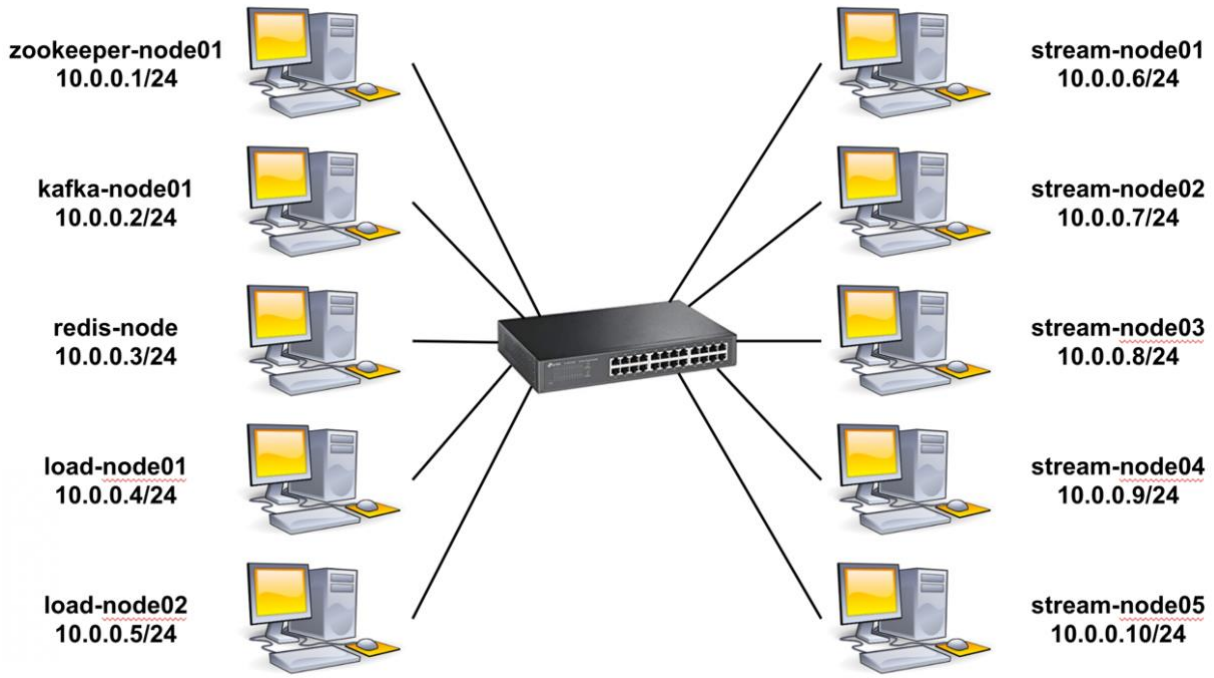

Figura 4.3. Dirección IP y nombre de cada máquina del clúster

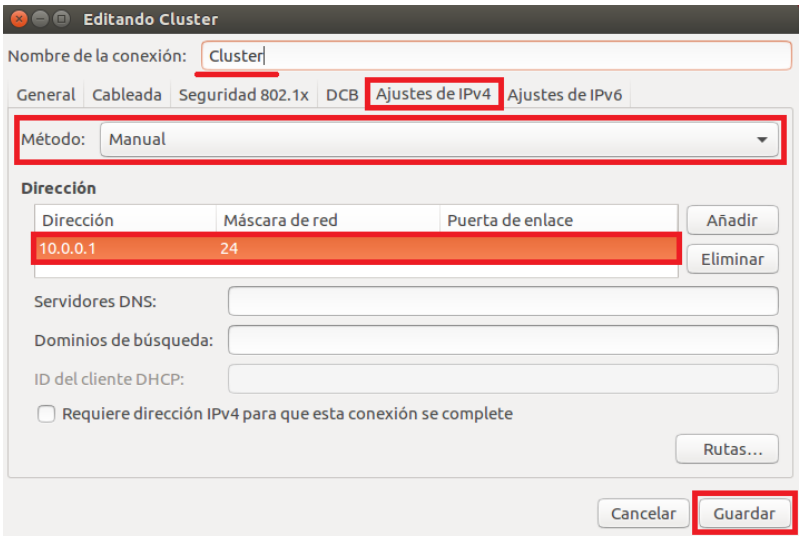

Figura 4.4. Configurar red privada en Ubuntu

Una vez creada esta red cableada en las máquinas, cada máquina debe estar conectada a ella. Para comprobar que hay conexión entre todas las máquinas se puede utilizar el comando "**ping [IP]**". Cuando todas las máquinas tengan conexión entre sí, se puede pasar al siguiente paso.

Tras establecer una red privada con las máquinas se va a cambiar el nombre de cada máquina, para que coincida con la función que va a realizar dentro del clúster. El nombre asignado a cada nodo aparece en la Figura 4.3. Para esto hay que modificar el archivo "**/etc/hostname**" y especificar en este archivo el nombre de la máquina. A continuación, es necesario reiniciar cada máquina para que utilicen el nuevo nombre.

Para facilitar la utilización de comandos, también se va a modificar el archivo "**/etc/hosts**" para asignar al nombre de cada máquina su dirección IP, como muestra la Figura 4.3. De esta forma se puede utilizar el nombre de las máquinas e internamente se hace la traducción a dirección IP. Se puede utilizar nuevamente el comando "***ping [hostname]***" para comprobar la conectividad.

Tras estos pasos para conectar las máquinas dentro de la red privada, se va a utilizar ssh para poder ejecutar comandos entre máquinas de forma remota. Para ello se ejecuta en cada máquina el comando "**ssh-keygen**" para generar las claves y luego, en cada máquina, ejecutar "**ssh-copy-id [hostname]**" para almacenar en cada máquina la clave del resto de máquinas. De esta forma se puede realizar una conexión desde una máquina a cualquier otra sin que se solicite la clave.

A continuación, se va a explicar como instalar los diferentes nodos según su función dentro del clúster. Para ello se va a utilizar la información incluida en la carpeta "**install_nodes**" del proyecto. En primer lugar, es necesario ejecutar el script denominado **send.sh** para enviar a cada nodo el comprimido que necesita. Una vez enviados, es necesario ir a cada máquina y descomprimir el archivo comprimido, generando otro archivo comprimido y un archivo denominado **install.sh**, para facilitar la instalación. Solo es necesario modificar este archivo en los "**stream nodes**", ya que contiene parámetros de memoria y núcleos que varían según las máquinas que se utilizan en el clúster.

En apache Spark hay que modificar los siguientes parámetros:

- SPARK_DRIVER_MEMORY=[MEMORIA]
- SPARK_EXECUTOR_CORES=[NÚCLEOS]
- SPARK_EXECUTOR_MEMORY=[MEMORIA]
- SPARK_WORKER_CORES=[NÚCLEOS]
- SPARK_WORKERRRR_MEMORY=[MEMORIA]
- SPARK_DAEMON_MEMORY=[MEMORIA]

En apache Flink es necesario cambiar los siguientes parámetros:

- jobmanager.heap.size: [MEMORIA]
- taskmanager.heap.size: [MEMORIA]
- taskmanager.numberOfTaskSlots: [NÚCLEOS]
- parallelism.default: [Nº MÁQUINAS-1]*[NÚCLEOS]

Después de modificar el archivo *install.sh* (si es necesario), ejecutar para realizar la instalación y configuración del nodo. El software se instala en el directorio de usuario.

Tras tener instalado el software necesario para realizar las pruebas, eliminar la carpeta **install_nodes** del proyecto. Después, hay que enviar la carpeta completa del proyecto al resto de máquinas utilizando el comando **scp**. Una vez que cada máquina tiene el proyecto, hay que acceder a cada máquina y compilar el proyecto. Para realizar la compilación hay que abrir un terminal, situar el directorio de trabajo en la

carpeta del proyecto con el comando “**cd**” y ejecutar el comando “**mvn clean compile assembly:single**”.

Con todos estos pasos ya están todos los nodos del clúster listos para realizar la puesta en marcha para realizar las experimentaciones.

Nota: Si algún script no se puede ejecutar, es posible que sea porque no tiene permisos de ejecución. En este caso hay que asignarle estos permisos con el comando “**chmod +x [ruta script]**”.

4.1.4. Puesta en marcha del clúster

En esta subsección se van a explicar los pasos necesarios para iniciar el clúster y realizar las experimentaciones. Es muy importante tener compilado el proyecto con el código de la comparativa.

En la carpeta **conf** del proyecto se proporcionan unos archivos que contienen los parámetros de configuración que se van a utilizar en la experimentación. Existen cuatro archivos:

- **zk.properties**: Contiene los parámetros sobre Zookeeper que son necesarios para ejecutar las aplicaciones en Samza. Entre ellos se puede modificar (si es necesario) “**job.name**”, que indica el nombre de la aplicación.
- **samza.properties**: Contiene los parámetros que va a utilizar la aplicación de procesamiento en Samza. Es importante que los parámetros “**app.name**” y “**job.name**” sea igual que el especificado en el archivo anterior. Además, se debe especificar en el parámetro “**yarn.package.path**” la ruta donde se almacena el paquete compilado que se va a ejecutar en la plataforma Samza.
- **localConf.yaml**: Contiene los parámetros utilizados por las aplicaciones de procesamiento para realizar pruebas en modo local. Incluye:
    - **kafka.brokers**: Dirección IP de las máquinas que ejecutan el bróker de Kafka.

  - **zookeeper.servers**: Dirección IP de las máquinas que ejecutan Zookeeper.
  - **kafka.port**: Puerto en el que escucha el bróker de Kafka.
  - **zookeeper.port**: Puerto en el que escucha Zookeeper.
  - **redis.host**: Dirección IP del nodo con Redis.
  - **kafka.topic**: *Topic* utilizado para enviar/recibir datos durante la experimentación.
  - **kafka.partitions**: Número de particiones que utiliza el *topic* de Kafka. Normalmente coincide con el número de nodos esclavos.
  - **process.hosts**: Número de máquinas esclavas.
  - **process.cores**: Número de núcleos que utiliza cada máquina esclava.
  - **storm.workers**: Número de máquinas esclavas.
  - **storm.ack**: Indica si se habilita el envío de *ack* o no.
  - **time.divisor**: Constante para convertir las unidades temporales de las campañas generadas durante la experimentación.
  - **spark.batchtime**: Intervalo de *batch* utilizado en Spark, en milisegundos.
  - **spark.master**: Dirección del nodo maestro en Spark.
  - **spark.app.name**: Nombre de la aplicación ejecutada en Spark.
- **benchmarkConf.yaml**: Contiene los parámetros utilizados por las aplicaciones de procesamiento para realizar pruebas en modo clúster. Incluye, además de los parámetros de "localConf.yaml":
  - **test.time**: Tiempo en segundos que dura cada ejecución.
  - **tps.tps**: Valor inicial para *tps*.
  - **tps.range**: Incremento de valor del *tps* tras cada experimentación.
  - **tps.limit**: Límite superior de valor para *tps*.
  - **time.short**: Tiempo de espera corto.
  - **time.long**: Tiempo de espera largo.
  - **time.wait**: Tiempo de espera muy largo.
  - **ssh.user**: Nombre de usuario *ssh* utilizado en las máquinas.
  - **kafka.folder**: Directorio en el que se encuentra Kafka.

Tras modificar la configuración de la comparativa ya se pueden ejecutar experimentos. Para ello se proporcionan tres scripts que facilitan esta tarea:

- **variable.sh**: Incluye algunas variables utilizadas por los otros dos scripts y los comandos necesarios para iniciar o finalizar cualquiera de las plataformas y para ejecutar las aplicaciones de procesamiento. Es necesario indicar el número de nodos que se dedica a cada tarea en: ***STREAM_SERVER_COUNT***, ***LOAD_SERVER_COUNT***, ***ZOOKEEPER_SERVER_COUNT*** y ***KAFKA_SERVER_COUNT***. Además, se debe especificar el TPS de inicio (***INITIAL_TPS***), el TPS de fin (***TPS_LIMIT***), el incremento TPS (***TPS_RANGE***) y la duración de cada prueba (***TEST_TIME***).
- **stream-bench.sh**: Contiene los comandos necesarios para iniciar o finalizar cualquier plataforma; o para iniciar o finalizar la aplicación de procesamiento de forma local.
- **run.sh**: Este es el script principal que se ejecutará para realizar las pruebas de cada experimentación. Contiene funciones para ejecutar comandos en las otras máquinas utilizando el comando ssh. Estos comandos pueden ser para iniciar o finalizar cualquier plataforma (Zookeeper, Kafka, Redis, Spark, Storm, …), para ejecutar las aplicaciones de procesamiento o para realizar un envío de resultados de todas las máquinas al nodo maestro.

Para ejecutar los experimentos hay que abrir un terminal y situar el directorio de trabajo en la carpeta del proyecto con el comando "**cd**". Después, ejecutar el script "**run.sh**" pasándole como parámetro "**all**" para ejecutar la experimentación con todas las plataformas (Spark *Streaming*, Spark *Structured Streaming*, Storm, Flink y Samza). Para cada una de ellas, se ejecuta un bucle que se repite mientras el valor de **TPS** no supere el ***TPS_LIMIT***. En cada ejecución del bucle, lo primero que se hace es iniciar Zookeeper, Kafka, Redis y la plataforma de procesamiento que se vaya a utilizar. Después se ejecuta la aplicación de procesamiento y se inicia el envío de datos. La aplicación procesa datos durante el tiempo de ejecución especificado en ***TEST_TIME***. Esto almacena unos resultados en cada máquina. Al terminar la ejecución, se detienen la aplicación de procesamiento, Redis, Kafka y Zookeeper. Finalmente, se envían los

resultados de cada máquina al nodo maestro con el comando "**scp**". A continuación, se incrementa el valor de ***TPS*** en ***TPS_RANGE*** y se repite el bucle.

Finalmente, todos los resultados se encuentran en la carpeta "**result**". Para cada plataforma, hay una carpeta que contiene todos sus datos. Dentro de la carpeta de cada plataforma, se incluye una carpeta por cada TPS analizado (en nuestro caso 12 carpetas). Esta carpeta contiene información de Redis (datos procesados) y uso de recursos de cada máquina durante la ejecución (memoria y CPU). Para generar las gráficas a partir de estos resultados hay que ejecutar el comando "**R**" para abrir la consola de R y después ejecutar en ella el comando "**Rscript reporting/reporting.r [plataforma] [INITIAL_TPS] [DURACIÓN] [TPS_COUNT]**", donde:

- [plataforma] puede ser spark_dstream, spark_dataset, storm, flink o samza.
- [INITIAL_TPS] es el TPS inicial de cada prueba.
- [DURACIÓN] es la duración de cada ejecución.
- [TPS_COUNT] es el número de ejecuciones que se han realizado en la prueba.

Con este comando se generan una serie de gráficas en formato **pdf** en la carpeta de **result**. En la siguiente subsección se van a comentar los resultados obtenidos tras realizar todas las experimentaciones.

4.1.5. Experimentación

En esta comparativa se han realizado diferentes ejecuciones para cada plataforma de procesamiento de datos, cada una con un TPS (Tuplas Por Segundo) diferente. Se ha iniciado desde 4000 TPS hasta 48000 TPS con un incremento de 4000 TPS, realizando un total de 12 ejecuciones de 10 minutos por cada plataforma. El resultado se ha representado a través de diferentes gráficas (una por plataforma), como se ha explicado en la subsección anterior. En esta subsección se van a comentar los resultados obtenidos para cada plataforma y al final se va a realizar una comparativa general entre ellas. En estos resultados, se puede observar la latencia obtenida en cada experimento para los diferentes TPS utilizados. El objetivo de este

experimento es buscar a partir de que TPS se produce un cuello de botella en la latencia, lo que indica que la plataforma de procesamiento está recibiendo más datos de los que puede procesar por unidad de tiempo.

Dependiendo del tiempo de procesamiento y el tiempo de ventana la latencia puede ser: más o menos constante si el tiempo de procesamiento es menor que el tiempo de ventana; o aumentar linealmente o incluso exponencialmente, porque el tiempo de procesamiento es mayor y los datos que llegan al sistema tienen que esperar a que termine el procesamiento (dependiendo de la cantidad de datos recibida por unidad de tiempo el aumento será lineal o exponencial). Cuando la latencia aumenta exponencialmente, esto indica que existe un cuello de botella.

En la Tabla 4.2 se muestra un resumen de los resultados obtenidos en cada experimentación. En la primera fila aparece el TPS a partir del que se produce el cuello de botella en cada plataforma. Las dos últimas filas muestran las latencias mayor y menor obtenidas para cada plataforma de procesamiento, respectivamente.

Cada sub-ilustración en la Figura 4.5 muestra la evolución de la latencia por el porcentaje de tuplas completadas para las diferentes plataformas de procesamiento analizadas. La latencia aparece multiplicada por dos porque hay dos nodos que generan los datos (cuando el TPS es 4000 el TPS real es 8000).

	Spark Streaming	Spark Structured Streaming	Storm	Flink	Samza
Cuello de botella (TPS)	88K	88K	16K	88K	72K
Menor latencia (S)	4	12.5	25	4	4
Mayor latencia (S)	58	32.5	550	10	400

Tabla 4.2. Resumen de resultados de la comparativa I

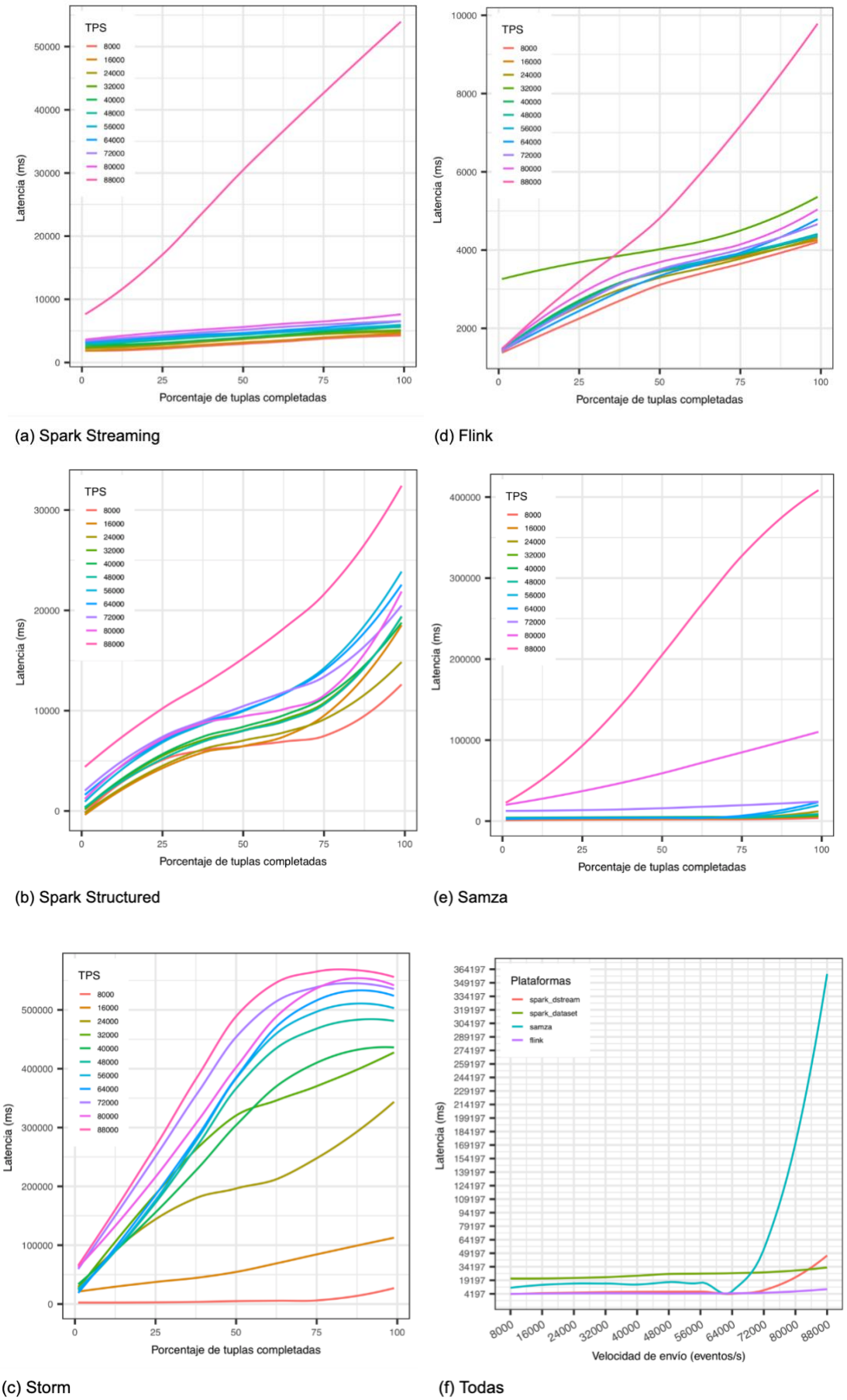

Figura 4.5. Resultados obtenidos en la comparativa I

En la Figura 4.5 (a) se puede ver que la latencia de **Spark *Streaming*** está por debajo de 10 segundos entre los 8K TPS y los 80K TPS y que aumenta linealmente. En 88K TPS aparece el cuello de botella, porque la latencia aumenta exponencialmente y alcanza unos 55 segundos. De esta manera, los resultados obtenidos muestran que Spark *Streaming* tiene un buen rendimiento para esta configuración de clúster cuando el TPS está por debajo de 80K.

La segunda plataforma de procesamiento analizada fue **Spark *Structured Streaming***, cuyos resultados se incluyen en la Figura 4.5 (b). Entre 8K TPS y 80K TPS la latencia se mantiene por debajo de 24 segundos y para 88K TPS alcanza los 33 segundos. Esto indica que el cuello de botella aparece a partir de 80K TPS, pero la latencia es menor que en Spark *Streaming* para 80K TPS. Por esta razón Spark *Structured Streaming* tiene un mejor rendimiento para esta configuración de clúster que Spark *Streaming* y Samza cuando el TPS es alto.

En la Figura 4.5 (c) se puede observar que las latencias de **Storm** son demasiado altas en comparación con las otras plataformas de procesamiento. En esta plataforma el cuello de botella aparece demasiado pronto. Como muestran los resultados obtenidos Storm no tiene un buen rendimiento para el problema de *streaming* propuesto en *Yahoo Streaming Benchmark*.

En **Flink** se han obtenido las mejores latencias de la comparativa para todos los TPS. En la Figura 4.5 (d) se muestra que entre 8K TPS y 80K TPS la latencia tiene un aumento más o menos lineal, con latencias entre 4 segundos y algo más de 5 segundos. El cuello de botella aparece a los 88K TPS, pero la latencia es inferior a 10 segundos. Esta es la menor latencia obtenida en la comparativa.

Los resultados de **Samza** se incluyen en la Figura 4.5 (e), donde se puede observar que la latencia está por debajo de 10 segundos entre 8K TPS y 48K TPS. Entre 48K TPS y 72K TPS la latencia alcanza unos 20 segundos y para TPS superiores a 72K la latencia aumenta exponencialmente, superando los 400 segundos en el caso de 88K TPS. Estos resultados muestran que Samza tiene un buen rendimiento para esta configuración de clúster cuando el TPS es bajo.

Finalmente, los resultados de Spark *Streaming* (spark_dstream), Spark *Structured Streaming* (spark_dataset), Flink y Samza se comparan en la Figura 4.5 (f) utilizando la misma escala en los ejes de coordenadas para evaluar mejor el rendimiento. Storm no se ha incluido en esta comparativa final porque tiene un rendimiento demasiado bajo con respecto al resto de plataformas. Spark *Streaming* tiene menor latencia que Spark *Structured Streaming* para 80K TPS o menos, pero para más de 80K TPS Spark *Structured Streaming* mantiene un buen rendimiento mientras que la latencia de Spark *Streaming* aumenta exponencialmente. Samza tiene un buen rendimiento hasta 64K TPS, pero después la latencia tiene un aumento exponencial para TPS altos. Para todos los TPS, la latencia de Flink es más o menos constante alrededor de 5 segundos, pero para TPS altos la latencia empieza a aumentar más rápido, pero siempre por debajo del resto de plataformas de procesamiento.

De acuerdo a los resultados obtenidos Flink tiene un mejor comportamiento en esta comparativa que las otras plataformas de procesamiento en *streaming*, porque tiene la menor latencia para todos los TPS analizados.

4.2. Comparativa II: Machine Learning Benchmark

En esta subsección se va a detallar la segunda comparativa realizada en la que se utiliza un algoritmo de *machine learning* en *streaming* para medir el rendimiento de las plataformas de procesamiento. En este caso solo se utilizan Flink y Spark, porque fueron las dos mejores en la comparativa anterior. Aunque Spark *Structured Streaming* ha demostrado tener un mejor rendimiento que Spark *Streaming* en la comparativa anterior, actualmente no tiene librerías con algoritmos de *machine learning* en *streaming*. Por este motivo, para esta segunda comparativa se ha decidido utilizar Spark *Streaming*, porque tiene la librería MLLIB que contiene algunos algoritmos de *streaming* implementados.

En primer lugar, se describe como funciona la metodología utilizada en esta comparativa. En las siguientes subsecciones se describe el clúster utilizado, detallando la arquitectura de nodos utilizados, como se ha realizado su configuración e instalación y la puesta en marcha del sistema para ejecutar las experimentaciones.

Finalmente, en la última subsección se explican las pruebas realizadas y se analizan los resultados obtenidos.

4.2.1. Introducción

En esta comparativa se ha decidido utilizar el algoritmo ***streaming k-means***, que es un algoritmo de *clustering* que crea diferentes agrupaciones a partir de los datos. Para ello, inicializa unos centroides en posiciones aleatorias y los va ajustando con los nuevos datos que llegan al sistema.

El objetivo es medir el tiempo de entrenamiento que tardan Flink y Spark *Streaming* en procesar diferentes *datasets*. Para esta comparativa se han utilizado cuatro *datasets* con diferentes tamaños. De estos cuatro *datasets* dos son reales y dos son sintéticos. Las principales características de cada *dataset* se indican a continuación:

- **Covtype**: es un *dataset* real que contiene 581.012 instancias y 54 atributos enteros y modela el problema de predecir el tipo de bosque a partir de variables cartográficas.
- **Power**: es otro *dataset* real que contiene 2.049.280 instancias (después de eliminar las instancias con valores perdidos) y 7 atributos decimales y mide el consumo eléctrico en una vivienda con una muestra cada minuto durante un periodo de cuatro años.
- **SEA**: es un *dataset* sintético generado con MOA [5] que contiene 5.000.000 instancias y 3 atributos decimales, de los que solo 2 son relevantes: $x_i \in [0,10]$, con i=1,2,3. El concepto es $x_1 + x_2 \leq \beta$, con $\beta \in \{7,8,9,9.5\}$. Para esta comparativa se ha utilizado $\beta = 8$.
- **RBF**: es otro *dataset* sintético generado con MOA [5] que contiene 10.000.000 instancias y 20 atributos decimales. Este generador primero crea un número fijo de centroides aleatorios y, para generar cada instancia, primero selecciona un centroide y después genera una instancia dentro del rango de este centroide.

Como se mencionó anteriormente, Spark *Streaming* si tiene una implementación del algoritmo *streaming k-means* en la librería MLLIB, pero Flink no tiene una implementación. Por este motivo se ha realizado la implementación de dicho algoritmo para Flink. Esta implementación en Flink es el mismo algoritmo que hay en MLLIB, pero utiliza ventanas secuenciales en lugar de ventanas temporales. En Spark *Streaming* el modelo se actualiza tras finalizar cada intervalo de *batch*, mientras que en Flink el modelo se actualiza tras recibir un determinado número de instancias. Para ello la implementación de Flink utiliza una frecuencia de actualización que indica cuando se actualiza el modelo. El modelo se actualiza de la misma manera en ambas plataformas, pero utilizando diferentes tipos de ventana.

4.2.2. Arquitectura del clúster

La arquitectura del clúster utilizado para realizar esta comparativa se puede observar en la Figura 4.6. En este caso se ha reutilizado el clúster creado para la comparativa anterior, pero se ha eliminado un productor Kafka y el nodo Redis.

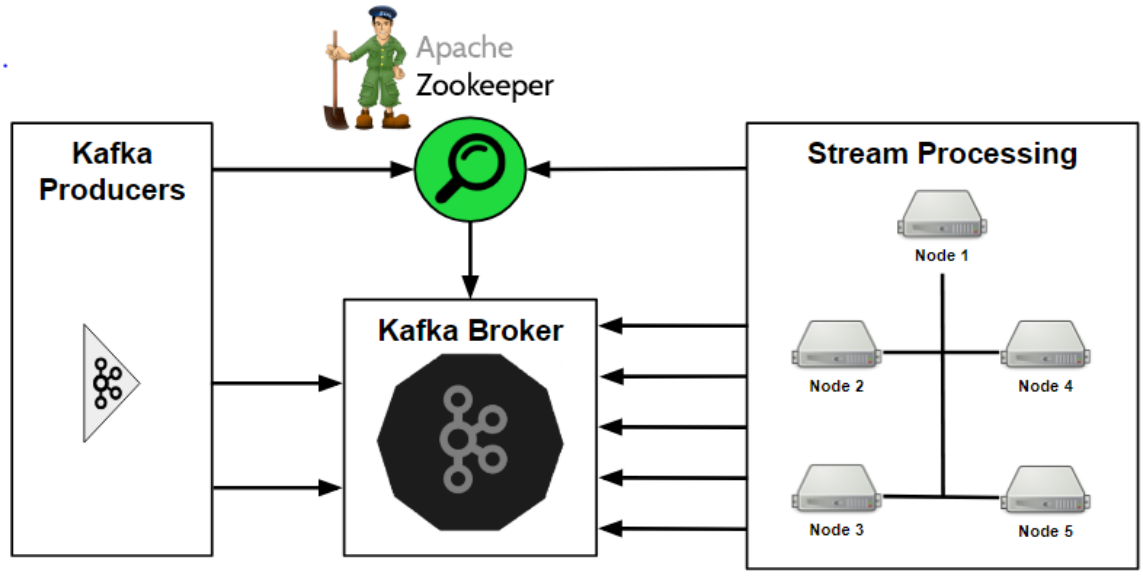

Figura 4.6. Arquitectura del clúster en la comparativa II

Para realizar esta comparativa se pretende crear una arquitectura similar a la de la primera comparativa, en la que un productor de datos genera eventos que son enviados y almacenados en el almacén de Kafka. Una vez que los mensajes están almacenados en Kafka, los nodos de procesamiento empiezan a leer y procesar los datos en paralelo. Finalmente, cuando se termina de procesar todo el *dataset* se

muestra el tiempo empleado por consola. Por otro lado, Zookeeper gestiona y coordina los diferentes servicios del clúster.

En el clúster utilizado para realizar la experimentación todos los nodos tienen una configuración homogénea. Cada nodo tiene un procesador Intel Core i5 750 2.67GHz, 4 núcleos y 4GB de memoria RAM. Además, se han utilizado conexiones de 1GB/s entre los nodos. En total se han utilizado 8 nodos: 5 nodos de procesamiento (4 esclavos y 1 maestro); 1 nodo que funciona como almacén Kafka; 1 nodo Zookeeper; y 1 nodo productor de datos.

Una vez que se conoce cual es la arquitectura del clúster necesario para realizar la experimentación, es necesario instalar y configurar el clúster. Como este proceso es idéntico al utilizado para la primera comparativa, se puede ver en la subsección 4.1.3. En la siguiente subsección se va a explicar como poner el clúster en marcha para ejecutar la experimentación.

4.2.3. Puesta en marcha del clúster

En primer lugar, es necesario descargar el proyecto que contiene las aplicaciones en Spark *Streaming* y Flink, que se incluye junto a este proyecto. Este proyecto contiene:

- **kafkaProducer**: Es la carpeta con el proyecto del productor Kafka. En esta carpeta están incluidos los 4 *datasets*.
- **ejemploSpark**: Es la aplicación de procesamiento en Spark *Streaming*.
- **Flink-StreamingKMeans**: Es la aplicación de procesamiento en Flink.
- **spark.conf**: Contiene algunos parámetros de configuración para la aplicación de Spark *Streaming*.

Para ejecutar las diferentes plataformas se pueden utilizar los scripts mencionados en la subsección 4.1.4. Se puede utilizar el script "**run.sh**" para realizar las siguientes acciones:

- **Iniciar Zookeeper**: opción "*start zoo*".
- **Iniciar Kafka Broker**: opción "*start kafka*".

- **Iniciar clúster Spark**: opción "*start spark*".
- **Iniciar clúster Flink**: opción "*start flink*".
- **Detener Zookeeper**: opción "*stop zoo*".
- **Detener Kafka Broker**: opción "*stop kafka*".
- **Detener clúster Spark**: opción "*stop spark*".
- **Detener clúster Flink**: opción "*stop flink*".

Una vez que el clúster está iniciado, se pueden ejecutar las aplicaciones de procesamiento en ellos. Para ejecutar la aplicación de Spark *Streaming* primero hay que acceder al código y especificar el número de centroides (k), el factor de penalización y los centroides iniciales. Después hay que compilar el proyecto con el comando "**sbt assembly**". Antes de ejecutar la aplicación, es necesario ir al archivo "**spark.conf**" y asignar el valor correcto a "**kafka.topic.train**", "**kafka.topic.test**" y "**kafka.partitions**". En los dos primeros se asigna el nombre para el *topic* de entrenamiento y el de prueba (aunque solo se utiliza el de entrenamiento) y en el último parámetro se especifica el número de CPUs totales. Después se puede ejecutar la aplicación de procesamiento con el comando "**spark/bin/spark-submit –master spark://stream-node01:7077 –class Main [ruta jar] [ruta spark.conf]**".

Para ejecutar la aplicación de Flink primero hay que acceder al código y especificar el número de nodos, la frecuencia de actualización, el número de centroides, el factor de penalización y los centroides iniciales. Después hay que compilar el proyecto con "**sbt assembly**". Finalmente se ejecuta la aplicación de procesamiento con el comando "**flink/bin/flink run [ruta jar]**".

Las dos aplicaciones de procesamiento deben estar en el nodo maestro y la aplicación de Kafka que va a generar los datos debe estar en el nodo productor. El proceso que se ha seguido para realizar esta comparativa es el siguiente:

1. Iniciar Zookeeper
2. Iniciar Kafka broker
3. Iniciar Spark/Flink
4. Iniciar aplicación Spark/Flink
5. Iniciar envío de datos
6. Esperar a que la aplicación procese todos los datos

7. Detener aplicación Spark/Flink
8. Detener Spark/Flink
9. Volver a 3 hasta que se realizan todas las pruebas
10. Detener Kafka broker
11. Detener Zookeeper

En esta comparativa, para cada *dataset* se han realizado 10 ejecuciones y el resultado es la media de estas ejecuciones. Al iniciar la aplicación de procesamiento se realiza una espera de 20 segundos para que de tiempo a que el nodo maestro envíe el jar a los nodos esclavos y lo ejecuten. Después de esta espera se inicia el envío de datos. La aplicación va recibiendo los datos conforme se envían y los procesa. Cuando termina este procesamiento, se almacena el tiempo empleado en este procesamiento y se detiene la aplicación de procesamiento y después la plataforma de procesamiento. A continuación, se vuelve a realizar el mismo proceso hasta 10 veces por cada *dataset*, para tener una medida más fiable.

4.2.4. Experimentación

Como se ha comentado antes, en la Tabla 4.3 aparece la media de 10 ejecuciones en Spark *Streaming* y Flink para cada configuración de clúster. En las diferentes experimentaciones se ha ido modificando el número de nodos esclavos del clúster, incrementando este número de uno a cuatro. Los resultados muestran que Flink tiene un mejor rendimiento que Spark *Streaming* en los *datasets* Power, SEA y RBF, empleando más o menos la mitad del tiempo en procesar los *datasets*. Como se puede observar, el tiempo de procesamiento disminuye proporcionalmente al número de nodos que procesan datos. Por ejemplo, suponed que con un nodo el tiempo de entrenamiento es 40 segundos. Si el número de nodos esclavos aumenta a 2, el tiempo disminuye a 20 segundos (la mitad) y con 4 nodos a 10 segundos (la cuarta parte). Pero como puede verse en la Tabla 4.3, algunos valores no disminuyen como deberían. Esto se debe a que el envío de datos limita el tiempo de procesamiento, porque el envío es más lento que las capacidades de procesamiento del clúster.

Dataset	Workers	Spark Streaming time (s)	Flink time (s)
Covtype	1	**13,6313**	18,2095
	2	**8,4619**	10,6988
	3	**6,3911**	7,9011
	4	**6,1384**	6,4233
Power	1	21,4361	**12,7472**
	2	11,8503	**8,5622**
	3	7,6643	**7,442**
	4	7,1309	**6,758**
SEA	1	59,6979	**29,172**
	2	32,1117	**22,8102**
	3	23,7461	**19,6678**
	4	20,7679	**18,7681**
RBF	1	676,6879	**320,9581**
	2	360,5873	**248,321**
	3	269,9693	**197,9395**
	4	237,5595	**185,5117**

Tabla 4.3. Resultados obtenidos en la comparativa II (I)

Para el *dataset* Covtype, Spark *Streaming* tiene mejor rendimiento que Flink, por lo que se ha realizado una nueva experimentación con este *dataset* para buscar las causas de esto. En este segundo experimento se utilizan otros dos *datasets*. Un *dataset* (denominado **Covtype2**) es el mismo *dataset* Covtype, pero tiene 8 veces su contenido. Este *dataset* empieza con una copia de Covtype y, cuando el envío de este *dataset* termina, comienza un nuevo envío, hasta 8 veces. El *dataset* resultante contiene 4.648.096 instancias y 54 atributos enteros. Con este primer *dataset* se quiere comprobar si la causa es el número de instancias del *dataset*. El otro *dataset* (denominado **Covtype3**) es el mismo *dataset* que Covtype2, pero reemplazando los ceros por un valor decimal cualquiera. Este *dataset* también contiene 4.648.096 instancias y 54 atributos enteros y reales.

Los resultados del segundo experimento se incluyen en la Tabla 4.4. El experimento es igual que el anterior, con la media de 10 ejecuciones, pero utilizando los nuevos *datasets*. En Covtype2 el rendimiento de Spark *Streaming* sigue siendo mejor que el de Flink en todos los experimentos excepto cuando el número de nodos esclavos es 4. Con este *dataset* se puede observar que la causa no es el número de instancias en el *dataset*. Cuando se observa el contenido del *dataset* se puede apreciar que el *dataset* contiene muchos ceros, por lo que se decide probar el mismo *dataset*, pero reemplazando los ceros por un valor decimal menor que 1. En Covtype3 Flink muestra un mejor rendimiento que Spark *Streaming* en todos los experimentos,

por lo que se puede concluir que la causa es la estructura de datos que utilizan. En Spark *Streaming* se utiliza una estructura denominada ***Vector*** en lugar de ***Array***, que es más eficiente cuando se procesan instancias con muchos ceros en sus atributos. La implementación de *streaming k-means* para Flink realizada en este trabajo utiliza *Array*, por lo que es menos eficiente que Spark *Streaming* en estos casos.

Dataset	Workers	Spark Streaming time (s)	Flink time (s)
Covtype	1	**13,6313**	18,2095
	2	**8,4619**	10,6988
	3	**6,3911**	7,9011
	4	**6,1384**	6,4233
Covtype2	1	**113,1811**	130,3124
	2	**60,5999**	68,94
	3	**43,8494**	47,64275
	4	38,3538	**37,88**
Covtype3	1	177,081	**134,2934**
	2	96,6267	**71,6498**
	3	73,1975	**53,5453**
	4	62,7564	**50,6911**

Tabla 4.4. Resultados obtenidos en la comparativa II (II)

Tanto Spark *Streaming* como Flink escalan bien según el número de nodos esclavos en el clúster, pero en general Flink muestra un mejor rendimiento con respecto a Spark *Streaming* excepto en casos en los que el tiene muchos ceros. Estos resultados muestran que los detalles de implementación y las estructuras de datos utilizadas son factores importantes a tener en cuenta para garantizar la buena eficiencia de los algoritmos.

5. IMPLEMENTACIONES PROPIAS DE ALGORITMOS DE MACHINE LEARNING

Con el paso del tiempo han ido apareciendo nuevos algoritmos de *machine learning* en *streaming* o mejoras sobre los ya existentes. Diferentes estudios del estado del arte en técnicas de *streaming*, aplicadas a tareas de clasificación, se pueden encontrar en [12] [36] [37] [38] [39] [40]. En la revisión realizada, los modelos que más se utilizan son árboles de decisión, modelos basados en reglas, vecinos más cercanos, máquinas de vectores de soporte, redes neuronales y *ensembles*. Aparecen, sin embargo, pocos modelos basados en redes neuronales, uno de los grandes paradigmas en *machine learning*. Esto puede deberse a que en *streaming* el tiempo de respuesta ha de ser mínimo y normalmente el aprendizaje en los modelos basados en redes neuronales suele tener un alto coste computacional. De hecho, en los modelos basados en redes neuronales que aparecen, la arquitectura de red suele ser simple con pocas capas ocultas, normalmente solo una capa. Dado que es más escasa la implementación de modelos basados en redes neuronales para clasificación en *streaming*, este trabajo se centra en este tipo de algoritmos y como el tiempo de entrenamiento es un factor clave, específicamente se van a implementar algoritmos basados en la red ELM (*Extreme Learning Machine*) [41].

Como la red ELM es de los modelos de red neuronal con menor coste computacional, en este trabajo se van a implementar varios algoritmos basados en este tipo utilizando computación distribuida que nos permita reducir el tiempo de ejecución cuando los algoritmos trabajan en un entorno *Big Data*, como pasa en *streaming*. Para ello, se hará uso de apache Spark *Streaming*, una plataforma de procesamiento en *streaming* muy utilizada por la comunidad investigadora y en el ámbito empresarial.

Esta sección se va a organizar de la siguiente forma: en la subsección 5.1 incluye una revisión sobre los distintos tipos existentes de redes ELM; en la subsección 5.2 se explica la implementación del productor de Kafka, necesario para poder enviar los datos a los distintos algoritmos; en las subsecciones 5.3, 5.4, 5.5, 5.6 y 5.7 se detallan los algoritmos implementados ELM, OS-ELM, EOS-ELM, VOS-ELM y CD-VOS-ELM, respectivamente; finalmente, en la subsección 5.8 se va a mostrar y a analizar el

resultado de las experimentaciones realizadas, de forma que se compare el rendimiento de los algoritmos implementados.

5.1. Tipos de redes ELM existentes

En [36] se realiza una revisión de algoritmos para *streaming* basados en la red **ELM** [41], que es una red neuronal que solo tiene una capa oculta y las conexiones entre las neuronas de la capa de entrada y la capa oculta están ponderadas mediante pesos, que se establecen de forma aleatoria en el algoritmo de entrenamiento. El modelo de entrenamiento tiene que aprender los pesos de las conexiones entre las neuronas de la capa oculta y la capa de salida, a partir de los datos de entrada. Esta red se ideó para utilizarse en un entorno tradicional (no *streaming*), por lo que se diseñó un método de entrenamiento incremental a partir de él denominado **OS-ELM** [42] (*Online Sequential Extreme Learning Machine*), para poder utilizar este tipo de red sobre *streams* de datos.

A partir de OS-ELM se crean muchas modificaciones para diferentes tipos de problemas. Esta red tiene un número fijo de neuronas en la capa oculta que debe especificar el usuario. Esto motiva la creación de **CEOS-ELM** [43] (*Constructive Enhancement for OS-ELM*), que es una red OS-ELM que permite adaptar automáticamente el número de neuronas de la capa oculta añadiendo neuronas cuando la precisión del clasificador disminuye por debajo de un límite.

Otra propuesta de mejora fue **EOS-ELM** [44] (*Ensemble of OS-ELM*), que crea un *ensemble* de varios clasificadores OS-ELM. Con este algoritmo se intenta justificar el hecho de que al utilizar un *ensemble* la precisión se mantiene más estable que con un único clasificador, con la desventaja de que requiere mucho más tiempo de entrenamiento que un único clasificador OS-ELM. Sobre este algoritmo se propusieron 2 variantes. La primera se denomina **FOS-ELM** [45] (*OS-ELM with Forgetting mechanism*) para dar más importancia en el entrenamiento a los últimos datos almacena los datos de los “s” bloques más recientes (cuando llega un nuevo bloque sustituye al más antiguo). La segunda variante se denomina **ESOS-ELM** [46] (*Ensemble of Subset OS-ELM*) para tener en cuenta las clases no balanceadas en

problemas de clasificación binaria. En este algoritmo cada clasificador del *ensemble* se entrena con un conjunto de datos balanceado.

En un congreso realizado en 2013 se propuso una variación en la técnica de combinación de soluciones del *ensemble* EOS-ELM. Este método se denomina **VOS-ELM** [47] (*Voting based OS-ELM*) y es el mismo algoritmo EOS-ELM, pero en lugar de combinar las soluciones calculando la media de ellas utiliza una estrategia de voto mayoritario. Con VOS-ELM se intenta justificar que al utilizar la estrategia de voto mayoritario se mejora la precisión del *ensemble*.

La red OS-ELM tiene el problema de que no tiene en cuenta las clases no balanceadas y ante estos problemas suele tener un mal rendimiento. Por este motivo se propone un nuevo método denominado **WOS-ELM** [48] (*Weighted OS-ELM*), donde a cada dato se le asigna un peso según si pertenece a la clase mayoritaria o minoritaria.

La red WOS-ELM solo se puede utilizar en clasificación binaria, por lo que se propone un nuevo método que combina VOS-ELM y WOS-ELM para poder aplicarlo a clasificación multiclase, denominado **VWOS-ELM** [49] (*Voting based WOS-ELM*). Este método es un *ensemble* de clasificadores WOS-ELM, de forma que cada clasificador es muy bueno clasificando una determinada clase. La salida final se obtiene utilizando una estrategia de voto mayoritario.

Recientemente han surgido nuevos métodos basados en OS-ELM que utilizan un método de detección de cambio de concepto para que el modelo se adapte más rápido a nuevos conceptos. Uno de ellos es **IDS-ELM** [50] (*Incremental Extreme Learning Machine for Data stream*), que es un *ensemble* de clasificadores OS-ELM con un método de detección de cambio de concepto basado en el error del *ensemble*. Este método va añadiendo un clasificador con cada nuevo bloque de datos hasta completar el *ensemble*. A partir de este momento clasifica cada nuevo bloque de datos y con esta clasificación obtiene el error del *ensemble*. En este método se define un nivel de decisión. Si el error es inferior a ese límite, todos los clasificadores del *ensemble* se actualizan incrementalmente con el bloque de datos. En caso de superar dicho límite, se ordenan los clasificadores en orden de error y se eliminan la mitad de

los peores, manteniendo la otra mitad. A continuación, se añaden clasificadores con cada bloque y se repite el proceso.

Otro método que utiliza un detector de cambio de concepto es **DELM** [51] (*Dynamic Extreme Learning Machine for Data stream*). Este método utiliza una red ELM de 2 capas ocultas con diferente número de neuronas en cada uno. Además, su mecanismo de detección se basa en el error del clasificador y se definen dos niveles de decisión: *warning* y *drift*. Si el error está por debajo del nivel de *warning*, el modelo se actualiza incrementalmente. Si el error está entre el nivel de *warning* y de *drift*, se modifica el número de neuronas de las capas ocultas y se actualiza el modelo incrementalmente. En caso de que el error supere el nivel de *drift*, se considera que se ha producido un cambio de concepto, por lo que se descarta el modelo actual y se genera uno desde cero.

Finalmente, se presentó un modelo de *ensemble* de clasificadores OS-ELM (similar a EOS-ELM) con un detector de cambio de concepto basado en el error del *ensemble*, denominado **CELM** [52] (*ensemble Extreme Learning Machine with Concept drift detection*). Este método es similar a VOS-ELM, pero añadiendo un mecanismo de detección de cambio de concepto. Se definen dos niveles de decisión: *warning* y *drift*. Si el error está por debajo del nivel de *warning* (no hay cambio), no se actualiza el modelo. Si el error está entre los niveles de *warning* y *drift* (posible cambio), se actualizan los clasificadores del *ensemble* incrementalmente. Si el error supera el nivel de *drift* (existe cambio), se descarta el modelo actual y se crea otro desde cero.

Como se ha comentado en secciones anteriores, actualmente no existen muchas implementaciones de algoritmos de *machine learning* en *streaming* para plataformas de procesamiento distribuidas. Por este motivo, se ha decidido implementar varios modelos basados en OS-ELM utilizando la plataforma Spark, una plataforma muy utilizada que permite el procesamiento distribuido de datos en tiempo real, mediante su módulo de Spark *Streaming*.

5.2. Productor Kafka

Para poder simular el flujo continuo de datos, ha sido necesario implementar un productor Kafka. Este es un programa en **Java** que lee un archivo almacenado localmente y envía su contenido al *Broker* de Kafka. En este programa se utiliza la clase ***BufferedReader*** para leer los archivos. En primer lugar, se carga un archivo denominado "***spark-ELM.conf***" que contiene los parámetros que necesita el productor para realizar el envío de datos. Este archivo tiene dos líneas: la primera indica el número de particiones (**[*PARTITIONS*]**) y la segunda indica la ruta local del archivo que quiere enviarse (**[*PATH*]**). Este productor está programado para enviar los datos al *topic* "***TRAIN-[PARTITIONS]***". Después de leer este archivo, se definen los parámetros de configuración del productor que sean necesarios. En este caso se han especificado los valores que aparecen en la Tabla 5.1. Con estos parámetros se crea una instancia del productor y a continuación se lee el archivo especificado en [*PATH*]. Este archivo se lee línea por línea (cada una es una instancia diferente) y se envía al *Broker* con el método ***send(…)***. Cuando termina el envío se cierra la conexión con los métodos ***flush()*** y ***close()***.

El resto de implementaciones son las de los algoritmos de *machine learning* en Spark, que serán los encargados de procesar los datos en *streaming*, en este caso, enviados desde el productor implementado. Para estas implementaciones se ha creado un proyecto en ***Scala*** que contiene una clase por cada algoritmo (**ELM**, **OS_ELM**, **EOS_ELM**, **VOS_ELM** y **CD_EOS_ELM**). En la implementación de Spark se utilizan dos tipos de estructuras de datos distribuidas: **RDD[((Int,Int),Double)]** y **RDD[Array[Double]]**. Como muestra la Figura 5.1 (b) la primera estructura consiste en distribuir cada celda de la matriz original. Para cada celda se tiene un par ((Int,Int),Double), donde el primer elemento hace referencia a la posición de la matriz (fila,columna) a la que pertenece el dato y el segundo elemento indica el valor que tiene esa celda en la matriz original. La segunda estructura se muestra en la Figura 5.1 (c), que consiste en distribuir cada fila de la matriz. Esta estructura se utiliza en el producto y la inversa de matrices.

Parámetro	Valor	Descripción
key.serializer	LongSerializer	Clase que serializa la clave
value.serializer	StringSerializer	Clase que serializa el valor
bootstrap.servers	localhost:9092	Lista de máquinas del clúster kafka
num.partitions	[PARTITIONS]	Número de particiones para cada *topic*
acks	1	Número de asentimientos positivos para considerar que un envío se ha realizado con éxito
batch.size	98304	Tamaño del bloque que se envía a Kafka. Kafka agrupa *records* hasta llenar un bloque
buffer.memory	100000000	Cantidad de memoria en *bytes* que puede usar el productor para almacenar mensajes que esperan ser enviados
client.id	Producer	Identificador del productor
compression.type	none	Tipo de compresión de los datos
connections.max.idle.ms	300000	Tiempo que debe transcurrir para cerrar una conexión vacía
delivery.timeout.ms	120000	Tiempo de espera para la confirmación de recepción de mensaje
enable.idempotence	false	Si está activo solo se envía una copia de cada *record*, aunque falle. Si no está activo se reenvían *records* que no son confirmados
max.block.ms	30000	Tiempo máximo para agrupar *records* en un bloque
max.in.flight.requests.per.connection	6	Número máximo de respuestas no confirmadas que el productor envía antes de cerrar una conexión
max.request.size	1048576	Tamaño máximo de las respuestas en *bytes*
metadata.max.age.ms	300000	Tiempo de espera para actualizar los metadatos
partitioner.class	org.apache.clients.producer.internals.DefaultPartitioner	Clase que se encarga de las particiones
reconnect.backoff.max.ms	1000	Tiempo máximo de espera para realizar una reconexión
reconnect.backoff.ms	100	Tiempo de espera para realizar una reconexión
request.timeout.ms	30000	Tiempo máximo de espera para recibir una respuesta
retries	2147483647	Número de reintentos de envío
retry.backoff.ms	100	

Tabla 5.1. Parámetros de configuración del productor de Kafka

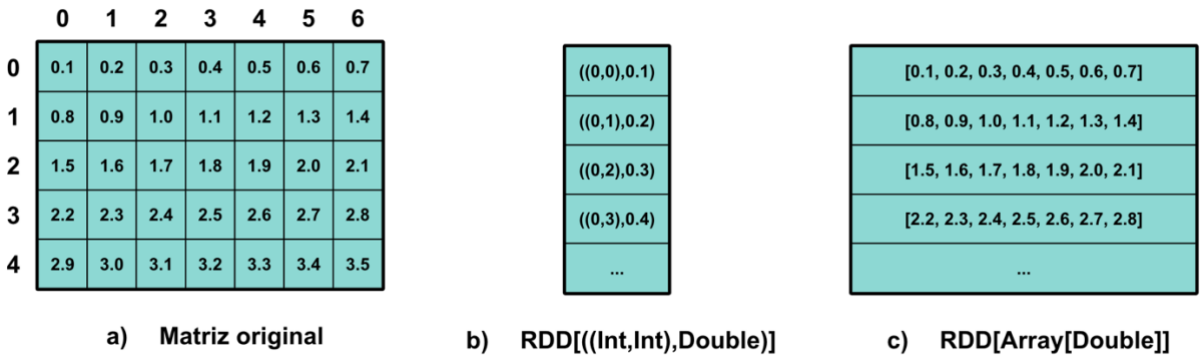

Figura 5.1. Estructuras de datos distribuidas para las implementaciones

Además de las clases mencionadas en el párrafo anterior, el proyecto incluye la implementación de las siguientes clases y objetos:

- ***ActivationFunctions***: Contiene las funciones de activación que pueden utilizar las neuronas de la red neuronal. En este caso solo está implementada la función ***sigmoid***.
- ***Matrix***: Contiene todas las operaciones con matrices en paralelo que se utilizan en estas implementaciones:
    - ***parallelDot***: Esta función calcula el producto de dos matrices. Para ello distribuye las filas de la primera matriz entre los diferentes nodos y, a cada nodo, le envía una copia de la segunda matriz completa. Cada nodo calcula algunas filas de la matriz resultante. Al final se distribuyen las celdas resultantes y se devuelve una estructura del primer tipo.
    - ***parallelTrans***: Esta función calcula la traspuesta de una matriz. Recibe como entrada una matriz distribuida del primer tipo (por celdas) y esta función lo que hace es invertir los índices de fila y columna para todas las celdas de la matriz.
    - ***parallelInv***: Calcula la inversa de una matriz cuadrada M utilizando el método ***SVD*** (Descomposición en Valores Singulares) de MLLIB. Este método utiliza una estructura del segundo tipo (distribuida por filas) y el método descompone la matriz en tres matrices (U, V, S). Aplicando la expresión $M^{-1} = VS^{-1}U^T$ se obtiene la inversa de la matriz M.
    - ***parallelSum***: Calcula la suma de dos matrices distribuidas por celdas. Para ello, une las dos matrices en un mismo RDD y luego suma las celdas con el mismo índice (Int,Int), utilizando un ***reduceByKey()***. Suponed que se quieren sumar las siguientes celdas procedentes de los dos RDDs de entrada: ((0,0),1.3) y ((0,0),4.2), respectivamente. El resultado de la suma sería la celda ((0,0),5.5).
    - ***parallelRes***: Igual que la suma, pero modificando la función del ***reduceByKey()***, ya que en la suma no influye el orden de los factores, pero en la resta si influye. Para el ejemplo anterior, no es

lo mismo ((0,0)1,3) – ((0,0),4.2) que ((0,0),4.2) – ((0,0),1.3). En este caso se le ha añadido un identificador a cada celda que indica si pertenece al primer factor de la resta o al segundo. Sabiendo cual es el orden correcto de los factores para restarlos la operación se puede hacer en paralelo sin problema.

- ***Params***: Es un objeto que incluye los parámetros de configuración de la aplicación de Spark *Streaming* y de los algoritmos de *machine learning*. Los parámetros que se incluyen son los que aparecen en la Tabla 5.2.
- ***Prequential***: Almacena el error actual del modelo.
- ***Results***: Se encarga de almacenar en ficheros los resultados de tiempo y error.
- ***TimeMetric***: Almacena el tiempo total que se ha empleado en el entrenamiento del modelo.
- ***Utils***: Es un objeto que tiene diferentes funciones para agilizar la implementación. Incluye métodos para mostrar una matriz por consola y para pasar de un tipo de datos a otro (por ejemplo, de RDD[((Int,Int),Double)] a Array[Array[Double]]).

Tras explicar las clases y métodos generales utilizadas en el proyecto junto con los aspectos de configuración, en las siguientes subsecciones se detallan cada una de las implementaciones realizadas.

Parámetro	Descripción
EXP_NUMBER	Número de experimento
EXP_NAME	Nombre del experimento. Se utiliza junto a *EXP_NUMBER* para generar los archivos con los datos de cada ejecución
SEEDS	Es un *array* con las semillas utilizadas
SEED	Es la semilla utilizada en la ejecución actual. Es la semilla en *SEEDS* que está en la posición *EXP_NUMBER*
SPARK_MASTER	URL del nodo maestro del clúster Spark
SPARK_APP_NAME	Nombre de la aplicación que se ejecuta
SPARK_BATCH_INTERVAL	Tiempo en segundos del intervalo de *batch*
SPARK_KAFKA_RATE	Indica si se habilita la limitación de envío o no
KAFKA_BROKERS	Lista de *Brokers* a los que se conecta la aplicación de Spark *Streaming* para recibir los datos
TOPIC	Es el *topic* del que se obtienen los datos
KAFKA_RATE	Es la velocidad de envío de datos del *Broker* al sistema Spark, indicado en instancias por segundo
NODES	Número de nodos esclavos en el clúster de Spark
CORES_NODE	Número de núcleos que tiene cada nodo del clúster de Spark
PARTITIONS	Número de particiones utilizadas
n	Número de atributos de entrada
L	Número de neuronas de la capa oculta
m	Número de clases disponibles
P	Número de clasificadores en los algoritmos de *ensemble*
MAX	*Array* con el valor máximo que puede tomar cada atributo
MIN	*Array* con el valor mínimo que puede tomar cada atributo
FILE_PATH	Ruta del archivo con los datos de entrada. Esto se utiliza solo en los algoritmos tradicionales (no *streaming*)
TRAIN_PROPORTION	Valor entre 0 y 1 que indica la proporción de datos para entrenamiento
TEST_PROPORTION	Valor entre 0 y 1 que indica la proporción de datos para evaluación
lUp	Límite superior (*drift*) utilizado en el algoritmo *CD_VOS_ELM*
lDown	Límite inferior (*warning*) utilizado en el algoritmo *CD_VOS_ELM*

Tabla 5.2. Parámetros de configuración de la aplicación Spark Streaming

5.3. ELM

ELM [41] (*Extreme Machine Learning*) es un algoritmo de red neuronal que tiene solo una capa oculta y se puede utilizar para problemas de regresión y clasificación en modo tradicional. Su estructura general se presenta en la Figura 5.2, donde se pueden apreciar las siguientes capas:

- **Capa de entrada**: contiene n neuronas, donde n coincide con el número de atributos de los datos. Cada enlace que une la capa de entrada con la capa oculta tiene un peso asociado.

- **Capa oculta**: contiene L neuronas, donde L es un parámetro que introduce el usuario. Cada neurona de la capa oculta tiene asociado un valor de bias.
- **Capa de salida**: contiene m neuronas, donde m coincide con el número de clases distintas que pueden aparecer. Cada enlace que une la capa oculta con la capa de salida tiene un peso asociado.

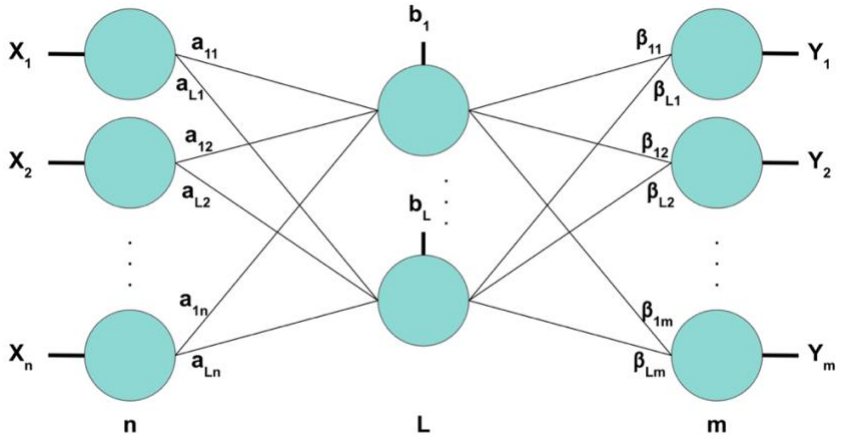

Figura 5.2. Estructura general de una red ELM

En este algoritmo los valores de los pesos de entrada y los de bias se generan aleatoriamente, normalmente con valores comprendidos en el intervalo [-1,1]. Después, como se conoce el valor de salida de cada instancia, se calcula que valor deberían tener los pesos de salida para que a partir de esas entradas se obtengan esas salidas.

Suponed que se tiene un bloque de datos de tamaño N con distintas instancias (X_i, Y_i), donde $X_i = [X_{i1}, X_{i2}, \ldots, X_{in}] \in R^n$ y $Y_i = [Y_{i1}, Y_{i2}, \ldots, Y_{im}] \in R^m$. La salida de una red neuronal con L neuronas en la capa oculta es:

$$f(X_i) = \sum_{j=1}^{L} \beta_j g(a_j, b_j, X_j), \; para\; i = 1, 2, \ldots, N \qquad \textit{(Fórmula 5.1)}$$

Donde g(X) es una función de activación (en nuestro caso la función **sigmoid**, $g(a, b, X) = \frac{1}{1+\exp(-(a*X+b))}$), $a_j = [a_{j1}, a_{j2}, \ldots, a_{jn}]$ es el vector de pesos que unen la capa de entrada con la neurona j, y b_j es el valor de bias de la neurona j. $\beta_j = [\beta_{j1}, \beta_{j2}, \ldots, \beta_{jm}]$ es el vector de pesos que unen la neurona j con la capa de salida. La formula anterior se puede escribir también de la siguiente forma:

$$H\beta = Y \qquad \textit{(Fórmula 2.2)}$$

donde

$$H = \begin{bmatrix} g(a_1, b_1, X_1) & \cdots & g(a_L, b_L, X_1) \\ \vdots & \ddots & \vdots \\ g(a_1, b_1, X_N) & \cdots & g(a_L, b_L, X_N) \end{bmatrix}_{NxL}$$

$$\beta = [\beta_1, \dots, \beta_L]^T = \begin{bmatrix} \beta_{11} & \cdots & \beta_{1m} \\ \vdots & \ddots & \vdots \\ \beta_{L1} & \cdots & \beta_{Lm} \end{bmatrix}_{Lxm}$$

$$Y = [Y_1, \dots, Y_N]^T = \begin{bmatrix} Y_{11} & \cdots & Y_{1m} \\ \vdots & \ddots & \vdots \\ Y_{N1} & \cdots & Y_{Nm} \end{bmatrix}_{Nxm}$$

H se denomina matriz de salidas de la capa oculta. Esta matriz tiene una fila por cada instancia del bloque de datos de entrada (1, …, N) y una columna por cada neurona de la capa oculta (1, …, L). Cada celda (i,j) de la matriz contiene la salida de la función de activación cuando recibe como entrada el dato i (X_i) y los pesos de entrada y el valor de bias de la neurona j (a_j y b_j, respectivamente). Cada fila i de la matriz indica el valor de salida de cada neurona de la capa oculta para la instancia i.

β es la matriz de pesos de salida de la capa oculta. Esta matriz tiene una fila por cada neurona de la capa oculta (1, …, L) y una columna por cada neurona de la capa de salida (1, …, m). Cada celda (i,j) de la matriz contiene el peso que une la neurona oculta i con la neurona de salida j. Cada fila i de la matriz indica los pesos de salida que unen la neurona oculta i con cada neurona de salida.

Y es la matriz de salidas, cuyas celdas contienen un 0 o un 1. Esta matriz tiene una fila por cada instancia del bloque de datos de entrada (1, …, N) y una columna por cada neurona de la capa de salida (1, …, m). Cada celda (i,j) de la matriz contiene el valor de salida de cada neurona de salida j para la instancia i. Cada fila i de la matriz indica la neurona de salida que está activa para la instancia i.

Las matrices H e Y son conocidas. El objetivo es calcular β. Despejando en la fórmula 5.1 se obtiene la siguiente fórmula:

$$\beta = H^{-1}Y \qquad \textit{(Fórmula 5.3)}$$

Normalmente la matriz H no va a ser una matriz cuadrada y para calcular la inversa es necesario que la matriz sea cuadrada. Por este motivo, en este algoritmo se utiliza la **pseudo-inversa de Moore-Penrose** ($H^{-1} = H^{\dagger}$):

- Si $L > N$

$$H^{\dagger} = H^{T}(HH^{T})^{-1} \qquad \textit{(Fórmula 5.4)}$$

- Si $L = N$

$$H^{\dagger} = H^{-1} \qquad \textit{(Fórmula 5.5)}$$

- Si $L < N$

$$H^{\dagger} = (H^{T}H)^{-1}H^{T} \qquad \textit{(Fórmula 5.6)}$$

El pseudo-código de este algoritmo es el siguiente:

1. Generar a y b de forma aleatoria
2. Calcular la matriz H utilizando a y b
3. Calcular la pseudo-inversa $H^{\dagger}$
4. Obtener los pesos de salida β con la Fórmula 5.3

ELM es un algoritmo en modo tradicional. Este algoritmo lee los datos de un fichero almacenado localmente (con el método ***textFile(…)***), le aplica un pre-procesamiento, que consiste en separar los elementos de cada línea por comas (el fichero está en formato **CSV**), crear un ***RDD[LabeledPoint]*** (consta de dos elementos: atributos y etiqueta (último elemento de la línea)), **normalizar** el valor de cada atributo y, finalmente, **dividir** los datos en dos conjuntos (***train*** y ***test***). Con el conjunto de datos dividido, primero se entrena el modelo con los datos de *train* y luego se evalúa el modelo resultante con los datos de *test*. Para implementar el modelo ELM se han creado dos clases:

- ***ELMModel***: Es la clase que incluye todos los elementos de una red ELM, que son los pesos de entrada, los valores de bias y los pesos de salida. Esta clase incluye el método ***predict(…)***, que calcula la salida de la red neuronal para un dato de entrada (aplicando la fórmula 5.1).

- ***ELM***: Es la clase que implementa la actualización del modelo, con el método ***run(…)***. Este método recibe el bloque de datos de entrenamiento y a partir de ellos se aplica el pseudo-código del algoritmo ELM, utilizando operaciones paralelas. Finalmente devuelve el nuevo modelo entrenado.

5.4. OS-ELM

El algoritmo ELM solo se puede utilizar en modo tradicional (no se puede utilizar en streaming). Por este motivo surge el algoritmo OS-ELM [42] (*Online Sequential Extreme Learning Machine*), que es una versión incremental del anterior. OS-ELM puede actualizarse dato a dato o por bloques con un tamaño fijo o variable. En este caso, en lugar de tener un único bloque de datos se va a tener una secuencia de bloques de datos (*stream*) para entrenar el modelo.

Supongamos que se reciben bloques de tamaño N_k, donde k es el número del bloque (k = 0, 1, …). El tamaño de cada bloque puede variar (no tiene porque ser el mismo). Este algoritmo se divide en dos fases:

- **Fase de inicialización**: En esta fase se inicializa una red ELM con el bloque $\aleph_0 = \{X_i, Y_i\}_{i=1}^{N_0}$. Es necesario que se cumpla la condición $N_0 > L$ para poder realizar esta fase. Además, es necesario calcular H_0, y a partir de esta matriz calcular $K_0 = (H_0^T H_0)$. Finalmente, se obtiene el valor de los pesos de salida $\beta_0 = K_0^{-1} H_0^T Y_0$, donde:

$$H_0 = \begin{bmatrix} g(a_1, b_1, X_1) & \cdots & g(a_L, b_L, X_1) \\ \vdots & \ddots & \vdots \\ g(a_1, b_1, X_{N_0}) & \cdots & g(a_L, b_L, X_{N_0}) \end{bmatrix}_{N_0 x L}$$

$$\beta_0 = [\beta_1, \dots, \beta_L]^T = \begin{bmatrix} \beta_{11} & \cdots & \beta_{1m} \\ \vdots & \ddots & \vdots \\ \beta_{L1} & \cdots & \beta_{Lm} \end{bmatrix}_{Lxm}$$

$$Y_0 = [Y_1, \dots, Y_{N_0}]^T = \begin{bmatrix} Y_{11} & \cdots & Y_{1m} \\ \vdots & \ddots & \vdots \\ Y_{N_0 1} & \cdots & Y_{N_0 m} \end{bmatrix}_{N_0 xm}$$

- **Fase secuencial**: Esta fase actualiza incrementalmente la red ELM con el bloque $\aleph_k = \{X_i, Y_i\}_{i=N_{k-1}}^{N_k}$. Con este bloque de datos se obtiene la matriz H_k, y a partir de esta matriz se calcula $K_k = K_{k-1} + H_k^T H_k$. Finalmente, se calcula $\beta_k = \beta_{k-1} + K_k^{-1} H_k^T (Y_k - H_k \beta_{k-1})$, donde:

$$H_k = \begin{bmatrix} g(a_1, b_1, X_{N_{k-1}}) & \cdots & g(a_L, b_L, X_{N_{k-1}}) \\ \vdots & \ddots & \vdots \\ g(a_1, b_1, X_{N_k}) & \cdots & g(a_L, b_L, X_{N_k}) \end{bmatrix}_{N_k x L}$$

$$\beta_k = [\beta_1, \dots, \beta_L]^T = \begin{bmatrix} \beta_{11} & \cdots & \beta_{1m} \\ \vdots & \ddots & \vdots \\ \beta_{L1} & \cdots & \beta_{Lm} \end{bmatrix}_{Lxm}$$

$$Y_k = [Y_{N_{k-1}}, \dots, Y_{N_k}]^T = \begin{bmatrix} Y_{N_{k-1}1} & \cdots & Y_{N_{k-1}m} \\ \vdots & \ddots & \vdots \\ Y_{N_k 1} & \cdots & Y_{N_k m} \end{bmatrix}_{N_k x m}$$

En caso de que el tamaño del primer bloque coincida con el tamaño total de datos ($N_0 = N$), el resultado del algoritmo OS-ELM coincide con el de ELM, debido a que la fase de inicialización coincide con el proceso realizado en el algoritmo ELM.

El pseudo-código de este algoritmo es el siguiente:

1. Fase de inicialización:
   1.1. Generar a y b de forma aleatoria
   1.2. Calcular la matriz H_0 utilizando a y b
   1.3. Calcular $K_0 = (H_0^T H_0)$
   1.4. Calcular los pesos de salida $\beta_0 = K_0^{-1} H_0^T Y_0$
2. Fase secuencial:
   2.1. Calcular la matriz H_k utilizando a y b
   2.2. Calcular $K_k = K_{k-1} + H_k^T H_k$
   2.3. Calcular los nuevos pesos de salida $\beta_k = \beta_{k-1} + K_k^{-1} H_k^T (Y_k - H_k \beta_{k-1})$
   2.4. Volver a 2.1.

La implementación del algoritmo OS_ELM tiene dos versiones (incluidas en el *main*): la primera es en **modo tradicional** (para comparar con el algoritmo tradicional

ELM) y la segunda es en **modo *streaming***. La diferencia es que el modo tradicional lee los datos de un fichero almacenado localmente, los divide en dos conjuntos (***train*** y ***test***) y aplica el algoritmo, mientras que el segundo recibe los datos de Kafka por bloques y el algoritmo primero se evalúa con el bloque recibido y luego se entrena. Esta implementación también realiza el mismo pre-procesamiento que ELM. La implementación del modelo OS_ELM se ha realizado con dos clases:

- ***OS_ELMModel***: Es igual que ELMModel, pero incluyendo un atributo denominado ***K***, que se utiliza en la fórmula incremental de la fase secuencial. La forma de calcular la salida para un dato de entrada también es igual que en el algoritmo ELM.
- ***OS_ELM***: Esta clase implementa la actualización del modelo OS_ELMModel, con el método ***update(…)***. Este método recibe un bloque de datos de entrada y según en que fase se encuentre la ejecución se realiza una inicialización o una actualización del modelo. Para conocer si estamos en la fase de inicialización o no, se utiliza una variable denominada ***initialized***, que se inicializa a *false*. Cuando el modelo se inicializa se cambia su valor a *true*. Para actualizar el modelo se ha seguido el pseudo-código del algoritmo OS-ELM, utilizando operaciones paralelas. Finalmente, tras cada actualización se devuelve un modelo que incluye los valores de la red ELM y la matriz K, pera utilizarla en la fórmula incremental con el siguiente bloque de datos. Como es un método en *streaming* (continuamente se reciben bloque de datos), esta clase tiene dos métodos denominados ***trainOn*** y ***testOn***. Cada método se ejecuta con cada nuevo bloque de datos que llega al sistema. Además incluye una función denominada ***trainModel(…)***, que se utiliza para entrenar el modelo en los algoritmos de *ensemble*.

5.5. EOS-ELM

La utilización de *ensembles* de varios clasificadores permite que se tenga una mejor generalización de los datos. Un método representativo de *ensemble* es EOS-ELM [44] (*Ensemble of OS-ELM*), que es un *ensemble* de redes OS-ELM. Como se puede apreciar en la Figura 5.3 el *ensemble* está formado por P clasificadores OS-ELM, cada uno de ellos inicializado de forma aleatoria (son diferentes). Cada vez que llega al sistema un bloque de datos, cada clasificador del *ensemble* se actualiza con él. Como cada clasificador es diferente, aunque se entrenen con los mismos datos el resultado va a seguir siendo diferente. Cuando hay que clasificar un dato, cada clasificador toma una decisión y el resultado final del *ensemble* será la media de todas las soluciones.

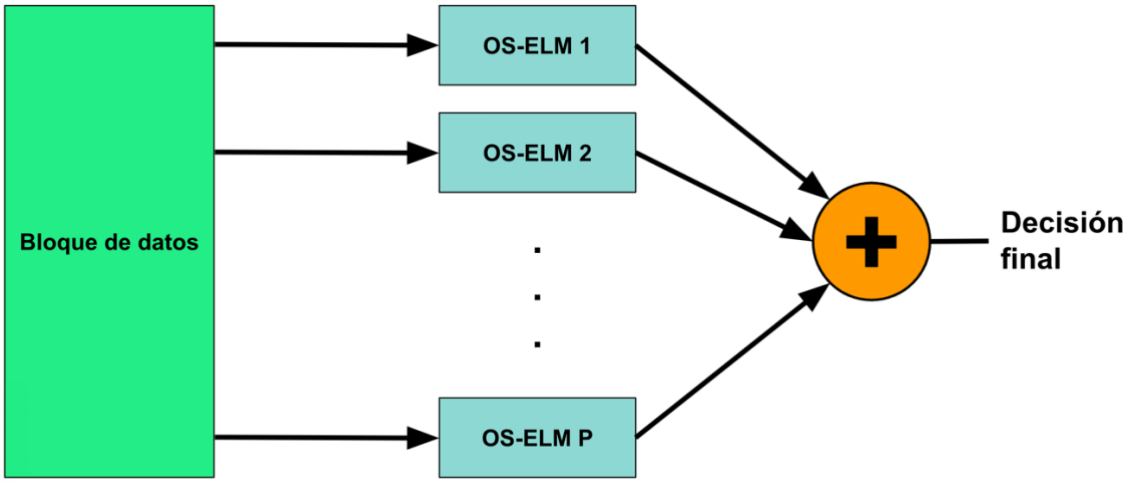

Figura 5.3. Ensemble de OS-ELM

El pseudo-código de este algoritmo es el siguiente:

1. `Inicializar P redes OS-ELM aleatoriamente`
2. `Actualizar las P redes con el nuevo bloque de datos (inicialización/secuencial)`
3. `Volver a 2`

Utilizando la implementación del algoritmo OS_ELM, se implementa el método de *ensemble* EOS-ELM. El método *main* es idéntico a la parte de *streaming* del algoritmo OS_ELM (cambiando el modelo utilizado por EOS-ELM). Para el modelo se han utilizado dos clases:

- ***EOS_ELMModel***: Esta clase incluyo un array de redes OS_ELM y dos funciones para calcular la salida del ensemble para un dato de entrada: ***predict*** (EOS-ELM) y ***predict2*** (VOS-ELM). Esta clase incluye el *predict* del algoritmo VOS-ELM porque los algoritmos son idénticos, excepto la evaluación de los datos. Para agilizar las pruebas, el *ensemble* se entrena con los bloques y al evaluar se utilizan las dos funciones para obtener los resultados de cada algoritmo.
- ***EOS_ELM***: Clase que actualiza el modelo EOS_ELMModel, con el método ***update(…)***. Cuando el modelo se inicializa o se actualiza, este método contiene un bucle ***for*** que actualiza cada clasificador OS_ELM con el bloque actual. El método de actualización para cada clasificador es el indicado en OS_ELM, debido a que se invoca la función ***trainModel*** del OS_ELM.

5.6. VOS-ELM

El algoritmo EOS-ELM es un algoritmo de *ensemble* y como tal utiliza una estrategia de combinación de las soluciones parciales de cada clasificador para generar la solución final. En un *ensemble* se pueden utilizar diferentes estrategias para tomar la decisión final y la elección de la misma es un factor muy importante que afecta al rendimiento del *ensemble*. Por este motivo VOS-ELM [47] (*Voting based OS-ELM*) fue propuesto para analizar el algoritmo EOS-ELM utilizando otra estrategia diferente. La estrategia utilizada por el algoritmo VOS-ELM es el voto mayoritario, es decir, cada clasificador toma una decisión, y en lugar de hacer la media se elige la más votada. En el artículo en el que se propone este algoritmo se puede apreciar que VOS-ELM iguala e incluso supera la precisión de EOS-ELM.

La implementación del algoritmo VOS-ELM es idéntica al algoritmo EOS-ELM, pero utilizando la función ***predict2(…)***, que combina las soluciones mediante la estrategia de voto mayoritario. Cada clasificador toma una decisión y al final se elige la decisión más votada.

5.7. CD-VOS-ELM

El algoritmo EOS-ELM actualiza todos los clasificadores del *ensemble* con cada nuevo bloque que llega al sistema, independientemente de si existe o no un cambio de concepto. Este algoritmo CD-VOS-ELM (*VOS-ELM with Concept Drift detection*) es el mismo algoritmo, pero añadiéndole un método de detección de cambio de concepto basado en el error. Este método de detección está inspirado en el algoritmo CELM [52], que define dos niveles de decisión (*warning* y *drift*), como muestra la Figura 5.4. Con cada nuevo bloque que llega al sistema, primero se evalúa con el modelo actual y se obtiene el error. Si este error está por debajo del nivel de *warning*, el *ensemble* no se actualiza porque se considera que no hay cambio de concepto. Si el error está entre el nivel de *warning* y el de *drift*, se considera que es posible que exista un cambio de concepto, pero no se puede asegurar, por lo que en este caso se actualizan los clasificadores del *ensemble*. Finalmente, si el error está por encima del nivel de *drift*, se descarta el *ensemble* actual y se genera uno nuevo desde cero.

El pseudo-código de este algoritmo es el siguiente:

1. Obtener el error de *ensemble* con el bloque de datos actual
2. Si el error es inferior al nivel de *warning*, no hacer nada
3. Si el error está entre los niveles de *warning* y *drift*, actualizar *ensemble*
4. Si el error es superior al nivel *drift*, eliminar *ensemble* y generar nuevo
5. Volver a 1

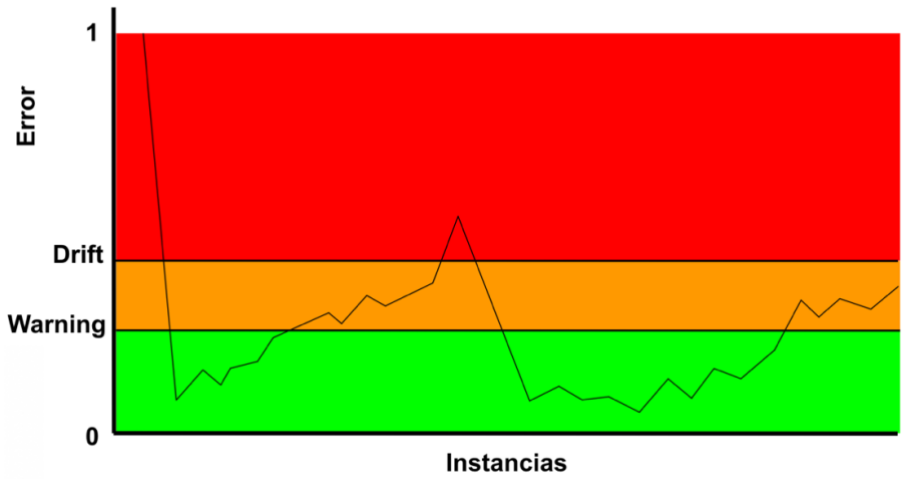

Figura 5.4. Método de detección de cambio de concepto

Este algoritmo también es muy similar al algoritmo VOS-ELM, ya que es el mismo algoritmo, pero añadiendo un mecanismo de detección de cambio de concepto (basado en el error). El único cambio de su implementación está en el método ***trainOn(…)***, que es el que se invoca cuando se recibe un nuevo bloque de datos. En este caso, primero se evalúa el modelo con el nuevo bloque de datos y después se realizan dos comprobaciones:

- Si el error es mayor o igual que el límite superior (***drift***), se descarta el modelo (la variable *initializaed* se establece a *false*).
- Si el error se superior al límite inferior (***warning***), se actualiza el modelo con el nuevo bloque de datos.

De esta forma, si el error supera el límite de *drift* también va a superar el límite de *warning*, por lo que primero comprobamos si el modelo se descarta o no y luego comprobamos si hay que invocar el método ***update(…)*** o no.

5.8. Experimentación y análisis de los resultados

Una vez descrita la implementación de los distintos algoritmos, en esta sección se va a comparar el rendimiento de los algoritmos implementados utilizando diferentes *datasets*, tanto sintéticos como reales. Para realizar esta experimentación se ha utilizado un clúster Spark de 5 máquinas y una máquina extra para apache Kafka, Zookeeper y productor (solo se utiliza en la parte de *streaming*). La instalación y configuración se ha realizado de la misma forma que en la sección 4. En esta experimentación todos los nodos del clúster Spark tienen una configuración homogénea. Cada nodo tiene un procesador Intel Core i7-2600 3.40GHz, 8 núcleos y 4GB de memoria RAM. El nodo utilizado para ejecutar apache Kafka, Zookeeper y el productor Kafka tiene un procesador Intel Core i7-8700 3.20GHz, 12 núcleos y 16GB de memoria RAM. Además, se han utilizado conexiones de 1GB/s entre los nodos.

La experimentación se va a dividir en dos partes. En la primera se va a comparar el algoritmo ELM (tradicional) con su versión en *streaming* (OS-ELM), realizando las ejecuciones en un entorno tradicional. En la segunda comparativa se evalúa en un entorno en *streaming* los algoritmos OS-ELM, EOS-ELM, VOS-ELM y CD-VOS-ELM,

midiendo sus rendimientos (error y tiempo de entrenamiento) y mostrando cual ofrece un mejor comportamiento sobre los diferentes *datasets* utilizados.

Los *datasets* utilizados son los siguientes:

- Reales: Obtenidos del repositorio UCI [53].
  - **Satellite Image (SatImage)**: Contiene una serie de imágenes multi-espectrales tomadas por satélite. Cada dato del *dataset* se corresponde con una región de 3x3 píxeles. El objetivo es clasificar el píxel central de cada región en una de las 6 clases disponibles.
  - **Diabetes**: Fue generada en el laboratorio de físicas aplicadas de la universidad de Johns Hopkins. El *dataset* contiene información sobre 768 mujeres sobre la edad de 21 años residentes en Phoenix, Arizona. El objetivo es detectar si un paciente muestra signos de diabetes según el criterio de la organización mundial de la salud.
  - **Letter**: Contiene imágenes de letras basadas en 20 fuentes diferentes que están representadas por 16 atributos que han sido escaladas en el rango entre 0 y 15. El objetivo es identificar a qué letra de las 26 letras en el alfabeto inglés pertenece cada dato.
  - **Poker**: Cada instancia contiene un ejemplo de mano que consiste en 5 cartas de una baraja de 52 cartas. Cada carta se describe utilizando 2 atributos (palo y número), por lo que cada instancia tiene un total de 10 atributos. El objetivo es clasificar cada instancia para ver que tipo de mano es, teniendo en cuenta que el orden de las cartas es importante.
- Sintéticos: Generados con la herramienta MOA [5].
  - **SEA**: Cada instancia contiene 3 atributos decimales, de los que solo 2 son relevantes: $x_i \in [0,10]$, con i=1,2,3. El concepto es $x_1+x_2 \leq \beta$, con $\beta \in \{7,8,9,9.5\}$. Si se cumple la expresión pertenece a la clase negativa y sino a la positiva. Cada uno de estos 4 conceptos se puede apreciar en la Figura 5.5.

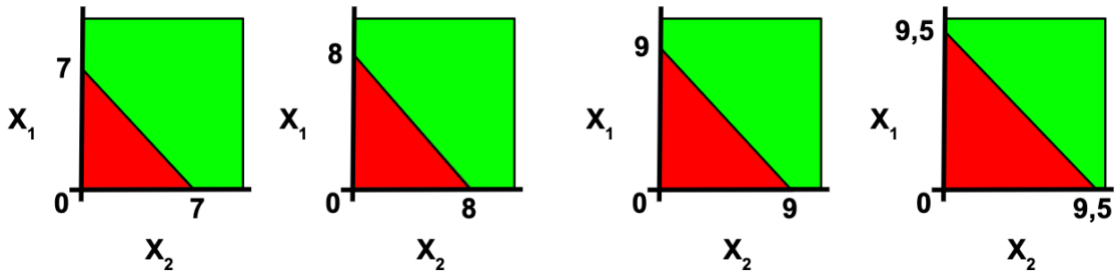

Figura 5.5. SEA: Tipos de conceptos

- **SINE**: Cada instancia contiene 4 atributos decimales, de los que solo 2 son relevantes: $x_i \in [0,1]$, con i=1,2,3,4. Los 4 conceptos disponibles son (Figura 5.6):
  - $x_2 < \sin(x_1)$
  - $x_2 \geq \sin(x_1)$
  - $x_2 < 0.5 + 0.3\ \sin(3\pi x_1)$
  - $x_2 \geq 0.5 + 0.3\ \sin(3\pi x_1)$

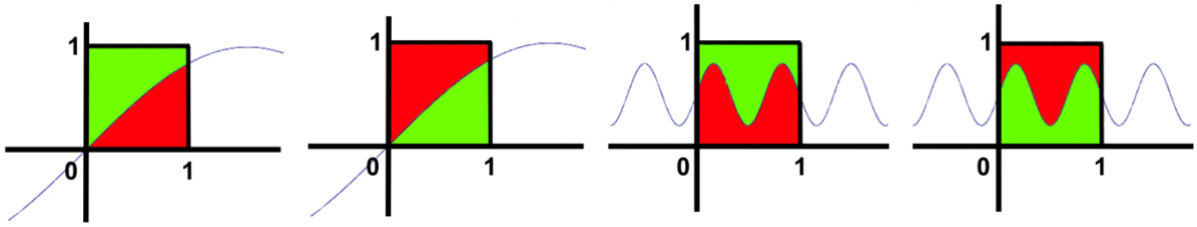

Figura 5.6. SINE: Tipos de conceptos

De estos *datasets* solo se ejecutan en paralelo los *datasets* artificiales, ya que los reales no tienen muchas instancias y el sistema no escala cuando se tienen pocas instancias. Esto ocurre porque se deben hacer envíos entre las máquinas para poder paralelizar y con pocas instancias se emplea más tiempo enviando datos que procesando. Los *datasets* pequeños se han utilizado para comparar el rendimiento de las implementaciones de este trabajo con las versiones originales y ver que se obtienen resultados similares.

5.8.1. Experimentación en un entorno tradicional

En esta experimentación se van a ejecutar los algoritmos ELM y OS-ELM en un entorno tradicional. Esto significa que los datos van a estar almacenados localmente y todo el *dataset* está disponible desde el principio. El objetivo es comprobar que los

algoritmos generan resultados similares a los originales propuestos en [41] [42] y que el sistema escala los tiempos de procesamiento al aumentar el número de nodos del clúster. Los *datasets* utilizados en esta experimentación se han dividido en un conjunto de *train* y otro de *test*. Los detalles de los *datasets* utilizados en esta prueba se muestran en la Tabla 5.3.

Dataset	Nº atributos	Nº clases	Nº instancias	Train	Test
SEA (Abrupto)	3	2	1.000.000	100.000	900.000
SEA (Gradual)	3	2	1.000.000	100.000	900.000
SatImage	36	6	6.435	4.435	2.000
Diabetes	8	2	768	576	192
Letter	16	26	20.000	18.000	2.000
Poker	10	10	1.025.010	100.000	925.010

Tabla 5.3. Detalles de los datasets utilizados en la experimentación tradicional

Para preparar la experimentación, primero se han copiado los *datasets* localmente en un directorio fijo (en este caso *~/spark/datasets*) y se ha enviado el *jar* con el código de los algoritmos. Para crear el *jar* se ha utilizado el comando "***sbt assembly***" y para enviarlo a una máquina el comando ***scp***. Junto con el *jar* está el archivo "***spark-ELM.conf***" que contiene los parámetros de configuración necesarios para ejecutar la aplicación. En este caso hay que especificar, como mínimo, los valores que aparecen en la Tabla 5.4.

Parámetro	Valor
EXP_NUMBER	El número de ejecución [1, 2, …, 10]
SEEDS	5,7,11,13,17,19,23,29,31,37
SPARK_MASTER	local[*]
SPARK_BATCH_INTERVAL	1
NODES	1
CORES_NODE	8
n	Número de atributos
L	Número de neuronas de la capa oculta
m	Número de clases
MAX	*Array* con el valor máximo para cada atributo de entrada
MIN	*Array* con el valor mínimo para cada atributo de entrada
FILE_PATH	Ruta del archivo con los datos de entrada
TRAIN_PROPORTION	Valor entre 0 y 1 que indica la proporción de datos de entrenamiento
TEST_PROPORTION	Valor entre 0 y 1 que indica la proporción de datos de evaluación

Tabla 5.4. Parámetros de configuración de la experimentación tradicional

Cuando todos estos archivos están en el nodo maestro y se configura la ejecución, se ejecuta el comando ***"~/spark/bin/spark-submit --master spark://Spark_Master:7077 --class [algoritmo] [ruta al jar]***" para iniciar cada prueba.

Para esta experimentación se han realizado 10 ejecuciones con cada *dataset* utilizando una semilla diferente en cada una (5, 7, 11, 13, 17, 19, 23, 29, 31, 37), y para obtener el resultado final se ha hecho la media y la desviación típica. En cada ejecución el algoritmo utilizado entrena con todo el conjunto de *train* y luego evalúa el error obtenido sobre el conjunto de *test*. Para un *dataset*, se configuran todos los parámetros anteriores y, para cada una de las 10 pruebas, solo se cambia el valor de EXP_NUMBER. Tras realizar las 10 ejecuciones de un *dataset* se repite el proceso con otro, hasta analizar todos.

La Tabla 5.5 muestra el error y la desviación típica obtenidas en los *datasets* SatImage, Diabetes, Letter y Poker, para ambos algoritmos. Para estas ejecuciones se ha utilizado el mismo número de neuronas ocultas que en los artículos originales, excepto en el *dataset* Letter, que por limitaciones de RAM no se han podido utilizar las 721 neuronas. Como se puede observar los valores son muy similares a los de las propuestas originales. En el *dataset* SatImage se obtiene un peor resultado que el original y en Diabetes se mejora el error obtenido. Para el *dataset* Letter se obtiene un peor resultado debido a que se utilizan menos neuronas, pero aun así el resultado se acerca al original. Estas variaciones en los resultados pueden deberse a que no se han utilizado las mismas semillas, ya que los artículos originales no las especifican. En el caso de Poker no hay artículos que lo utilicen con estos algoritmos, pero como es un problema de clasificación multiclase muy utilizado se ha querido ofrecer un análisis del mismo, además de por ser uno de los más grandes reales. Como se puede observar el error es casi un 50%. Esto se debe a que el *dataset* no está balanceado y estos algoritmos no están preparados para problemas de este tipo.

Dataset	L	Nº nodos	ELM	OS-ELM	Originales
SatImage	400	Local	0,1515 ± 0,0026	0,1515 ± 0,0026	**0,1103**
Diabetes	20	Local	**0,2078 ± 0,0108**	**0,2078 ± 0,0108**	0,2346
Letter	200	Local	0,2077 ± 0,0047	0,2077 ± 0,0047	**0,1222* (L=721)**
Poker	25	Local	**0,4934 ± 0,0058**	**0,4934 ± 0,0058**	---

Tabla 5.5. Comparativa ELM vs versiones originales (error ± desviación típica)

Los resultados para los *datasets* SEA (Abrupto) y SEA (Gradual) se muestran en la Tabla 5.6. En este caso los *datasets* se han ejecutado con diferente número de nodos, para comprobar que el error se mantiene prácticamente igual y el tiempo de entrenamiento disminuye. En el caso del error hay pequeñas variaciones porque se ha utilizado una función de MLLIB (svd) y no siempre devuelve el mismo resultado (probablemente por el orden en el que se distribuyen los datos entre los nodos). Aun

así, el error obtenido es prácticamente igual al aumentar el número de nodos, por lo que se puede afirmar que el error se mantiene.

Dataset	L	Nº nodos	ELM	OS-ELM
SEA (Abrupto)	25	Local	**0,1124 ± 0,0065**	**0,1124 ± 0,0065**
		2	0,1095 ± 0,0026	**0,1094 ± 0,0025**
		3	0,1106 ± 0,0029	**0,1099 ± 0,0025**
		4	0,1114 ± 0,0036	**0,1100 ± 0,0025**
SEA (Gradual)	25	Local	**0,1130 ± 0,0061**	**0,1130 ± 0,0061**
		2	0,1118 ± 0,0078	**0,1096 ± 0,0019**
		3	**0,1096 ± 0,0013**	0,1112 ± 0,0042
		4	0,1111 ± 0,0033	**0,1099 ± 0,0033**

Tabla 5.6. Resultados obtenidos para ELM vs OS-ELM

Finalmente, los tiempos de ejecución para todos los *datasets* analizados se muestra en la Tabla 5.7. Como se puede apreciar en los datasets SEA (Abrupto) y SEA (Gradual), al aumentar el número de nodos los tiempos de entrenamiento disminuyen. Esta disminución del tiempo es mayor cuando el clúster pasa de tener un nodo a tener dos. Después la disminución de tiempo es cada vez menor, debido a que cada vez hay más tiempo de envío de datos entre las máquinas del clúster.

Dataset	L	Nº nodos	ELM (ms)	OS-ELM (ms)
SatImage	400	Local	**83.447**	118.539
Diabetes	20	Local	**2.178**	2.309
Letter	200	Local	**212.399**	263.721
Poker	25	Local	**79.664**	86.739
SEA (Abrupto)	25	Local	70.358	**68.590**
		2	**46.317**	46.496
		3	**38.959**	39.131
		4	35.211	**34.889**
SEA (Gradual)	25	Local	69.314	**67.704**
		2	47.379	**46.247**
		3	**38.582**	38.923
		4	35.720	**35.126**

Tabla 5.7. Tiempos obtenidos para ELM vs OS-ELM

Como conclusiones de esta experimentación se puede destacar que los resultados de las implementaciones distribuidas realizadas en este trabajo son similares a los originales, con variaciones por las semillas utilizadas. Además, los valores de error se mantienen al aumentar el número de nodos y el tiempo se va reduciendo, hasta un determinado número de nodos. Finalmente, en el caso de *datasets* con pocos datos se invierte más tiempo enviando que procesando, por lo que los tiempos obtenidos son elevados y al añadir nodos no se mejoran.

5.8.2. Experimentación en un entorno streaming

En esta experimentación se van a ejecutar los algoritmos OS-ELM, EOS-ELM, VOS-ELM y CD-VOS-ELM en un entorno *streaming*. Esto significa que los datos van a llegar al sistema en tiempo real a través de Kafka, por lo que no se tiene todo el *dataset* disponible desde el principio. El objetivo es comprobar que los algoritmos generan resultados similares a los originales propuestos en [42] [44] [47] y que, al estar implementados de forma que se pueden ejecutar en paralelo, permiten mejorar el procesamiento de grandes volúmenes de datos, ya que el tiempo de procesamiento disminuye a medida que se aumenta el número de nodos en el clúster. Los *datasets* SatImage, Diabetes y Letter se han dividido en un conjunto de *train* y otro de *test*, debido a que son *datasets* con muy pocos datos. El resto de *datasets* no se han dividido porque se va a aplicar la técnica *Test-then-Train*, en la que se procesa el *dataset* completo por bloques (primero se evalúa y luego se entrena). Los detalles de los *datasets* utilizados en esta prueba se muestran en la Tabla 5.8.

Dataset	Nº atributos	Nº clases	Nº instancias	Instancias Train	Instancias Test
SEA (Abrupto)	3	2	5.000.000	---	---
SEA (Gradual)	3	2	5.000.000	---	---
SINE	4	2	5.000.000	---	---
SatImage	36	6	6.435	4.435	2.000
Diabetes	8	2	768	576	192
Letter	16	26	20.000	18.000	2.000
Poker	10	10	1.025.010	---	---

Tabla 5.8. Detalles de los datasets utilizados en la experimentación en streaming

Para poder realizar las pruebas hay que copiar el productor y los *datasets* en el nodo encargado de ello. El productor tiene un archivo asociado denominado "***producer.conf***" que tiene 2 líneas: la primera indica el número de particiones (número de núcleos totales) y la segunda la ruta del *dataset* que se va a enviar. Estos valores dependen del número de nodos del clúster y del *dataset* que se está analizando.

En cuanto al clúster de Spark, para cada ejecución, si ha sido en local, no ha sido necesario iniciar el clúster. Para las ejecuciones con 2, 3 o 4 nodos, el clúster solo tenía ese mismo número de nodos activos. Para ir cambiando el número de nodos activos hay que ir al archivo ***"~/spark/conf/slaves***" y en el comentar o descomentar los nodos necesarios. Para iniciar el clúster se ha utilizado el comando "**~/spark/sbin/start-all.sh**" y para detenerlo el comando "**~/spark/sbin/stop-all.sh**".

Además, hay que modificar la configuración en el fichero "**~/spark/spark-ELM.conf**", asignándole los valores que aparecen en la Tabla 5.9.

Parámetro	Descripción
EXP_NUMBER	El número de ejecución [1, 2, …, 10]
SEEDS	5,7,11,13,17,19,23,29,31,37
SPARK_MASTER	spark://Spark-Master:7077
SPARK_BATCH_INTERVAL	1
SPARK_KAFKA_RATE	true
KAFKA_BROKERS	[IP_Kafka]:9092
TOPIC	TRAIN
KAFKA_RATE	100.000
NODES	[1, 2, 3, 4]
CORES_NODE	8
n	Número de atributos
L	Número de neuronas de la capa oculta
m	Número de clases
P	5
MAX	*Array* con el valor máximo que puede tomar cada atributo
MIN	*Array* con el valor mínimo que puede tomar cada atributo
lUp	0.15
lDown	0.05

Tabla 5.9. Parámetros de configuración de la experimentación streaming

Debido a la mayor duración de las pruebas, en este caso los resultados son la media de 3 ejecuciones. Los algoritmos EOS-ELM y VOS-ELM son idénticos, excepto en la técnica de predicción, por lo que para su evaluación solo se ha ejecutado el algoritmo EOS-ELM y se ha evaluado con las dos estrategias a la vez, para obtener los resultados de ambos algoritmos. Por este motivo el tiempo de ejecución de ambos es el mismo.

Esta experimentación se ha dividido en dos partes. En esta primera parte se analizan los *datasets* SatImage, Diabetes y Letter para comparar los resultados con las versiones originales. En cada ejecución, el algoritmo entrena con todo el conjunto de *train* por bloques, es decir, divide el conjunto de *train* en bloques del mismo tamaño y los procesa en orden de uno en uno (el primer bloque inicializa el modelo y el resto lo actualiza). Después de entrenar, se evalúa el conjunto de *test* y se obtiene el error. Como en este caso el tiempo tampoco escala debido al número de datos, solo se han ejecutado en local (1 nodo). En SatImage y Latter el valor de P es 5 y en Diabetes es 7. Para ejecutar cada prueba, primero se ha iniciado la aplicación de Spark *Streaming* y, tras 30 segundos, se ha iniciado el productor para enviar los datos.

Los resultados de las implementaciones de este trabajo se muestran en la Tabla 5.10 y los de las implementaciones originales en la Tabla 5.11. Los tiempos de

entrenamiento aparecen en la Tabla 5.12. Se puede observar que son prácticamente iguales, con algunas variaciones debidas a las semillas utilizadas, que en este caso tampoco se especifican en la bibliografía existente. En general se puede observar como las técnicas de *ensemble* superan por poca diferencia a la utilización de un único modelo, pero el tiempo de entrenamiento aumenta considerablemente. Esto hace que se pueda considerar si merece la pena el uso de un único modelo o un *ensemble*. En los datasets SatImage y Letter el algoritmo VOS-ELM obtiene mejores resultados que EOS-ELM. Esto es porque son *datasets* multiclase y en ellos la técnica que utiliza VOS-ELM es mejor.

Dataset	L	Tam. Bloque	OS-ELM	EOS-ELM	VOS-ELM	CD-VOS-ELM
SatImage	400	20	0,1295±0,0023	0,1537±0,0062	0,1325±0,0028	0,1325±0,0028
Diabetes	20	20	0,2066±0,0030	0,2153±0,0120	0,2153±0,0120	0,2153±0,0120
Letter	200	500	0,1963±0,0032	0,2628±0,0028	0,1808±0,0043	0,1808±0,0043

Tabla 5.10. Resultados streaming de las nuevas implementaciones

Dataset	OS-ELM	EOS-ELM	VOS-ELM	CD-VOS-ELM
SatImage	0,111	0,1099	---	---
Diabetes	0,2254	---	0,2176	---
Letter	0,2295	0,236	0,2176	---

Tabla 5.11. Resultados streaming de las implementaciones originales

Dataset	L	Tam. Bloque	OS-ELM	EOS-ELM	VOS-ELM	CD-VOS-ELM
SatImage	400	20	600.977	3.077.930	3.077.930	3.061.363
Diabetes	20	20	12.879	63.082	63.082	62.920
Letter	200	500	78.434	363.641	363.641	383.537

Tabla 5.12. Tiempos de entrenamiento streaming vs original

En la siguiente experimentación se utiliza la técnica *Test-then-Train* para ver como evoluciona el error a lo largo del tiempo y como se adapta cada modelo ante una situación de cambio de concepto. Los *datasets* SEA (Abrupto) y SINE tienen un cambio de concepto abrupto justo en la mitad del *dataset*, mientras que en SEA (Gradual) tiene un cambio de concepto gradual que empieza en la instancia 1.500.000 y el cambio se produce hasta la 3.500.000, donde se estabiliza el nuevo concepto. Para los *datasets* SEA el cambio de concepto es de $\beta = 9.5$ a $\beta = 7$. En SINE el cambio es de la función 1 a la función 3. Estos 3 *datasets* son procesados por bloques de 100.000 instancias porque los recursos *hardware* disponibles no permiten un tamaño mayor, pero el *dataset* Poker se procesa en bloques de 50.000 debido a que tiene menos instancias, además de generar más puntos para representar en la gráfica. En los 4 *datasets* el valor de P es 5. Para todos los algoritmos el valor de L es 25.

En la Figura 5.7 se muestran los resultados obtenidos para SEA (Abrupto) con el algoritmo OS-ELM utilizando diferente número de nodos en el clúster. Como se puede observar se detecta el cambio de concepto, pero se adapta poco a poco. Esto se debe a que utiliza una fórmula incremental y el modelo se adapta a una velocidad constante.

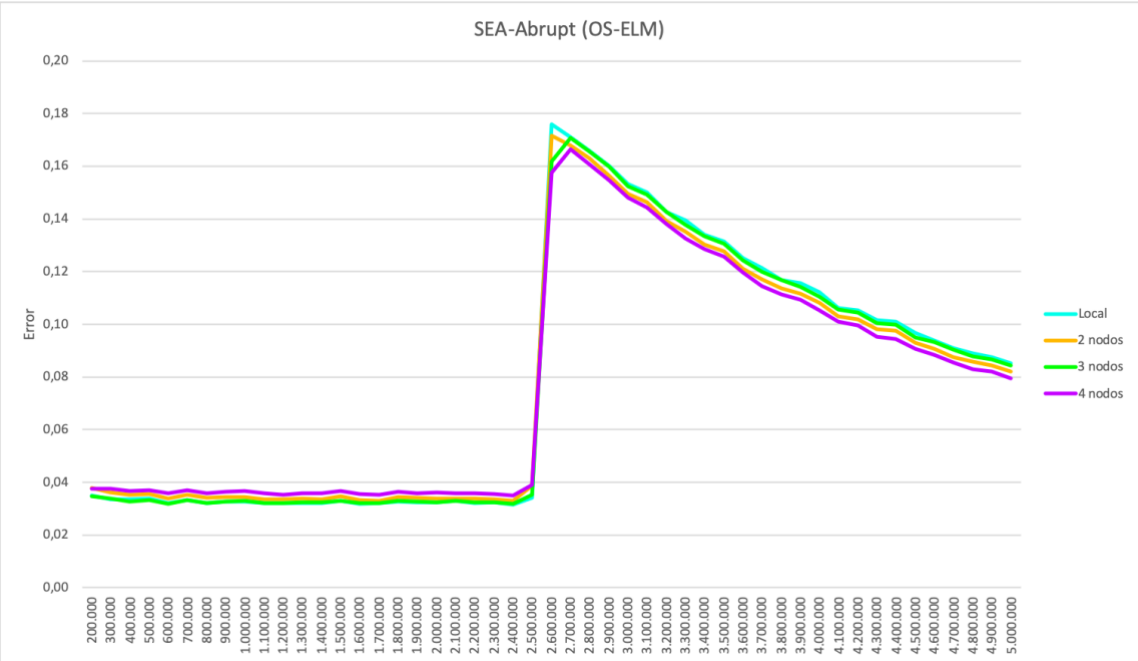

Figura 5.7. Test-thenTrain en SEA (Abrupto) con OS-ELM

En la Figura 5.8 se muestran los resultados del algoritmo EOS-ELM para el *dataset* SEA (Abrupt). En este caso el error es algo inferior que en OS-ELM, pero en el cambio de concepto se sigue adaptando a una velocidad constante, al igual que OS-ELM.

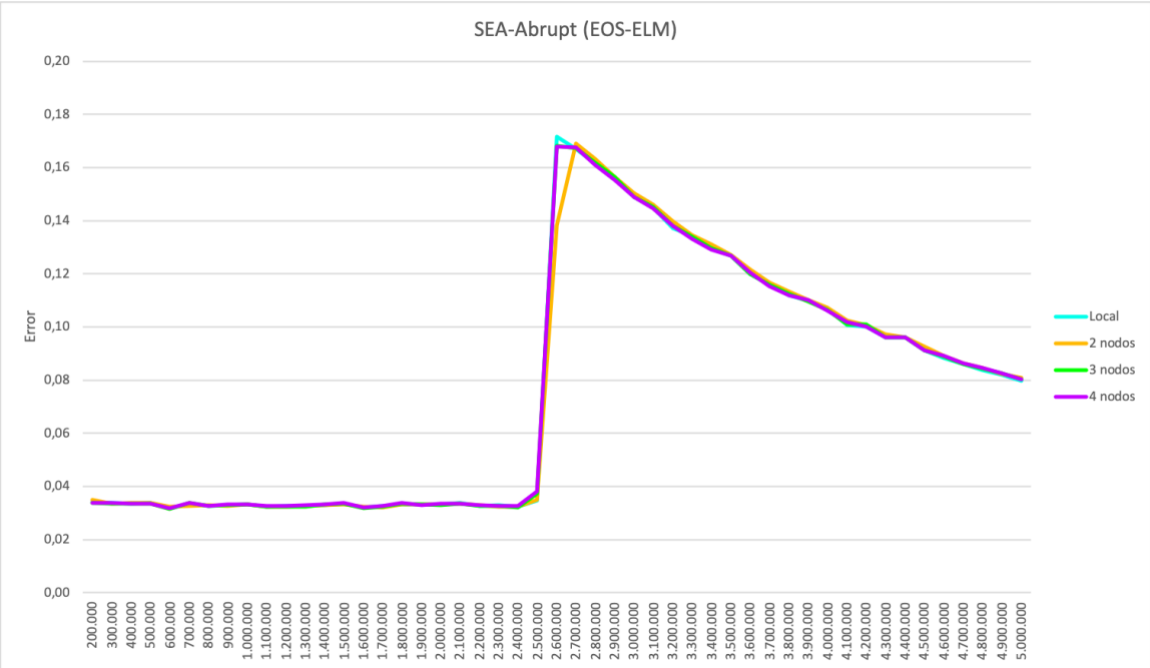

Figura 5.8. Test-thenTrain en SEA (Abrupto) con EOS-ELM

Como el *dataset* SEA (Abrupto) solo tiene dos clases, al utilizar EOS-ELM y VOS-ELM se obtienen los mismos resultados. Los resultados se muestran en la Figura 5.9.

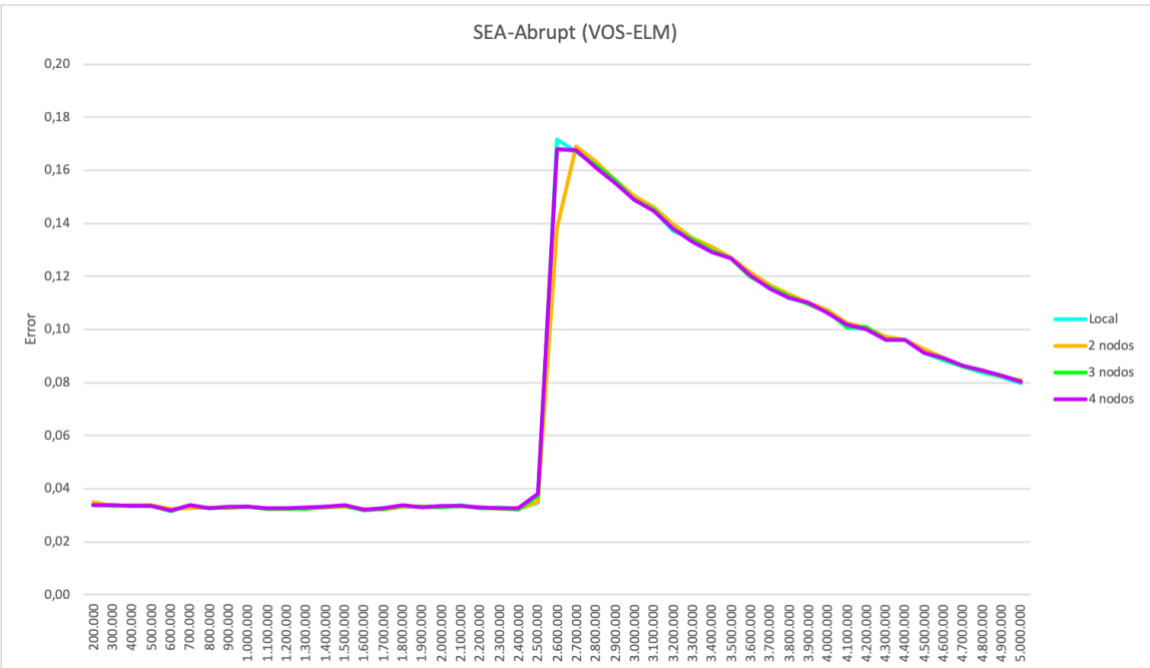

Figura 5.9. Test-thenTrain en SEA (Abrupto) con VOS-ELM

Como se puede apreciar en la Figura 5.10, en este caso el algoritmo se adapta muy rápido al nuevo concepto. Esto es porque se han definido los límites superior e inferior a 0,15 y 0,05, respectivamente. Cuando se produce el cambio de concepto, el

error sube por encima de 0,15. Al superar el límite superior, el *ensemble* se descarta y se genera otro con los nuevos datos. Como el nuevo bloque solo tiene datos del segundo concepto, el modelo se adapta al instante al nuevo concepto.

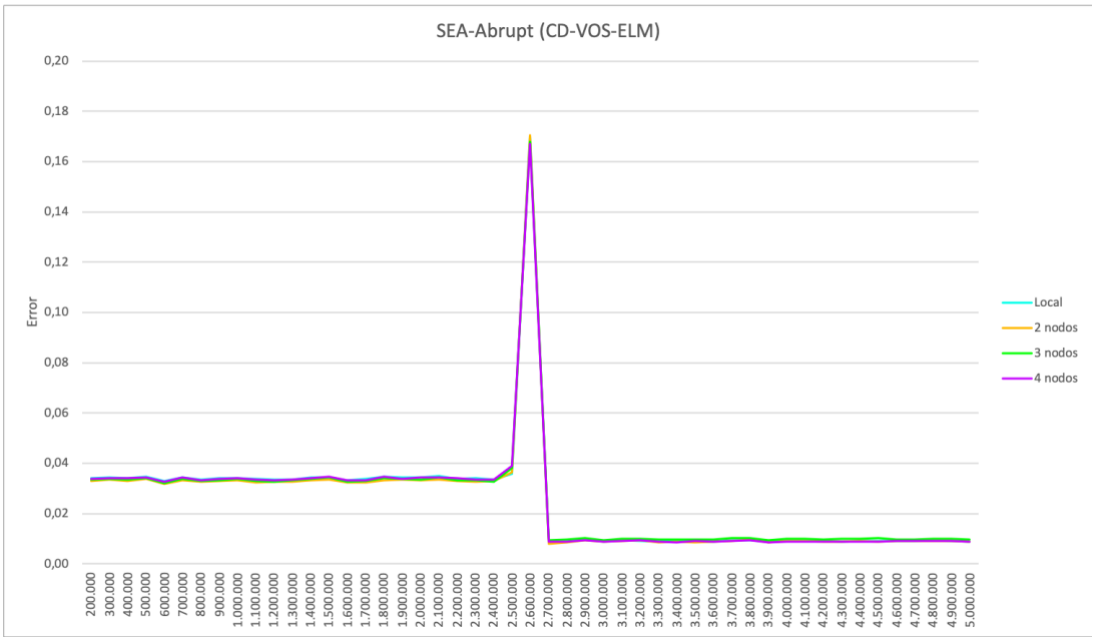

Figura 5.10. Test-thenTrain en SEA (Abrupto) con CD-VOS-ELM

En la Figura 5.11 se puede observar el comportamiento de cada algoritmo con el *dataset* SEA (Abrupto). OS-ELM es el que mayor error tiene de todos, seguido de EOS-ELM y VOS-ELM y, finalmente, el que mejor se adapta es CD-VOS-ELM, gracias a su mecanismo de detección de cambio de concepto.

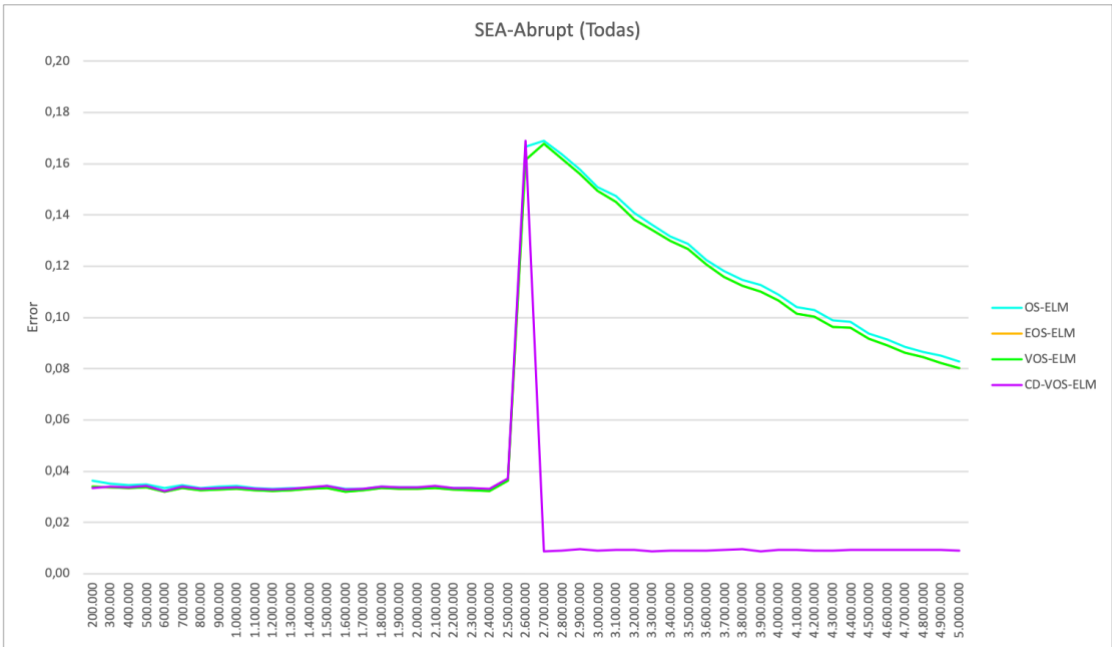

Figura 5.11. Test-thenTrain en SEA (Abrupto) con todos

Tras analizar el dataset SEA (Abrupto), se va a proceder a analizar el dataset SEA (Gradual). Como se puede observar en la Figura 5.12, el error aumenta poco a poco en lugar de aumentar de golpe, como en el caso anterior. Como el modelo se va adaptando continuamente, el error no tiene una subida muy elevada. El error aumenta mientras se produce el cambio de concepto, hasta que se empieza a estabilizar el nuevo concepto. Cuando el segundo concepto es estable, el error comienza a disminuir.

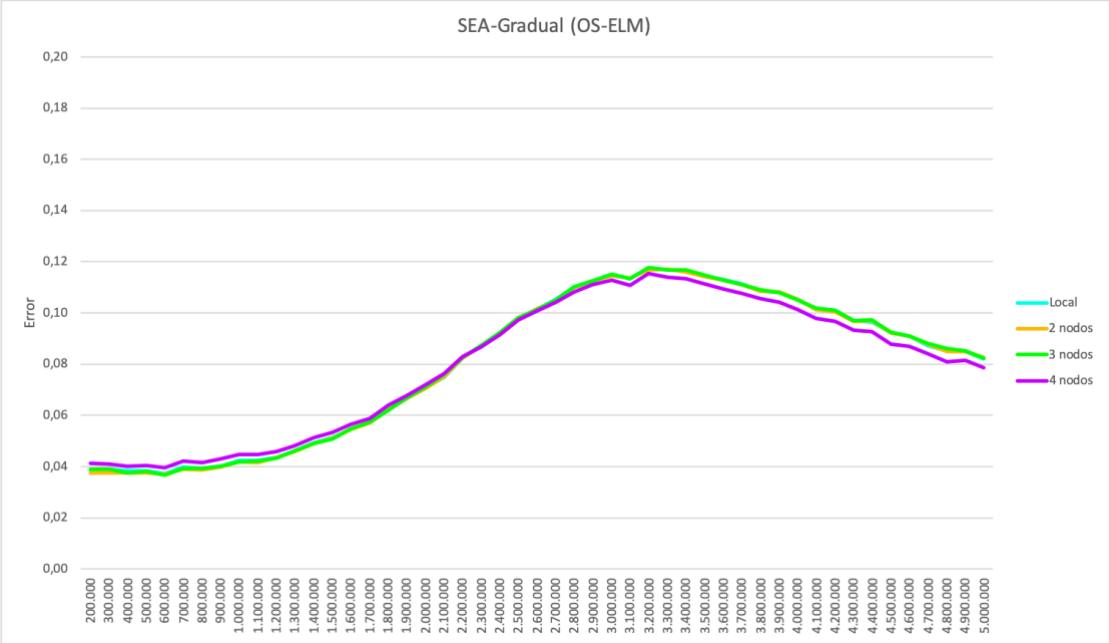

Figura 5.12. Test-thenTrain en SEA (Gradual) con OS-ELM

Como se puede apreciar en la Figura 5.13, el error de EOS-ELM es algo menor que en OS-ELM y se mantiene más estable. La respuesta del algoritmo es igual que en OS-ELM, adaptándose poco a poco al nuevo concepto.

Al igual que en el caso de SEA (Abrupto), la Figura 5.14 muestra que la respuesta de VOS-ELM es exactamente igual que EOS-ELM. Esto se debe a que este *dataset* también es binario (solo tiene dos clases), por lo que hacer la media de las respuestas y elegir el más votado genera el mismo resultado.

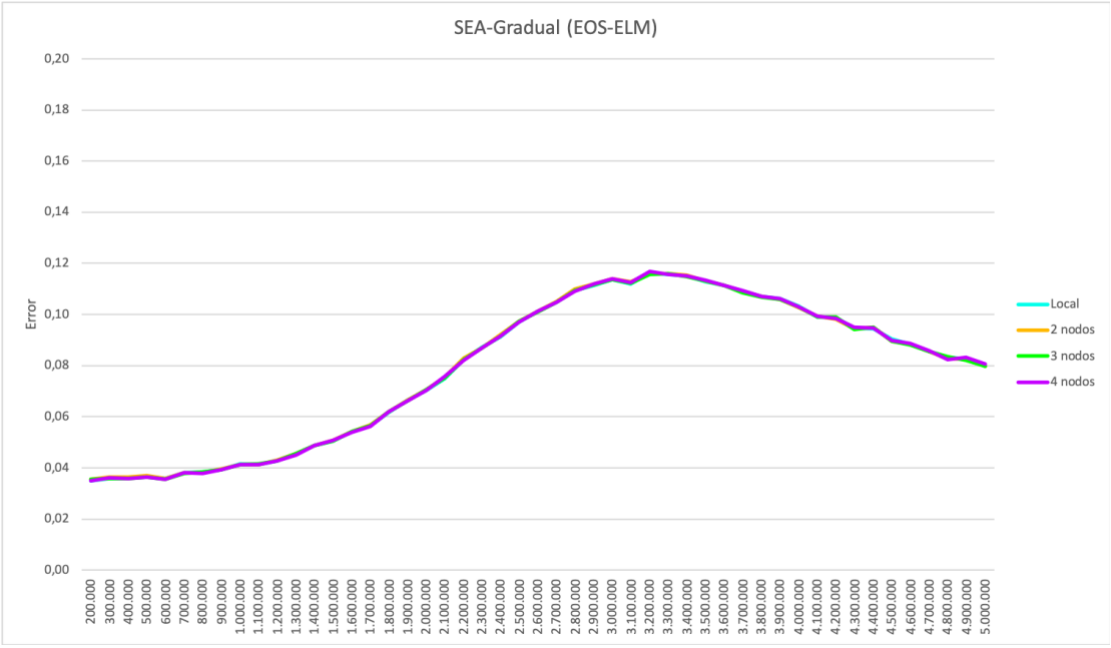

Figura 5.13. Test-thenTrain en SEA (Gradual) con EOS-ELM

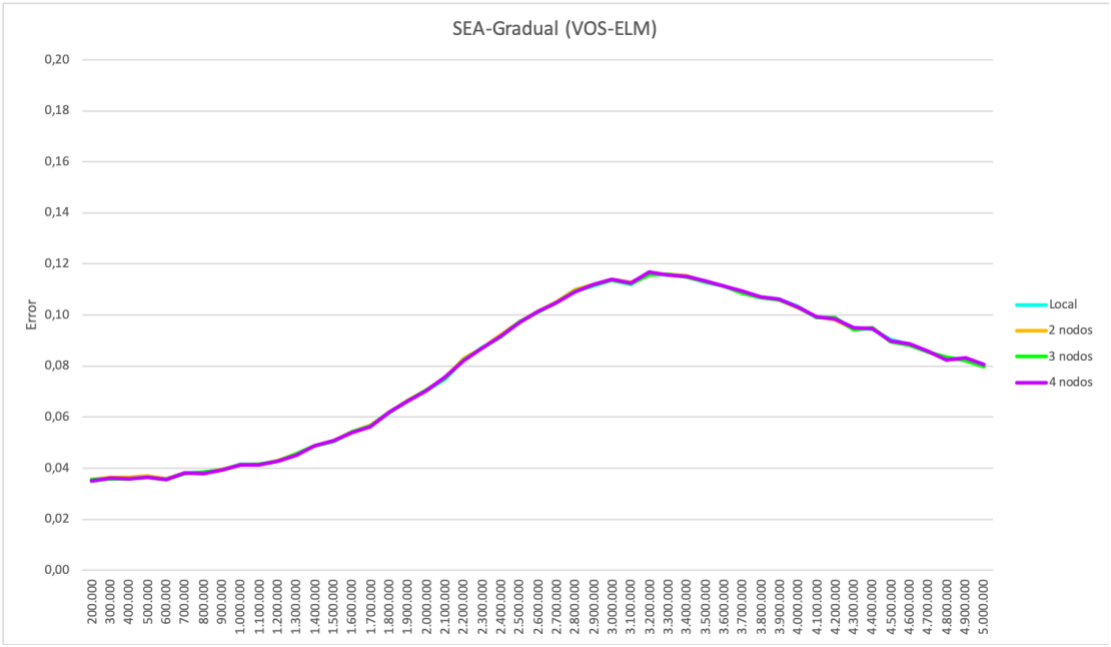

Figura 5.14. Test-thenTrain en SEA (Gradual) con VOS-ELM

Para el *dataset* SEA (Gradual), el algoritmo CD-VOS-ELM también se adapta más rápido, como muestra la Figura 5.15. En este caso el algoritmo no empieza a actualizar el modelo hasta que el error supera el 0,05. A partir de este valor, el modelo empieza a actualizarse (igual que EOS-ELM). El problema es que ESO-ELM se ha adaptado mejor al primer concepto, porque ha hecho una actualización constante, mientras que CD-VOS-ELM ha empezado a actualizar en la instancia 1.500.000. Esto le ha permitido olvidar el primer concepto más rápido.

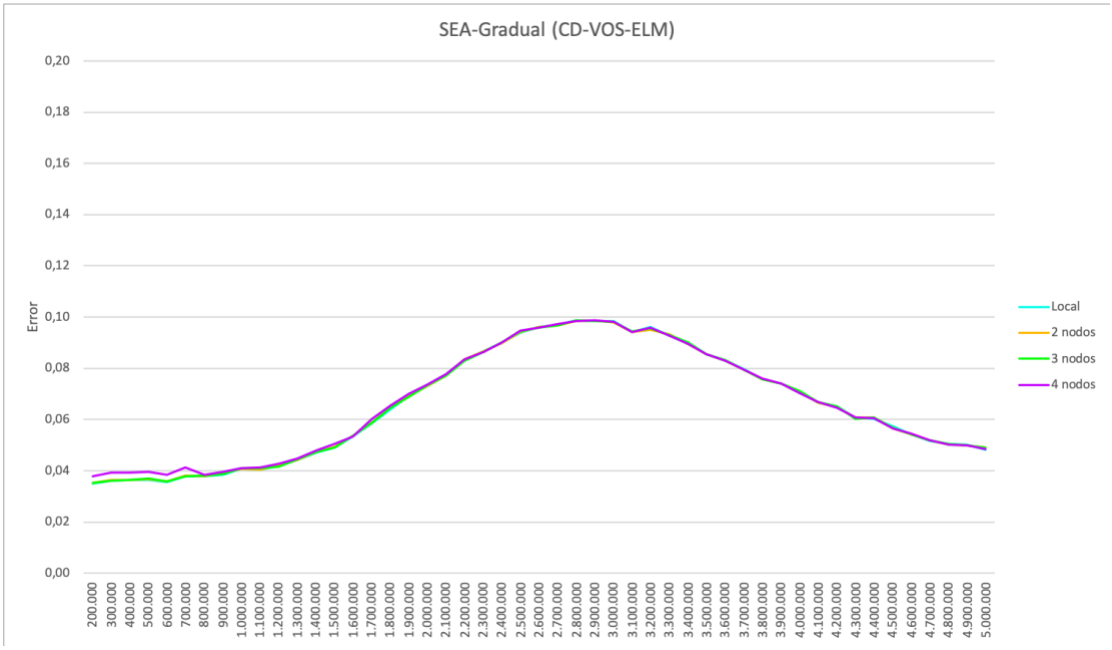

Figura 5.15. Test-thenTrain en SEA (Gradual) con CD-VOS-ELM

En la Figura 5.16 se muestra una comparativa de los 4 algoritmos con el *dataset* SEA (Gradual). En este caso pasa algo similar al análisis de SEA (Abrupto). El algoritmo OS-ELM tiene un error superior al resto, seguido de EOS-ELM y VOS-ELM y, finalmente, el que se adapta más rápido al nuevo concepto es CD-VOS-ELM. La diferencia es que en este caso no se adapta al instante, porque es un cambio gradual y el error se queda entre los límites superior e inferior. Esto obliga al algoritmo a actualizar el modelo, pero no descartarlo.

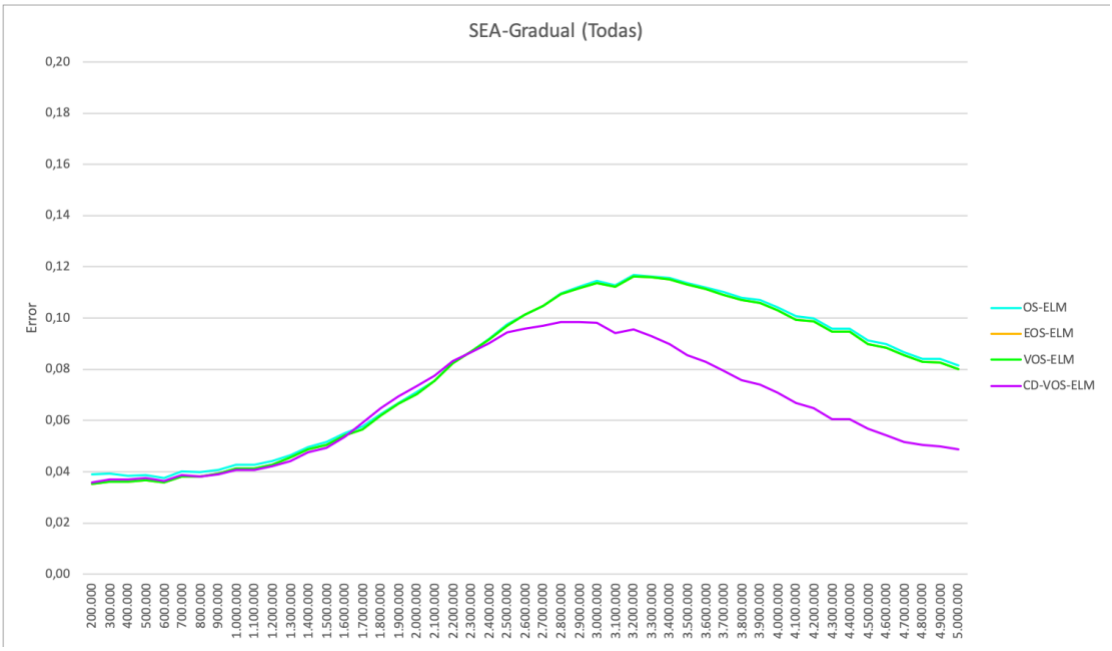

Figura 5.16. Test-thenTrain en SEA (Gradual) con todos

A continuación, se va a analizar el *dataset* SINE, que tiene un cambio abrupto en mitad del *dataset*. En este caso la Figura 5.17 muestra que se detecta el cambio de concepto, pero la actualización es muy lenta.

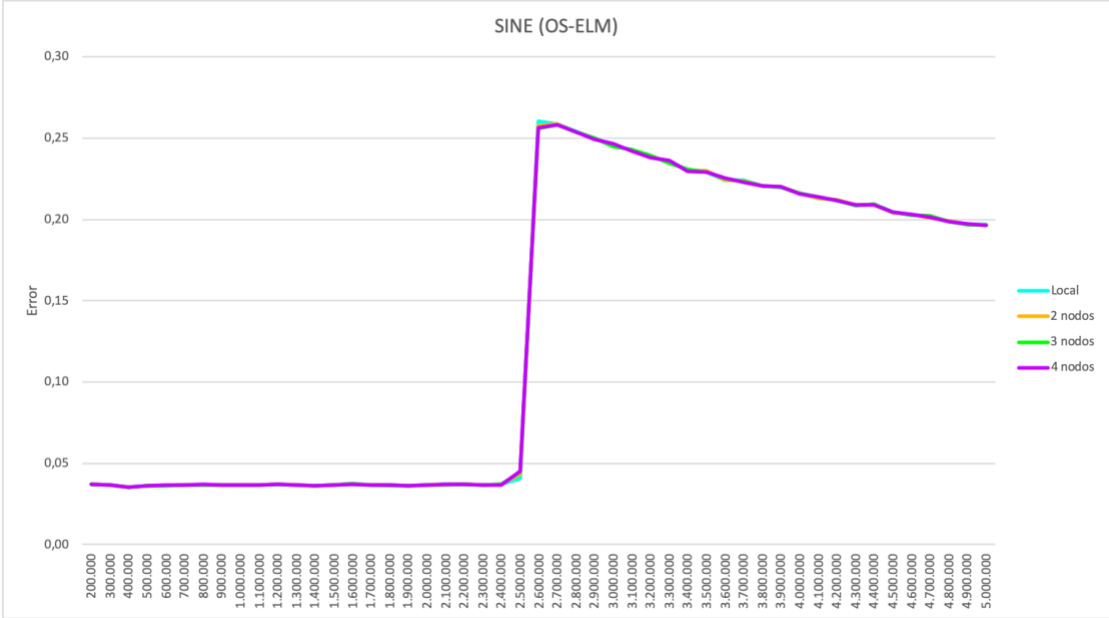

Figura 5.17. Test-thenTrain en SINE con OS-ELM

Como muestra la Figura 5.18, el algoritmo EOS-ELM inicia con un error menor que el algoritmo OS-ELM durante el primer concepto, pero al producirse el cambio de concepto el error es muy similar al de OS-ELM. A demás la actualización también es muy lenta.

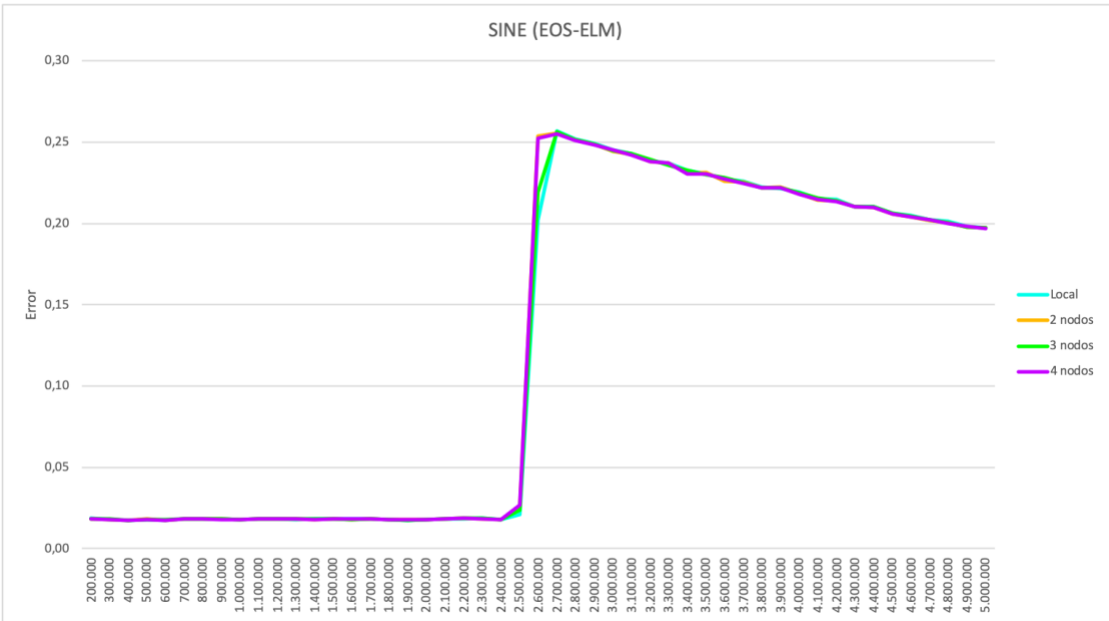

Figura 5.18. Test-thenTrain en SINE con EOS-ELM

El resultado para el algoritmo VOS-ELM se puede ver en la Figura 5.19. En este caso el *dataset* también es binario y genera el mismo resultado que EOS-ELM.

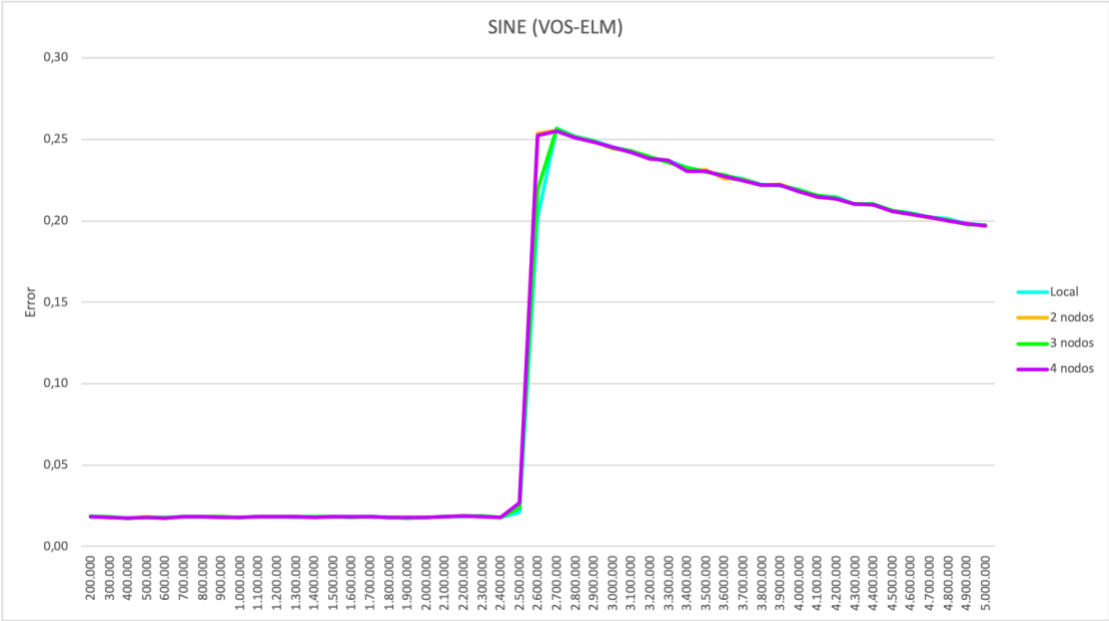

Figura 5.19. Test-thenTrain en SINE con VOS-ELM

La Figura 5.20 muestra el comportamiento del algoritmo CD-VOS-ELM con el *dataset* SINE. En este caso se vuelve a superar el límite superior, por lo que se elimina el modelo y se genera uno nuevo. El problema de este *dataset* es que estos algoritmos no pueden aprender bien el segundo concepto, por lo que el error se mantiene alto.

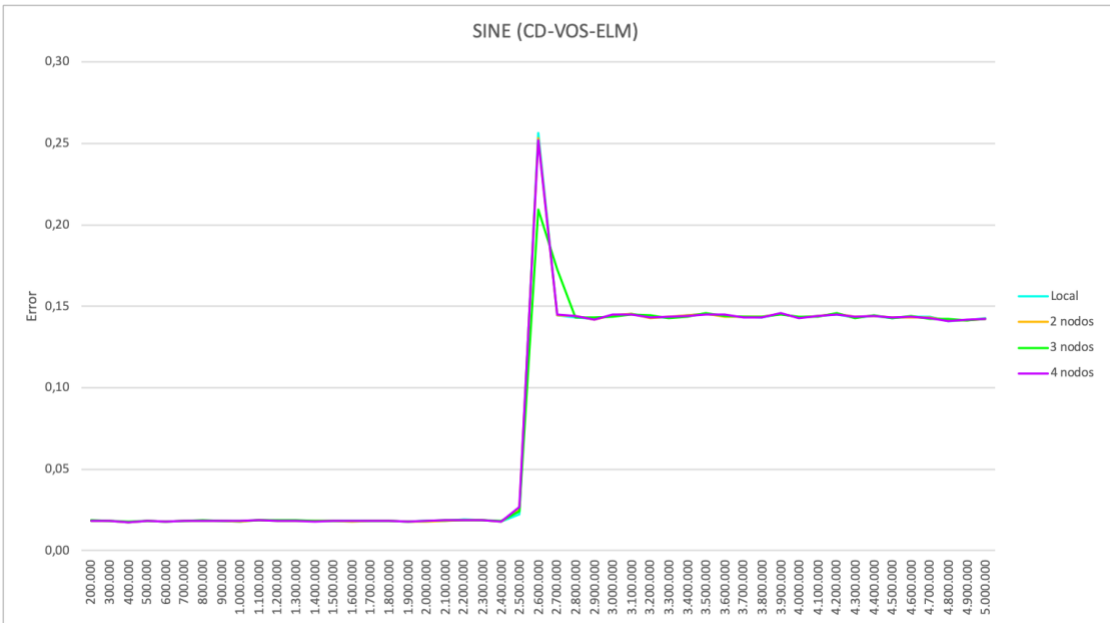

Figura 5.20. Test-thenTrain en SINE con CD-VOS-ELM

En este caso la Figura 5.21 muestra que el comportamiento es similar. OS-ELM tiene un mayor error, seguido de EOS-ELM y VOS-ELM y, finalmente, el algoritmo CD-VOS-ELM vuelve a ser el que mejor se adapta.

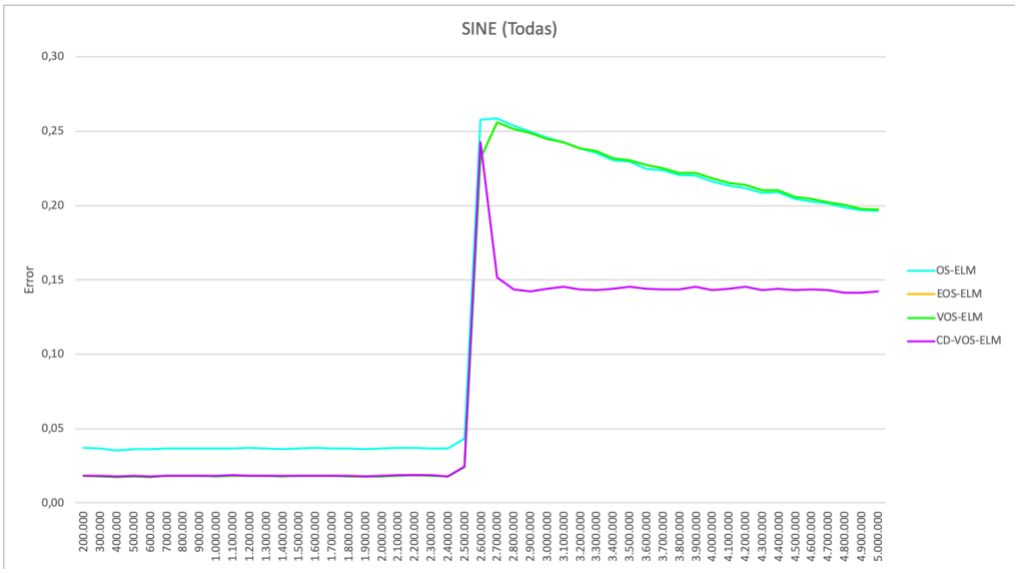

Figura 5.21. Test-thenTrain en SINE con todos

Finalmente, se va a analizar el comportamiento de los algoritmos implementados con el *dataset* Poker. Este *dataset* solo se ha ejecutado en modo local debido a que no tiene un gran número de instancias. El resultado se muestra en Figura 5.22, donde se puede apreciar que el error oscila entorno al 50%. En este caso EOS-ELM y VOS-ELM ofrecen el mismo resultado, aunque el *dataset* es multiclase. Esto se debe a que el *dataset* no está balanceado y estos algoritmos no están preparados para esto.

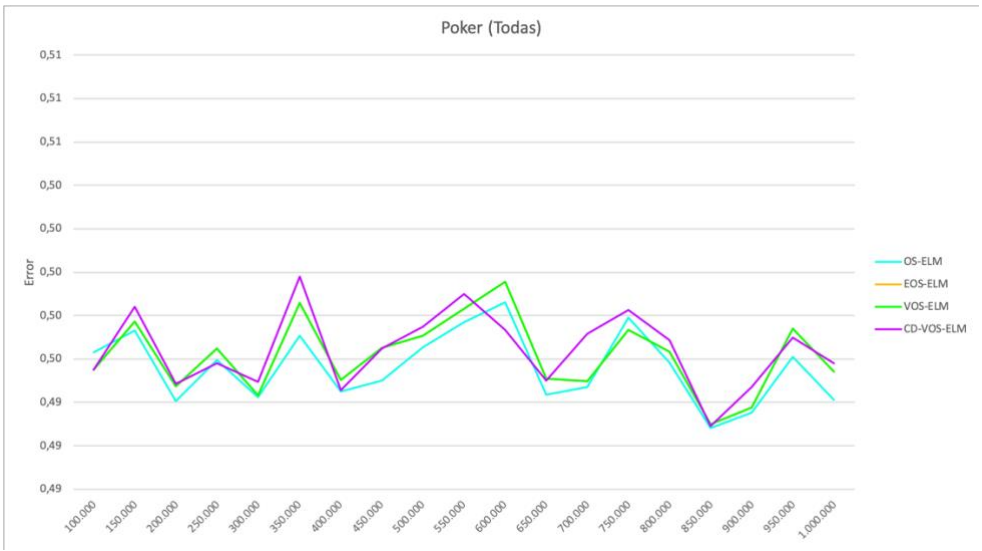

Figura 5.22. Test-thenTrain en Poker con todos

Para terminar esta sección, se van a analizar los tiempos de ejecución que aparecen en la Tabla 5.13. Como se puede apreciar al aumentar el número de nodos el tiempo de ejecución disminuye en todos los casos. Además, se aprecia que el tiempo de ejecución de EOS-ELM y VOS-ELM depende del número de modelos OS-ELM utilizados, ya que en lugar de entrenar un modelo hay que entrenar varios. Finalmente, se observa que el algoritmo CD-VOS-ELM es muy bueno en entornos con cambio de concepto (en especial cambios abruptos). Cuando el cambio es abrupto, como no tiene que ir actualizando el modelo el tiempo de ejecución es menor incluso que el algoritmo OS-ELM (SEA (Abrupto)). En los cambios graduales, hay una parte del *dataset* que no realiza actualización, por lo que el tiempo es bastante menor que el de EOS-ELM y VOS-ELM. En el caso del *dataset* Poker, como el error se mantiene entorno al 50%, el algoritmo está descartando el *ensemble* y generando uno nuevo con cada bloque. Esto hace que el tiempo sea mayor que en el resto de casos.

Dataset	Tam Bloque	Nº nodos	OS-ELM	EOS-ELM	VOS-ELM	CD-VOS-ELM
SEA (Abrupto)	100.000	Local	1.031.581	5.321.761	5.321.761	**442.146**
		2	732.536	3.630.931	3.630.931	**321.457**
		3	645.546	3.261.486	3.261.486	**288.471**
		4	623.284	3.134.568	3.134.568	**286.772**
SEA (Gradual)	100.000	Local	**1.029.285**	5.206.248	5.206.248	3.593.565
		2	**731.312**	3.610.355	3.610.355	2.509.502
		3	**649.203**	3.211.651	3.211.651	2.231.589
		4	**623.463**	3.126.891	3.126.891	2.126.192
Sine	100.000	Local	**1.030.081**	5.321.868	5.321.868	2.886.053
		2	**739.951**	3.652.328	3.652.328	2.016.071
		3	**651.544**	3.228.134	3.228.134	1.790.300
		4	**628.724**	3.112.745	3.112.745	1.750.297
Poker	50.000	Local	**259.469**	1.255.758	1.255.758	2.013.177

Tabla 5.13. Tiempos de entrenamiento streaming (en ms)

Como conclusiones de esta sección se destacan los siguientes puntos:

- Las implementaciones distribuidas de este trabajo ejecutadas en modo local generan resultados similares a las originales ejecutadas en secuencial sobre una máquina.
- Los tiempos obtenidos no son iguales que en los trabajos originales. Esto se debe a que en ellos se utilizan *datasets* muy pequeños y los algoritmos implementados emplean más tiempo enviando que procesando. Sin embargo, los algoritmos distribuidos van a ofrecer mejores tiempos para *datasets* con un elevado número de instancias.

- Normalmente en la bibliografía se demuestra que la precisión de los *ensembles* es superior a la de un clasificador único, aunque en esta experimentación el algoritmo OS-ELM ofrece resultados similares a EOS-ELM. Esto puede depender del número de clasificadores utilizados en el *ensemble* y del *dataset* clasificado.
- Para *datasets* binarios los resultados obtenidos en EOS-ELM y VOS-ELM son exactamente iguales. Esto se debe a que si, por ejemplo, hay más clasificadores que votan la clase 0, el resultado de la media (EOS-ELM) va a tender a 0 y el resultado de elegir la respuesta mayoritaria (VOS-ELM) también va a ser 0.
- En el entorno *streaming*, los algoritmos se adaptan incrementalmente al nuevo concepto, pero lo hacen a una velocidad constante. Los resultados obtenidos muestran que el uso de un detector de cambio de concepto (CD-VOS-ELM) mejora mucho el rendimiento del sistema (sobre todo para cambios de concepto abruptos). Esto hace que la velocidad de adaptación del modelo aumente, generando menos error y tiempos de ejecución.

6. CONCLUSIONES

En este trabajo se han estudiado diferentes plataformas que permiten definir un sistema de procesamiento en *streaming*. Se ha visto que en un sistema en *streaming* se pueden identificar 3 fases: recolección, procesamiento y visualización. Es este estudio solo nos centramos en las dos primeras.

Para la fase de recolección se han analizado apache Kafka, Flume y Nifi. De ellas, se ha demostrado que la más utilizada es Kafka debido a que ofrece ventajas sobre el resto, como una estructura dinámica en la que se pueden añadir/eliminar fuentes origen y destino sin necesidad de reiniciar el sistema y replicación de datos. Además, tiene una gran cantidad de documentación y compatibilidad con las plataformas de procesamiento debido a que es muy utilizada por la comunidad investigadora y en el ámbito empresarial.

Por otro lado, en la fase de procesamiento se han analizado apache Spark (*Streaming* y *Structured Streaming*), Storm, Flink, Samza, Apex y Beam. Para todas ellas, se ha llevado a cabo un estudio de las principales características de cada una, siendo mejor opción apache Spark, debido a que permite realizar un procesamiento con estado o sin estado con garantías de procesamiento exactamente una vez. Además, al ser muy utilizada, tiene una gran cantidad de documentación y cuenta con una librería de *machine learning* propia denominada MLLIB.

Para medir el rendimiento de las plataformas de procesamiento se han realizado dos comparativas de rendimiento. En la primera se ha utilizado el método *Yahoo Streaming Benchmark*, que simula una aplicación de anuncios. En esta prueba se mide el rendimiento y la latencia de las plataformas y la que mejores resultados ha generado es Flink, seguida de Spark. Tras obtener estos resultados, se ha realizado una segunda comparativa solo con Flink y Spark, ejecutándolas para procesar un *dataset* utilizando un algoritmo de *machine learning* denominado *streaming k-Means*. En este caso Spark tiene la implementación en MLLIB, pero Flink no, por lo que se ha implementado el algoritmo en Flink para realizar esta comparativa. El resultado muestra que Flink vuelve a ofrecer mejores resultados de tiempo de ejecución que Spark. Aunque se han obtenido mejores tiempos de procesamiento en Flink, en este trabajo se ha decidido utilizar Spark porque es más utilizado por la comunidad

investigadora y tiene una librería de *machine learning* más madura, lo que facilita el desarrollo de algoritmos.

Después de analizar diferentes plataformas disponibles para *streaming* y elegir apache Spark como plataforma de procesamiento, se han implementado distintos algoritmos de clasificación sobre Spark. Los algoritmos implementados son ELM, OS-ELM, EOS-ELM, VOS-ELM y CD-VOS-ELM, porque hay muy pocas implementaciones de redes neuronales en *streaming* y las pocas que existen no son en paralelo. De estos, ELM es un método tradicional y el resto en *streaming*. Para implementarlos, se ha utilizado la API de Spark (ELM) y Spark *Streaming* junto con la librería MLLIB. Se han realizado dos comparativas: una para comparar ELM con su versión en *streaming* (OS-ELM) en un entorno tradicional y la segunda para comparar OS-ELM, EOS-ELM, VOS-ELM y CD-VOS-ELM en un entorno en *streaming*. En la primera comparativa se puede observar que ambos algoritmos ofrecen la misma respuesta, por lo que se puede afirmar que la fase de inicialización coincide con el entrenamiento realizado en el algoritmo ELM. Además, los tiempos de entrenamiento son similares. En la segunda comparativa se puede destacar que el uso de *ensembles* estabiliza el error, pero aumenta los tiempos de ejecución al tener que entrenar varios modelos con cada bloque. Esto muestra que, dependiendo del problema, hay que evaluar si es útil utilizar un *ensemble* en lugar de una red OS-ELM. Además, se puede destacar que el uso de un método de detección de cambio de concepto mejora mucho el rendimiento general del sistema, ya que aumenta la velocidad con la que el modelo se adapta al concepto y solo actualiza el modelo si es necesario.

Finalmente, de los resultados obtenidos en estas dos experimentaciones sobre redes ELM, se puede afirmar que los tiempos se reducen bien con el número de nodos y merece la pena utilizar las versiones distribuidas cuando el tamaño del *dataset* (o de los bloques) es muy grande. Tal y como se esperaba, en el caso de *datasets* con pocos datos, la paralelización no solo no reduce el tiempo de ejecución, sino que lo puede hasta incrementar, dado que las implementaciones distribuidas tardan más tiempo en paralelizar que en procesar los datos.

Como conclusiones personales, he aprendido que existen muchas plataformas que se pueden utilizar en un sistema en *streaming*. Cada una ofrece unas características propias y, dependiendo de nuestras necesidades, puede ser mejor

utilizar unas u otras. El objetivo de este trabajo es enfocar el sistema en *streaming* a analizar datos utilizando técnicas de *machine learning*, por lo que después de los análisis y estudios realizados se ha llegado a la conclusión de que es mejor utilizar Spark. Además, he descubierto el tipo de redes ELM, que son redes neuronales muy rápidas, y su versión en *streaming* (OS-ELM) y sus variantes. Al realizar estas implementaciones he aprendido a utilizar apache Spark *Streaming* y su librería MLLIB y durante las experimentaciones a utilizar apache Kafka como plataforma de recolección. Finalmente, he comprobado la importancia de utilizar un detector de cambio de concepto en un algoritmo de *streaming*, que es muy importante en estos entornos en los que pueden aparecer cambios de concepto.

Bibliografía

[1] J. Han, J. Pei y M. Kamber, Data mining concepts and techniques, Elsevier, 2011.

[2] J. Gama, Knowledge discovery from data streams, Chapman and Hall/CRC, 2010.

[3] C. Aggarwal, Data streams: models and algorithms, Springer, 2007.

[4] F. Gürcan y M. Berigel, «Real-time processing of big data streams: Lifecycle, tools, tasks, and challenges,» de *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, 2018.

[5] A. Bifet, G. Holmes, R. Kirkby y B. Pfahringer, «Moa: Massive online analysis,» *Journal of Machine Learning Research,* pp. 1601--1604, 2010.

[6] B. Krawczyk, L. Minku, J. Gama, J. Stefanowski y M. Wozniak, «Ensemble learning for data stream analysis: A survey,» *Information Fusion,* pp. 132--156, 2017.

[7] I. Khamassi, M. Sayed-Mouchaweh, M. Hammami y K. Ghedira, «Discussion and review on evolving data streams and concept drift adapting,» *Evolving systems,* pp. 1--23, 2018.

[8] J. Gama, I. Zliobaite, A. Bifet, M. Pechenizkiy y A. Bouchachia, «A survey on concept drift adaptation,» *ACM computing surveys (CSUR),* p. 44, 2014.

[9] S. Ramírez-Gallego, B. Krawczyk, S. García, M. Wozniak y F. Herrera, «A survey on data preprocessing for data stream mining: Current status and future directions,» *Neurocomputing,* pp. 39--57, 2017.

[10] M. Lazarescu, S. Venkatesh y H. Bui, «Using multiple windows to track concept drift,» *Intelligent data analysis,* pp. 29--59, 2004.

[11] J. Gama, R. Sebastião y P. P. Rodrigues, «On evaluating stream learning algorithms,» *Machine learning,* pp. 317--346, 2013.

[12] A. S. Iwashita y J. P. Papa, «An Overview on Concept Drift Learning,» *IEEE,* pp. 1532--1547, 2018.

[13] N. Heudecker y W. R. Schulte, «Market guide forevent stream processing,» 2018.

[14] W. R. Schulte y N. Heudecker, «Technology insightfor event stream processing,» 2017.

[15] MvnRepository, «MVN Repository,» 2019. [En línea]. Available: https://mvnrepository.com/. [Último acceso: 14 Enero 2019].

[16] A. S. F. «Apache Kafka,» 15 03 2019. [En línea]. Available: https://kafka.apache.org/documentation/. [Último acceso: 14 Enero 2019].

[17] A. S. F. «Apache Flume,» 15 03 2019. [En línea]. Available: https://flume.apache.org/documentation.html. [Último acceso: 14 Enero 2019].

[18] A. S. Foundation, «Apache Hadoop,» 2019. [En línea]. Available: https://hadoop.apache.org/. [Último acceso: 14 Enero 2019].

[19] A. S. F. «Apache Nifi,» 15 03 2019. [En línea]. Available: https://nifi.apache.org/docs.html. [Último acceso: 14 Enero 2019].

[20] A. S. F. «Apache Spark Streaming,» 15 03 2019. [En línea]. Available: https://spark.apache.org/docs/2.4.0/streaming-programming-guide.html. [Último acceso: 14 Enero 2019].

[21] A. S. F. «Apache Spark Structured Streaming,» 15 03 2019. [En línea]. Available: https://spark.apache.org/docs/2.4.0/structured-streaming-programming-guide.html. [Último acceso: 14 Enero 2019].

[22] A. S. F. «Apache Storm,» 15 03 2019. [En línea]. Available: https://storm.apache.org/releases/1.2.2/index.html. [Último acceso: 14 Enero 2019].

[23] A. S. F. «Apache Flink,» 15 03 2019. [En línea]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.7/. [Último acceso: 14 Enero 2019].

[24] A. S. F. «Apache Hadoop YARN,» 15 03 2019. [En línea]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. [Último acceso: 14 Enero 2019].

[25] A. S. F. «Apache Samza,» 15 03 2019. [En línea]. Available: http://samza.apache.org/learn/documentation/1.0.0/core-concepts/core-concepts.html. [Último acceso: 14 Enero 2019].

[26] A. S. F. «Apache Apex,» 15 03 2019. [En línea]. Available: https://apex.apache.org/docs.html. [Último acceso: 14 Enero 2019].

[27] A. S. F. «Apache Beam,» 15 03 2019. [En línea]. Available: https://beam.apache.org/documentation/. [Último acceso: 14 Enero 2019].

[28] A. S. F. «MLLIB,» 15 03 2019. [En línea]. Available: https://spark.apache.org/docs/latest/mllib-guide.html. [Último acceso: 14 Enero 2019].

[29] H. N. A. Lab, «StreamDM,» 15 03 2019. [En línea]. Available: http://huawei-noah.github.io/streamDM/. [Último acceso: 14 Enero 2019].

[30] A. S. F. «Apache SAMOA,» 15 03 2019. [En línea]. Available: https://samoa.incubator.apache.org/documentation/Home.html. [Último acceso: 14 Enero 2019].

[31] S. F. Programme, «Amidst-Toolbox,» 15 03 2019. [En línea]. Available: http://www.amidsttoolbox.com/documentation/. [Último acceso: 14 Enero 2019].

[32] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil y B. J. Peng, «Benchmarking streaming computation engines: Storm, flink and spark streaming,» de *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, 2016.

[33] E. Shahverdi, «Comparative Evaluation for the Performance of Big Data Stream Processing Systems,» 2018.

[34] T. Dasgupta, «Evaluation of two major data stream processing technologies,» 2016.

[35] Z. Karakaya, A. Yazici y M. Alayyoub, «A Comparison of Stream Processing Frameworks,» de *2017 International Conference on Computer and Applications (ICCA)*, 2017.

[36] L. Li, R. Sun, S. Cai, K. Zhao y Q. Zhang, «A review of improved extreme learning machine methods for data stream classification,» *Multimedia Tools and Applications,* pp. 1--26, 2019.

[37] H.-L. Nguyen, Y.-K. Woon y W.-K. Ng, «A survey on data stream clustering and classification,» *Knowledge and information systems,* vol. 45, nº 3, pp. 535--569, 2015.

[38] A. Haneen, A. Noraziah y M. H. Abd Wahab, «A Review on Data Stream Classification,» de *1st International Conference on Big Data and Cloud Computing (ICoBiC)*, 2017.

[39] C. Aggarwal, «A Survey of Stream Classification Algorithms,» de *Data Classification: Algorithms and Applications*, Chapman and Hall/CRC, 2014, pp. 245--273.

[40] H. M. Gomes, J. P. Barddal, F. Enembreck y A. Bifet, «A Survey on Ensemble Learning for Data Stream Classification,» *ACM Computing Surveys (CSUR),* vol. 50, nº 2, p. 23, 2017.

[41] G.-B. Huang, Q.-Y. Zhu y C.-K. Siew, «Extreme Learning Machine: A New Learning Scheme of Feedforward Neural Networks,» *Neural networks,* vol. 2, nº 985--990, 2004.

[42] N.-Y. Liang, G.-B. Huang, P. Saratchandran y N. Sundararajan, «A Fast and Accurate Online Sequential Learning Algorithm for Feedforward Networks,» *IEEE Transactions on neural networks,* vol. 17, nº 6, pp. 1411--1423, 2006.

[43] Y. Lan, Y. C. Soh y G.-B. Huang, «A Constructive Enhancement for Online Sequential Extreme Learning Machine,» de *2009 International Joint Conference on Neural Networks*, 2009.

[44] Y. Lan, Y. C. Soh y G.-B. Huang, «Ensemble of online sequential extreme learning machine,» *Neurocomputing,* vol. 72, nº 13-15, pp. 3391--3395, 2009.

[45] J. Zhao, Z. Wang y D. S. Park, «Online sequential extreme learning machine with forgetting mechanism,» *Neurocomputing,* vol. 87, pp. 79--89, 2012.

[46] B. Mirza, Z. Lin y N. Liu, «Ensemble of subset online sequential extreme learning machine for class imbalance and concept drift,» *Neurocomputing,* vol. 149, pp. 316--329, 2015.

[47] J. Cao, Z. Lin y G.-B. Huang, «Voting base Online Sequential Extreme Learning Machine for Multi-class Classification,» de *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, 2013.

[48] B. Mirza, Z. Lin y K.-A. Toh, «Weighted Online Sequential Extreme Learning Machine for Class Imbalance Learning,» *Neural processing letters,* vol. 38, nº 3, pp. 465--486, 2013.

[49] B. Mirza, Z. Lin, J. Cao y X. Lai, «Voting based Weighted Online Sequential Extreme Learning Machine for Imbalance Multi-Class Classification,» *2015 IEEE International Symposium on Circuits and Systems (ISCAS),* pp. 565--568, 2015.

[50] S. Xu y J. Wang, «A fast incremental extreme learning machine algorithm for data streams classification,» *Expert Systems with Applications,* vol. 65, pp. 332--344, 2016.

[51] S. Xu y J. Wang, «Dynamic extreme learning machine for data stream classification,» *Neurocomputing,* vol. 238, pp. 433--449, 2017.

[52] R. Yang, S. Xu y L. Feng, «An Ensemble Extreme Learning Machine for Data Stream Classification,» *Algorithms,* vol. 11, nº 7, 2018.

[53] N. S. Foundation, «UCI,» 2019. [En línea]. Available: https://archive.ics.uci.edu/ml/datasets.php. [Último acceso: 14 Enero 2019].